ARM Instructions

ARM Instruction
Data processing instructions
Branch instructions
Load-store instructions
SWI instruction
Loading instructions
Conditional Execution

Data Processing Instructions

- Perform arithmetic and logical operations on data values in registers
- Only instructions which modify data value
- All other instructions move data and control the sequence of program execution
- It requires two operands and produces a single result

Data Processing Instructions

- All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.
- The result, if there is one, is 32 bits wide and is placed in a register. (There is an exception here: long multiply instructions produce a 64-bit result)
- Each of the operand registers and the result register are independently specified in the instruction.
- The ARM uses a '3-address' format for these instructions.
- ADD r0, r1, r2   ; comment begins r0 = r1 + r2
- values in the source registers are 32 bits wide and may be considered to be either unsigned integers or signed 2's-complement integers.

Arithmetic operations

- These instructions perform binary arithmetic (addition, subtraction and reverse subtraction, which is subtraction with the operand order reversed) on two 32-bit operands.
- The operands may be unsigned or 2's-complement signed integers; the carry-in, when used, is the current value of the C bit in the CPSR.

Arithmetic operations

```
ADD    r0, r1, r2        ; r0 := r1 + r2
ADC    r0, r1, r2        ; r0 := r1 + r2 + C
SUB    r0, r1, r2        ; r0 := r1 - r2
SBC    r0, r1, r2        ; r0 := r1 - r2 + C - 1
RSB    r0, r1, r2        ; r0 := r2 - r1
RSC    r0, r1, r2        ; r0 := r2 - r1 + C - 1
```

RSC is reverse subtract with carry.

Immediate operands

Add a constant to a register, replace the second source operand with an immediate value, a literal constant preceded by #1

ADD r3, r3, r1; r3 = r3 + 1

AND r8, r7, #&FF ; r8 = r7 + FF Hexadecimal values are specified by adding &

Shifted register operands

- ADD r3, r2, r1, LSL #3
- allows the second register operand to be subject to a shift operation before it is combined with the first operand
- Single instruction executed in single clock cycle
- LSL - Logical shift left
- #3 - shift left 3 times, the number can be from 0 to 31. If 0, it indicates no shift
- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 31 places; fill the vacated bits at the most significant end of the word with zeros.

Shifted register operands

- ASL: Arithmetic shift left - synonym for LSL
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive or with ones if the source operand was negative
- ROR: rotate right by 0 to 32 places, the bits which fall off the least significant end of word are used, in order to fill the vacated bits at the most significant end of the word
- RRX: rotate right extended by 1 place, the vacated bit is filled with the old value of the C flag and the operand is shifted one place to the right.
- It is also possible to use a register value to specify the number of bits the second operand should be shifted by: ADD r5, r5, r3, LSL r2
- This is a 4-address instruction.

Shifted register operands

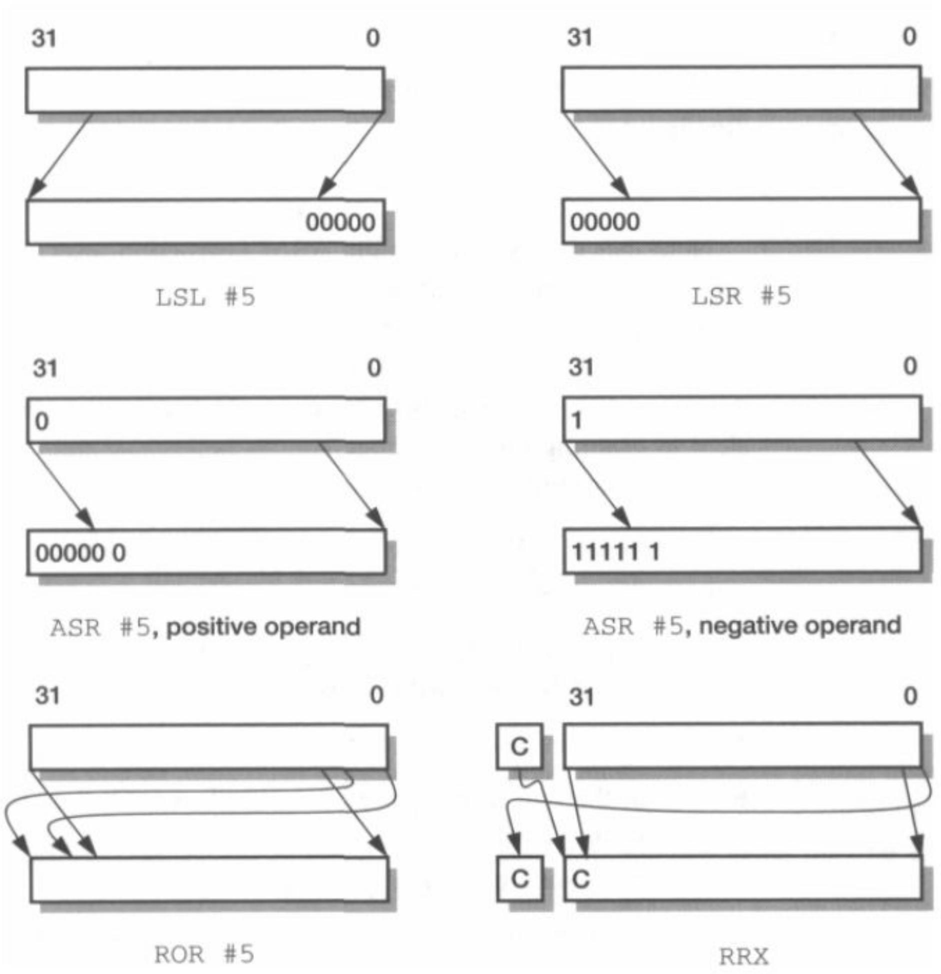
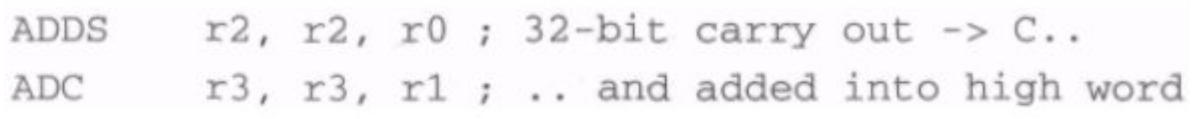


Figure 3.1 ARM shift operations

Setting the condition codes

Any data processing instruction can set the condition codes (N, Z, C and V) if the programmer wishes it to.

The comparison operations only set the condition codes, so there is no option with them, but for all other data processing instructions a specific request must be made.



