고려대학교
KOREA UNIVERSITY

COSE321 Computer Systems Design

Lecture 2. ARM Instructions

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

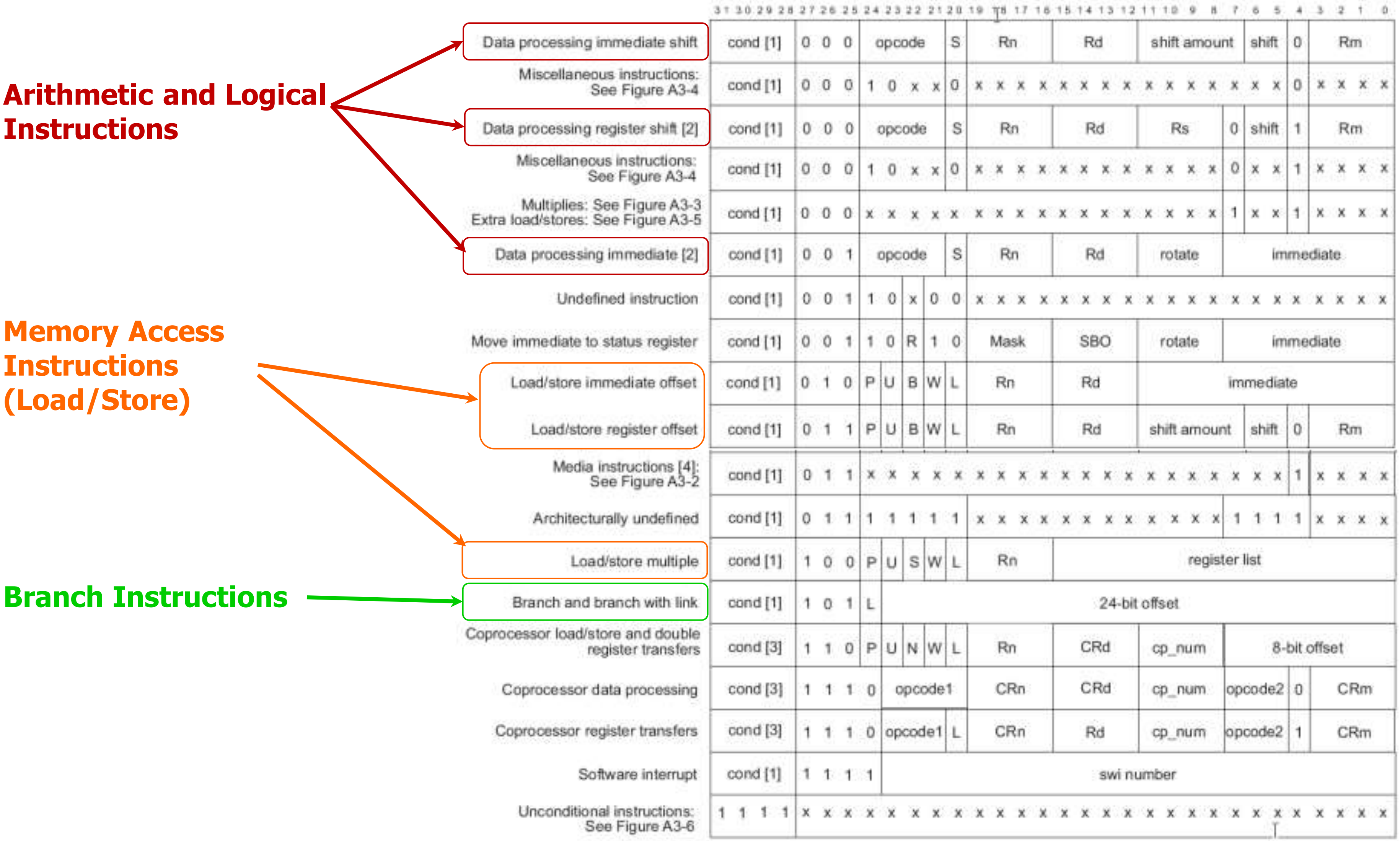
ARM Instruction Overview

- ARM is a **RISC** machine, so the instruction length is fixed
  - In ARM mode, instructions are 32-bit wide
  - In Thumb2 mode, instructions are 16-bit wide or 32-bit wide

- **Most ARM instructions** can be **conditionally executed**
  - It means that they have their normal effect only if the N (Negative), Z (Zero), C (Carry) and V (Overflow) flags in the CPSR satisfy a condition specified in the instruction
    - If the flags do not satisfy this condition, the instruction acts as a NOP (No Operation)
    - In other words, the instruction has no effect and advances to the next instruction

* Unconditional instruction examples: SRS (save return status), RFE (return from exception)

ARM Instruction Format



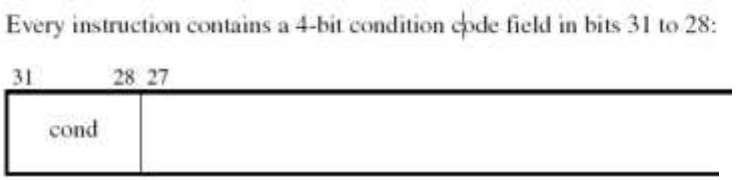
Arithmetic and Logical Instructions

Memory Access Instructions (Load/Store)

Branch Instructions

	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
Data processing immediate shift	cond [1]	0	0	0	opcode				S	Rn	Rd	shift amount		shift	0	Rm
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	x	x x	0	x x x x
Data processing register shift [2]	cond [1]	0	0	0	opcode				S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	0	x x	1	x x x x
Multiplies: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	1	x x	1	x x x x
Data processing immediate [2]	cond [1]	0	0	1	opcode				S	Rn	Rd	rotate	immediate			
Undefined instruction	cond [1]	0	0	1	1	0	x	0	0	x x x x	x x x x	x x x x	x	x x	x	x x x x
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask	SBO	rotate	immediate			
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn	Rd	immediate				
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn	Rd	shift amount		shift	0	Rm
Media instructions [4]: See Figure A3-2	cond [1]	0	1	1	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	1	x x x x
Architecturally undefined	cond [1]	0	1	1	1	1	1	1	1	x x x x	x x x x	x x x x	1	1 1	1	x x x x
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn	register list					
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset										
Coprocessor load/store and double register transfers	cond [3]	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8-bit offset			
Coprocessor data processing	cond [3]	1	1	1	0	opcode1				CRn	CRd	cp_num	opcode2		0	CRm
Coprocessor register transfers	cond [3]	1	1	1	0	opcode1			L	CRn	Rd	cp_num	opcode2		1	CRm
Software interrupt	cond [1]	1	1	1	1	swi number										
Unconditional instructions: See Figure A3-6	1 1 1 1	x	x	x	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	x	x x x x

Condition Field



Every instruction contains a 4-bit condition code field in bits 31 to 28:

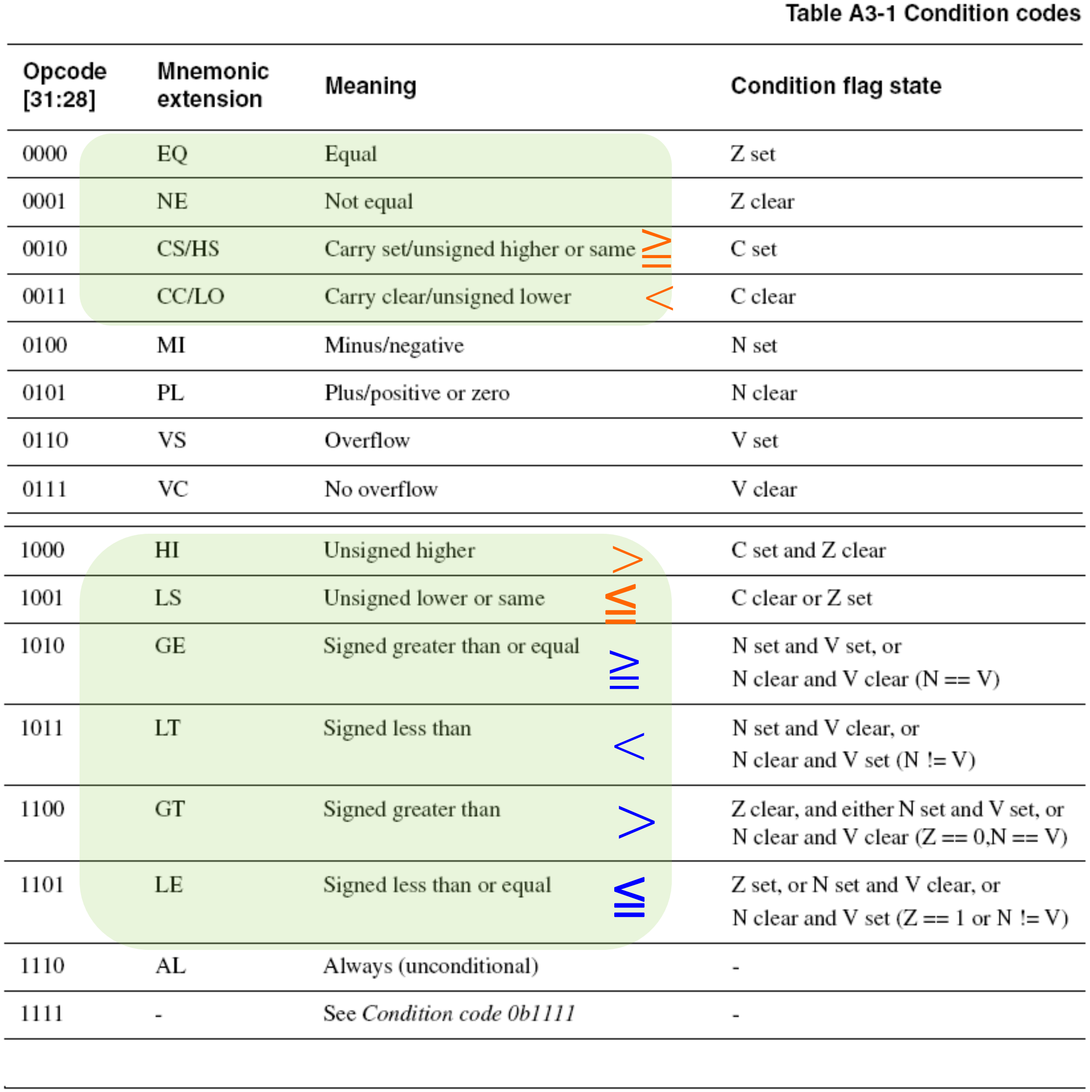


Table A3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same ≧	C set
0011	CC/LO	Carry clear/unsigned lower <	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher >	C set and Z clear
1001	LS	Unsigned lower or same ≦	C clear or Z set
1010	GE	Signed greater than or equal ≧	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than <	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than >	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal ≦	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-

Flags

	Which flags would you check? (N, Z, C, V)
Unsigned higher	ua > ub ? **C = 1**
Unsigned lower	ua < ub ? **C = 0**
Signed greater than	sa > sb ?
Signed less than	sa < sb ?

Signed greater than

sa > sb? Yes if (N == V)

- **(+) - (+) : N=0 & V=0**
- **(+) - (-) : N=0 & V=0 or**
  : N=1 & V=1
- **(-) - (+) : N=1 & V=0 or**
  : N=0 & V=1
- **(-) - (-) : N=0 & V=0**

Signed less than

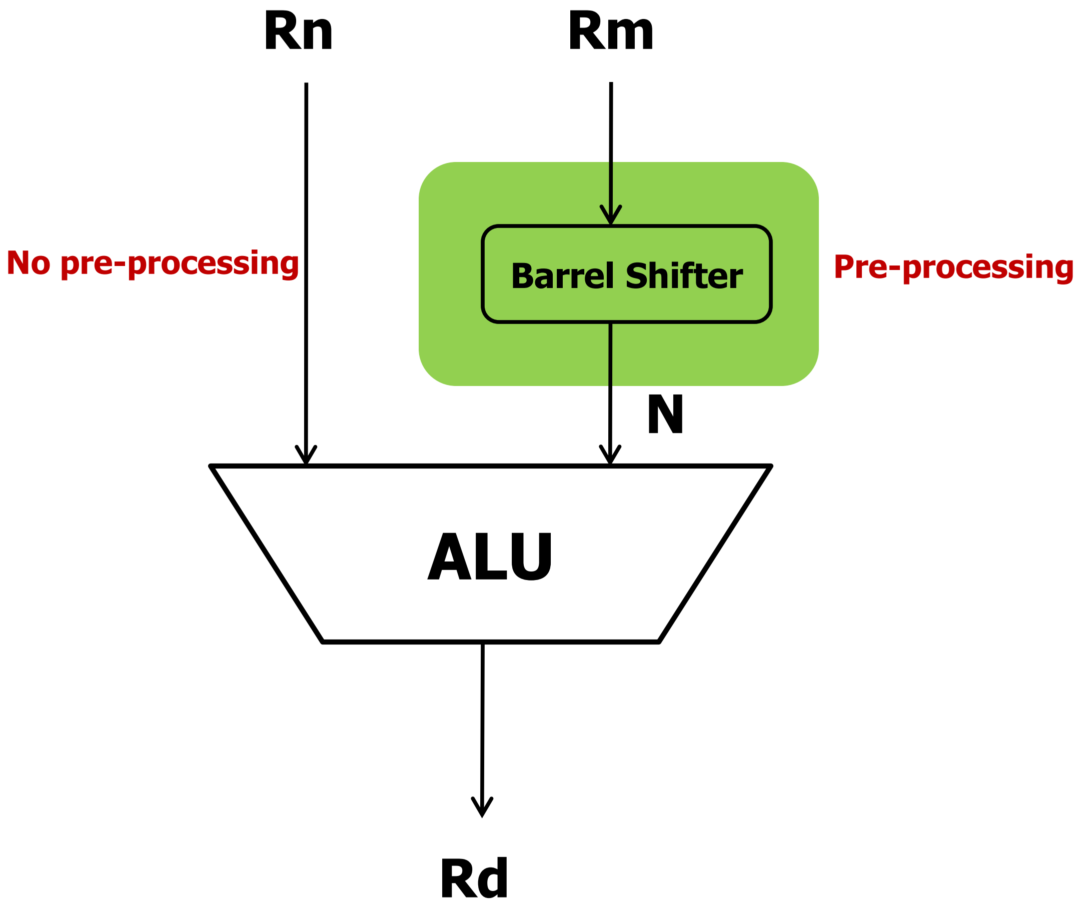
sa < sb? Yes if (N != V)

- **(+) - (+) : N=1 & V=0**
- **(+) - (-) : N=0 & V=0 or**
  : N=1 & V=1
- **(-) - (+) : N=1 & V=0 or**
  : N=0 & V=1
- **(-) - (-) : N=1 & V=0**

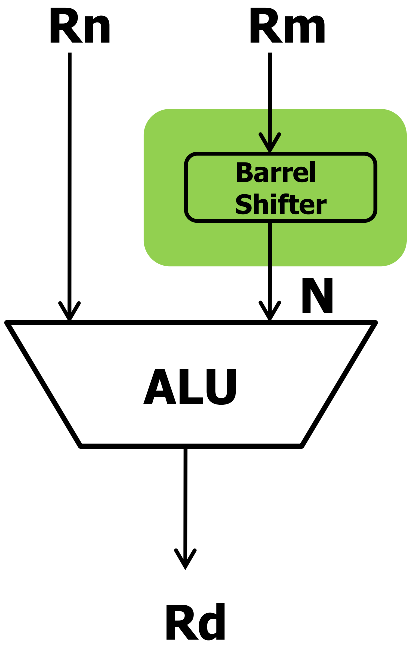
Data Processing Instructions

- Move instructions
- Arithmetic instructions
- Logical instructions
- Comparison instructions

Execution Unit in ARM



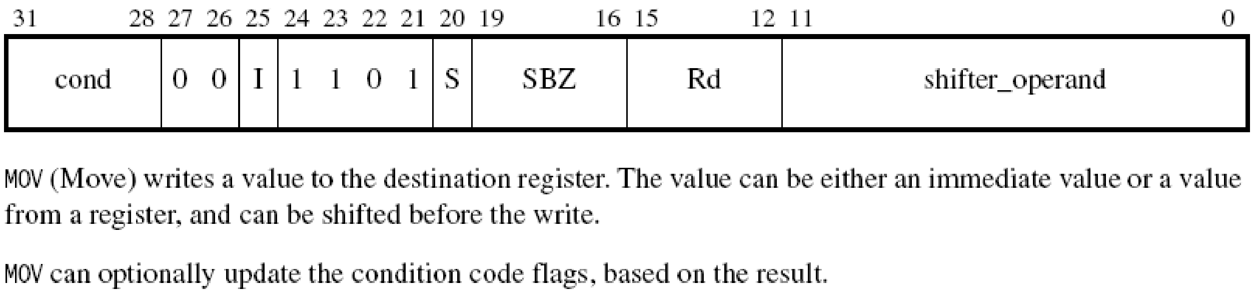
Move Instructions



Syntax: <instruction>{cond}{S} Rd, N

MOV	Move a 32-bit value into a register	*Rd = N*
MVN	Move the NOT of the 32-bit value into a register	*Rd = ~ N*

Move Instructions – MOV



31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 0 1	S	SBZ	Rd	shifter_operand

MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

MOV can optionally update the condition code flags, based on the result.

- MOV loads a value into the destination register (Rd) from another register, a shifted register, or an immediate value
  - **Useful to setting initial values and transferring data between registers**
  - It updates the carry flag (C), negative flag (N), and zero flag (Z) if S bit is set
    - C is set from the result of the barrel shifter

```
MOV R0, R0; move R0 to R0, Thus, no effect
MOV R0, R0, LSL#3 ;  R0 = R0 * 8
MOV PC, R14; (R14: link register) Used to return to caller
MOVS PC, R14; PC <- R14 (lr), CPSR <- SPSR
              ; Used to return from interrupt or exception
```

* **SBZ**: should be zeros

MOV Example

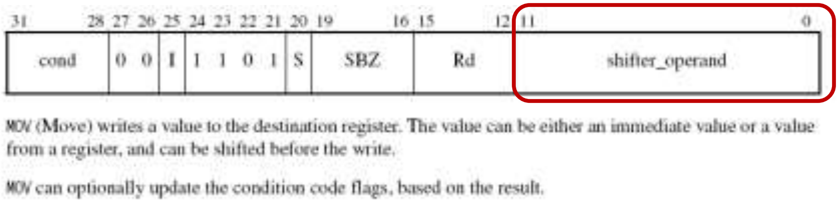
Before: cpsr = nzcv
r0 = 0x0000_0000
r1 = 0x8000_0004

MOVS r0, r1, LSL #1

After: cpsr = nz**C**v
r0 = 0x0000_0008
r1 = 0x8000_0004

Rm with Barrel Shifter

MOV r0, r1, LSL #1



31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	1	1	0	1	S	SBZ	Rd	shifter_operand

MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

MOV can optionally update the condition code flags, based on the result.

