



고려대학교
KOREA UNIVERSITY

COMP427 Embedded Systems

Lecture 4. 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 Thumb mode, instructions are 16-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

ARM Instruction Format

Arithmetic and Logical Instructions

Memory Access Instructions (Load/Store)

Branch Instructions

Software Interrupt Instruction

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 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 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 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																									
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 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 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																													

Condition Field

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	-

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



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

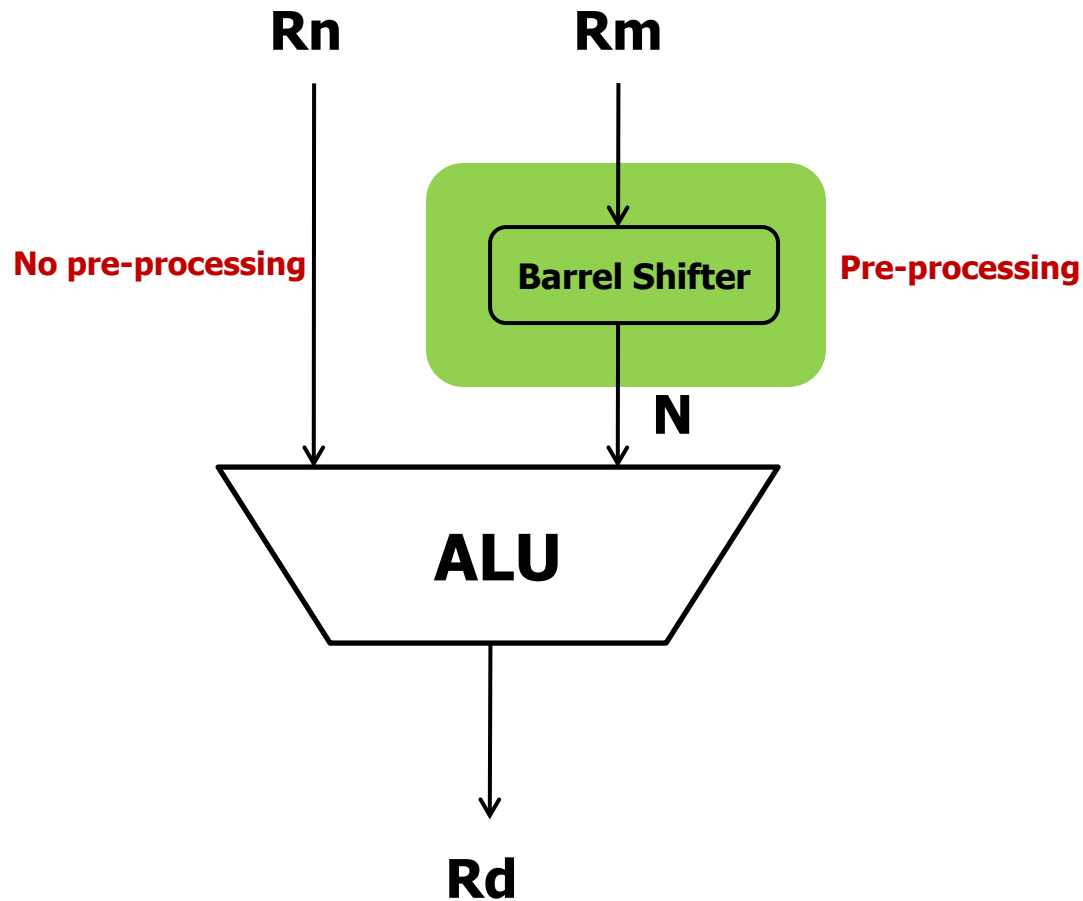
$sa < sb?$ Yes if (N != V)