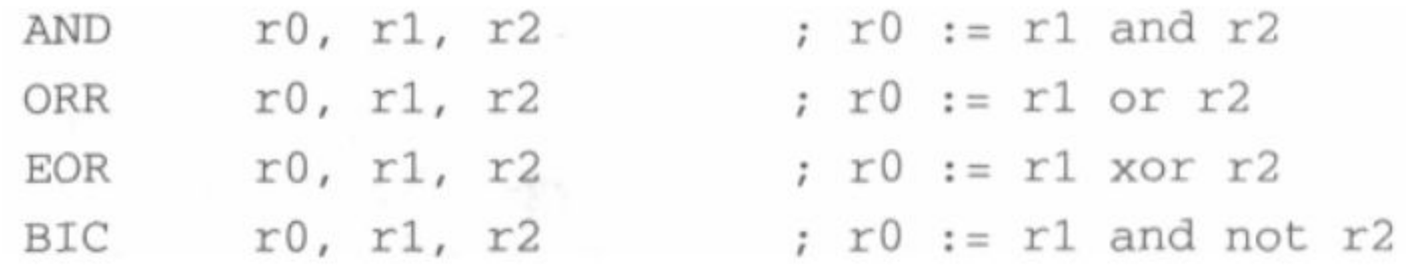
```
ADDS    r2, r2, r0 ; 32-bit carry out -> C..
ADC     r3, r3, r1 ; .. and added into high word
```

Condition Codes

- An arithmetic operation (which here includes CMP and CMN) sets all the flags according to the arithmetic result.
- A logical or move operation does not produce a meaningful value for C or V, so these operations set N and Z according to the result but preserve V, and either preserve C when there is no shift operation, or set C to the value of the last bit to fall off the end of the shift.
- The most important use of the condition codes, which is to control the program flow through the conditional branch instructions

Bit-wise logical operations

- perform the specified Boolean logic operation on each bit pair of the input operands, so in the first case r0[i]:= r1[i] AND r2[i] for each value of i from 0 to 31 inclusive, where r0[i] is the ith bit of r0.
- BIC stands for bit clear - every ' 1' in the second operand clears the corresponding bit in the first



```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

Register movement operations

- These instructions ignore the first operand
- move the second operand (possibly bit-wise inverted) to the destination
- 'MVN' mnemonic stands for 'move negated'
- result register is set to the value obtained by inverting every bit in the source operand

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

```
PRE     r5 = 5
        r7 = 8
        MOV    r7, r5     ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Barrel shifter

MOV Rd, N;

N - register or immediate value

or a register that has been preprocessed by the barrel shifter prior to being used by a data processing instruction

Barrel shifter

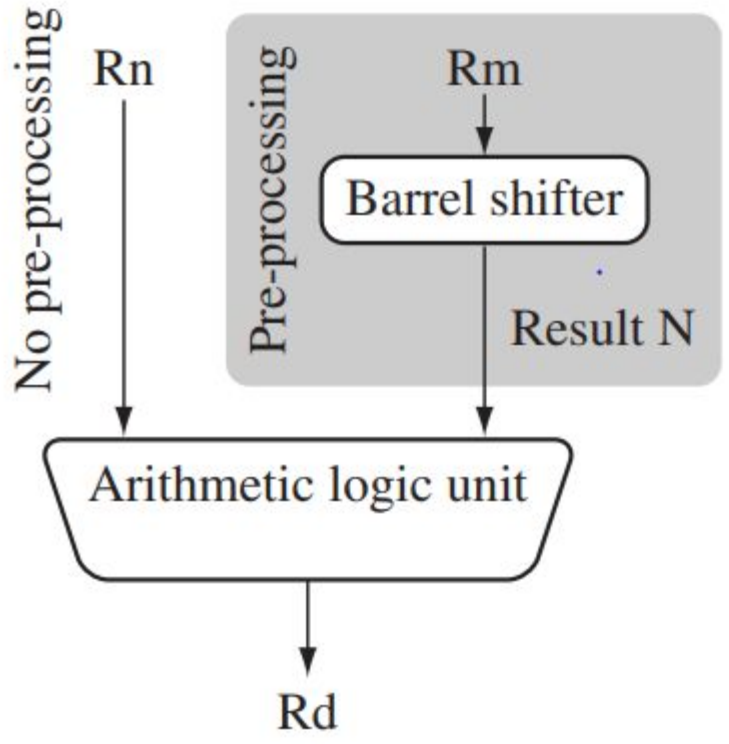
A unique and powerful feature of ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.

Data processing instructions - not using Barrel Shifter

- MUL, CLZ(count leading zeros), QADD(signed saturated 32-bit add).

Pre-processing or shift occurs within the cycle time of the instruction.

Barrel shifter



Register Rn enters the ALU without any pre-processing of registers.

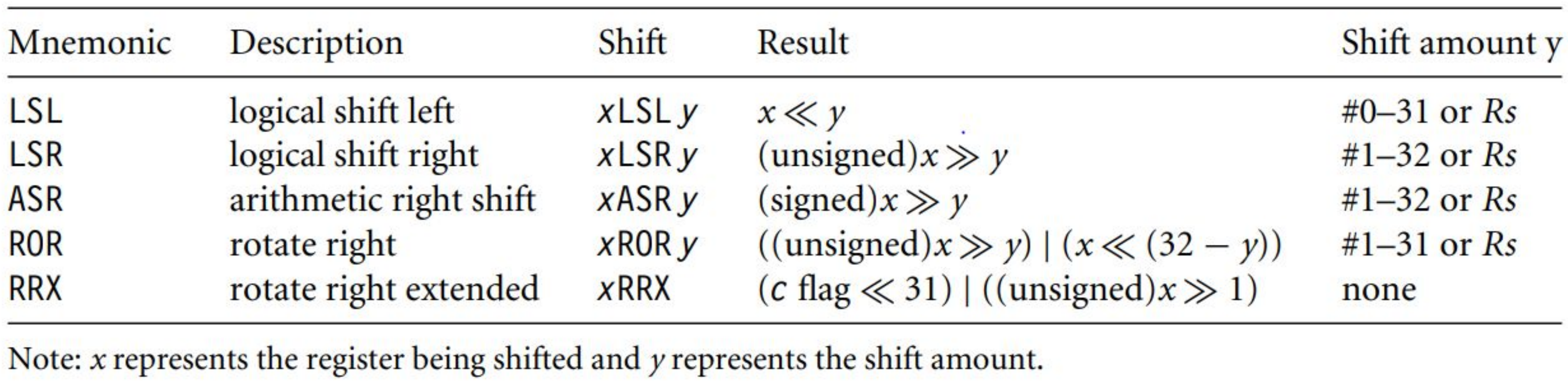
Pre r5 = 5, r7 = 8

MOV r7, r5, LSL #2; let r7 = r5*4 = (r5<<2)