Encoded here

Shift Operation (for Rm)	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right by register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #shift_imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend (Rotate right by 1-bit with carry)	Rm, RRX

LSL: Logical Shift Left

LSR: Logical Shift Right

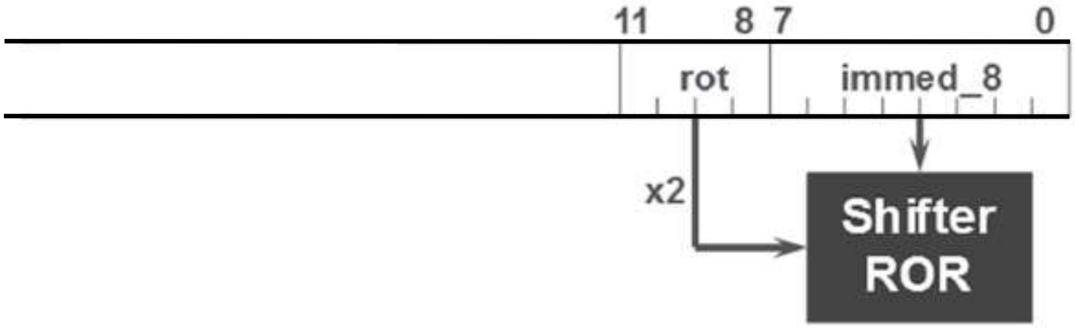
ASR: Arithmetic Shift Right

ROR: Rotate Right

RRX: Rotate Right with Extend

Immediate Constants

- No ARM instruction can contain a 32-bit immediate constant
- Data processing instruction format has 12-bits for N (operand2)
- Immediate N can be created by 8-bits rotated right by an even number of bit positions



Examples:

```
• mov r0, #0x12
• mov r0, #0x1200       // 0x12 rotate right by 24-bit
• mov r0, #0x12000000   // 0x12 rotate right by 8-bit
```

Arm has other instruction encodings to allow the 12-bit immediate or 16-bit immediate

- ‘mov r0, #0x123’ is converted to ‘movw r0, #0x123’
- ‘mov r0, #0x1234’ is converted to ‘movw r0, #0x1234’

Loading 32-bit Constants

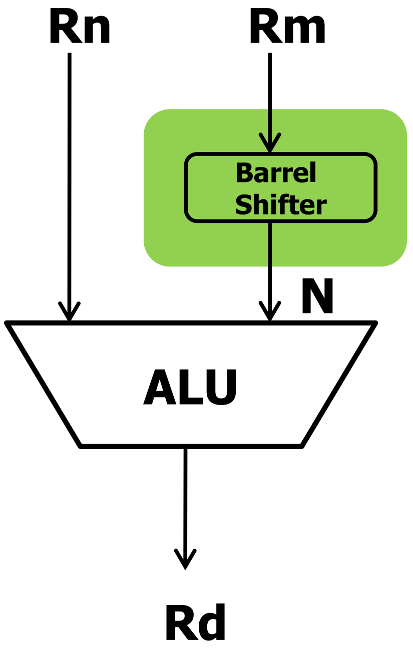
- To load large constants, the assembler provides a pseudo instruction:

```
LDR rd, =const
```

  - It will either produce a `MOV` or `MVN` instruction to generate the value if possible or generate a `LDR` instruction with a PC-relative address to read the constant from a literal pool

```
LDR r0, =0xFF            =>    MOV r0, #0xFF
LDR r0, =0x55555555      =>    LDR r0, [PC, #imm12]
                              ...
                              DCD  0x55555555


        LDR Rn, =<constant>
        LDR Rn, =label
        ...
label:  ADD …
```

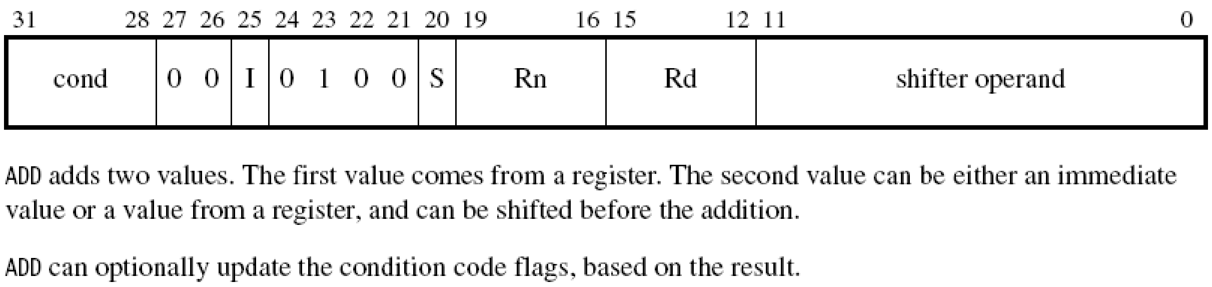
Arithmetic Instructions



Syntax: <instruction>{cond}{S} Rd, Rn, N

ADD	add two 32-bit values	*Rd = Rn + N*
ADC	add two 32-bit values with carry	*Rd = Rn + N + carry*
SUB	subtract two 32-bit values	*Rd = Rn - N*
SBC	subtract two 32-bit values with carry	*Rd = Rn - N - !C*
RSB	reverse subtract of two 32-bit values	*Rd = N – Rn*
RSC	reverse subtract of two 32-bit values with carry	*Rd = N – Rn - !C*

Arithmetic Instructions – ADD



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	0	0	S	Rn	Rd	shifter operand

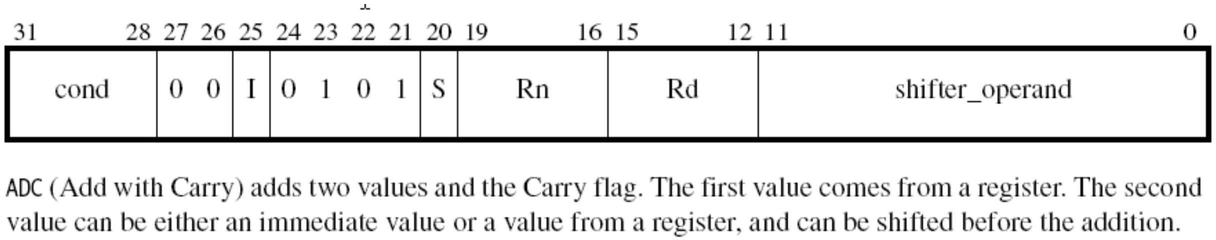
ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

- `ADD` adds two operands, placing the result in Rd
  - Use S suffix to update conditional field
  - The addition may be performed on signed or unsigned numbers

```
ADD  R0, R1, R2 ; R0 = R1 + R2
ADD  R0, R1, #256 ; R0 = R1 + 256
ADDS R0, R2, R3,LSL#1 ; R0 = R2 + (R3 << 1) and update flags
```

Arithmetic Instructions – ADC



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	0	1	S	Rn	Rd	shifter_operand

ADC (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

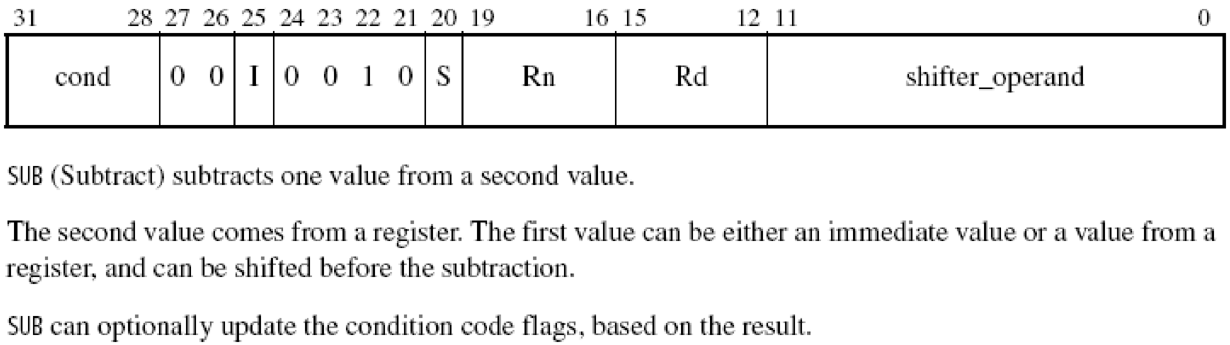
- ADC adds two operands with a carry bit, placing the result in Rd
  - It uses a carry bit, so can add numbers larger than 32 bits
  - Use S suffix to update conditional field

<64-bit addition>

64 bit 1st operand: R5 and R4
64 bit 2nd operand: R9 and R8
64 bit result: R1 and R0

```
ADDS R0, R4, R8 ; R0 = R4 + R8 and set carry accordingly
ADCS R1, R5, R9 ; R1 = R5 + R9 + (Carry flag)
```

Arithmetic Instructions – SUB



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	1	0	S	Rn	Rd	shifter_operand

SUB (Subtract) subtracts one value from a second value.

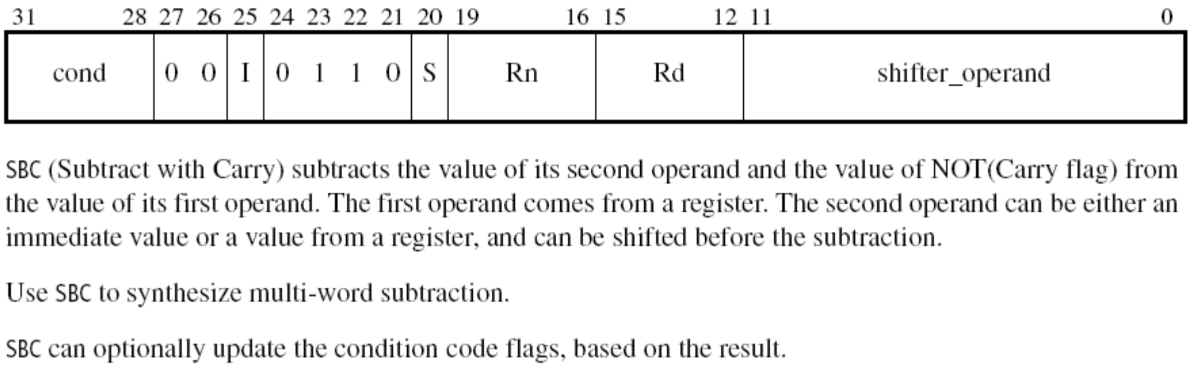
The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

SUB can optionally update the condition code flags, based on the result.

- `SUB` subtracts operand 2 from operand 1, placing the result in Rd
  - Use S suffix to update conditional field
  - The subtraction may be performed on signed or unsigned numbers

```
SUB   R0, R1, R2 ; R0 = R1 - R2
SUB   R0, R1, #256 ; R0 = R1 - 256
SUBS   R0, R2, R3,LSL#1 ; R0 = R2 - (R3 << 1) and update flags
```

Arithmetic Instructions – SBC



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	1	0	S	Rn	Rd	shifter_operand

SBC (Subtract with Carry) subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand. The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.

Use SBC to synthesize multi-word subtraction.

SBC can optionally update the condition code flags, based on the result.

- `SBC` subtracts operand 2 from operand 1 with the carry flag, placing the result in Rd
  - It uses a carry bit, so can subtract numbers larger than 32 bits.
  - Use S suffix to update conditional field

<64-bit Subtraction>

64 bit 1st operand: R5 and R4
64 bit 2nd operand: R9 and R8
64 bit result: R1 and R0

```
SUBS  R0, R4, R8 ; R0 = R4 – R8
SBC   R1, R5, R9 ; R1 = R5 – R9 - !(carry flag)
```

Examples

Before:

r0 = 0x0000_0000
r1 = 0x0000_0002
r2 = 0x0000_0001

```
SUB  r0, r1, r2
```

After:

r0 = 0x0000_0001
r1 = 0x0000_0002
r2 = 0x0000_0001

Before:

r0 = 0x0000_0000
r1 = 0x0000_0077

```
RSB  r0, r1, #0
 // r0 = 0x0 – r1
```

After:

r0 = 0xFFFF_FF89
r1 = 0x0000_0077

Before:

r0 = 0x0000_0000
r1 = 0x0000_0005

```
ADD  r0, r1, r1, LSL#1
```

After:

r0 = 0x0000_000F
r1 = 0x0000_0005

Examples

Before: cpsr = nzcv

r1 = 0x0000_0001

SUBS r1, r1, #1

After: cpsr = n**ZC**v

r1 = 0x0000_0000

- Why is the C flag set (C = 1)?

CLZ (Count Leading Zeros)

- `CLZ` returns the number of leading binary 0 bits before the first binary 1 bit appears

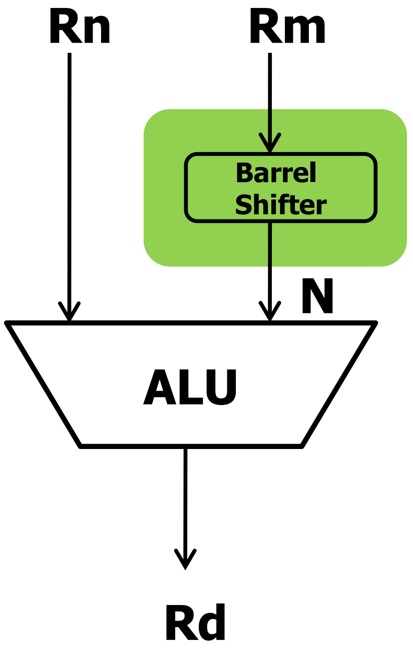
```
CLZ{cond}  Rd, Rm
```

  - Return 32 if no bits set
  - Return 0 if bit 31 is set

<Normalization Example>

```
R0 = 0000_0010_1100_...

CLZ  R1, R0;  // R1 = 6
MOV  R0, R0, LSL R1 ; R0 = 1011_00...
```

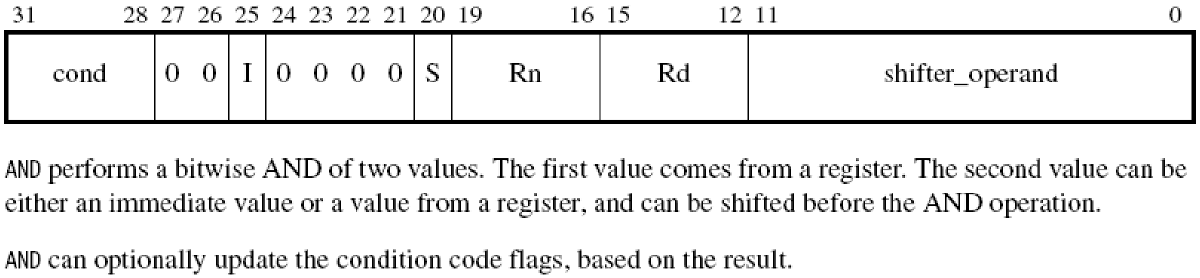
Logical Instructions



Syntax: <instruction>{cond}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	*Rd = Rn & N*
ORR	logical bitwise OR of two 32-bit values	*Rd = Rn | N*
EOR	logical exclusive OR of two 32-bit values	*Rd = Rn ^ N*
BIC	logical bit clear	*Rd = Rn & ~N*

Logical Instructions – AND



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	0	0	S	Rn	Rd	shifter_operand

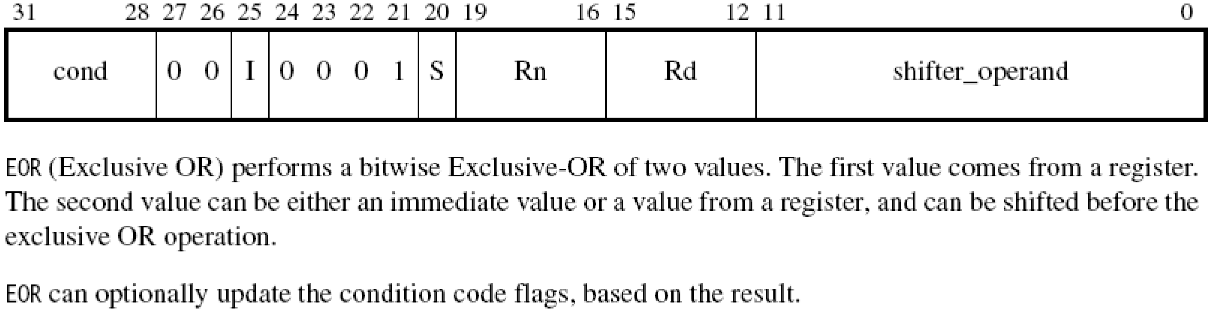
AND performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

AND can optionally update the condition code flags, based on the result.

- AND performs a logical AND between the two operands, placing the result in Rd
  - It is useful for masking the bits

```
AND  R0, R0, #3 ; Keep bits zero and one of R0 and discard the rest
```

Logical Instructions – EOR



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	0	1	S	Rn	Rd	shifter_operand

EOR (Exclusive OR) performs a bitwise Exclusive-OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the exclusive OR operation.

EOR can optionally update the condition code flags, based on the result.

- EOR performs a logical Exclusive OR between the two operands, placing the result in the destination register
  - It is useful for inverting certain bits

```
EOR  R0, R0, #3 ; Invert bits zero and one of R0
```

Examples

Before:

r0 = 0x0000_0000
r1 = 0x0204_0608
r2 = 0x1030_5070

ORR r0, r1, r2

After:

r0 = 0x1234_5678

Before:

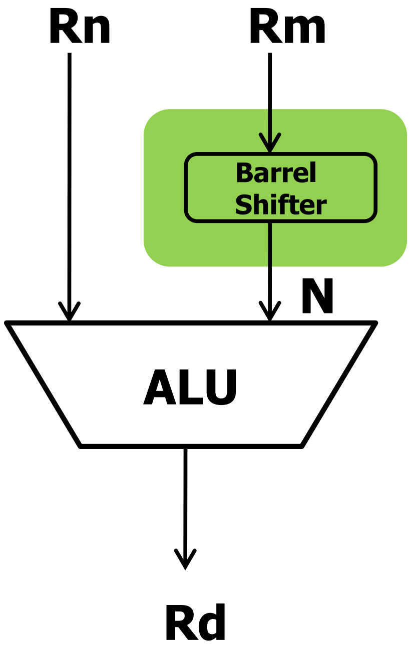
r1 = 0b1111
r2 = 0b0101

BIC r0, r1, r2

After:

r0 = 0b1010

Comparison Instructions



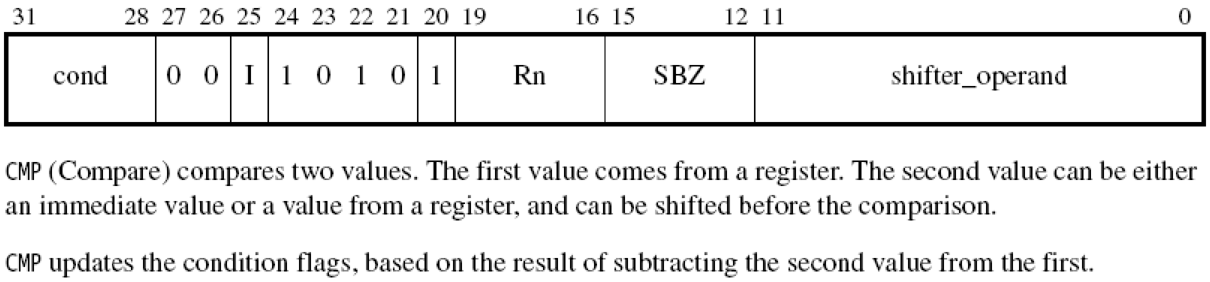
- The comparison instructions update the cpsr flags according to the result, but do **not** affect other registers

- After the bits have been set, the information can be used to change program flow by using conditional execution

Syntax: <instruction>{cond}{S} Rn, N

`CMP`	Compare	*Flags set as a result of Rn – N*
`CMN`	compare negated	*Flags set as a result of Rn + N*
`TEQ`	test for equality of two 32-bit values	*Flags set as a result of Rn ^ N*
`TST`	test bits of a 32-bit value	*Flags set as a result of Rn & N*

Comparison Instructions – CMP



31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 0	1	Rn	SBZ	shifter_operand

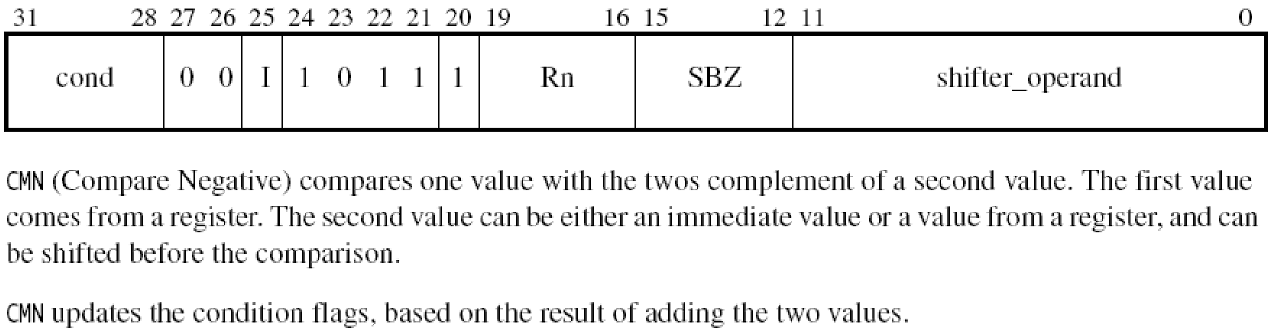
CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMP updates the condition flags, based on the result of subtracting the second value from the first.

- CMP compares two values by subtracting the second operand from the first operand
  - Note that there is no destination register
  - It only update N, Z, C, V flags in CPSR based on the execution result

```
CMP R0, R1;
```

Comparison Instructions – CMN



31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 1	1	Rn	SBZ	shifter_operand

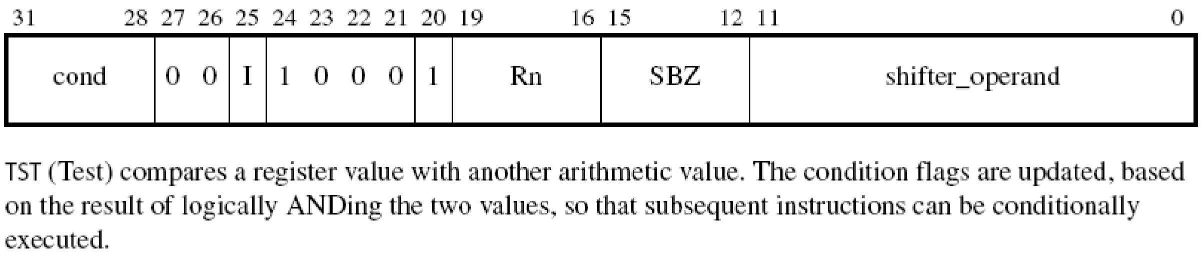
CMN (Compare Negative) compares one value with the twos complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMN updates the condition flags, based on the result of adding the two values.

- CMN compares one value with the 2's complement of a second value
  - It performs a comparison by adding the 2nd operand to the first operand
    - It is equivalent to subtracting the negative of the 2nd operand from the 1st operand
  - Note that there is no destination register
  - It only update N, Z, C, V flags in CPSR based on the execution result

```
CMN R0, R1;
```

Comparison Instructions – TST



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	0	0	0	1	Rn	SBZ	shifter_operand

TST (Test) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed.