- (+) - (+) : 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

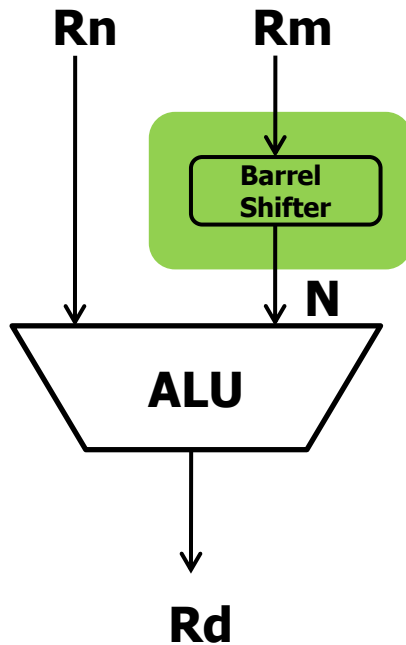
Data Processing Instructions

- **Move instructions**
- **Arithmetic instructions**
- **Logical instructions**
- **Comparison instructions**
- **Multiply instructions**

Execution Unit in ARM



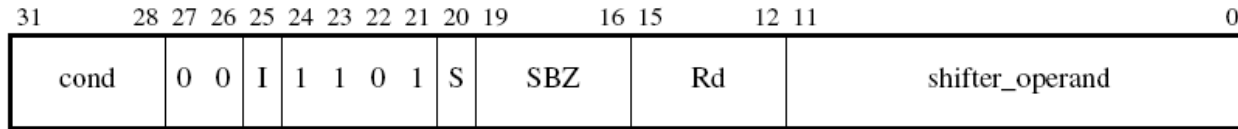
Move Instructions



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	$Rd = \sim N$

Move Instructions – MOV



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
```

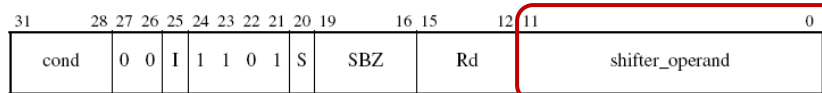
MOV Example

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



Encoded here

MOVS r0, r1, LSL #1

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.

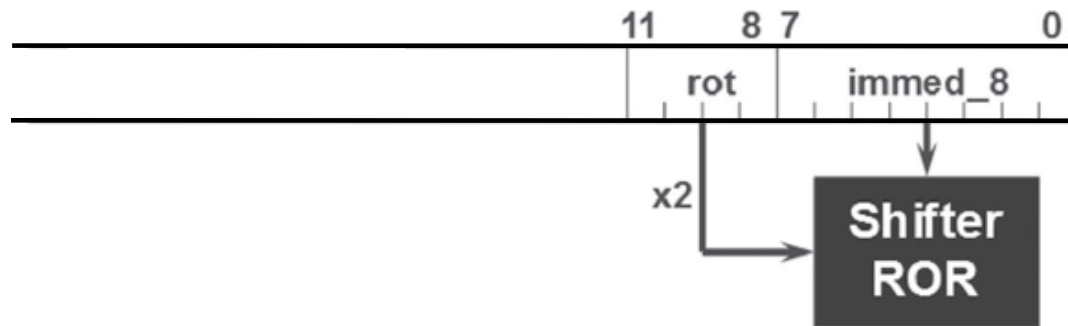
MOV can optionally update the condition code flags, based on the result.

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	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



Loading 32-bit Constants

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

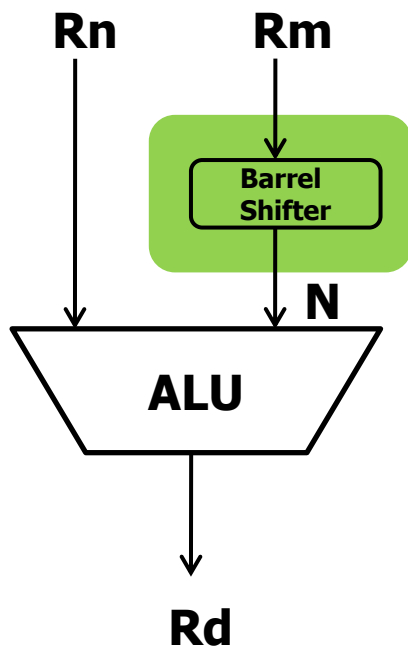
`LDR rd, =const`

- 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 ...
```