post r5 = 5

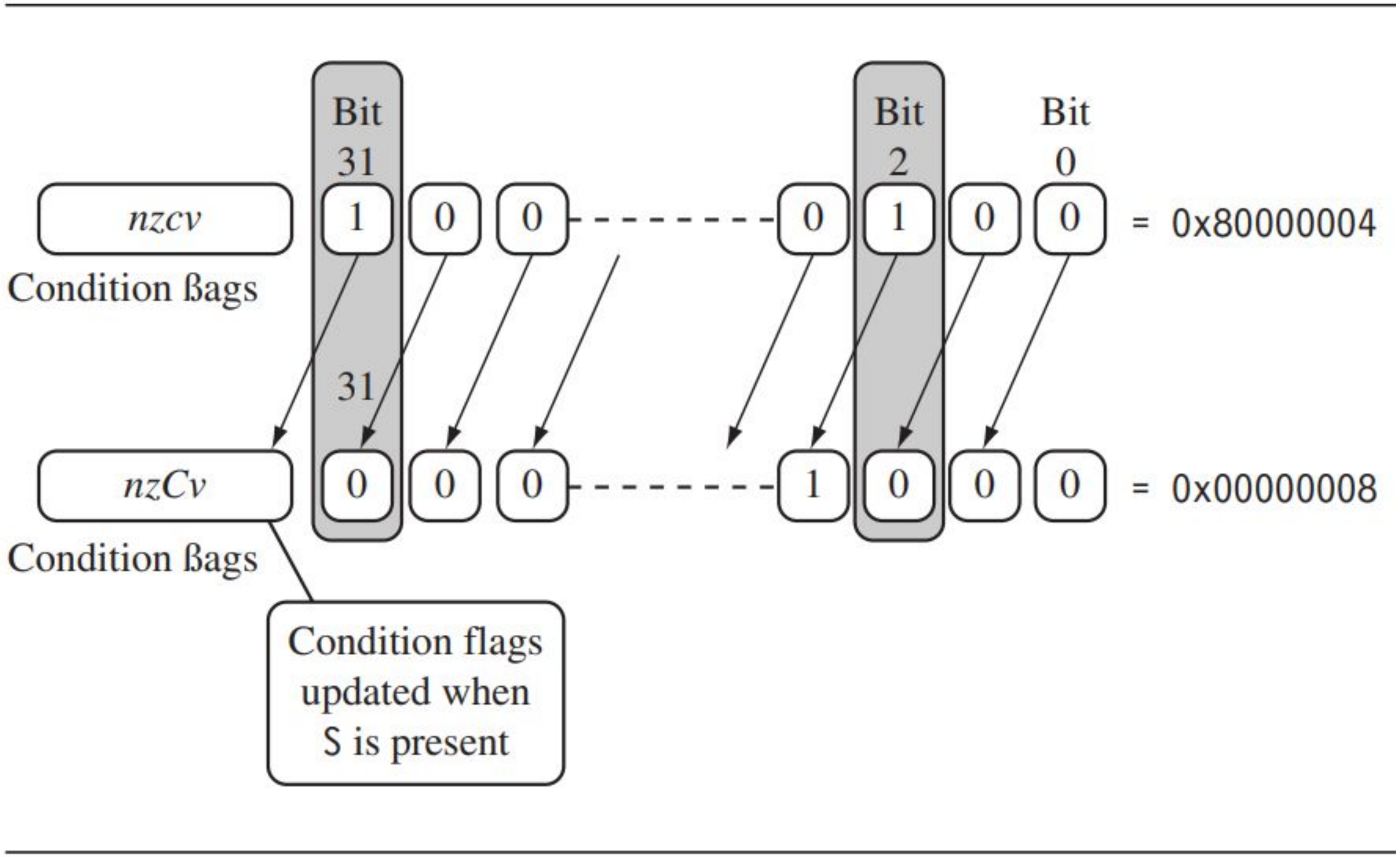
r7 = 20

multiplies register r5 by four and then places the result into register r7



Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	x LSL y	$x \ll y$	#0–31 or Rs
LSR	logical shift right	x LSR y	(unsigned)$x \gg y$	#1–32 or Rs
ASR	arithmetic right shift	x ASR y	(signed)$x \gg y$	#1–32 or Rs
ROR	rotate right	x ROR y	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or Rs
RRX	rotate right extended	x RRX	(c flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

Note: x represents the register being shifted and y represents the shift amount.



Logical shift left by one.

Barrel shift operation syntax for data processing instructions.

N shift operations	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 with 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`

A MOVS instruction shifts register r1 left by one bit.

This multiplies register r1 by a value 2.

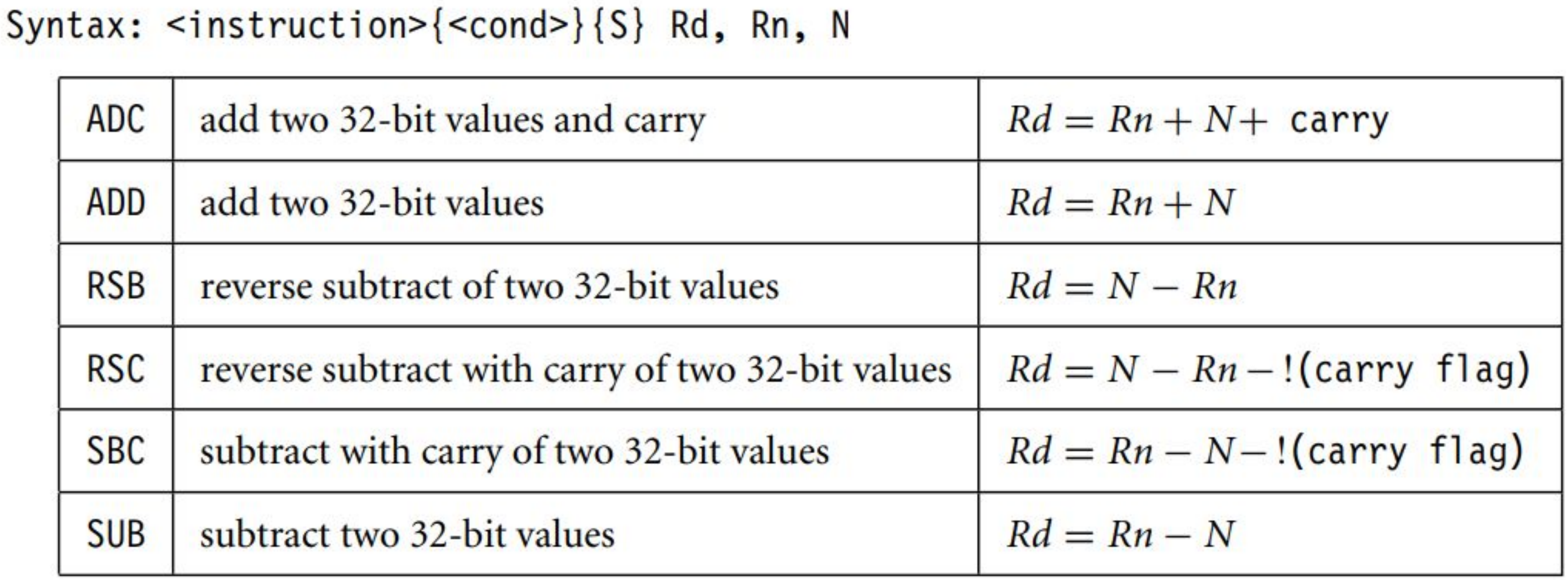
C flag is updated in the CPSR because the S suffix is present in the instruction mnemonic.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

         MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

The second operand N can be an immediate constant preceded by hash, a register value Rm, or the value of Rm processed by a shift.

ARITHMETIC INSTRUCTIONS

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.



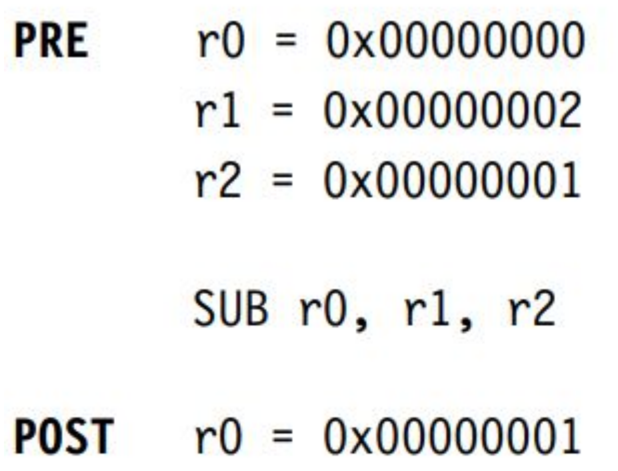
```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and 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 with carry of two 32-bit values	$Rd = N - Rn -$ `!(carry flag)`
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$ `!(carry flag)`
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operations.

Subtraction

Subtracts a value stored in register r2 from a value stored in register r1. The result is stored in register r0.

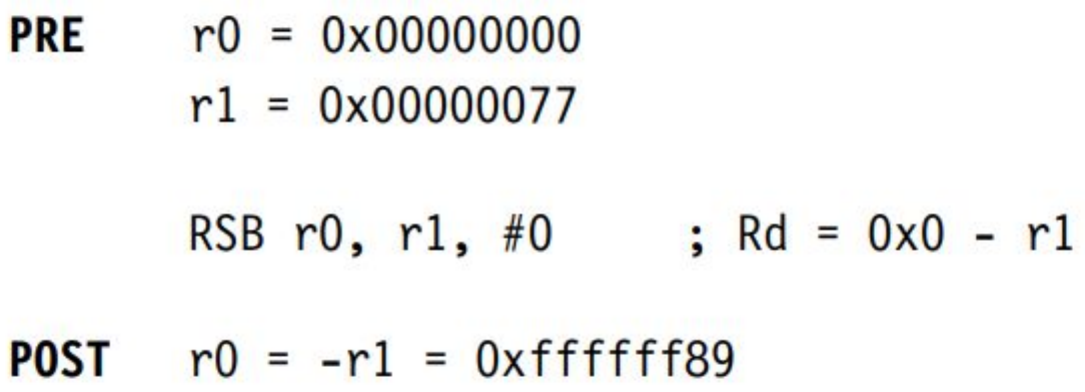


```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

Reverse subtract

Reverse subtract instruction subtracts r1 from the constant value 0, writing the result to r0.