- `TST` tests bits of two 32-bit values by logically ANDing the two operands
  - Note that there is no destination register
  - It only updates CPSR flags (N,Z,C) based on the execution result

- `TEQ` sets flags by EORing the two operands

Examples

Before: cpsr = nzcv
r0 = 4
r9 = 4

CMP r0, r9

After: cpsr = nZCv
r0 = 4
r9 = 4

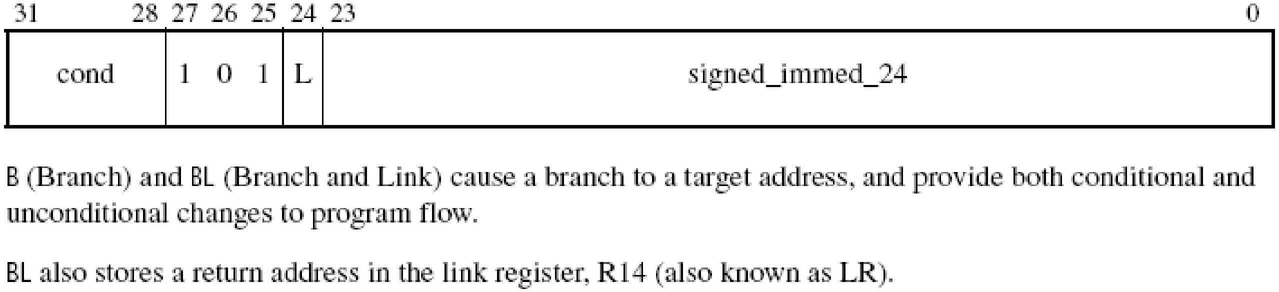
Branch Instructions

- A branch instruction changes the flow of execution or is used to call a function
  - This type of instructions allows programs to have subroutines, *if-then-else* structures, and loops

Syntax: B{cond} label
BL{cond} label

`B`	branch	***pc = label***
`BL`	branch with link	***pc = label*** ***lr = address of the next instruction after the BL***

B, BL



31–28	27	26	25	24	23–0
cond	1	0	1	L	signed_immed_24

B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

BL also stores a return address in the link register, R14 (also known as LR).

- `B` (branch) and `BL` (branch with link) are used for conditional or unconditional branch
  - `BL` is used for the subroutine (procedure, function) call
  - To return from a subroutine, use
    - `MOV PC, R14; (R14: link register) Used to return to caller`

- Branch target address
  - Sign-extend the 24-bit signed immediate (2's complement) to 30-bits
  - Left-shift the result by 2 bits
  - Add it to the current PC (actually, PC+8)
  - Thus, the branch target could be ±32MB away from the current instruction

- See backup slide for 'PC + 8'

Examples

```
          …
          B  forward
          ADD   r1,  r2, #4
          ADD   r0,  r6, #2
          ADD   r3,  r7, #4
forward:
          SUB   r1,  r2, #4
```

```
backward:
          ADD   r1,  r2, #4
          SUB   r1,  r2, #4
          ADD   r4,  r6, r7
          B  backward
          …
```

```
          BL  foo
          CMP  r1,  #5
          MOVEQ  r1, #0
          …..
foo:
           < subroutine code >
          MOV  pc, lr // return from subroutine
```

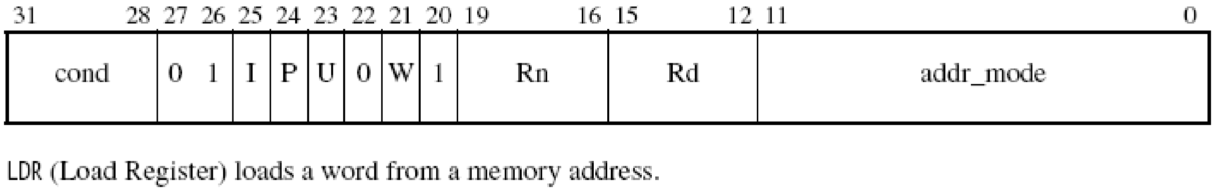
Memory Access Instructions

- Load-Store (memory access) instructions transfer data between memory and CPU registers
  - Single-register transfer
  - Multiple-register transfer
  - Swap instruction

Single-Register Transfer

LDR	Load (Read) a word (32-bit) from memory into a register	***Rd ← mem32[address]***
STR	Store (Write) a word from a register to memory	***Rd → mem32[address]***
LDRB	Load a **zero-extended byte** from memory into a register	***Rd ← Zero-extend (mem8[address])***
STRB	Store a byte from a register to memory	***Rd → mem8[address]***
LDRH	Load a **zero-extended half-word** from memory into a register	***Rd ← Zero-extend (mem16[address])***
STRH	Store a half-word from a register into memory	***Rd → mem16[address]***
LDRSB	Load a **sign-extended byte** from memory into a register	***Rd ← Sign-Extend (mem8[address])***
LDRSH	Load a **sign-extended half-word** from memory into a register	***Rd ← Sign-Extend (mem16[address])***

LDR (Load Register)



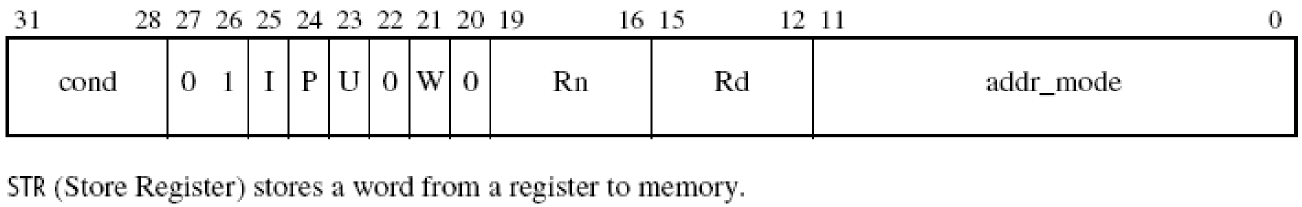
31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	1	Rn	Rd	addr_mode

LDR (Load Register) loads a word from a memory address.

- `LDR` loads a word from a memory location to a register
  - The memory location is specified in a very flexible manner with addressing mode

```
// Assume R1 = 0x0000_2000
LDR R0, [R1]  // R0 ← [R1]
LDR R0, [R1, #16] // R0 ← [R1+16]; 0x0000_2010
```

STR (Store Register)



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	0	Rn	Rd	addr_mode

STR (Store Register) stores a word from a register to memory.

- `STR` stores a word from a register to a memory location
  - The memory location is specified in a very flexible manner with a addressing mode

```
// Assume R1 = 0x0000_2000
STR R0, [R1]  // [R1] <- R0
STR R0, [R1, #16] // [R1+16] <- R0
```

Load-Store Addressing Mode

Addressing Mode	Syntax	Data access location	*Rn* after memory access	*Examples*
Offset addressing	[Rn + offset]	Rn + offset	No change	***LDR r0, [r1, #4]***
Pre-indexed addressing	[Rn + offset]!	Rn + offset	Rn = Rn + offset	***LDR r0, [r1, #4]!***
Post-indexed addressing	[Rn], offset	Rn	Rn = Rn + offset	***LDR r0, [r1], #4***

! indicates that the instruction writes the calculated address back to the base address register

Before:

r0 = 0x0000_0000
r1 = 0x0009_0000
Mem[0x0009_0000] = 0x01010101
Mem[0x0009_0004] = 0x02020202

LDR r0, [r1, #4]

After: r0 ← mem[0x0009_0004]
r0 = 0x0202_0202
r1 = 0x0009_0000

LDR r0, [r1, #4]!

After: r0 ← mem[0x0009_0004]
r0 = 0x0202_0202
r1 = 0x0009_0004

LDR r0, [r1], #4

After: r0 ← mem[0x0009_0000]
r0 = 0x0101_0101
r1 = 0x0009_0004

Multiple Register Transfer – LDM, STM

Syntax: <LDM/STM>{cond}<addressing mode> Rn{!}, <registers>^

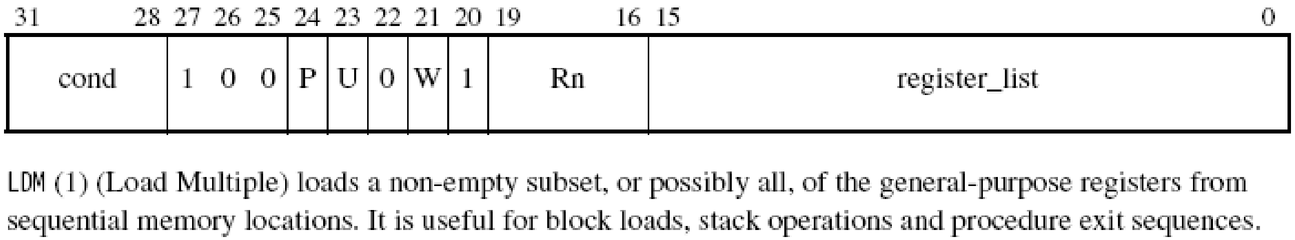
`LDM`	Load multiple registers
`STM`	Store multiple registers

default addressing mode (IA)

Addressing Mode	Description	Start address	End address	Rn!
IA	Increment After	Rn	Rn + 4 x N - 4	Rn + 4 x N
IB	Increment Before	Rn + 4	Rn + 4 x N	Rn + 4 x N
DA	Decrement after	Rn – 4 x N + 4	Rn	Rn – 4 x N
DB	Decrement Before	Rn – 4 x N	Rn – 4	Rn – 4 x N

N: number of words (registers) you want to transfer

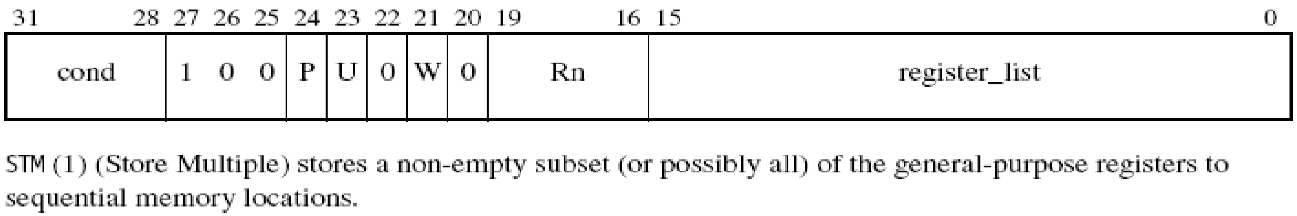
Multiple Register Transfer – LDM, STM



31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	0	W	1	Rn	register_list

LDM (1) (Load Multiple) loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.

- LDM (Load Multiple) loads general-purpose registers from sequential memory locations



31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	0	W	0	Rn	register_list

STM (1) (Store Multiple) stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations.

- STM (Store Multiple) stores general-purpose registers to sequential memory locations

LDM, STM - Multiple Data Transfer

- In multiple data transfer, the register list is given in a curly brackets {}

- It doesn't matter which order you specify the registers in
  - Lowest register (lowest numbered reg) – Lowest memory location
  - …
  - Highest register (highest numbered reg) – Highest memory location

```
STMDB  R13!  {R0, R1}   // R13 is updated
LDMIA  R13!  {R1, R0}   // R13 is updated
```

- A useful shorthand is "-"
  - It specifies the beginning and end of registers

```
STMDB  R13!, {R0-R12} // R13 is updated appropriately
LDMIA  R13!, {R0-R12} // R13 is updated appropriately
```

Examples

LDMIA r0!, {r1-r3}

Before:

Mem32[0x80018] = 0x3

Mem32[0x80014] = 0x2

Mem32[0x80010] = 0x1

r0 = 0x0008_0010

r1 = 0x0000_0000

r2 = 0x0000_0000

r3 = 0x0000_0000

After:

Mem32[0x80018] = 0x3

Mem32[0x80014] = 0x2

Mem32[0x80010] = 0x1

r0 = 0x0008_001C

r1 = 0x0000_0001

r2 = 0x0000_0002

r3 = 0x0000_0003

Stack Operation

- Multiple data transfer instructions (LDM and STM) are used to load and store multiple words of data from/to main memory

Stack	Other	Description
STMFA	STMIB	Pre-incremental store
STMEA	STMIA	Post-incremental store
STMFD	STMDB	Pre-decremental store
STMED	STMDA	Post-decremental store
LDMED	LDMIB	Pre-incremental load
LDMFD	LDMIA	Post-incremental load
LDMEA	LDMDB	Pre-decremental load
LDMFA	LDMDA	Post-decremental load

ARM default stack

- IA: Increment After
- IB: Increment Before
- DA: Decrement After
- DB: Decrement Before
- FA: Full Ascending (in stack)
- FD: Full Descending (in stack)
- EA: Empty Ascending (in stack)
- ED: Empty Descending (in stack)

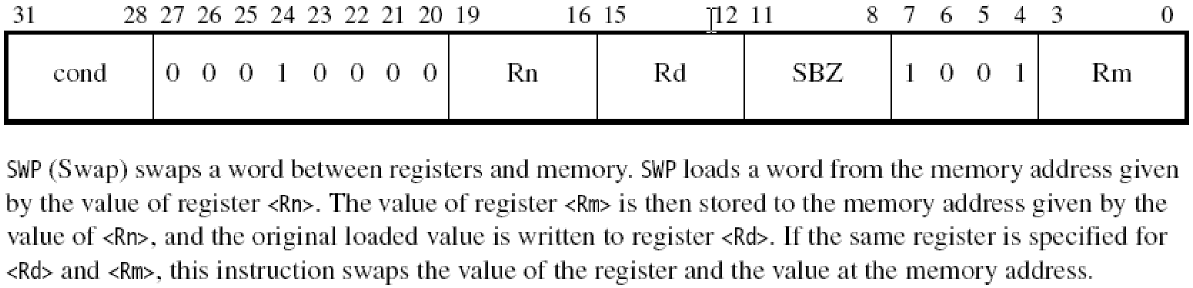
SWAP Instruction

Syntax: SWP{B}{cond} Rd, Rm, <Rn>

`SWP`	Swap a word between memory and a register	**tmp = mem32[Rn]** **mem32[Rn] = Rm** **Rd = tmp**
`SWPB`	Swap a byte between memory and a register	**tmp = mem8[Rn]** **mem8[Rn] = Rm** **Rd = tmp**

- **Deprecated in ARMv6, ARMv7**
- **Instead, use `ldrex/strex`**

SWAP Instruction (Deprecated in Armv7)



31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 0 0	Rn	Rd	SBZ	1 0 0 1	Rm

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.

- SWP swaps the contents of memory with the contents of a register
  - It is a special case of a load-store instruction
  - It performs a swap atomically meaning that it does not release the bus unitil it is done with the read and the write
  - It is useful to implement semaphores and mutual exclusion (mutex) in an OS

SWP r0, r1, [r2]

Before:

mem32[0x9000] = 0x1234_5678
r0 = 0x0000_0000
r1 = 0x1111_2222
r2 = 0x0000_9000

After:

mem32[0x9000] = 0x1111_2222
r0 = 0x1234_5678
r1 = 0x1111_2222
r2 = 0x0000_9000

Semaphore Example

```
Spin:
            MOV  r1, =semaphore; // r1 has an address for semaphore
            MOV  r2, #1
            SWP  r3, r2, [r1]
            CMP  r3, #1
            BEQ   spin
```

LDREX, STREX Instructions

- `ldrex` (load register exclusive) performs a load and flags the physical address for exclusive access
  - If the address has the shared memory attribute, the physical address is marked as exclusive access for the executing processor in a global monitor
  - It causes the executing processor to indicate an active exclusive access in the local monitor

Syntax: LDREX <Rt>, [Rn]

- `strex` (store register exclusive) performs a conditional store, which succeeds only if the address has previously been flagged for exclusive access
  - Write Rt to [Rn]
  - Rd will be set to 0 on success of the store operation. Otherwise, Rd will be set to 1

Syntax: STREX <Rd>, <Rt>, [Rn]

* Read pp344 and pp 392 for LL-SC operation in David Culler's Book "Parallel Computer Architecture: A Hardware / Software Approach"
(LL-SC: Load-Locked (or Load-Linked) – Store Conditional)

Example

```
lock:
        LDREX   r1, [r0];   // check if locked
        CMP     r1, #LOCKED
        BEQ     lock

         MOV     r1, #LOCKED
         STREX   r2, r1, [r0]  // attempt to lock
         CMP     r2, #0        // check if strex is successful
         BNE     lock
         DMB
```

```
unlock:
         DMB
         MOV     r1, #UNLOCKED
         STR     r1, [r0]      // write "unlocked" to lock
```

CLREX Instructions

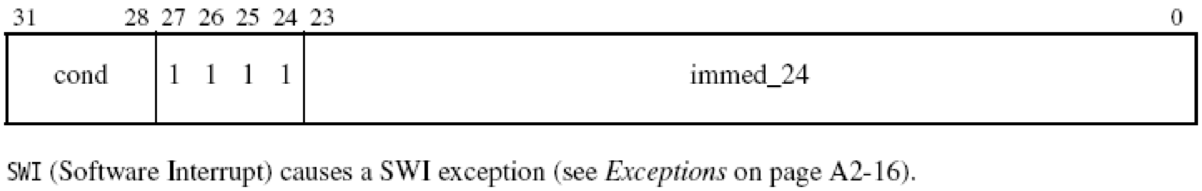
- `clrex` (clear exclusive) clears exclusive flags associated with a processor

Syntax: CLREX

Miscellaneous but Important Instructions

- Software interrupt instruction
- Program status register instructions

SVC
(previously SWI (Software Interrupt))



31 … 28	27 26 25 24	23 … 0
cond	1 1 1 1	immed_24

SWI (Software Interrupt) causes a SWI exception (see *Exceptions* on page A2-16).

- The SVC (Supervisor Call) instruction incurs a software interrupt
  - It is used by operating systems for system calls
  - 24-bit immediate value is ignored by the ARM processor, but can be used by the SVC exception handler in an operating system to determine what operating system service is being requested

Syntax: SVC{cond} SVC_number

SWI	Software interrupt	**• lr_svc (r14) = address of instruction following SWI** **• spsr_svc = cpsr** **•cpsr mode = SVC** **• cpsr 'I bit = 1 (it masks interrupts)** **• pc = 0x8**

- To return from the software interrupt, use
  - `MOVS PC, R14; PC <- R14 (lr), CPSR <- SPSR`

Example

0x0000_8000 SVC 0x123456

Before:

cpsr = nzcVqift_USER
pc = 0x0000_8000
lr = 0x003F_FFF0
r0 = 0x12

After:

cpsr = nzcVq**I**ft_**SVC**
spsr_svc = nzcVqift_USER
pc = 0x0000_0008
lr = 0x0000_8004
r0 = 0x12

SVC handler example

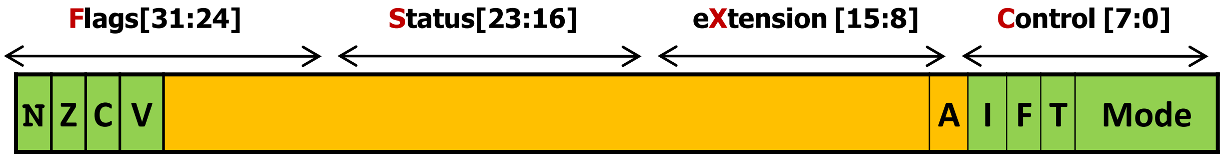
```
SVC_handler:
            STMFD   sp!, {r0-r12, lr}          // push registers to stack
            LDR     r10, [lr, #-4]             // r10 =  swi instruction
            BIC     r10, r10, #0xff000000      // r10 gets swi number
            BL      interrupt_service_routine
            LDMFD   sp!, {r0-r12, pc}^         // return from SWI hander
```

Program status register instructions

Syntax: MRS{cond} Rd, <cpsr | spsr>
MSR{cond} <cpsr | spsr>_<fields>, Rm
MSR{cond} <cpsr | spsr>_<fields>, #immediate

MRS	Copy program status register to a general-purpose register	**Rd = psr**
MSR	Copy a general-purpose register to a program status register	**psr[field] = Rm**
MSR	Copy an immediate value to a program status register	**psr[field] = immediate**

*** fields can be any combination of flags (f), status (s), extension (x), and control (c)**



Flags[31:24]

Status[23:16]

eXtension [15:8]

Control [7:0]

N Z C V A I F T Mode

MSR & MRS

- MSR: Move the value of a general-purpose register or an immediate constant to the CPSR or SPSR of the current mode

```
MSR CPSR_all, R0 ; Copy R0 into CPSR
MSR SPSR_all, R0 ; Copy R0 into SPSR
```

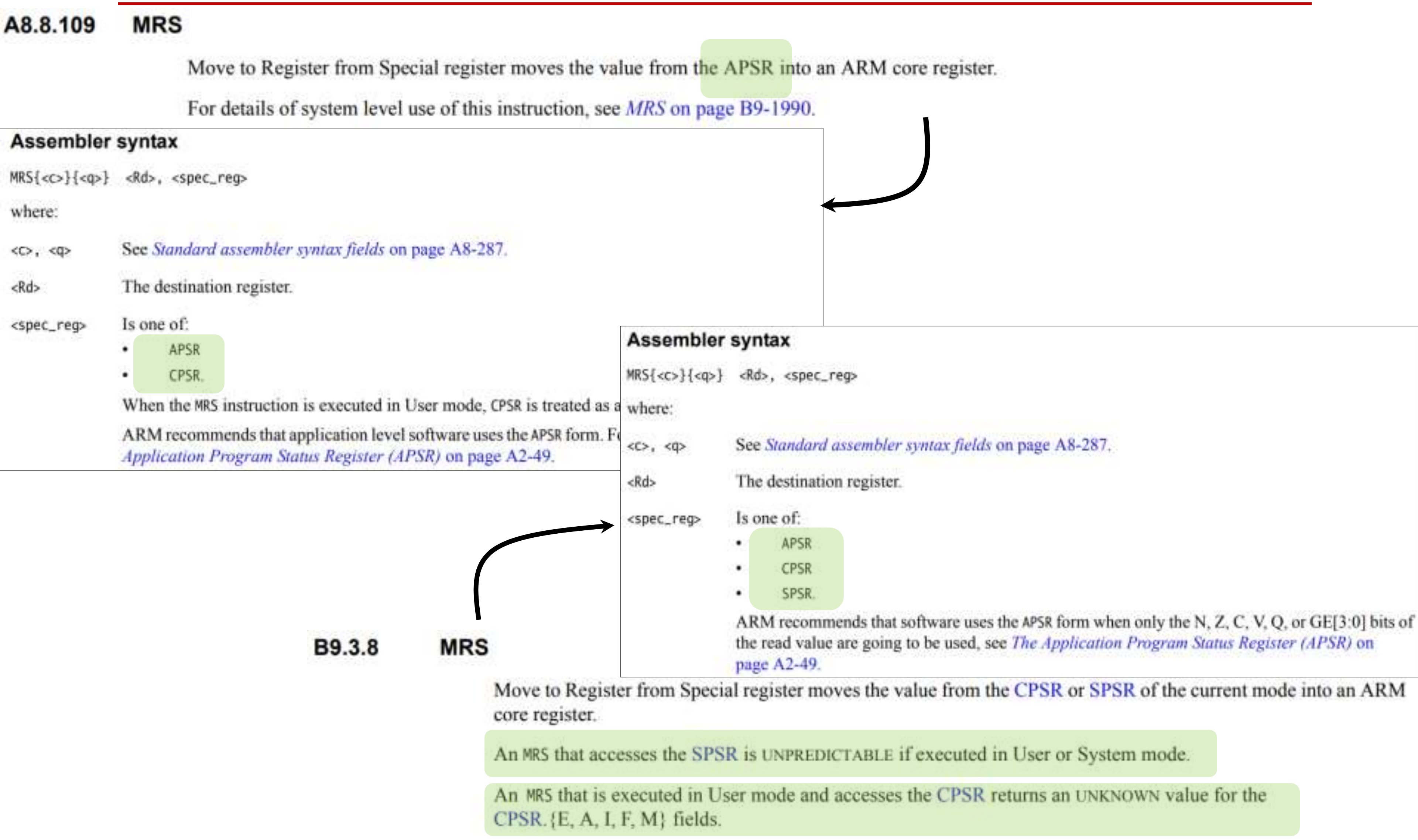
- MRS: Move the value of the CPSR or the SPSR of the current mode into a general-purpose register

```
MRS R0, CPSR_all ; Copy CPSR into R0
MRS R0, SPSR_all ; Copy SPSR into R0
```

- To change the operating mode, use the following code

```
// Change to the supervisor mode
MRS R0,CPSR ; Read CPSR
BIC R0,R0,#0x1F ; Remove current mode with bit clear instruction
ORR R0,R0,#0x13 ; Substitute to the Supervisor mode
MSR CPSR_c,R0 ; Write the result back to CPSR
```

Application-level vs System-level Access



A8.8.109 MRS

Move to Register from Special register moves the value from the APSR into an ARM core register.