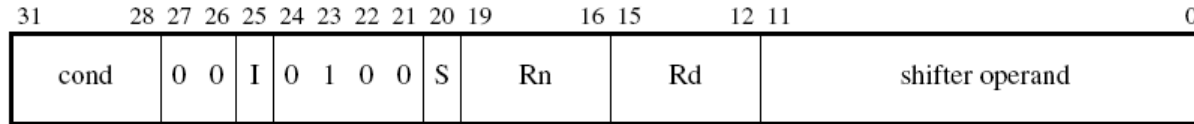
Arithmetic Instructions



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

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

Arithmetic Instructions – ADD



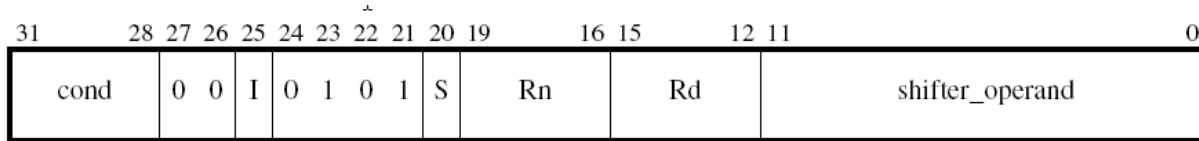
ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

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



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: R4 and R5

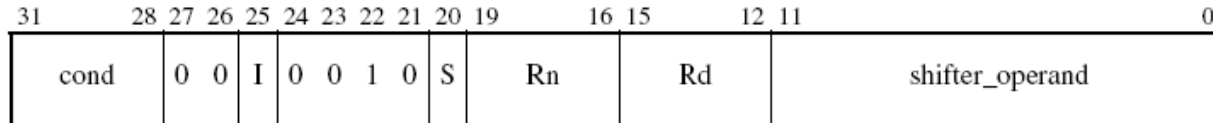
64 bit 2nd operand: R8 and R9

64 bit result: R0 and R1

`ADDS R0, R4, R8 ; R0 = R4 + R8 and set carry accordingly`

`ADCS R1, R5, R9 ; R1 = R5 + R9 + (Carry flag)`

Arithmetic Instructions – SUB



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

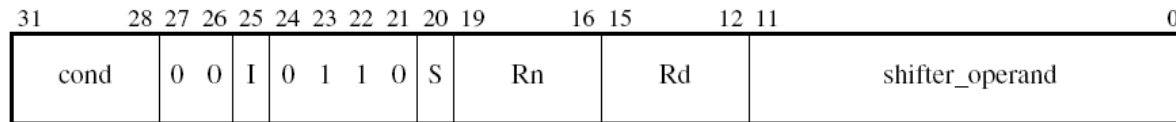
The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

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



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**: R4 and R5

64 bit **2nd operand**: R8 and R9

64 bit **result**: R0 and R1

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

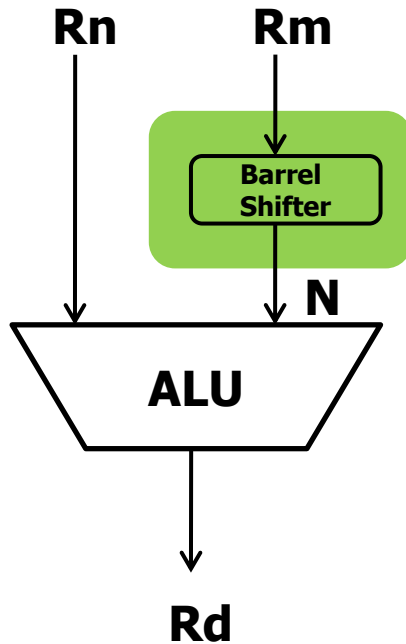
Before: cpsr = nzcv
 r1 = 0x0000_0001

SUBS r1, r1, #1

After: cpsr = n**Z**Cv
 r1 = 0x0000_0000

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

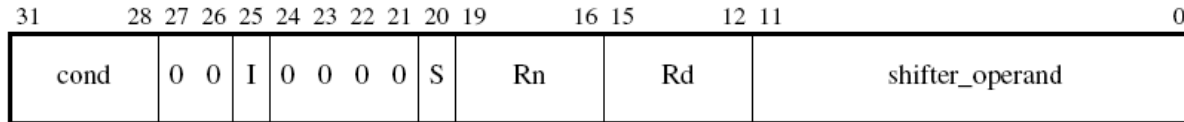
Logical Instructions



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	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear	$Rd = Rn \& \sim N$