This instruction can be used to negate numbers.



```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

SUBS

The SUBS instruction is useful for decrementing loop counters.

Subtract the immediate value 1 from the value stored in register r1.

The result value zero is written to register r1.

CPSR is updated with the ZC flags set.

```
PRE     cpsr = nzcvqiFt_USER        POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000001                     r1 = 0x00000000

        SUBS r1, r1, #1
```

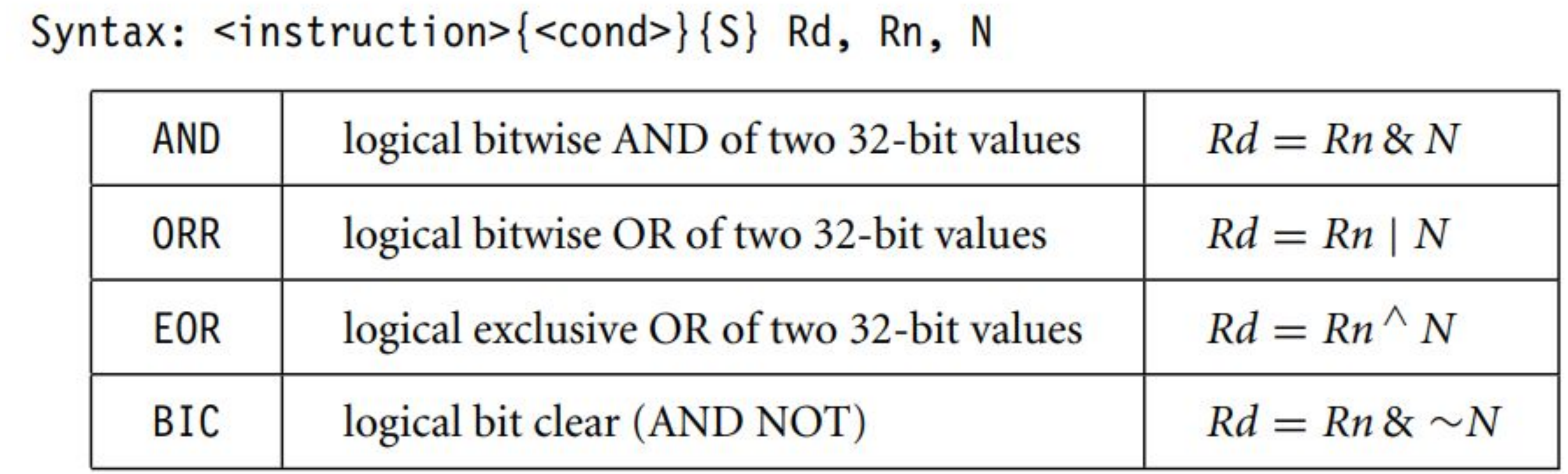
Barrel Shifter with Arithmetic Instructions

- The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set.
- Register r1 is shifted one location to the left to give the value of twice r1.
- ADD instruction, adds the result of the barrel shift operation to register r1.
- Final value in r0 is three times the value in r1

```
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

LOGICAL INSTRUCTIONS

Logical instructions perform bitwise logical operations on the two registers.

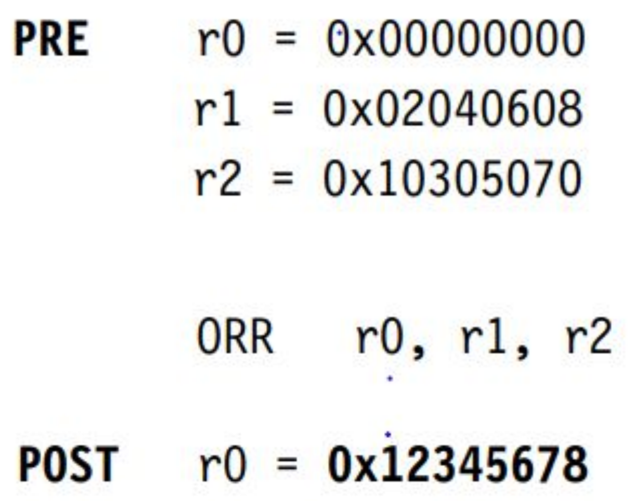


```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

`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	$Rd = Rn \,\hat{}\, N$
`BIC`	logical bit clear (AND NOT)	$Rd = Rn \,\&\, \sim N$

LOGICAL INSTRUCTIONS

Logical instructions perform bitwise logical operations on the two source registers.



```
PRE     r0 = 0x00000000
        r1 = 0x02040608
        r2 = 0x10305070

        ORR   r0, r1, r2

POST    r0 = 0x12345678
```

LOGICAL BIT CLEAR

BIC - logical bit clear

register r2 contains a binary pattern where every binary 1 in r2 clears a corresponding bit location in r1

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC    r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
        Rd = Rn AND NOT(N)
```

LOGICAL BIT CLEAR

BIC is useful when clearing status bits and is frequently used to change interrupt masks in the cpsr.

Logical instructions update the cpsr flags only if S suffix is present.

It can use barrel-shifted second operands in the same way as the arithmetic instructions.