For details of system level use of this instruction, see *MRS* on page B9-1990.

Assembler syntax

```
MRS{<c>}{<q>}  <Rd>, <spec_reg>
```

where:

<c>, <q> See *Standard assembler syntax fields* on page A8-287.

<Rd> The destination register.

<spec_reg> Is one of:

- APSR
- CPSR.

When the MRS instruction is executed in User mode, CPSR is treated as a

ARM recommends that application level software uses the APSR form. F
Application Program Status Register (APSR) on page A2-49.

Assembler syntax

```
MRS{<c>}{<q>}  <Rd>, <spec_reg>
```

where:

<c>, <q> See *Standard assembler syntax fields* on page A8-287.

<Rd> The destination register.

<spec_reg> Is one of:

- APSR
- CPSR
- SPSR.

ARM recommends that software uses the APSR form when only the N, Z, C, V, Q, or GE[3:0] bits of the read value are going to be used, see *The Application Program Status Register (APSR)* on page A2-49.

B9.3.8 MRS

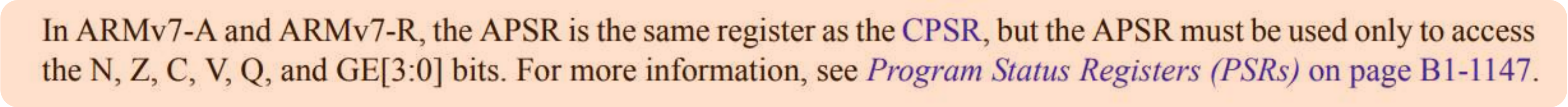
Move to Register from Special register moves the value from the CPSR or SPSR of the current mode into an ARM core register.

An MRS that accesses the SPSR is UNPREDICTABLE if executed in User or System mode.

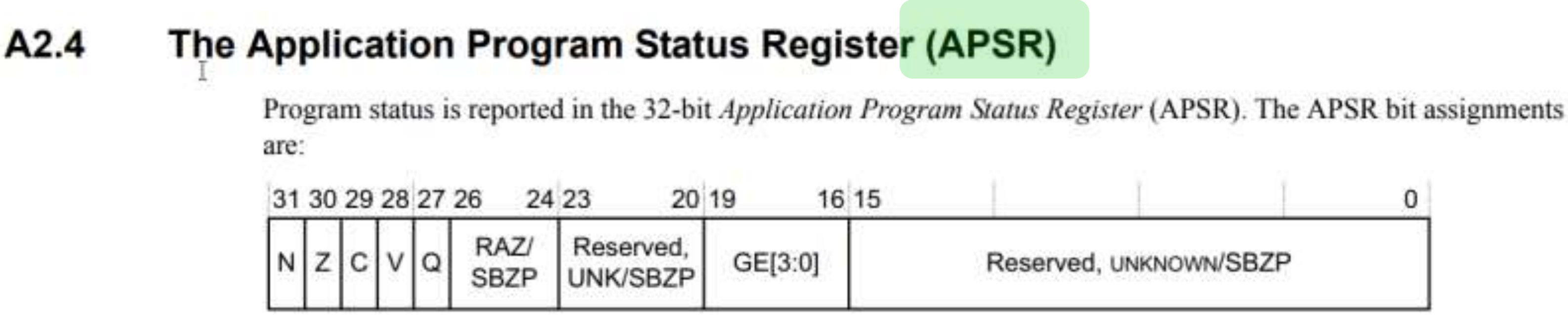
An MRS that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR.{E, A, I, F, M} fields.

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

APSR (Application Program Status Register)



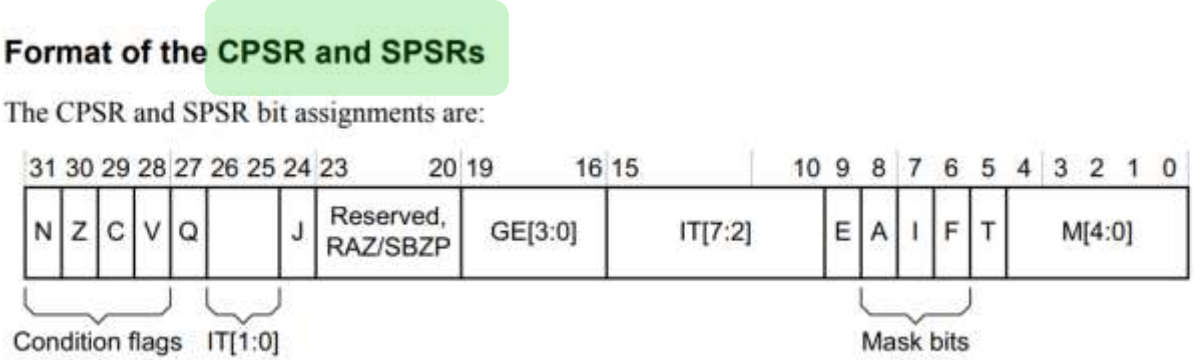
In ARMv7-A and ARMv7-R, the APSR is the same register as the CPSR, but the APSR must be used only to access the N, Z, C, V, Q, and GE[3:0] bits. For more information, see *Program Status Registers (PSRs)* on page B1-1147.



A2.4 The Application Program Status Register (APSR)

Program status is reported in the 32-bit *Application Program Status Register* (APSR). The APSR bit assignments are:

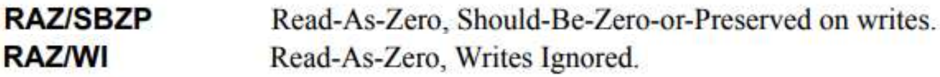
31	30	29	28	27	26 24	23 20	19 16	15 0
N	Z	C	V	Q	RAZ/ SBZP	Reserved, UNK/SBZP	GE[3:0]	Reserved, UNKNOWN/SBZP



Format of the CPSR and SPSRs

The CPSR and SPSR bit assignments are:

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 3 2 1 0
N	Z	C	V	Q		J	Reserved, RAZ/SBZP	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]



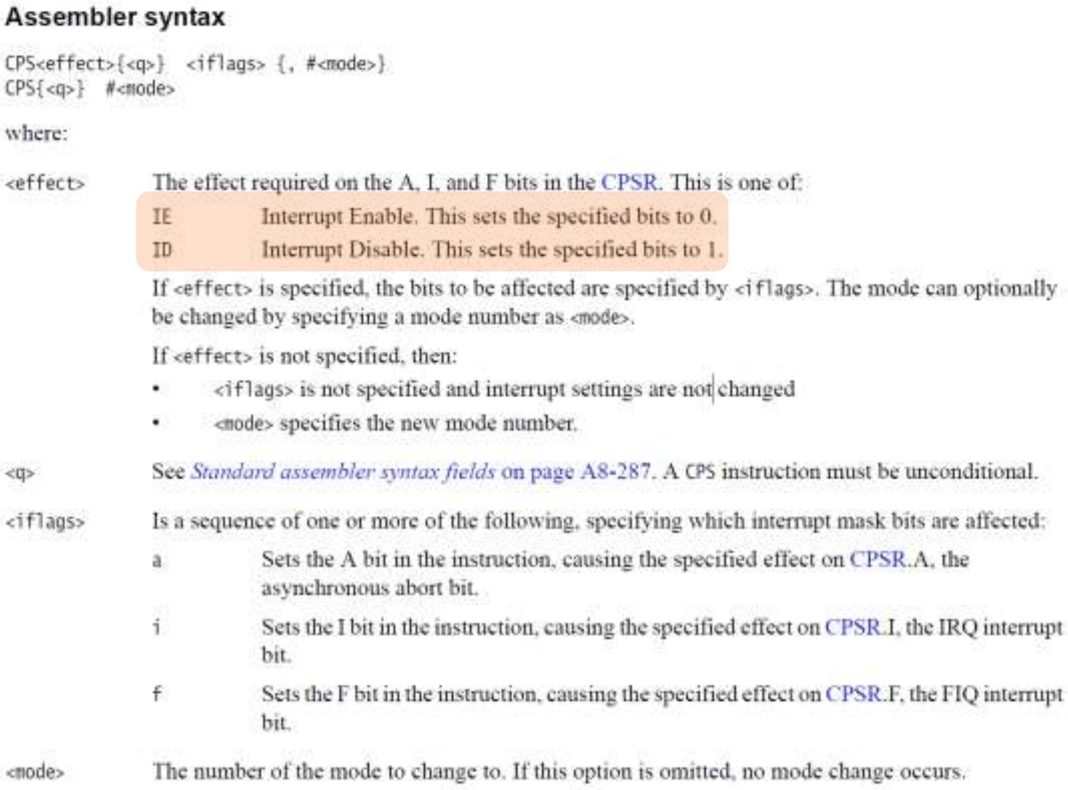
RAZ/SBZP Read-As-Zero, Should-Be-Zero-or-Preserved on writes.
RAZ/WI Read-As-Zero, Writes Ignored.

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

CPS (Change Processor State)

- CPS changes one or more of the CPSR.{A,I,F} interrupt mask bits and CPSR.M mode field without changing the other CPSR bits
  - CPS is treated as NOP if executed in User mode

```
CPS   #0x13 ; change to SVC mode
CPSIE aif, #0x10
; IE/ID to enable or disable specified
;       exceptions - one or more of AIF
; enable imprecise aborts, IRQ, and FIQ
```



Assembler syntax

```
CPS<effect>{<q>}  <iflags> {, #<mode>}
CPS{<q>}  #<mode>
```

where:

`<effect>` The effect required on the A, I, and F bits in the CPSR. This is one of:

IE Interrupt Enable. This sets the specified bits to 0.

ID Interrupt Disable. This sets the specified bits to 1.

If `<effect>` is specified, the bits to be affected are specified by `<iflags>`. The mode can optionally be changed by specifying a mode number as `<mode>`.

If `<effect>` is not specified, then:

- `<iflags>` is not specified and interrupt settings are not changed
- `<mode>` specifies the new mode number.

`<q>` See *Standard assembler syntax fields* on page A8-287. A CPS instruction must be unconditional.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

a Sets the A bit in the instruction, causing the specified effect on CPSR.A, the asynchronous abort bit.

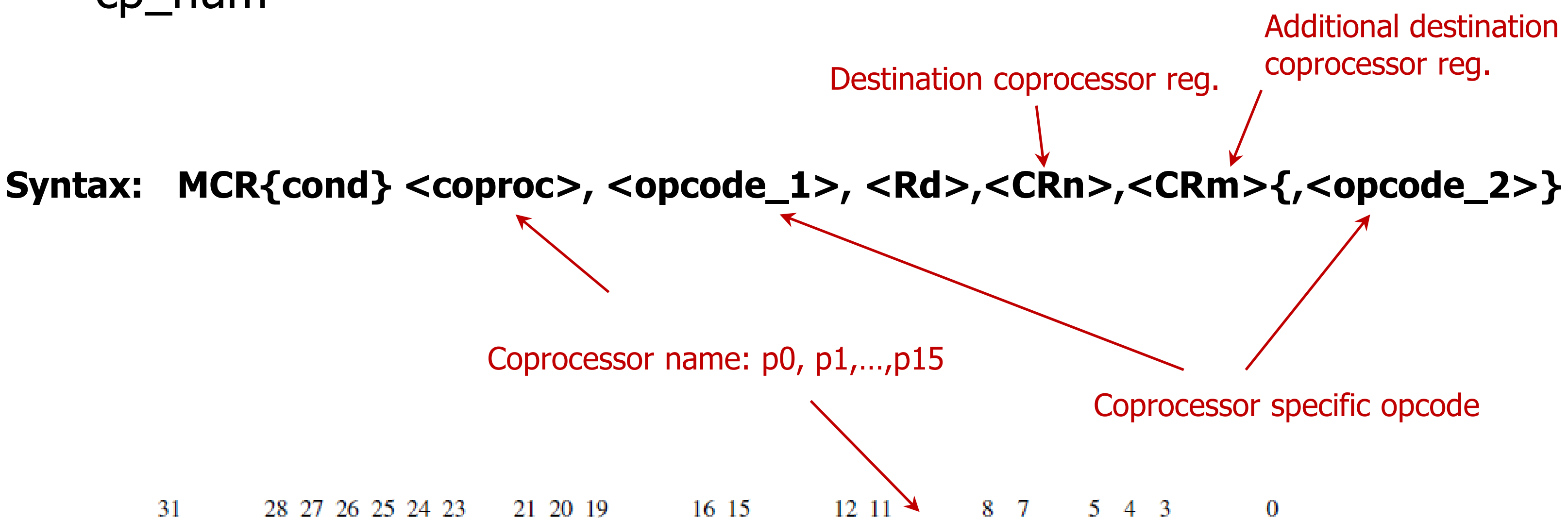
i Sets the I bit in the instruction, causing the specified effect on CPSR.I, the IRQ interrupt bit.

f Sets the F bit in the instruction, causing the specified effect on CPSR.F, the FIQ interrupt bit.

`<mode>` The number of the mode to change to. If this option is omitted, no mode change occurs.

MCR

- Move to Coprocessor from ARM Register
  - Pass the value of register Rd to the coprocessor whose number is cp_num



Additional destination coprocessor reg.

Destination coprocessor reg.

Syntax: MCR{cond} <coproc>, <opcode_1>, <Rd>,<CRn>,<CRm>{,<opcode_2>}

Coprocessor name: p0, p1,…,p15

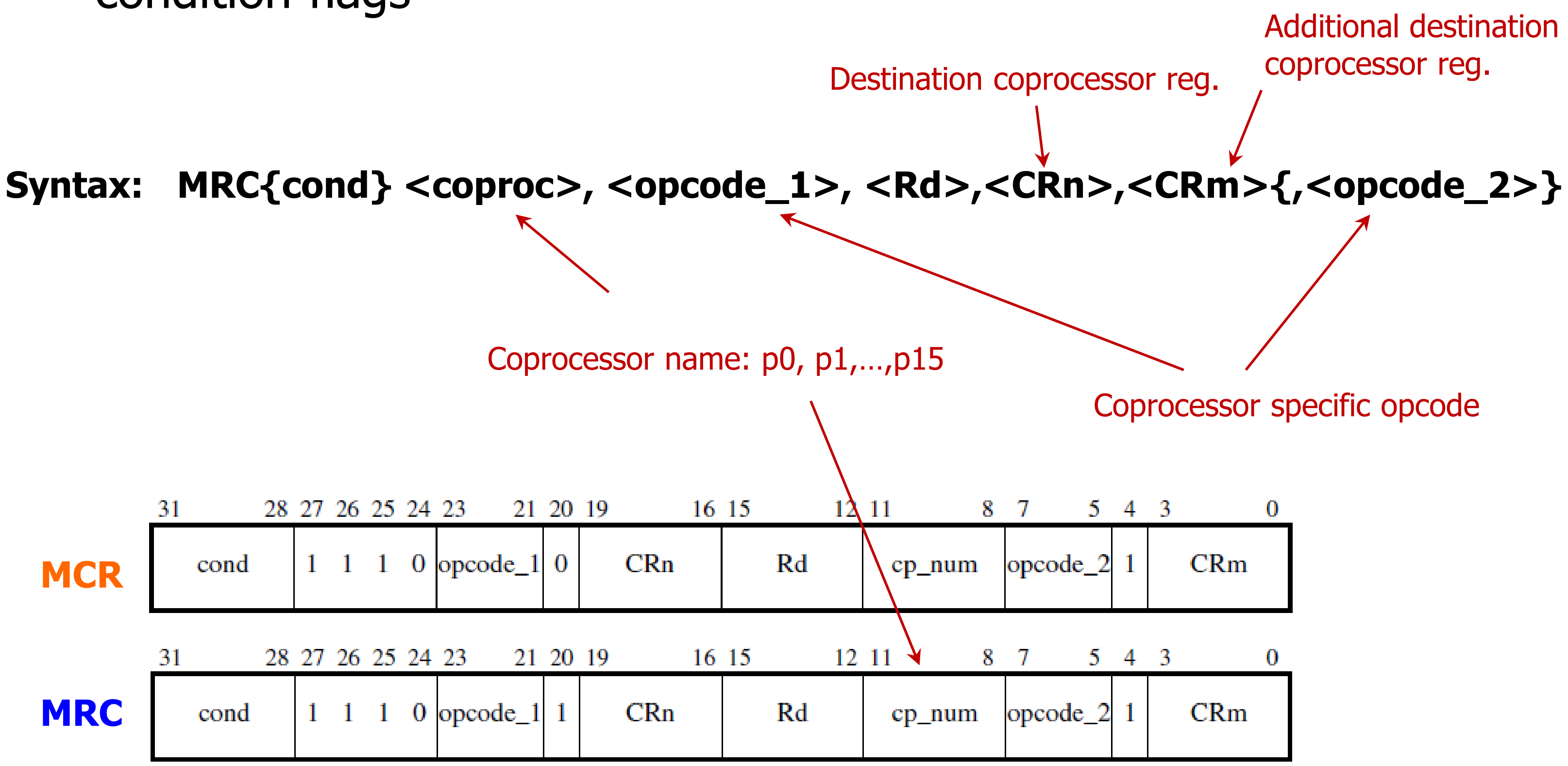
Coprocessor specific opcode

	31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
MCR	cond	1	1	1	0	opcode_1	0	CRn	Rd	cp_num	opcode_2	1	CRm

	31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
MRC	cond	1	1	1	0	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

MRC

- Move to ARM Register from Coprocessor
  - Cause a coprocessor to transfer a value to an ARM register or to the condition flags



Syntax: MRC{cond} <coproc>, <opcode_1>, <Rd>,<CRn>,<CRm>{,<opcode_2>}

Additional destination coprocessor reg.

Destination coprocessor reg.