Logical Instructions – AND



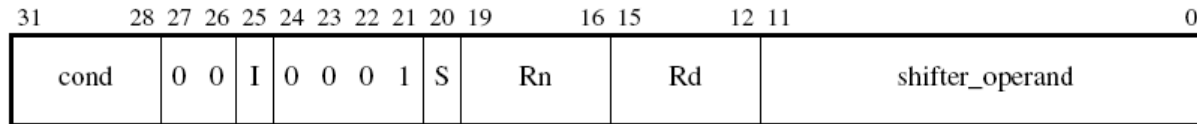
AND performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

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



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

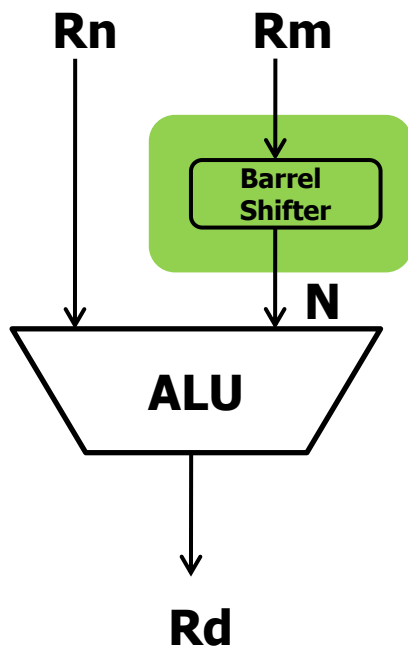
After: r0 = 0x1234_5678

Before: r1 = 0b1111
 r2 = 0b0101

BIC r0, r1, r2

After: r0 = 0b1010

Comparison Instructions

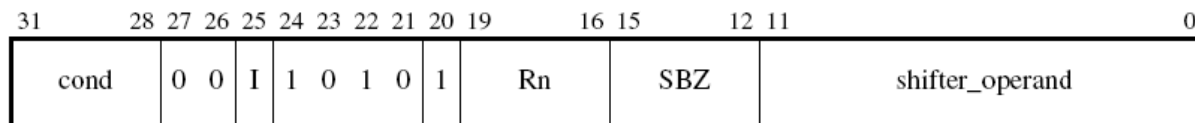


- The comparison instructions update the cpsr flags according to the result, but do **not** affect other registers
- After the bits have been set, the information can be used to change program flow by using conditional execution

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

CMN	compare negated	Flags set as a result of $Rn + N$
CMP	Compare	Flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	Flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	Flags set as a result of $Rn \& N$

Comparison Instructions – CMP



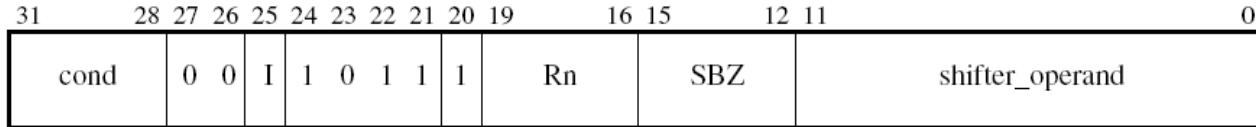
CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

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 cpsr flags based on the execution result

```
CMP R0, R1;
```

Comparison Instructions – CMN



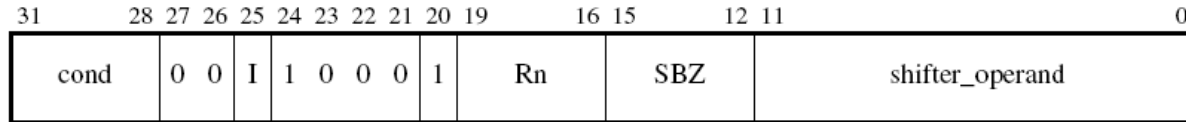
CMN (Compare Negative) compares one value with the two's complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