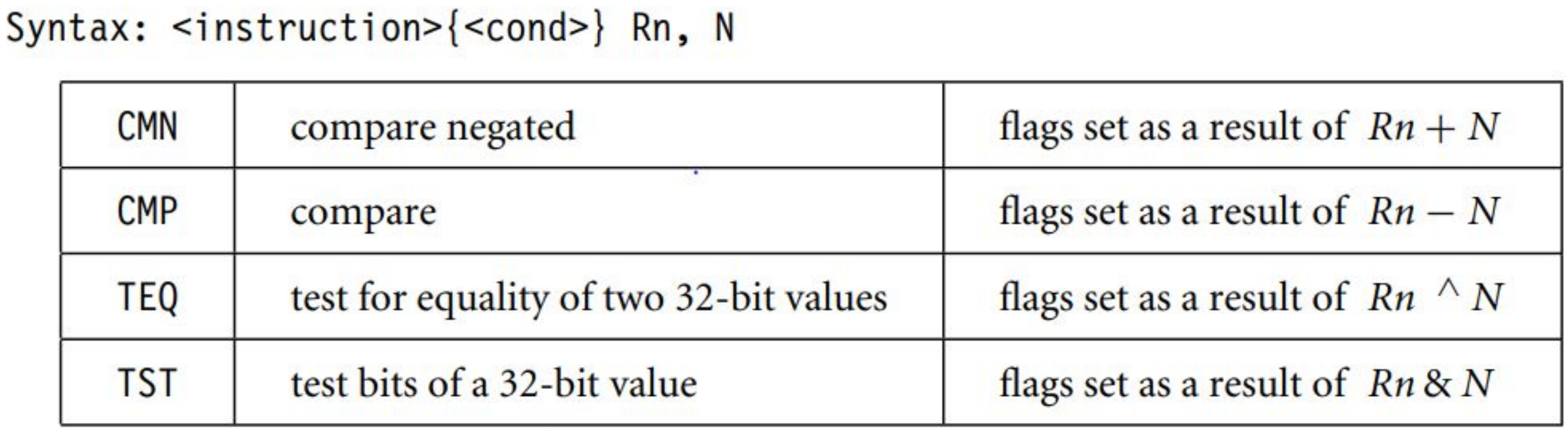
Comparison operations

- Compare or test a register with 32-bit value.
- These instructions do not produce a result but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.
- The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test' (TST) and 'test equal' (TEQ).

```
CMP CMN  r1, r2            ; set cc on r1 - r2
TST TEQ  r1, r2            ; set cc on r1 + r2
         r1, r2            ; set cc on r1 and r2
         r1, r2            ; set cc on r1 xor r2
```

Comparison operations

- do not need to apply the S suffix for comparison instructions to update the flags.
- N is the result of the shifter operations.

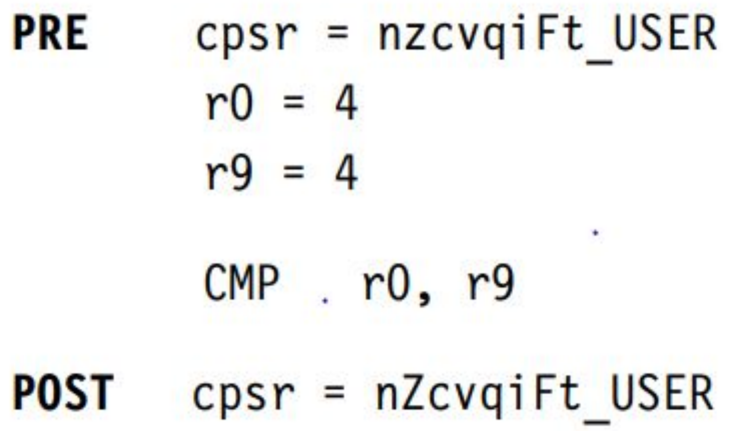


```
Syntax: <instruction>{<cond>} Rn, N
```

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 operations

- a subtract operation with result discarded
- r0 and r9 are equal before executing the instruction
- value of z flag prior to execution is zero, represented by lowercase z
- After execution z flag changes to 1 or an uppercase Z
- This change indicates equality.
- TST, TEQ - logical AND and logical ex-or



```
PRE     cpsr = nzcvqiFt_USER
         r0 = 4
         r9 = 4

         CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

Comparison operations

- Results are discarded for AND and TEQ, condition bits are updated in the cpsr
- Comparison instructions only modify the condition flags of the cpsr and do not affect the registers being compared.
-

Multiply Instructions

Multiplies

MUL r4, r3, r2 ; r4 = r3 * r2

There are some important differences from the other arithmetic instructions:

- Immediate second operands are not supported.
- The result register must not be the same as the first source register.
- If the ' s' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless.
- Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored.
- MLA r4, r3, r2, r1 ; r4 = (r3*r2 + r1)

Multiply Instructions

- multiply the contents of a pair of registers
- accumulates the results in with another register
- long multiplies accumulate onto a pair of registers representing 64-bit value
- final result is placed in a destination register or a pair of registers RdLo and RdHi.
- RdLo - lower 32 bits of the 64 bit result
- RdHi - higher 32 bits of the 64 bit result

Multiply Instructions

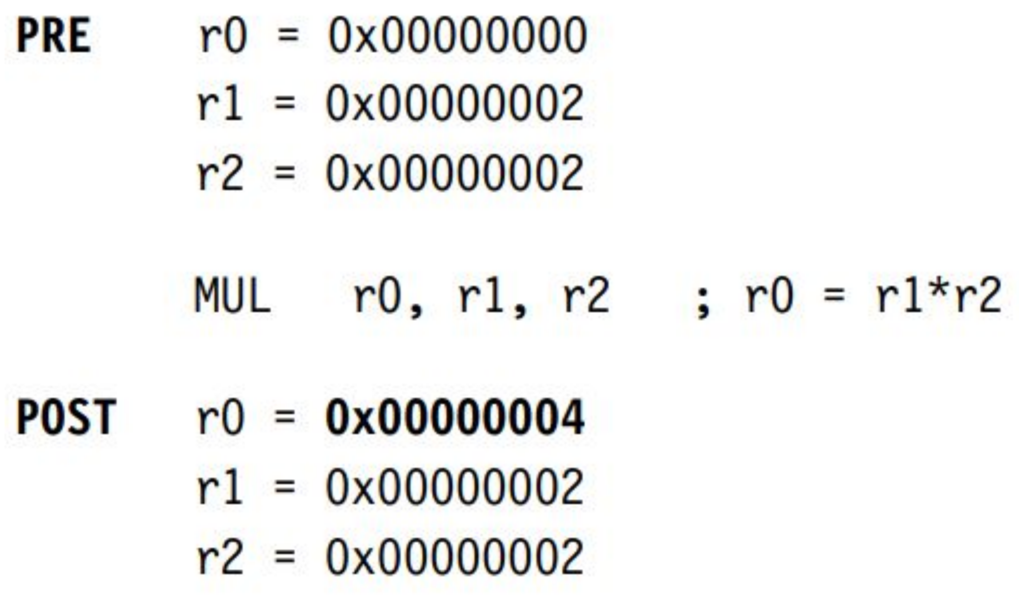
```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^{*} Rs) + Rn$
MUL	multiply	$Rd = Rm^{*} Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

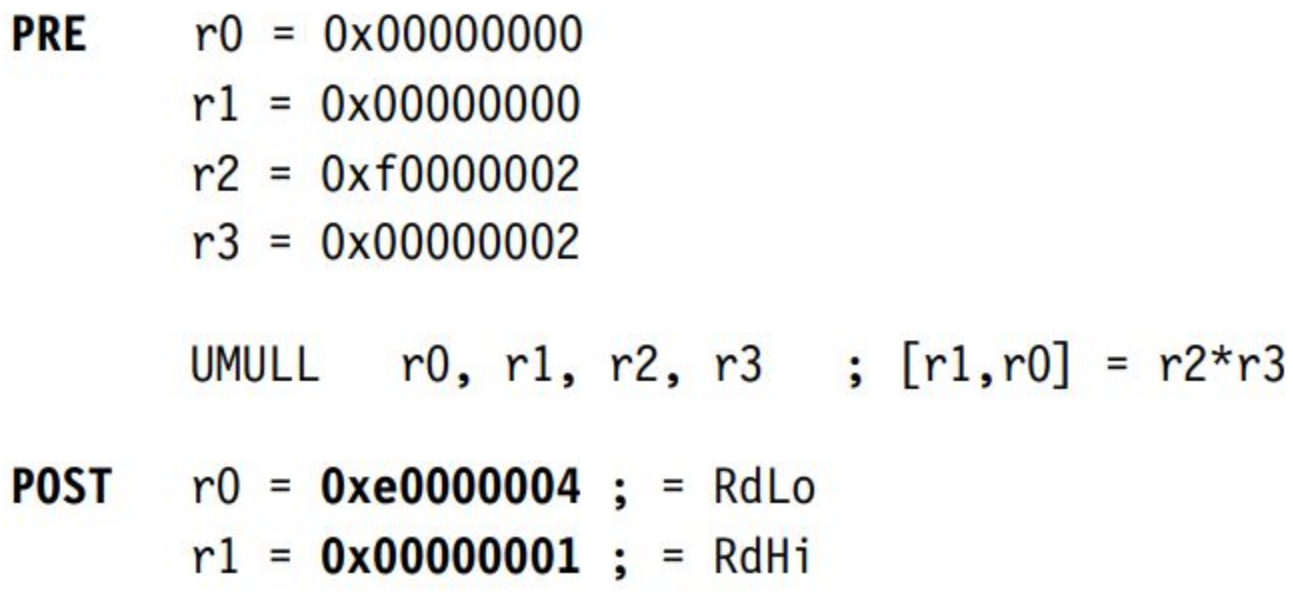
SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^{*} Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm^{*} Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^{*} Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm^{*} Rs$