Coprocessor name: p0, p1,…,p15

Coprocessor specific opcode

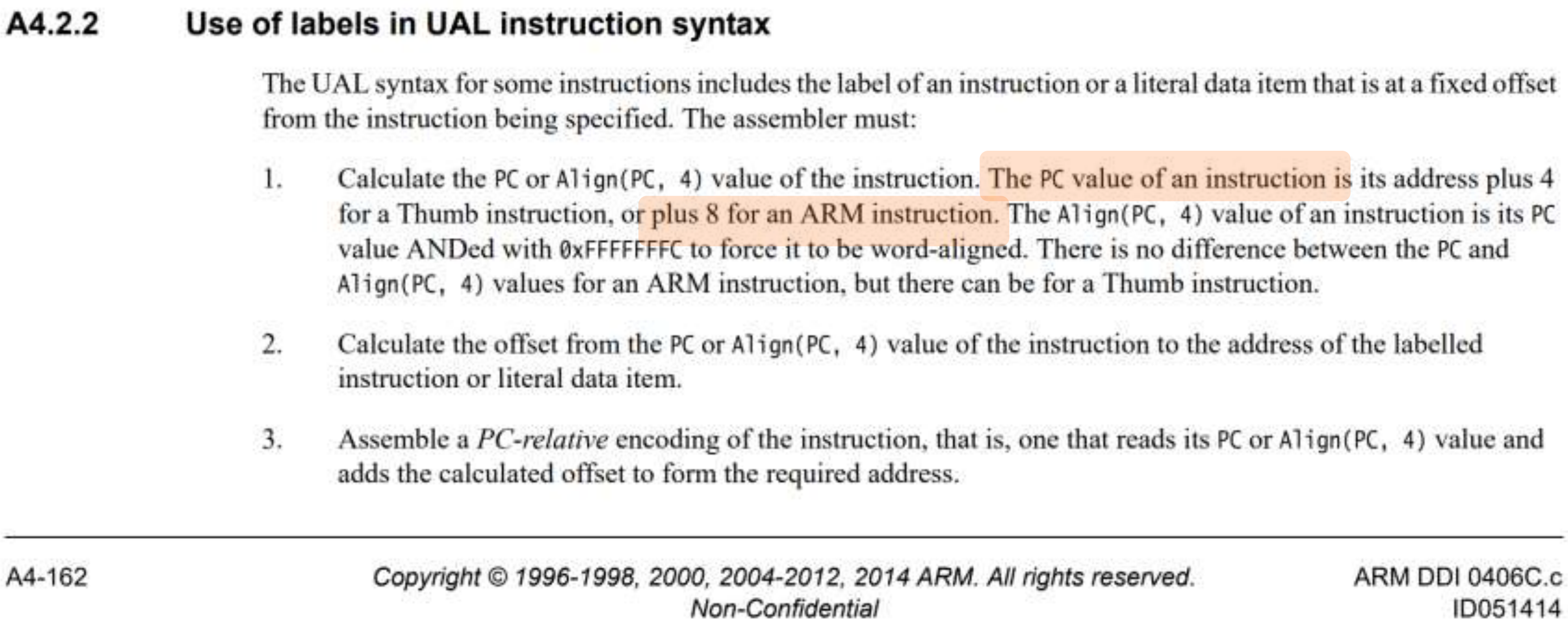
	31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
MCR	cond	1	1	1	0	opcode_1	0	CRn	Rd	cp_num	opcode_2	1	CRm
MRC	cond	1	1	1	0	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

(Assembly) Language

- There is no golden way to learn language
- You got to use and practice to get used to it

Backup Slides

PC + 8



A4.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC, 4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a Thumb instruction, or plus 8 for an ARM instruction. The `Align(PC, 4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC, 4)` values for an ARM instruction, but there can be for a Thumb instruction.

2. Calculate the offset from the `PC` or `Align(PC, 4)` value of the instruction to the address of the labelled instruction or literal data item.

3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC, 4)` value and adds the calculated offset to form the required address.

A4-162

Non-Confidential

ARM DDI 0406C.c
ID051414

N, Z, C, V in CPSR

- Flag update (NZCV) varies depending on data processing instructions
  - It is a nasty small detail in Armv7 instructions
  - But, can do programming intuitively most of the cases as shown below

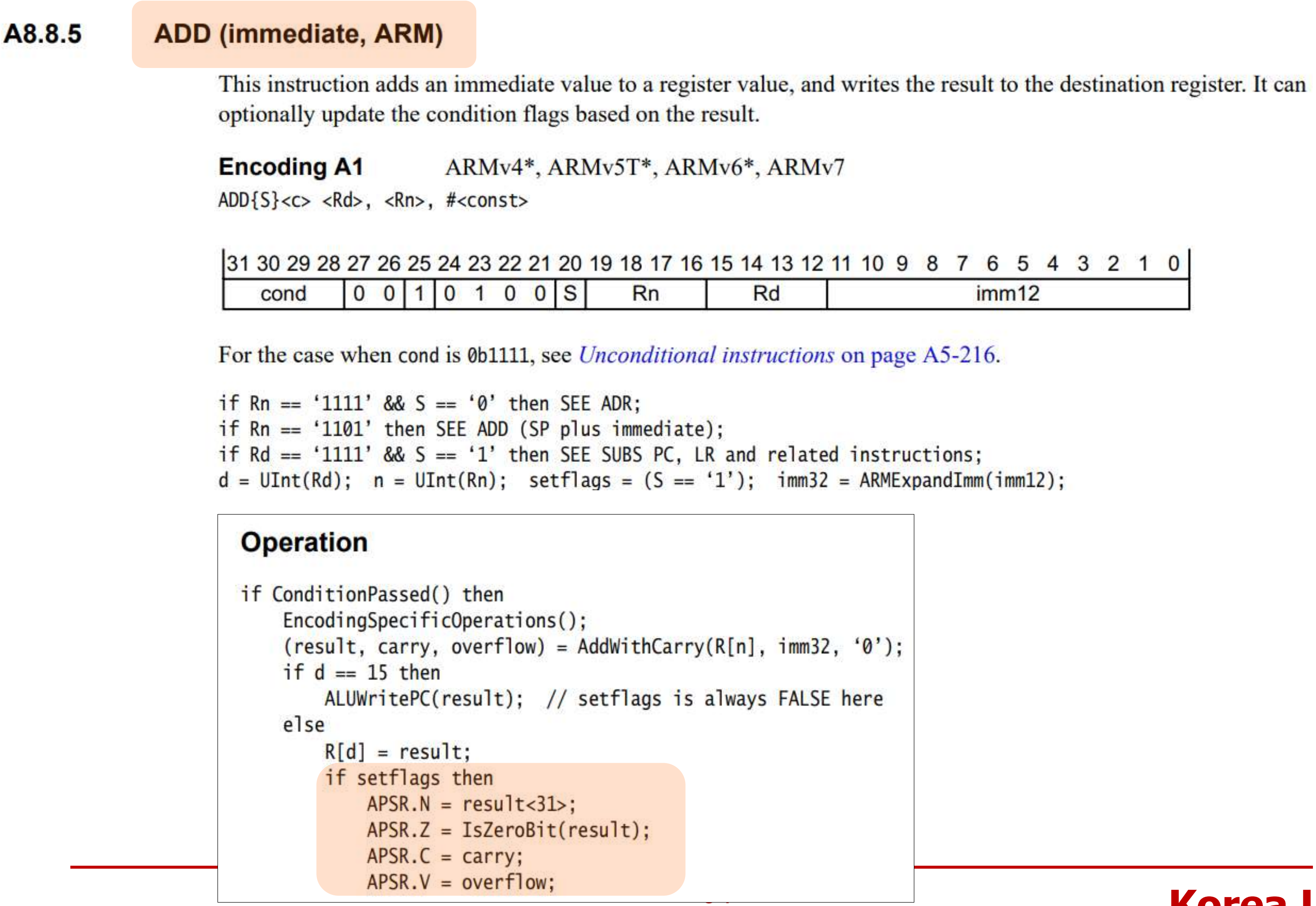
```
int a, b, c;

if (a == b)    c++;
else           c--;
```

```
cmp   r2, r3 // r2(=a), r3(=b)
addeq r4, r4, #1   // r4 (=c)
subne r4, r4, #1   // r4 (=c)
```

- Most arithmetic instructions update N,Z,C,V
  - Exception is MUL (multiplication)

- Logical instructions update N,Z,C (V unchanged)
  - C flag is updated from the barrel shifter output
  - Why do you want to set C-flag after the logical instruction though? Probably C-flag check is rarely done, meaning you don't have to worry too much about it. But if you have to do it, check out the C-flag update rules in Armv7 TRM.
    - For example, `BIT` (bit clear) instructions. There are 2 versions: `BIC (immediate), BIC (register-shifted)`. Why do you want to update and check C-flag after `BIC` instruction?

Flag-update examples According to Instructions



A8.8.5 ADD (immediate, ARM)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>, <Rn>, #<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 0 0	S	Rn	Rd	imm12

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

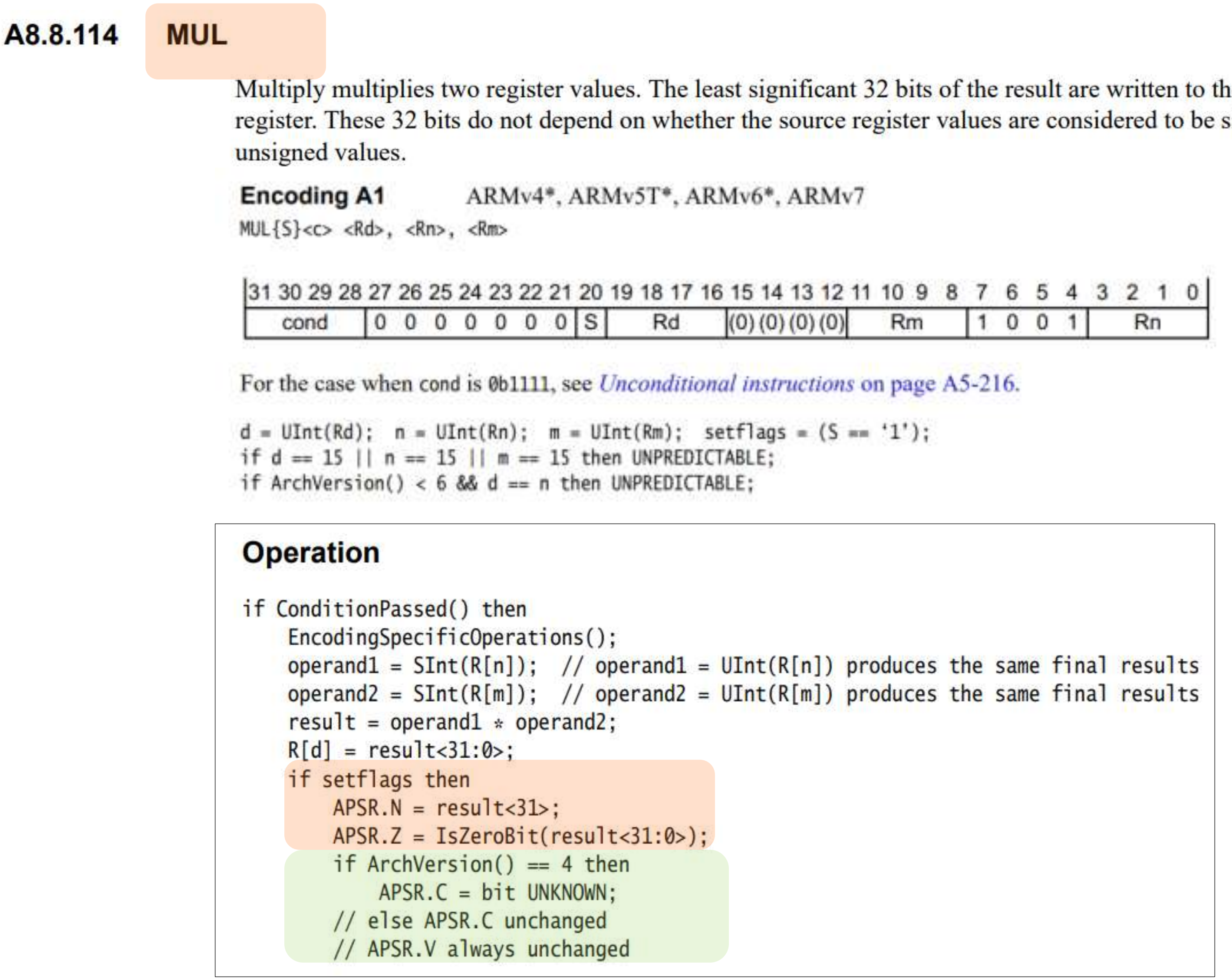
```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

Flag-update examples According to Instructions



A8.8.114 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

`MUL{S}<c> <Rd>, <Rn>, <Rm>`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	0	0	0	S	Rd				(0)	(0)	(0)	(0)	Rm				1	0	0	1	Rn			

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

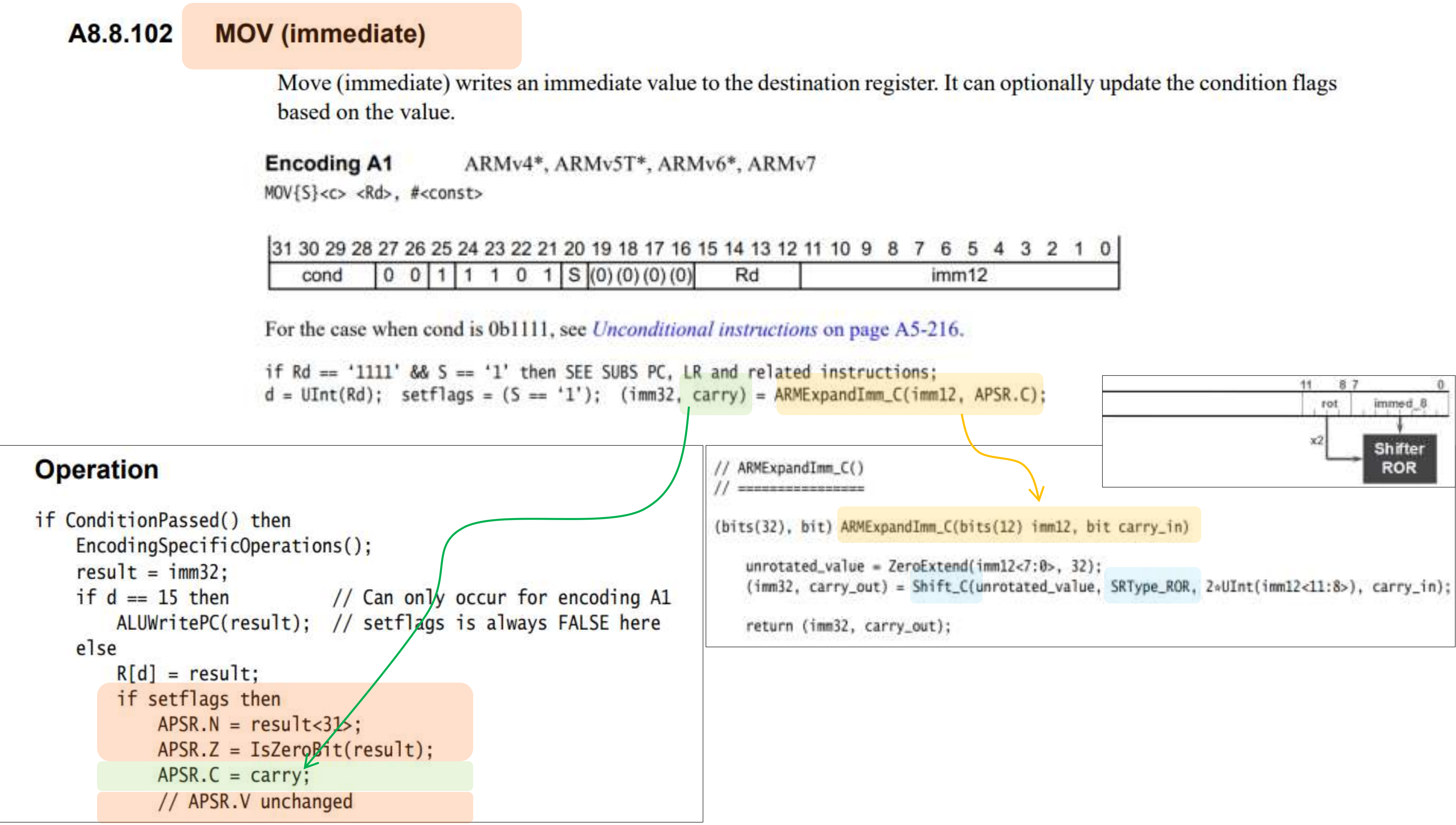
```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result<31:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
        // else APSR.C unchanged
        // APSR.V always unchanged
```

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

Flag-update examples According to Instructions



A8.8.102 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MOV{S}<c> <Rd>, #<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				imm12											

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

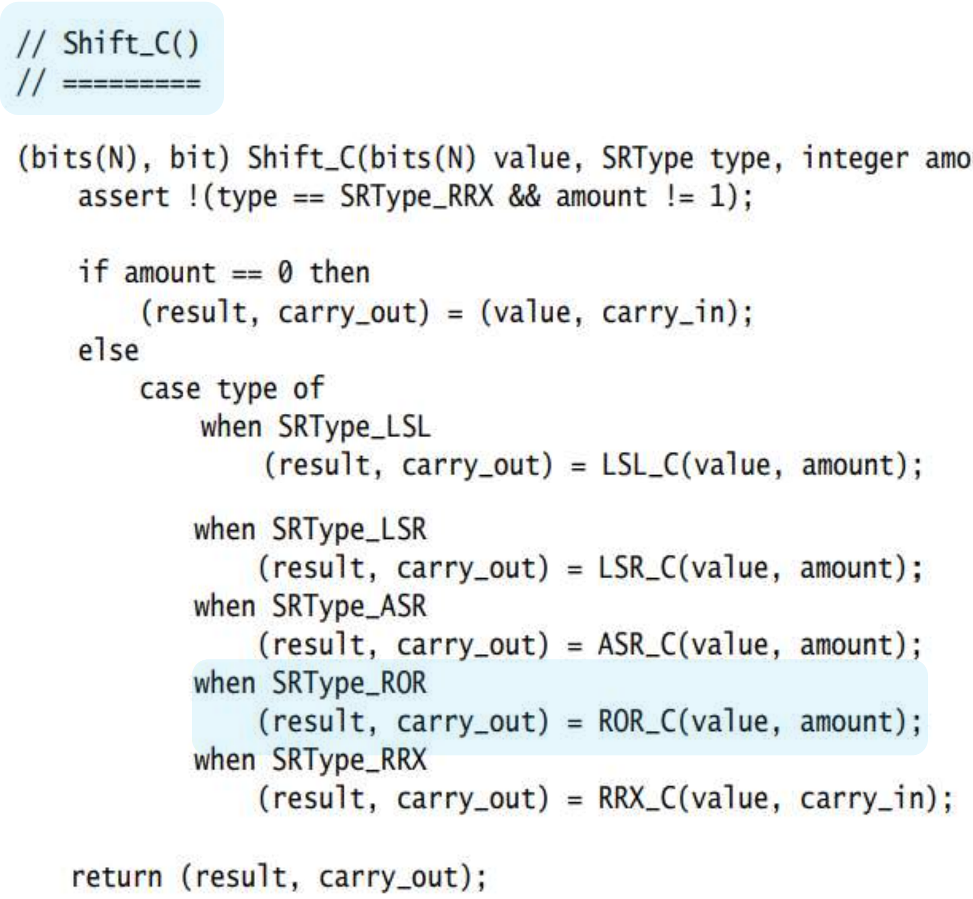
```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

```
// ARMExpandImm_C()
// ================

(bits(32), bit) ARMExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

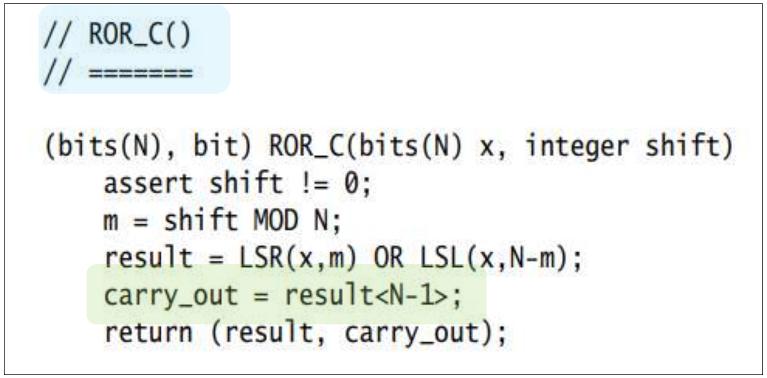
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf



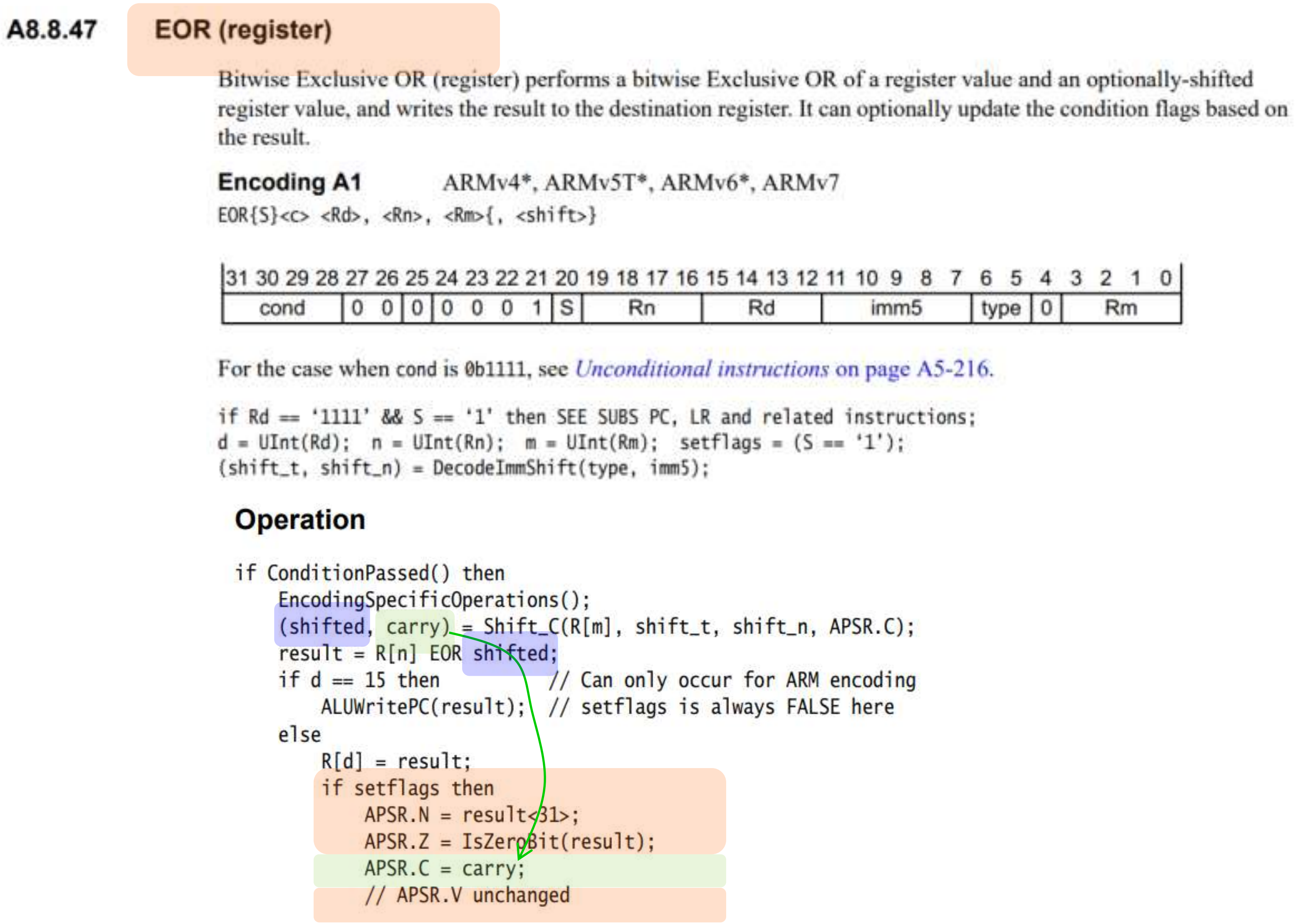
```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);