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 cpsr flags based on the execution result

```
CMN R0, R1;
```

Comparison Instructions – TST



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 update cpsr flags 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

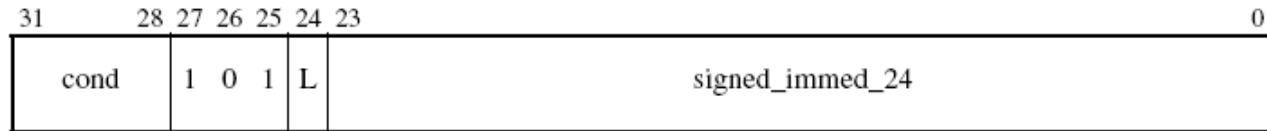
Branch Instructions

- 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	<i>pc = label</i>
BL	branch with link	<i>pc = label</i> <i>lr = address of the next instruction after the BL</i>

B, BL



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 $\pm 32\text{MB}$ away from the current instruction

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

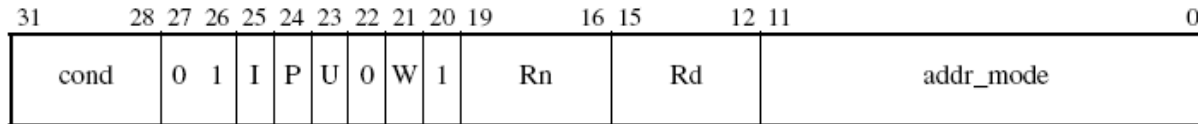
Memory Access Instructions

- 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 a word into a register	$Rd \leftarrow mem32[address]$
STR	Store a word from a register to memory	$Rd \rightarrow mem32[address]$
LDRB	Load a byte into a register	$Rd \leftarrow mem8[address]$
STRB	Store a byte from a register to memory	$Rd \rightarrow mem8[address]$
LDRH	Load a half-word into a register	$Rd \leftarrow mem16[address]$
STRH	Store a half-word into a register	$Rd \rightarrow mem16[address]$
LDRSB	Load a signed byte into a register	$Rd \leftarrow SignExtend (mem8[address])$
LDRSH	Load a signed half-word into a register	$Rd \leftarrow SignExtend (mem16[address])$

LDR (Load Register)

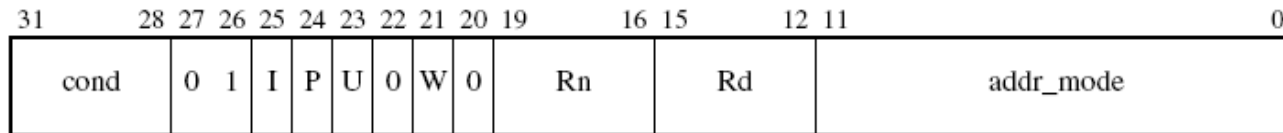


LDR (Load Register) loads a word from a memory address.

- 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)



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

Indexing Method	Data	Base Address register updated?	Example
Preindex with writeback	Mem[base + offset]	Yes (Base + offset)	<i>LDR r0, [r1, #4]!</i>
Preindex	Mem[base + offset]	No	<i>LDR r0, [r1, #4]</i>
Postindex	Mem[base]	Yes (Base + offset)	<i>LDR r0, [r1], #4</i>

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

Before:

r0 = 0x0000_0000

r1 = 0x0009_0000

Mem32[0x0009_0000] = 0x01010101

Mem32[0x0009_0004] = 0x02020202

LDR r0, [r1, #4]!