Multiply Instructions



```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

Multiply Instructions



```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3    ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

BRANCH INSTRUCTIONS

A branch instruction changes the flow of execution or is used to call a routine.

these instructions allows the programs to have subroutines, if-then-else statements, and loops.

The change of execution flow forces the program counter PC to point to a new address.

BRANCH INSTRUCTIONS

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

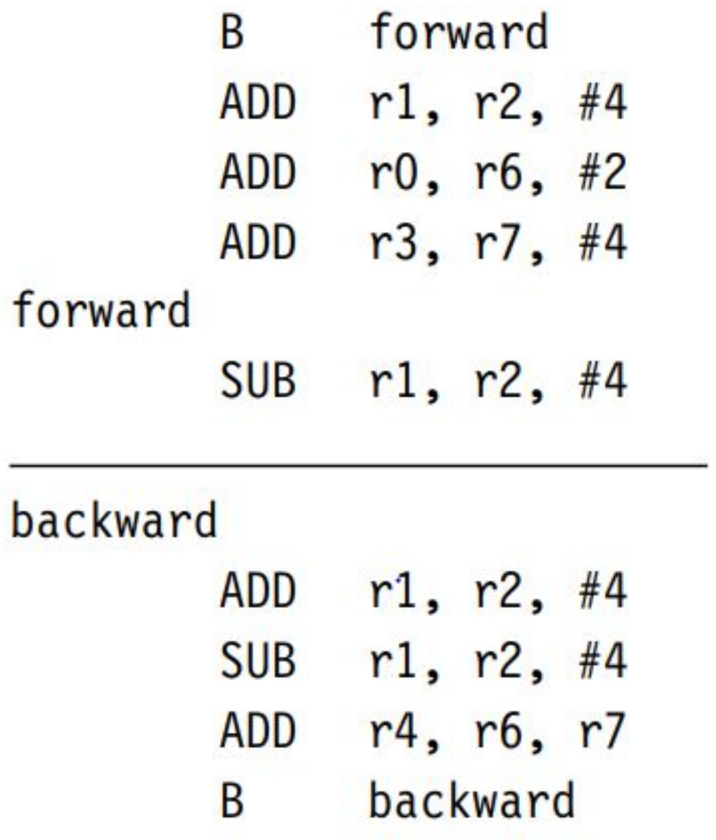
B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

BRANCH INSTRUCTIONS

- The address label is stored in the instruction as a signed pc-relative offset and must be within approximately 32 MB of the branch instruction.
- T refers to the Thumb bit in the cpsr.
- When instructions set T, the ARM switches to Thumb state.

Forward and Backward Jump

- Forward jump skips three instructions and backward jump creates an infinite loop.
- Branches are used to change execution flow



```
        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
```

Branch with Link

- BL is similar to B instruction but overwrites the link register lr with a return address.
- It performs a subroutine call.
- To return from a subroutine, copy the link register to the pc
- The branch exchange(BX) and BX with link(BLX)-BX uses an absolute address stored in register Rm.
- It is primarily used to branch to and from Thumb code.

Branch with Link

- The T bit in the cpsr is updated by the LSB of the branch register.
- BLX instruction updates the T bit of the cpsr with lsb and additionally sets the link register with the return address.

LOAD-STORE INSTRUCTIONS

Load-store instructions transfer data between memory and processor registers.

- Single-register transfer
- Multiple-register transfer
- Swap

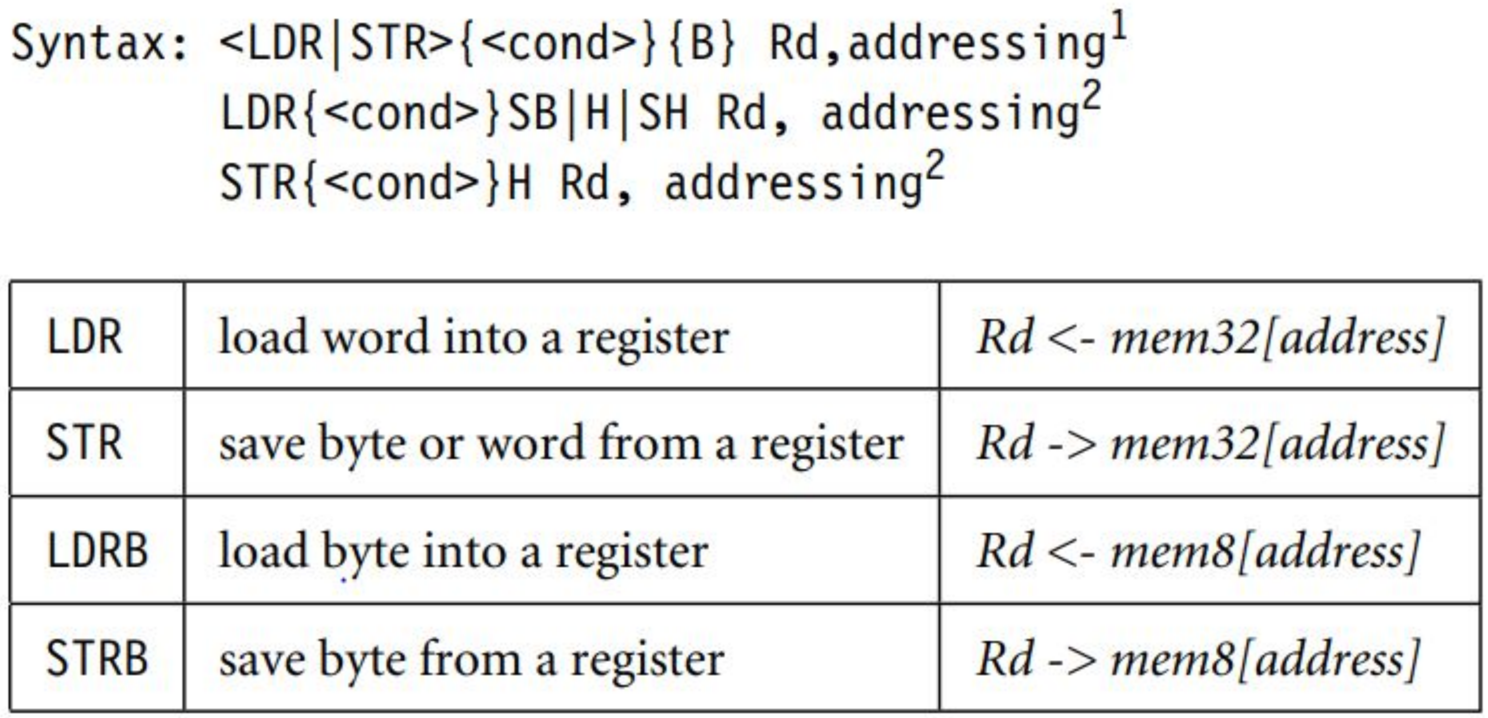
LOAD-STORE INSTRUCTIONS

Single-register transfer

Used for moving a single data item in and out of a register.