            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```



```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

Flag-update examples According to Instructions



A8.8.47 EOR (register)

Bitwise Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
EOR{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}
```

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0	0	0	0	0	0	1	S	Rn	Rd	imm5	type	0	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Example) N=32, shift=5
result = extended_x[36:5]
carry_out = extended_x[4]

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

LDRB, LDRSB

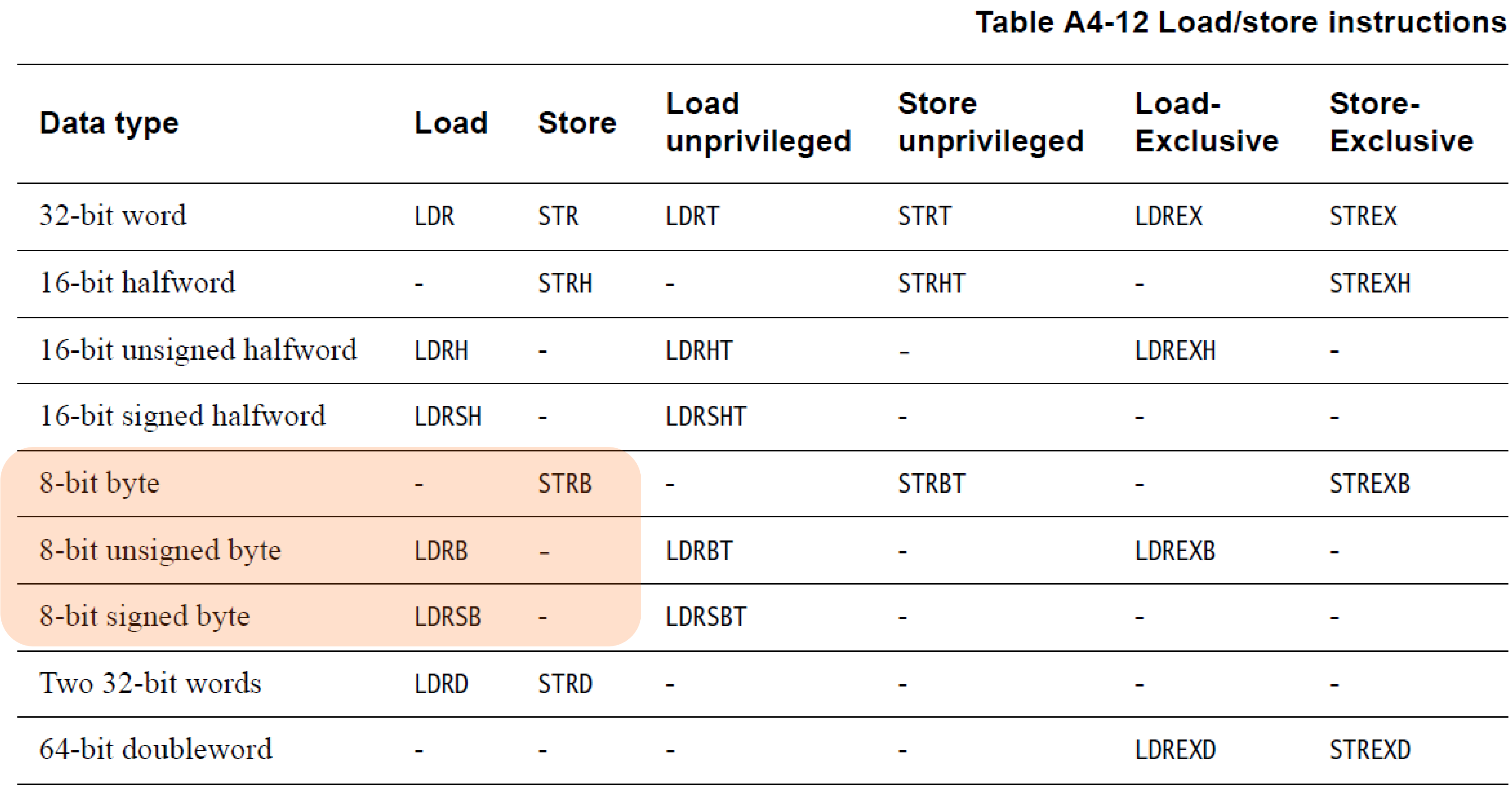
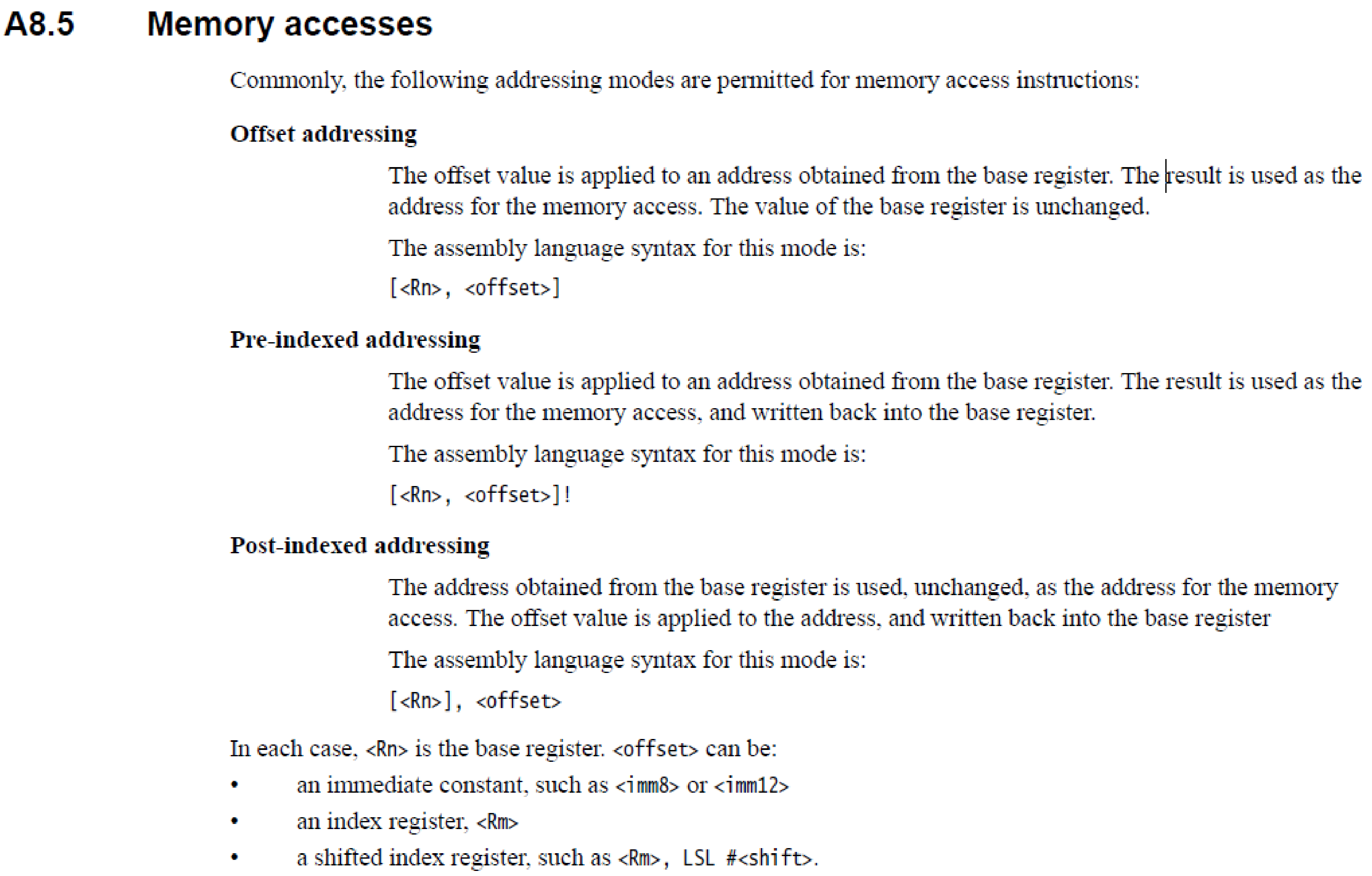


Table A4-12 Load/store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load-Exclusive	Store-Exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD



A8.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>, <offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register

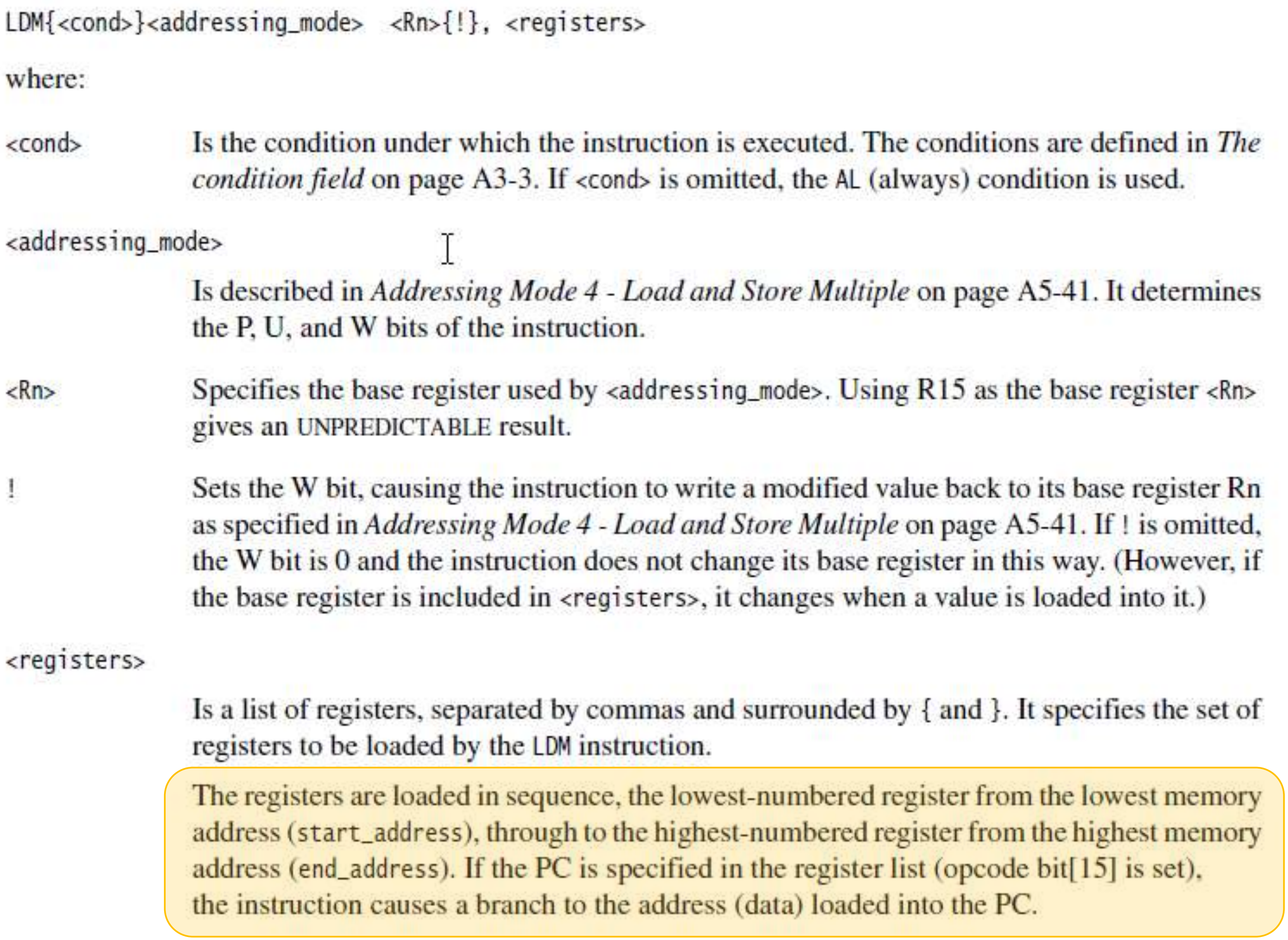
The assembly language syntax for this mode is:

```
[<Rn>], <offset>
```

In each case, <Rn> is the base register. <offset> can be:

- an immediate constant, such as <imm8> or <imm12>
- an index register, <Rm>
- a shifted index register, such as <Rm>, LSL #<shift>.