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

LDR r0, [r1, #4]

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

LDR r0, [r1], #4

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

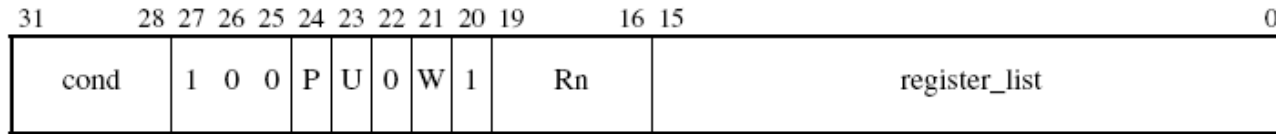
Multiple Register Transfer – LDM, STM

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

LDM	Load multiple registers
STM	Store multiple registers

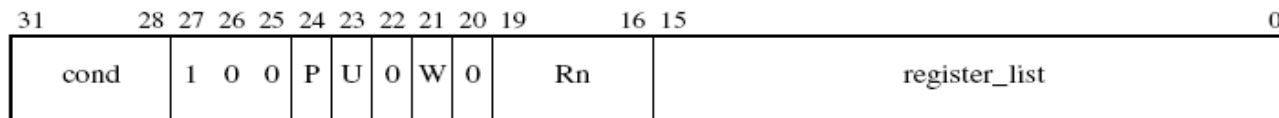
Addressing Mode	Description	Start address	End address	Rn!
IA	Increment After	Rn	$Rn + 4 \times N - 4$	$Rn + 4 \times N$
IB	Increment Before	$Rn + 4$	$Rn + 4 \times N$	$Rn + 4 \times N$
DA	Decrement after	$Rn - 4 \times N + 4$	Rn	$Rn - 4 \times N$
DB	Decrement Before	$Rn - 4 \times N$	$Rn - 4$	$Rn - 4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

Multiple Register Transfer – LDM, STM



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



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
 - They are stored from lowest to highest

```
STMFD R13! {R0, R1} // R13 is updated
LDMFD R13! {R1, R0} // R13 is updated
```

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

```
STMFD R13!, {R0-R12} // R13 is updated appropriately
LDMFD 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

- 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)

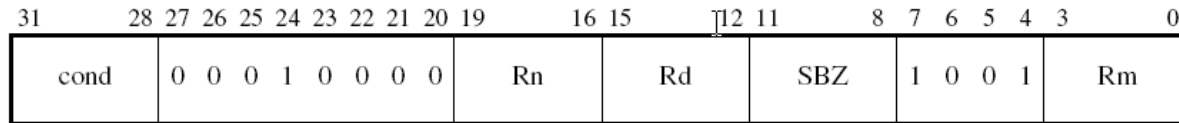
ARM default stack

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

SWAP Instruction



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 until it is done with the read and the write
 - It is useful to implement semaphores and mutual exclusion (mutex) in an OS

Before:

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

SWP r0, r1, [r2]

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
```

Miscellaneous but Important Instructions

- Software interrupt instruction
- Program status register instructions

SVC

(previously SWI (Software Interrupt))



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 = nzcVqift_**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 handler
```

Program status register instructions

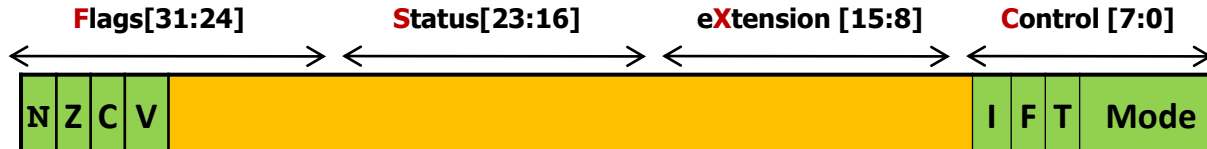
Syntax: **MRS{cond}** Rd, <cpsr | spsr>

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**)



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
```