Datatypes - signed, unsigned words(32-bit), halfwords(16-bits), and bytes.

LOAD-STORE INSTRUCTIONS

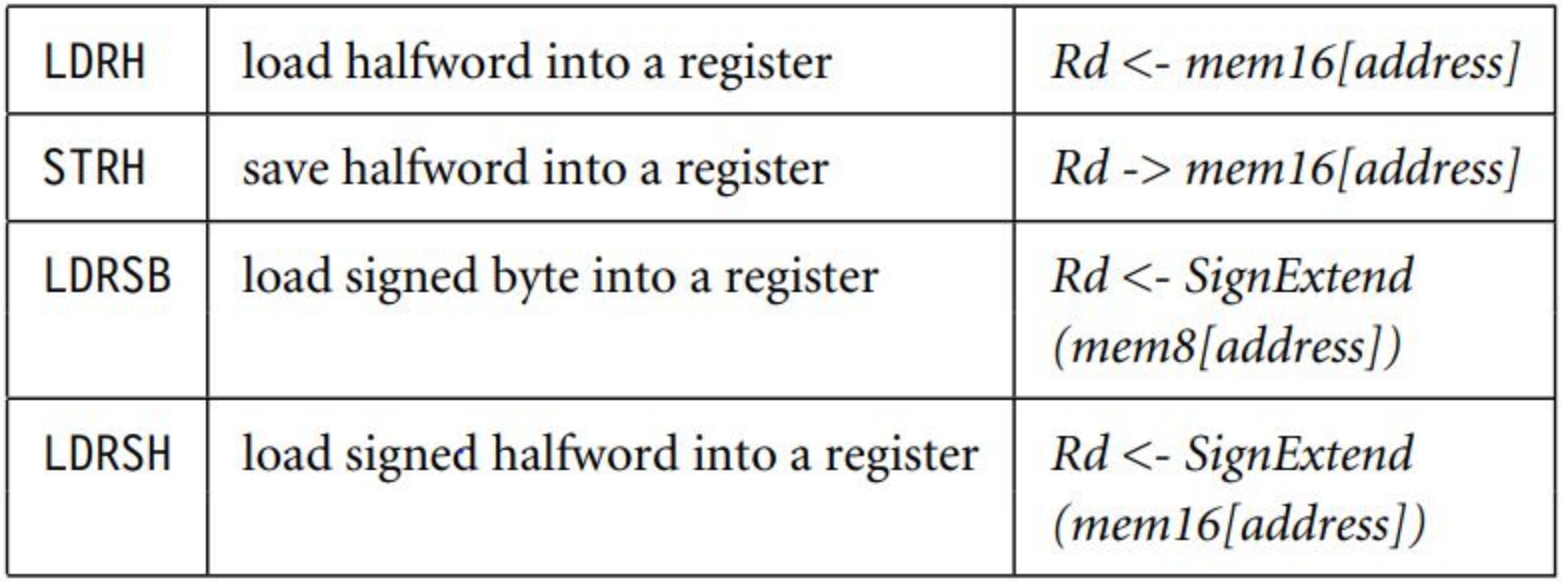


```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LOAD-STORE INSTRUCTIONS

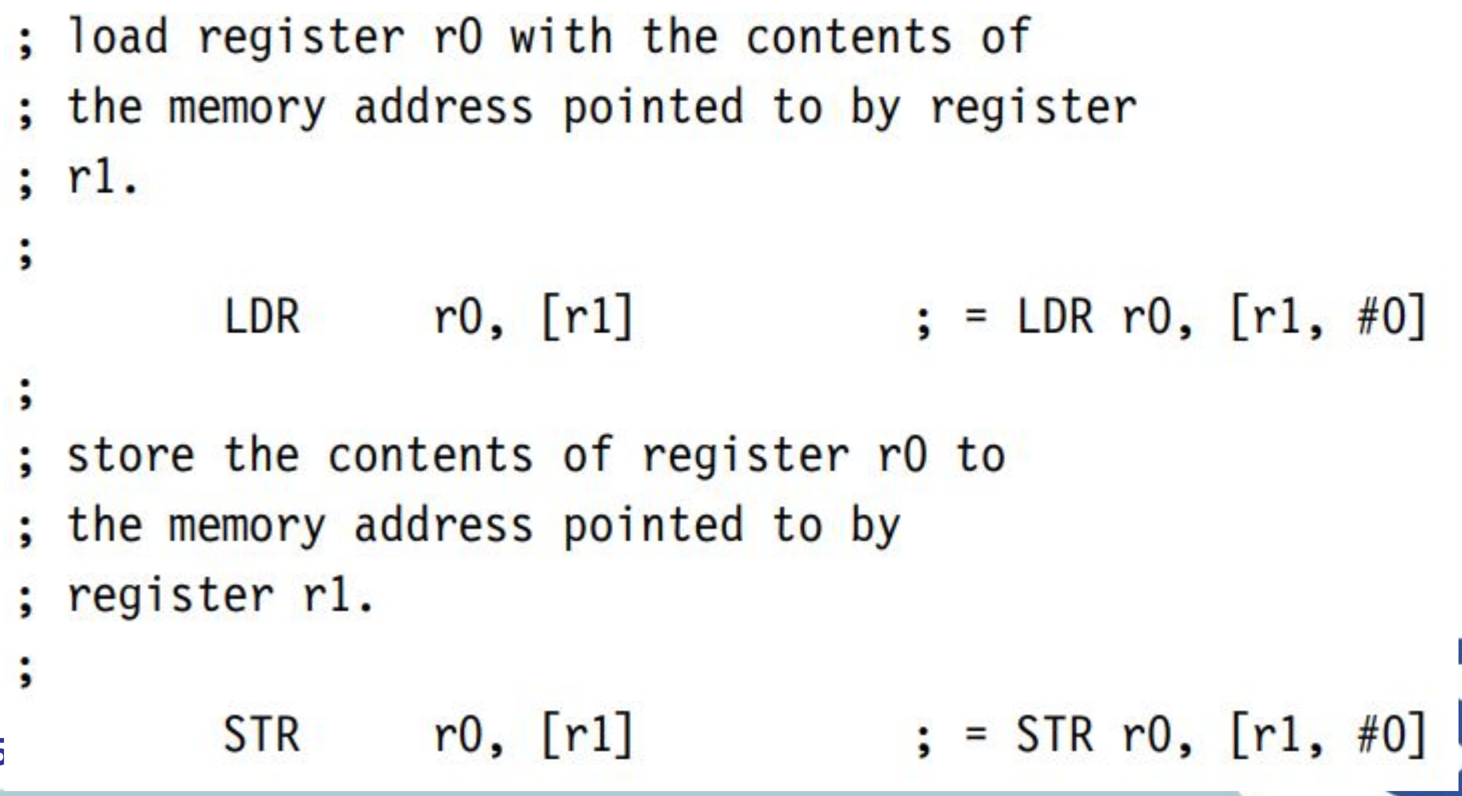
LDR and STR instructions can load and store data on a boundary alignment that is same as the datatype size being loaded or stored.



LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

LOAD-STORE INSTRUCTIONS

- LDR can only load 32-bit words on a memory address that is a multiple of 4 bytes.
- A load from a memory address contained in register r1, followed by a store back to the same address in memory.