LDM



```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P, U, and W bits of the instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as the base register `<Rn>` gives an UNPREDICTABLE result.

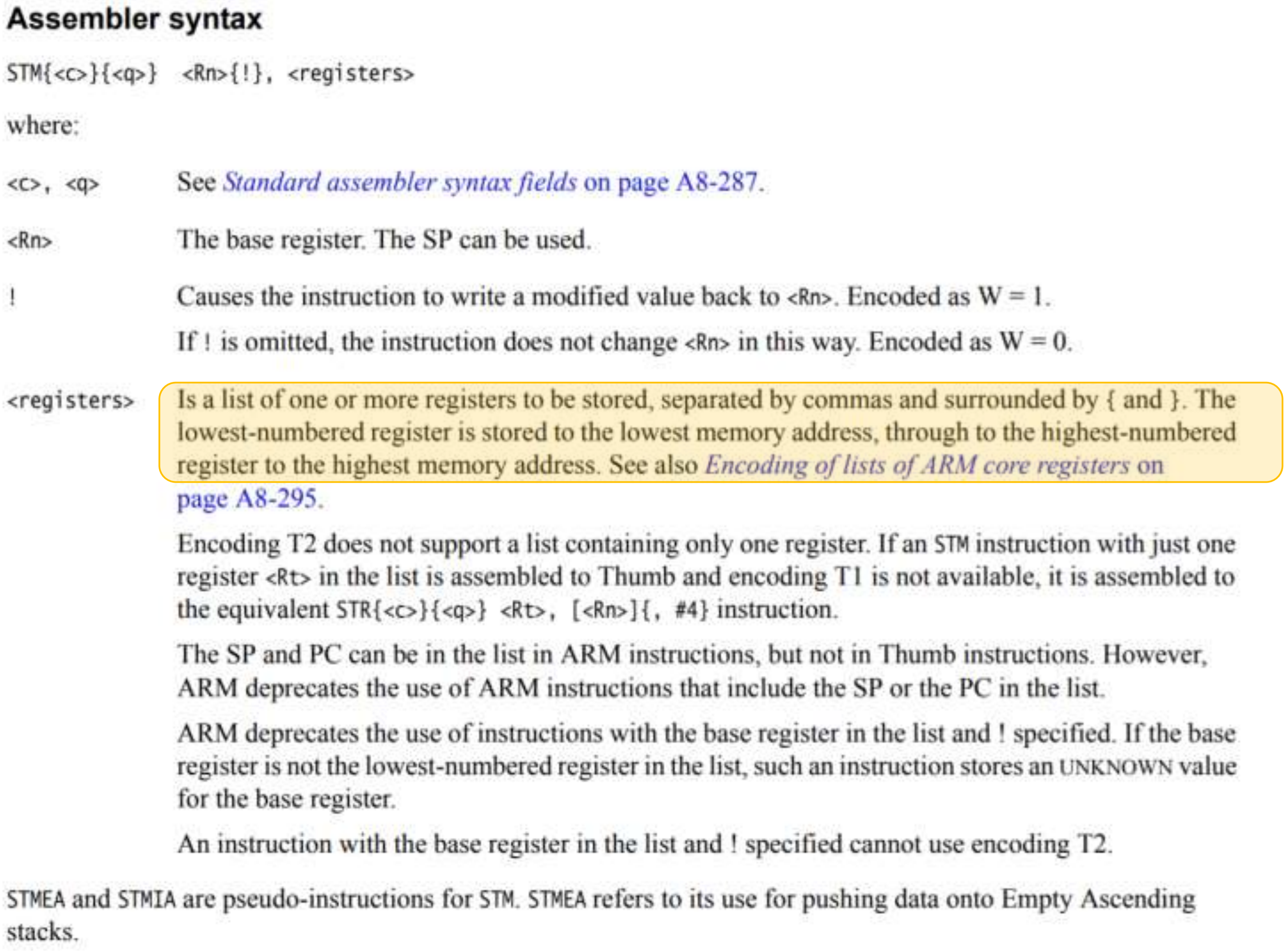
`!` Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If `!` is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in `<registers>`, it changes when a value is loaded into it.)

`<registers>`

Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`). If the PC is specified in the register list (opcode bit[15] is set), the instruction causes a branch to the address (data) loaded into the PC.

STM



Assembler syntax

`STM{<c>}{<q>}  <Rn>{!}, <registers>`

where:

`<c>, <q>` See *Standard assembler syntax fields* on page A8-287.

`<Rn>` The base register. The SP can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of ARM core registers* on page A8-295.

Encoding T2 does not support a list containing only one register. If an `STM` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `STR{<c>}{<q>} <Rt>, [<Rn>]{, #4}` instruction.

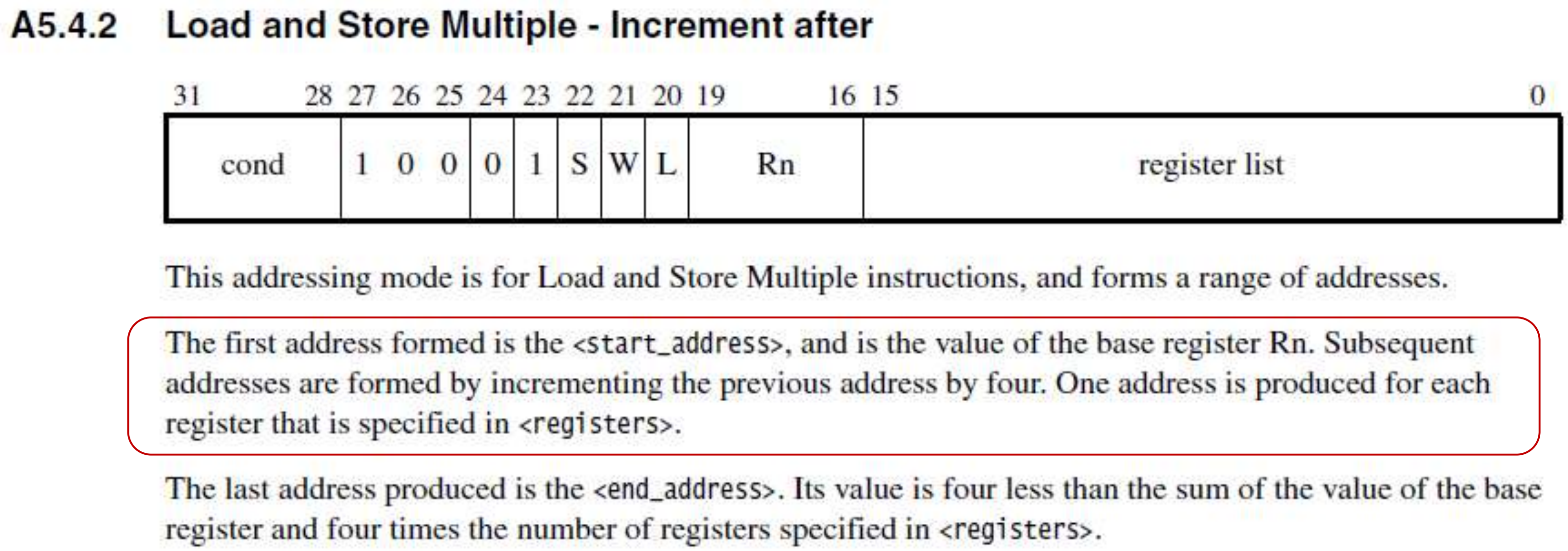
The SP and PC can be in the list in ARM instructions, but not in Thumb instructions. However, ARM deprecates the use of ARM instructions that include the SP or the PC in the list.

ARM deprecates the use of instructions with the base register in the list and ! specified. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

An instruction with the base register in the list and ! specified cannot use encoding T2.

`STMEA` and `STMIA` are pseudo-instructions for `STM`. `STMEA` refers to its use for pushing data onto Empty Ascending stacks.

Increment After, Decrement After



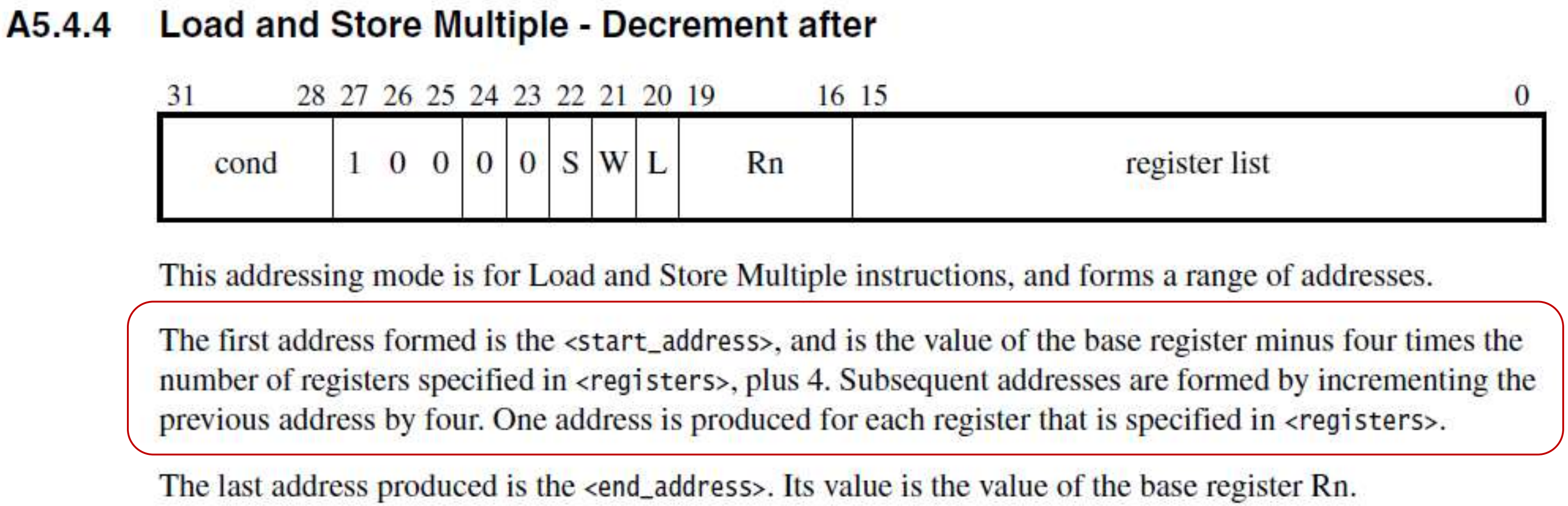
A5.4.2 Load and Store Multiple - Increment after

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	0	1	S	W	L	Rn	register list

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the <start_address>, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.

The last address produced is the <end_address>. Its value is four less than the sum of the value of the base register and four times the number of registers specified in <registers>.



A5.4.4 Load and Store Multiple - Decrement after

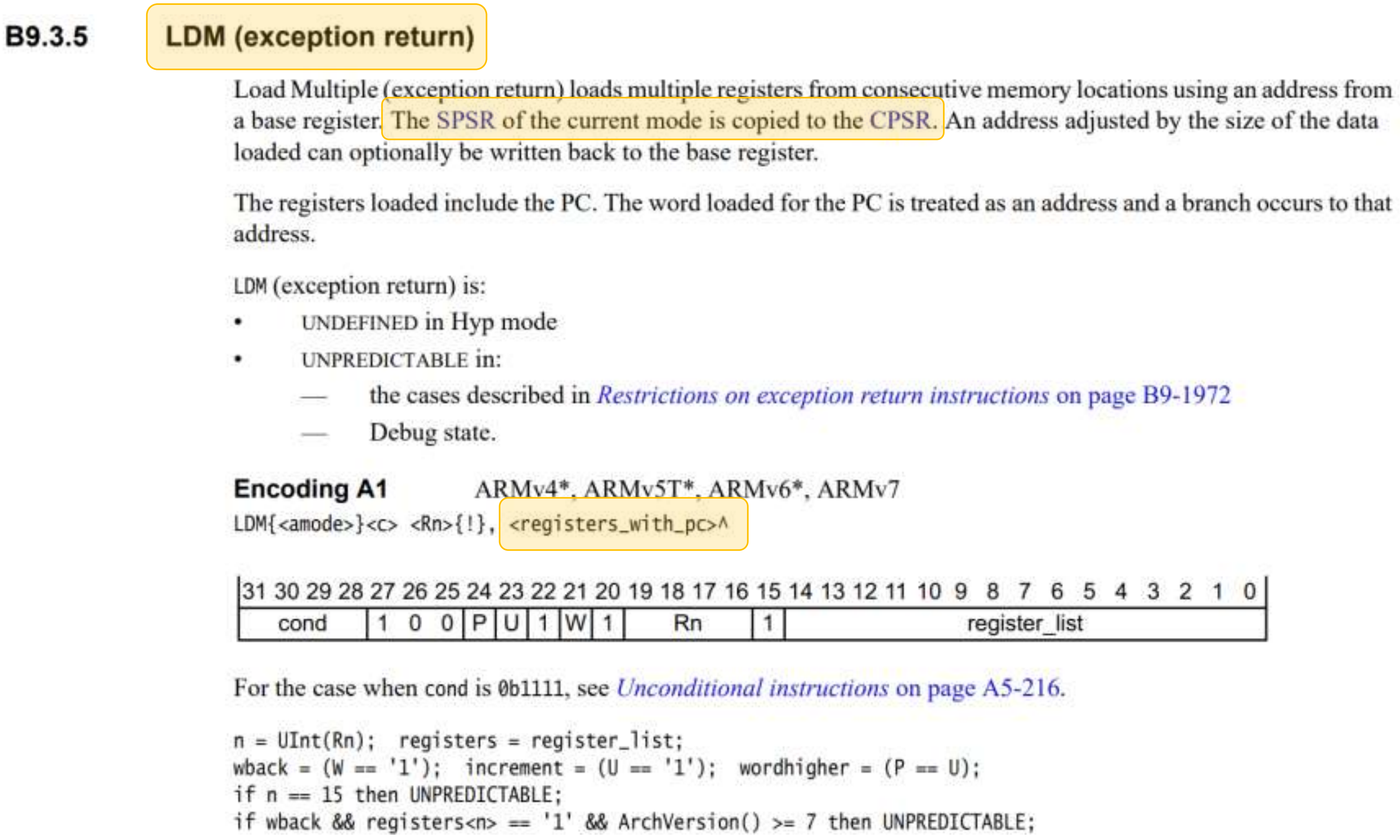
31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	0	0	S	W	L	Rn	register list

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the <start_address>, and is the value of the base register minus four times the number of registers specified in <registers>, plus 4. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.

The last address produced is the <end_address>. Its value is the value of the base register Rn.

Ex) LDMFD sp!, {r0-r15}^



B9.3.5 LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

LDM (exception return) is:

- UNDEFINED in Hyp mode
- UNPREDICTABLE in:
  - the cases described in *Restrictions on exception return instructions* on page B9-1972
  - Debug state.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

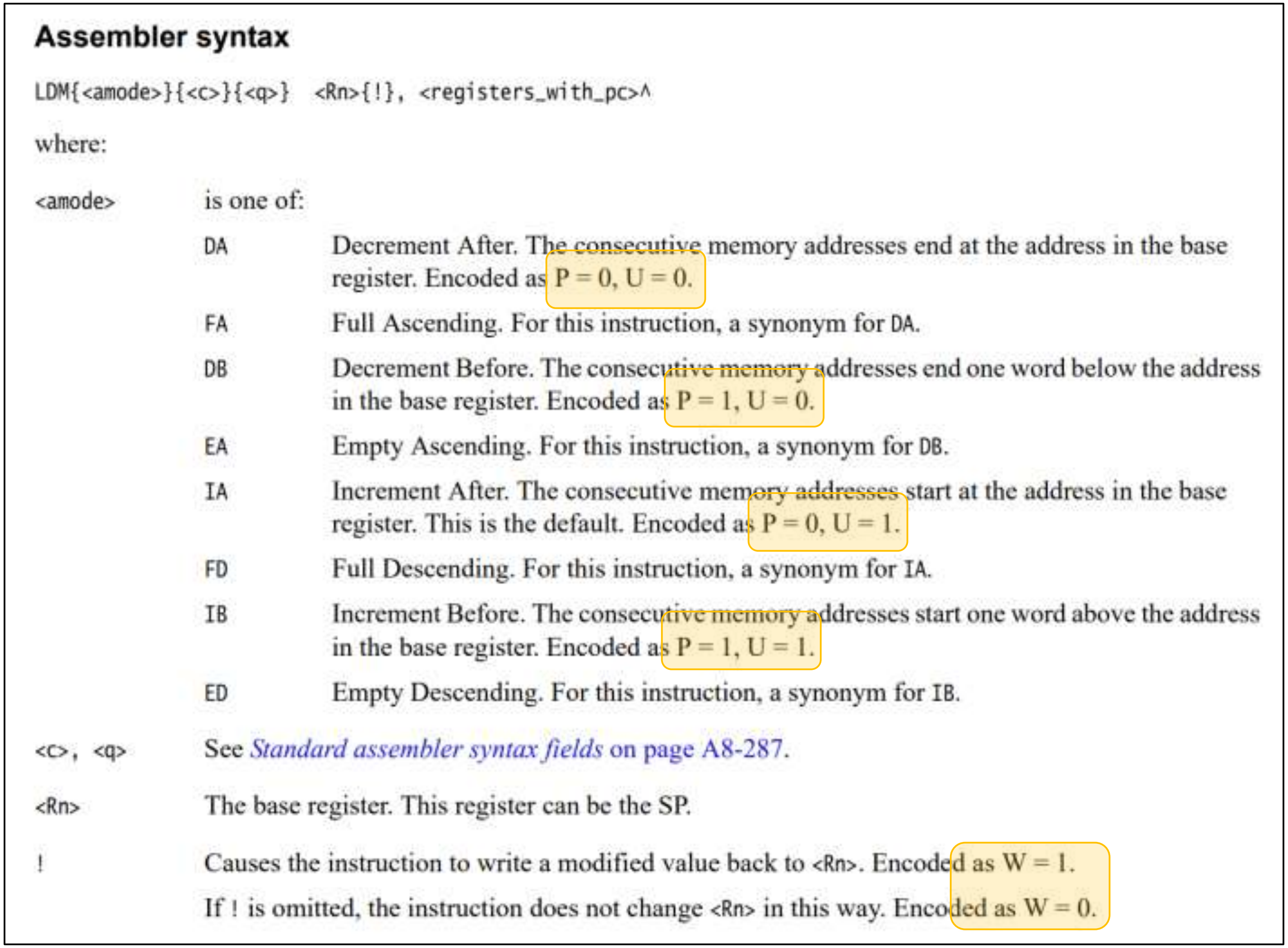
`LDM{<amode>}<c> <Rn>{!}, <registers_with_pc>^`

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1	0	0	P	U	1	W	1	Rn	1	register_list

For the case when cond is `0b1111`, see *Unconditional instructions* on page A5-216.

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

Ex) LDMFD sp!, {r0-r15}^



Assembler syntax

```
LDM{<amode>}{<c>}{<q>}  <Rn>{!}, <registers_with_pc>^
```

where:

`<amode>` is one of:

`DA` Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

`FA` Full Ascending. For this instruction, a synonym for `DA`.

`DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

`EA` Empty Ascending. For this instruction, a synonym for `DB`.

`IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

`FD` Full Descending. For this instruction, a synonym for `IA`.

`IB` Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

`ED` Empty Descending. For this instruction, a synonym for `IB`.

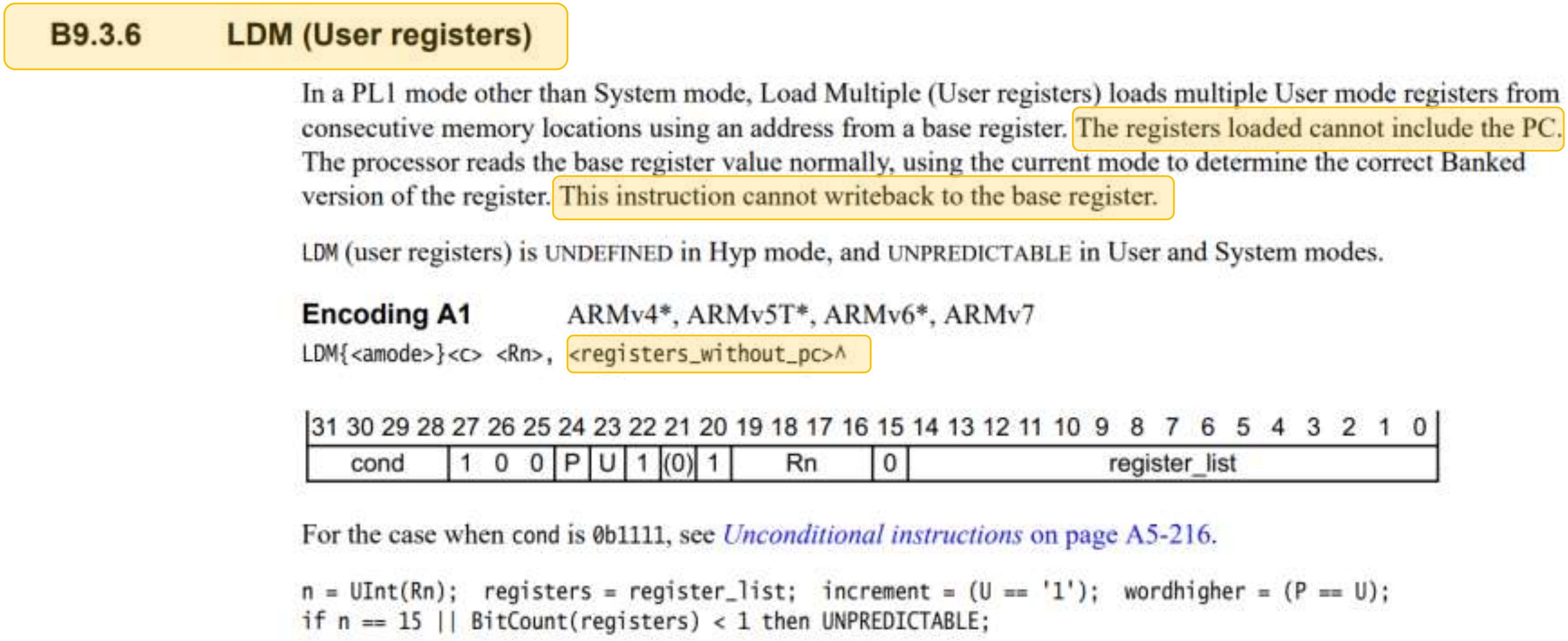
`<c>`, `<q>` See *Standard assembler syntax fields* on page A8-287.

`<Rn>` The base register. This register can be the SP.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

Ex) LDMFD sp, {r0-r14}^



B9.3.6 LDM (User registers)

In a PL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The processor reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

`LDM` (user registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

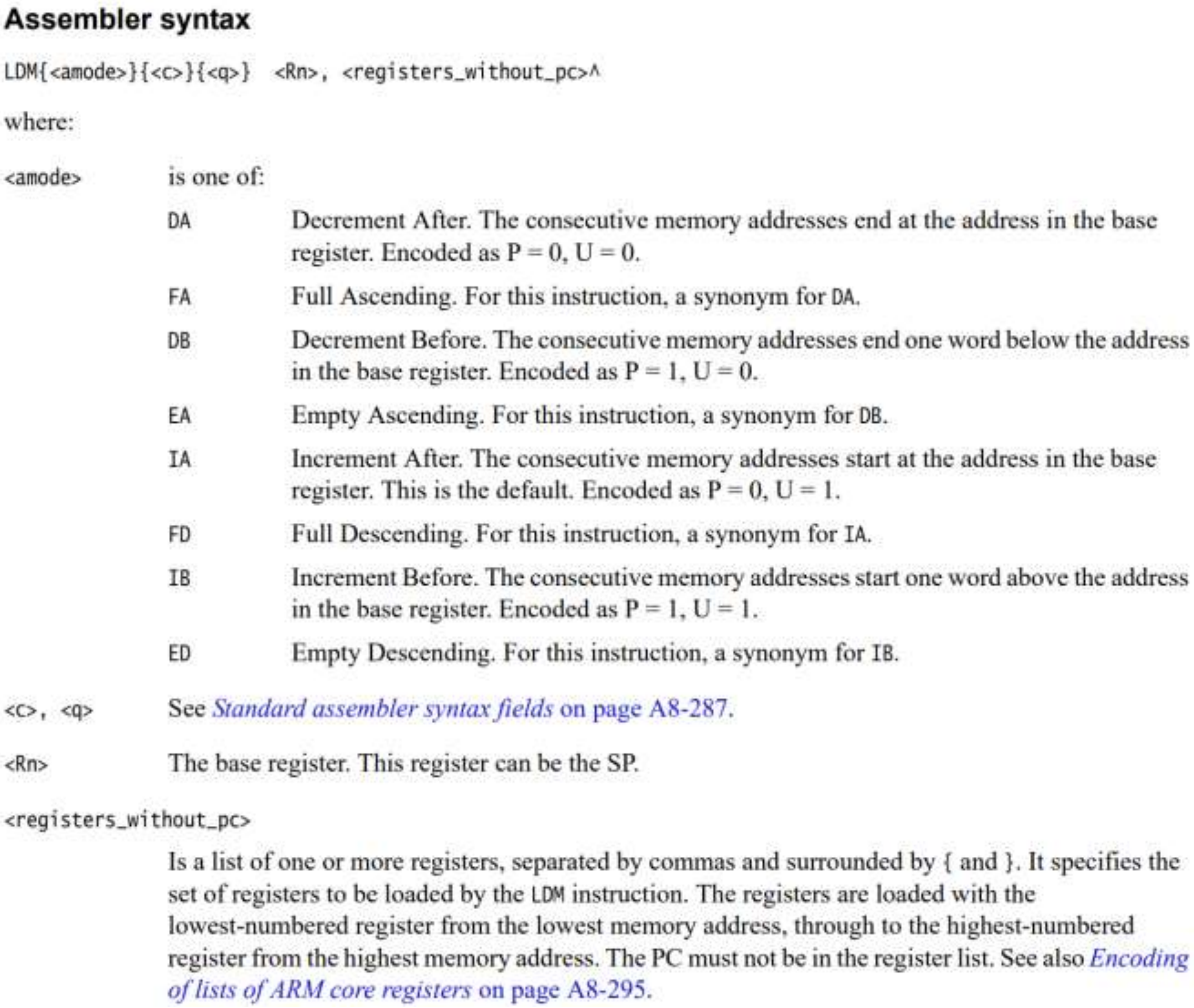
```
LDM{<amode>}<c> <Rn>, <registers_without_pc>^
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	P	U	1	(0)	1	Rn				0	register_list														

For the case when cond is `0b1111`, see *Unconditional instructions* on page A5-216.

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

Ex) LDMFD sp, {r0-r14}^



Assembler syntax

```
LDM{<amode>}{<c>}{<q>}  <Rn>, <registers_without_pc>^
```

where:

`<amode>` is one of:

`DA` Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

`FA` Full Ascending. For this instruction, a synonym for `DA`.

`DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

`EA` Empty Ascending. For this instruction, a synonym for `DB`.

`IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

`FD` Full Descending. For this instruction, a synonym for `IA`.

`IB` Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

`ED` Empty Descending. For this instruction, a synonym for `IB`.

`<c>, <q>` See *Standard assembler syntax fields* on page A8-287.