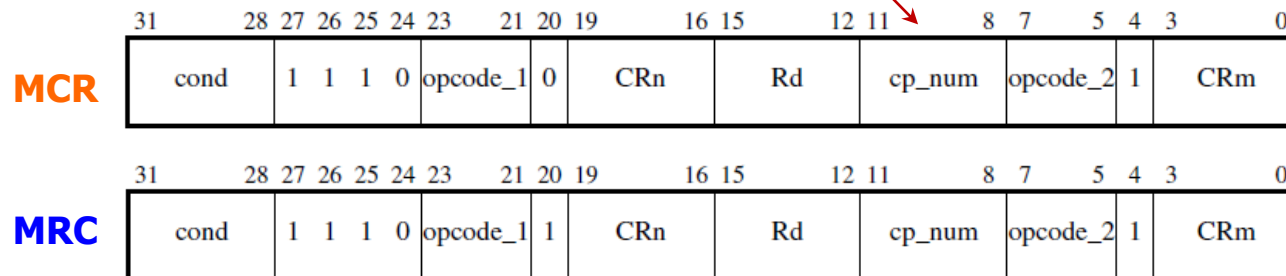
MCR

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

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

Annotations:

- Destination coprocessor reg. (points to <CRn>)
- Additional destination coprocessor reg. (points to <CRm>)
- Coprocessor name: p0, p1,...,p15 (points to <coproc>)
- Coprocessor specific opcode (points to <opcode_2>)



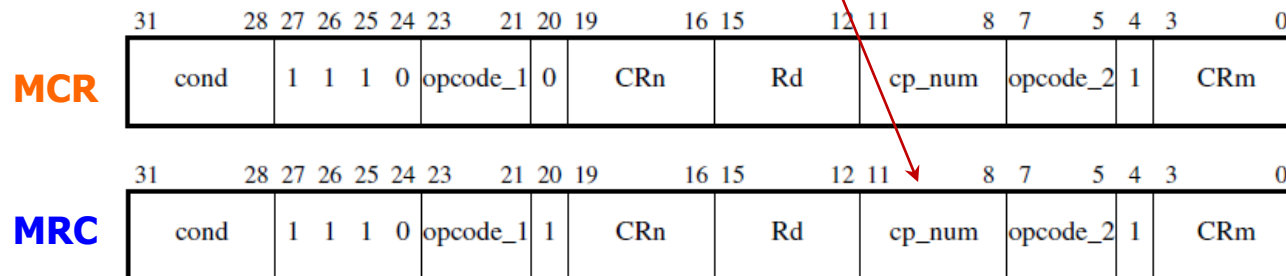
MRC

- 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>}

Annotations:

- Destination coprocessor reg. (points to <CRn>)
- Additional destination coprocessor reg. (points to <CRm>)
- Coprocessor name: p0, p1,...,p15 (points to <coproc>)
- Coprocessor specific opcode (points to <opcode_1>)



(Assembly) Language

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

Backup Slides

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.

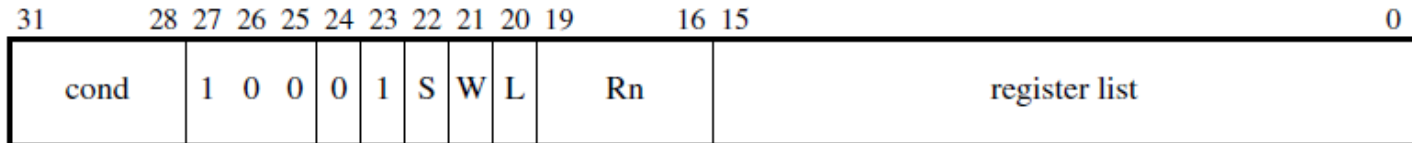
! Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register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

Increment After, Decrement After

A5.4.2 Load and Store Multiple - Increment after

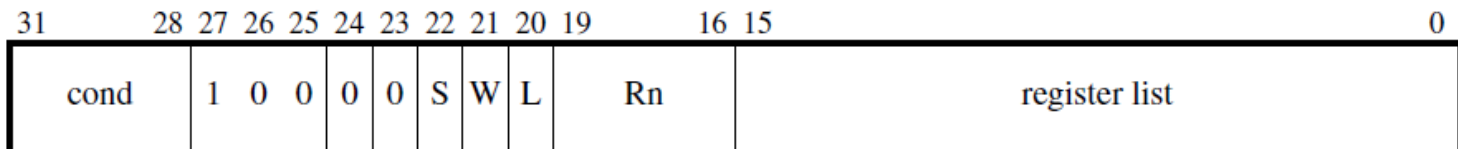


This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

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



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.