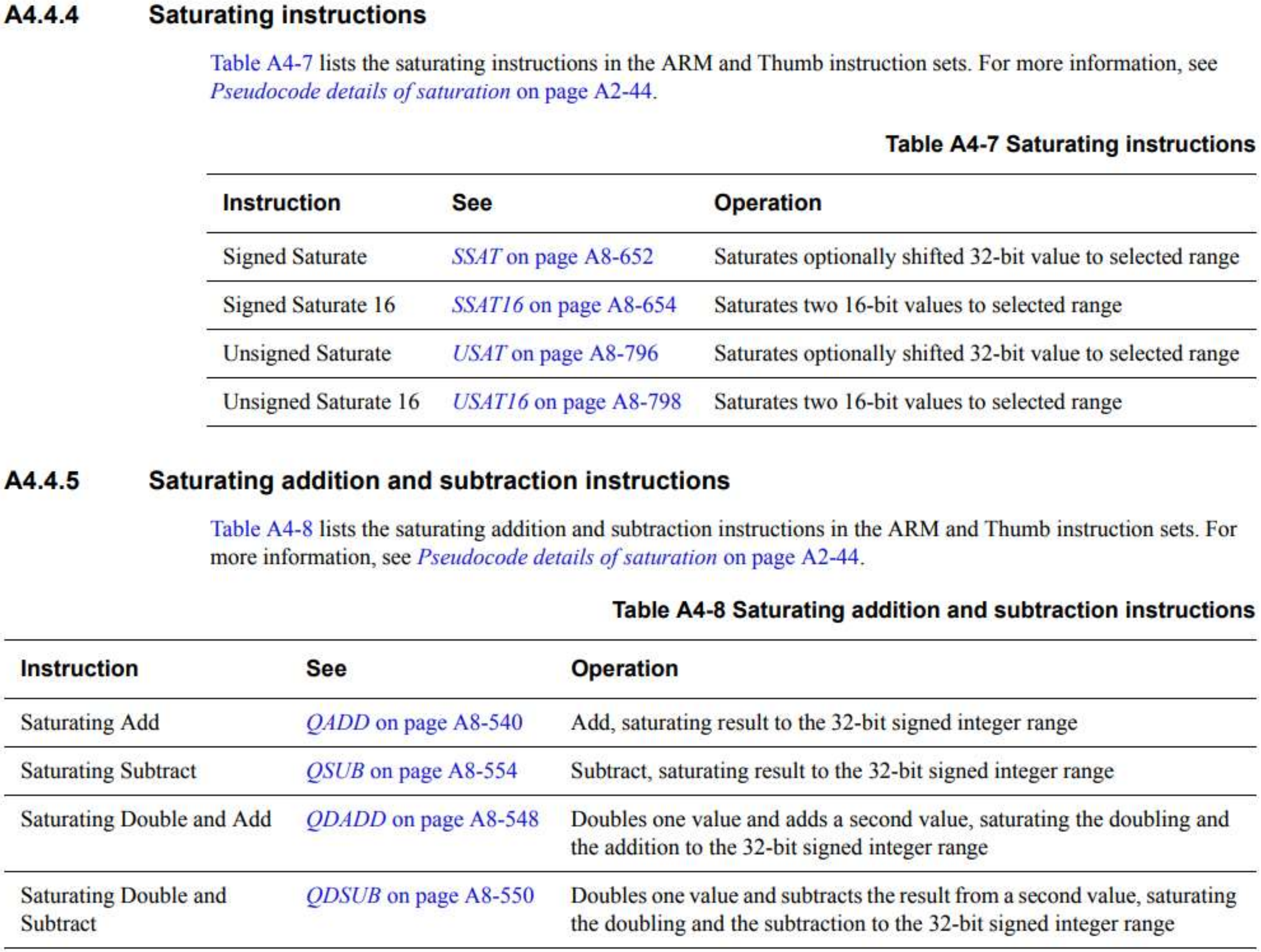
`<Rn>` The base register. This register can be the SP.

`<registers_without_pc>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the `LDM` instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also *Encoding of lists of ARM core registers* on page A8-295.

Saturating Instructions

CPSR'Q flag is set if saturation occurs after the execution of the following inst.



A4.4.4 Saturating instructions

Table A4-7 lists the saturating instructions in the ARM and Thumb instruction sets. For more information, see *Pseudocode details of saturation* on page A2-44.

Table A4-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	*SSAT* on page A8-652	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	*SSAT16* on page A8-654	Saturates two 16-bit values to selected range
Unsigned Saturate	*USAT* on page A8-796	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	*USAT16* on page A8-798	Saturates two 16-bit values to selected range

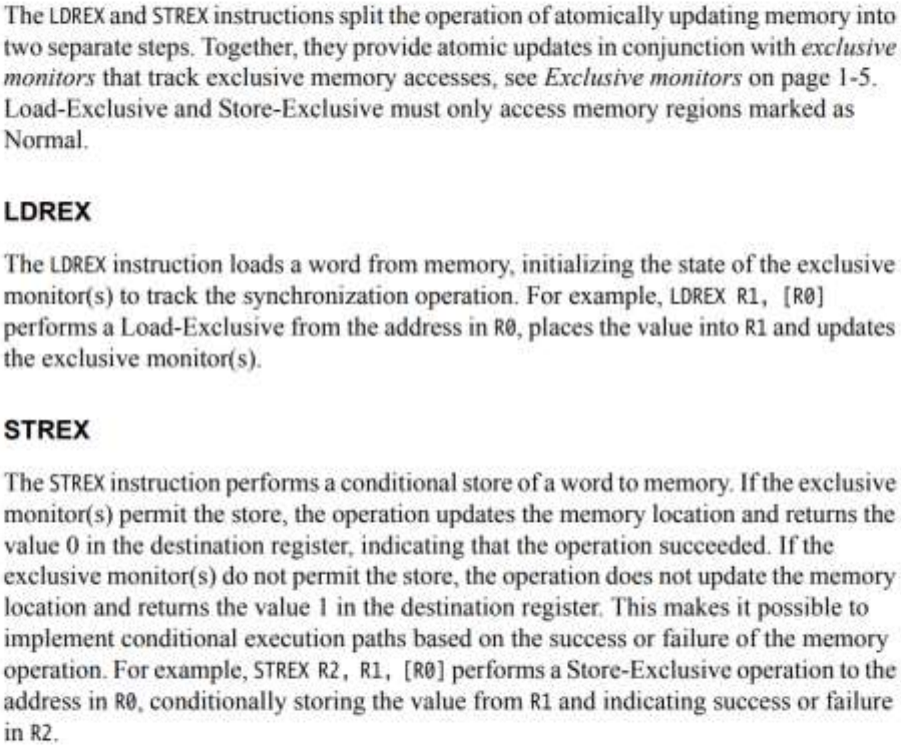
A4.4.5 Saturating addition and subtraction instructions

Table A4-8 lists the saturating addition and subtraction instructions in the ARM and Thumb instruction sets. For more information, see *Pseudocode details of saturation* on page A2-44.

Table A4-8 Saturating addition and subtraction instructions

Instruction	See	Operation
Saturating Add	*QADD* on page A8-540	Add, saturating result to the 32-bit signed integer range
Saturating Subtract	*QSUB* on page A8-554	Subtract, saturating result to the 32-bit signed integer range
Saturating Double and Add	*QDADD* on page A8-548	Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range
Saturating Double and Subtract	*QDSUB* on page A8-550	Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range

1.2.1 LDREX and STREX



The LDREX and STREX instructions split the operation of atomically updating memory into two separate steps. Together, they provide atomic updates in conjunction with *exclusive monitors* that track exclusive memory accesses, see *Exclusive monitors* on page 1-5. Load-Exclusive and Store-Exclusive must only access memory regions marked as Normal.

LDREX

The LDREX instruction loads a word from memory, initializing the state of the exclusive monitor(s) to track the synchronization operation. For example, `LDREX R1, [R0]` performs a Load-Exclusive from the address in R0, places the value into R1 and updates the exclusive monitor(s).

STREX

The STREX instruction performs a conditional store of a word to memory. If the exclusive monitor(s) permit the store, the operation updates the memory location and returns the value 0 in the destination register, indicating that the operation succeeded. If the exclusive monitor(s) do not permit the store, the operation does not update the memory location and returns the value 1 in the destination register. This makes it possible to implement conditional execution paths based on the success or failure of the memory operation. For example, `STREX R2, R1, [R0]` performs a Store-Exclusive operation to the address in R0, conditionally storing the value from R1 and indicating success or failure in R2.

Figure 1-1 shows an example system consisting of one Cortex™-A8 processor, one Cortex-R4 processor, and a memory device shared between the two.

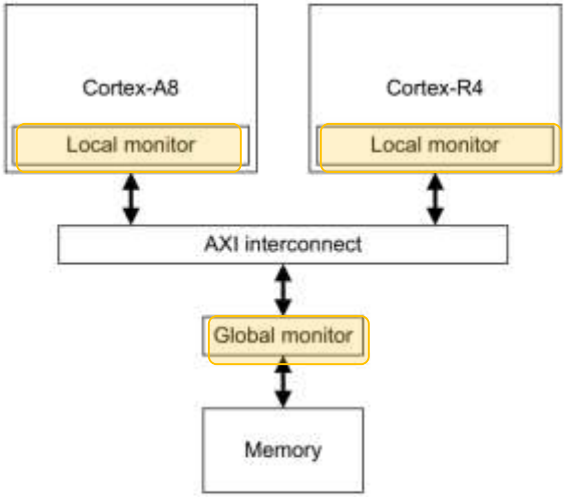
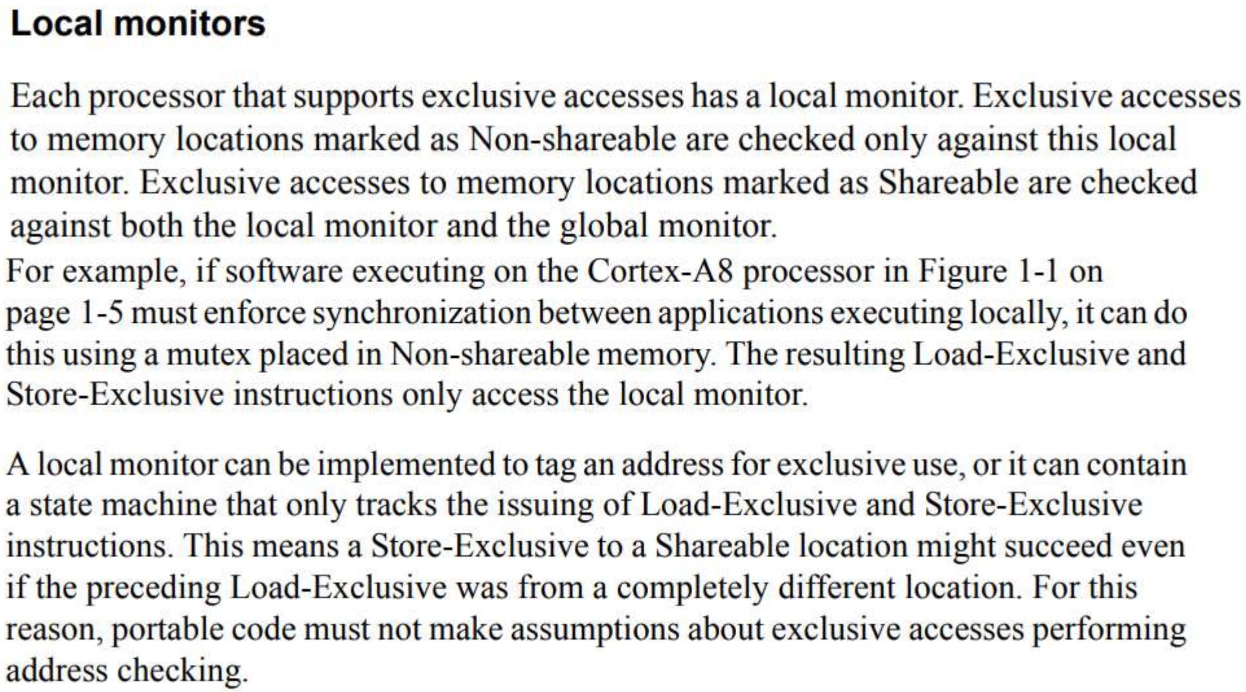


Figure 1-1 Local and global monitors in a multi-core system

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DHT0008A_arm_synchronization_primitives.pdf

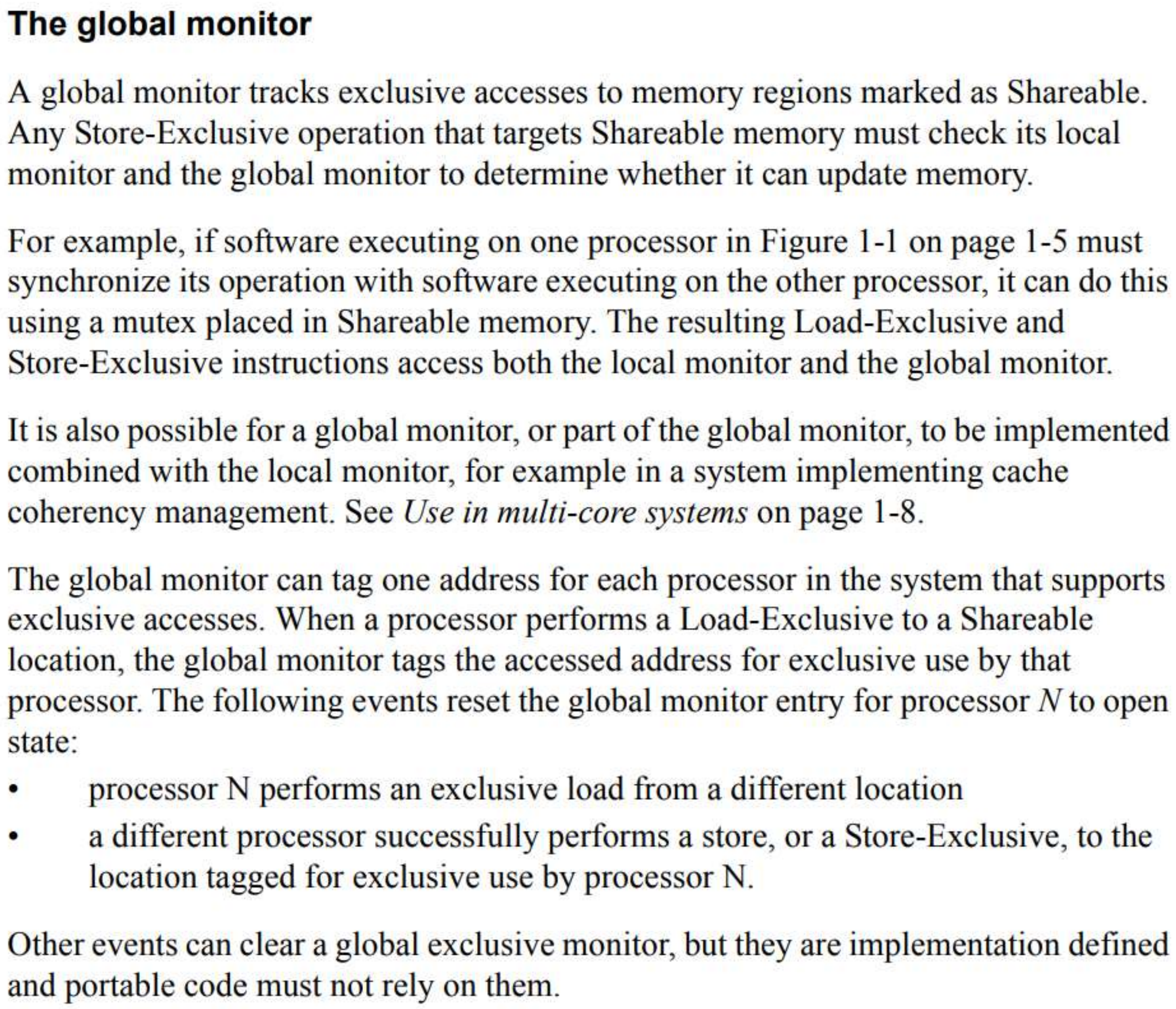


Local monitors

Each processor that supports exclusive accesses has a local monitor. Exclusive accesses to memory locations marked as Non-shareable are checked only against this local monitor. Exclusive accesses to memory locations marked as Shareable are checked against both the local monitor and the global monitor.

For example, if software executing on the Cortex-A8 processor in Figure 1-1 on page 1-5 must enforce synchronization between applications executing locally, it can do this using a mutex placed in Non-shareable memory. The resulting Load-Exclusive and Store-Exclusive instructions only access the local monitor.

A local monitor can be implemented to tag an address for exclusive use, or it can contain a state machine that only tracks the issuing of Load-Exclusive and Store-Exclusive instructions. This means a Store-Exclusive to a Shareable location might succeed even if the preceding Load-Exclusive was from a completely different location. For this reason, portable code must not make assumptions about exclusive accesses performing address checking.



The global monitor

A global monitor tracks exclusive accesses to memory regions marked as Shareable. Any Store-Exclusive operation that targets Shareable memory must check its local monitor and the global monitor to determine whether it can update memory.

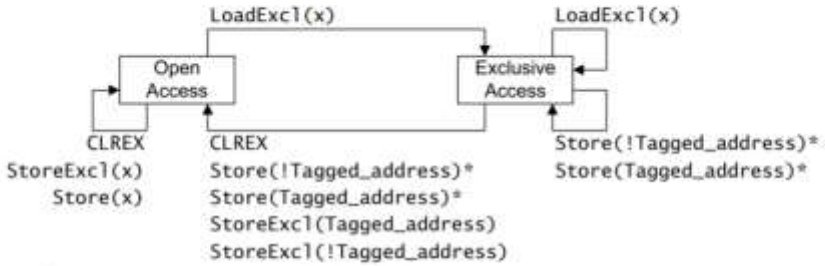
For example, if software executing on one processor in Figure 1-1 on page 1-5 must synchronize its operation with software executing on the other processor, it can do this using a mutex placed in Shareable memory. The resulting Load-Exclusive and Store-Exclusive instructions access both the local monitor and the global monitor.

It is also possible for a global monitor, or part of the global monitor, to be implemented combined with the local monitor, for example in a system implementing cache coherency management. See *Use in multi-core systems* on page 1-8.

The global monitor can tag one address for each processor in the system that supports exclusive accesses. When a processor performs a Load-Exclusive to a Shareable location, the global monitor tags the accessed address for exclusive use by that processor. The following events reset the global monitor entry for processor *N* to open state:

- processor N performs an exclusive load from a different location
- a different processor successfully performs a store, or a Store-Exclusive, to the location tagged for exclusive use by processor N.

Other events can clear a global exclusive monitor, but they are implementation defined and portable code must not rely on them.

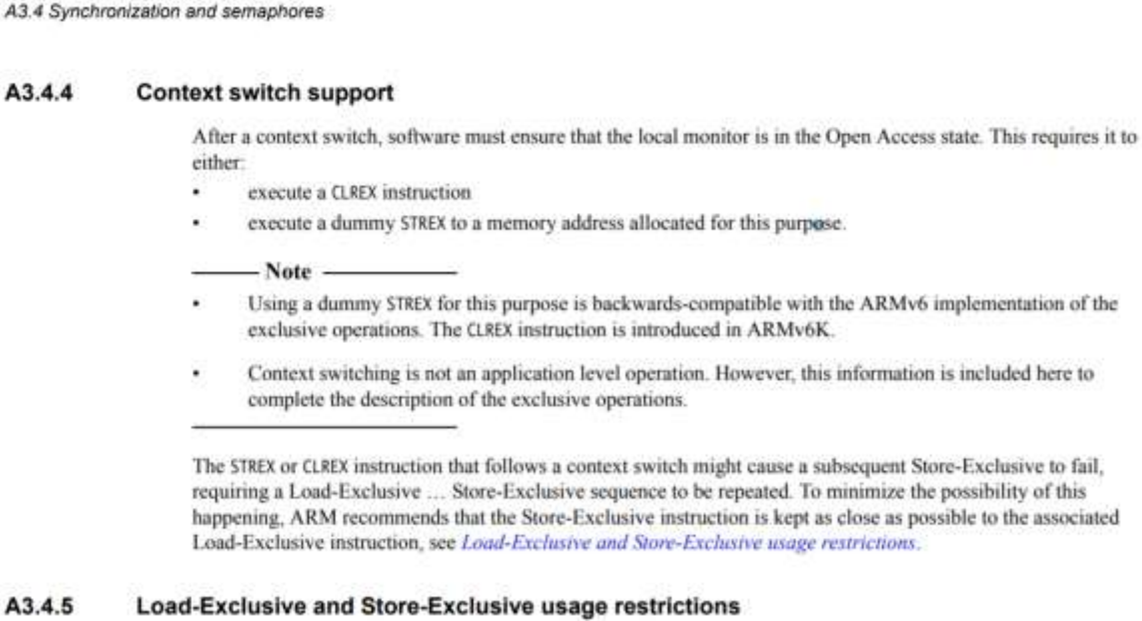


Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x used for the operation.

Figure A3-3 Local monitor state machine diagram

- Both data cache read misses and write misses are non-blocking with up to four outstanding data cache read misses and up to four outstanding data cache write misses being supported.
- The APU data caches offer full snoop coherency control utilizing the MESI algorithm.
- The data cache in Cortex-A9 contains local load/store exclusive monitor for LDREX/STREX synchronizations. These instructions are used to implement semaphores. The exclusive monitor handles one address only, with eight 8 words or one cache line granularity. Therefore, avoid interleaving LDREX/STREX sequences and always execute a CLREX instruction as part of any context switch.



A3.4.4 Context switch support

After a context switch, software must ensure that the local monitor is in the Open Access state. This requires it to either:

- execute a CLREX instruction
- execute a dummy STREX to a memory address allocated for this purpose.

—— **Note** ——

- Using a dummy STREX for this purpose is backwards-compatible with the ARMv6 implementation of the exclusive operations. The CLREX instruction is introduced in ARMv6K.
- Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

The STREX or CLREX instruction that follows a context switch might cause a subsequent Store-Exclusive to fail, requiring a Load-Exclusive … Store-Exclusive sequence to be repeated. To minimize the possibility of this happening, ARM recommends that the Store-Exclusive instruction is kept as close as possible to the associated Load-Exclusive instruction, see *Load-Exclusive and Store-Exclusive usage restrictions*.

A3.4.5 Load-Exclusive and Store-Exclusive usage restrictions