```
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]            ; = STR r0, [r1, #0]
```

LOAD-STORE INSTRUCTIONS

The ARM instruction set provides different modes for addressing memory.

Indexing methods:

- preindex with writeback
- preindex
- postindex

Index methods

- Preindex with writeback calculates an address from a base register plus address offset
- Updates the address base register with the new address
- Preindex offset is the same as the preindex with writeback but doesnot update the address base register
- Post index only updates the address base register after the address is used
- Preindex mode is useful for accessing an element in a data structure.
- Post index and preindex with writeback modes are useful for traversing an array.

Index methods

```
PRE        r0 = 0x00000000
           r1 = 0x00090000
           mem32[0x00009000] = 0x0101010
           mem32[0x00009004] = 0x0202020

           LDR      r0, [r1, #4]!
```

Preindexing with writeback:

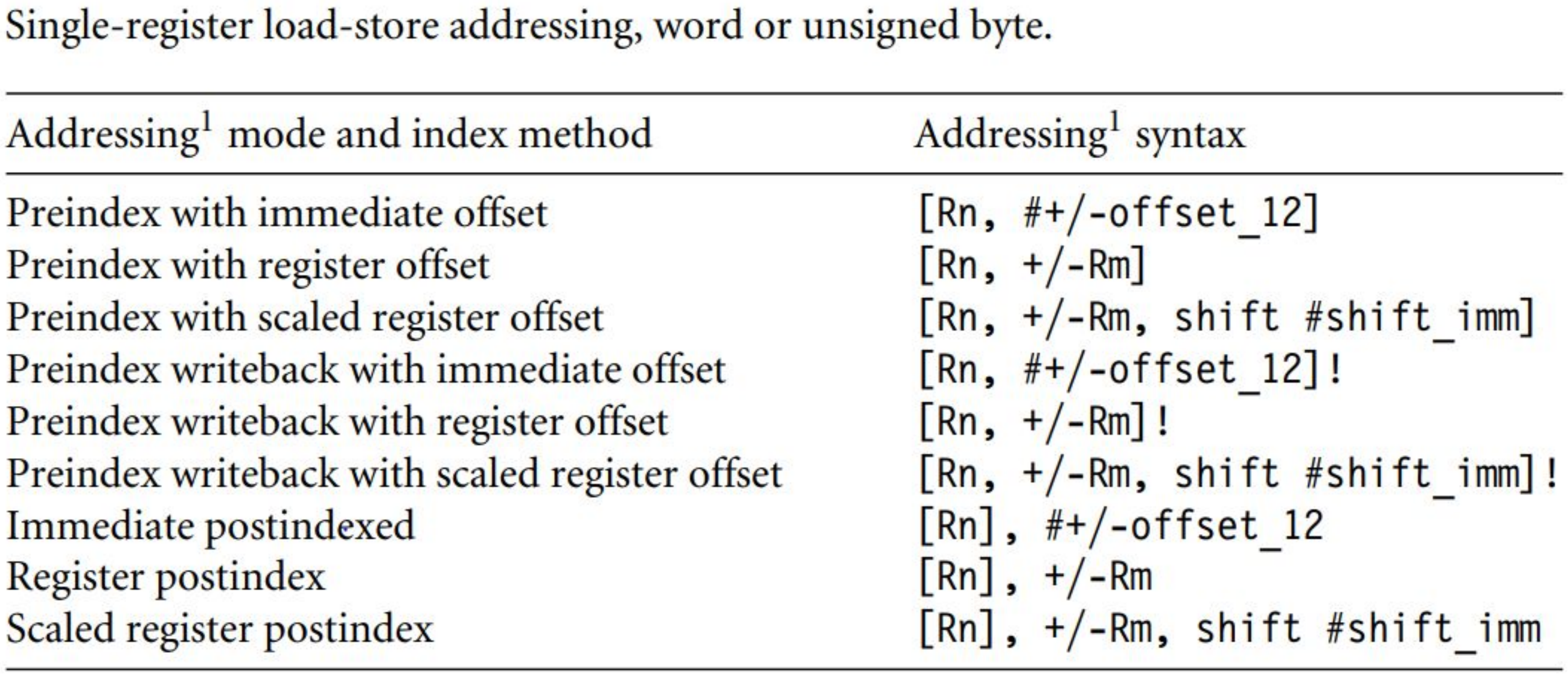
```
POST(1)    r0 = 0x02020202
           r1 = 0x00009004

           LDR      r0, [r1, #4]
```

Preindexing:

```
POST(2)    r0 = 0x02020202
           r1 = 0x00009000

           LDR      r0, [r1], #4
```

Postindexing:

```
POST(3)    r0 = 0x01010101
           r1 = 0x00009004
```



Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

- how each indexing method effects the address held in register r1 as well as the data loaded into r0.
- each instruction shows the result of the index method with the same pre-condition.

- A signed offset or register is denoted by “+/-”, identifying that it is either a positive or negative offset from the base address register Rn.
- The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.

Swap Instruction

- The swap instruction is a special case of a load store instruction.
- It swaps the contents of memory with the contents of a register.
- This is an atomic operation.
- It reads and writes a location in the same bus operation, preventing any other instruction from reading and writing to that location until it completes.
- Swap cannot be interrupted by any other instruction or any other bus access.

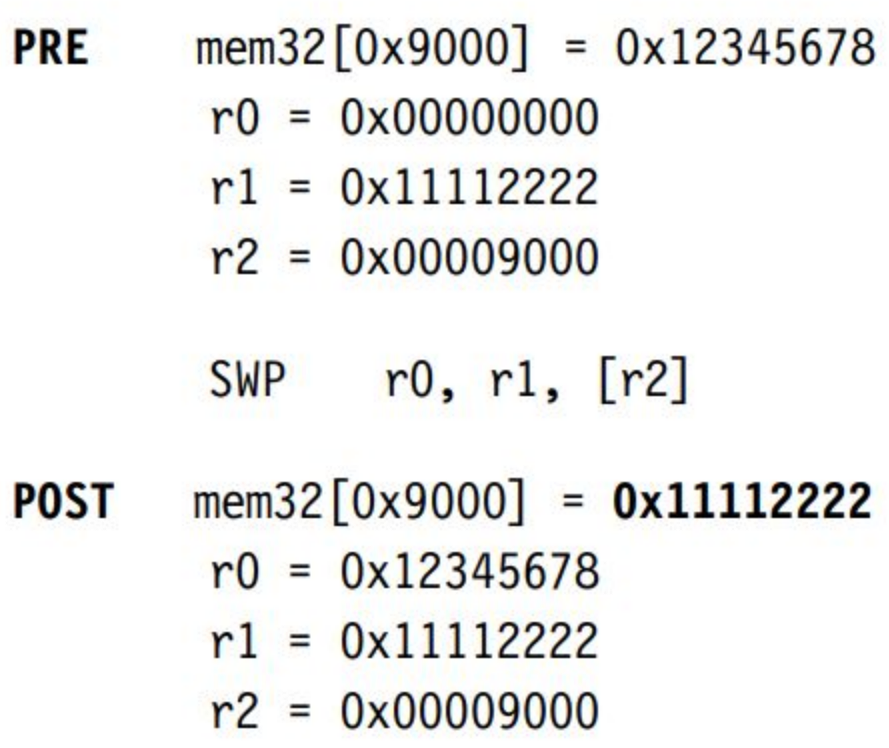
Swap Instruction

```
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	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

- loads a word from memory into register r0, and overwrites the memory with register r1
- useful while implementing semaphores and mutual exclusion in an os.

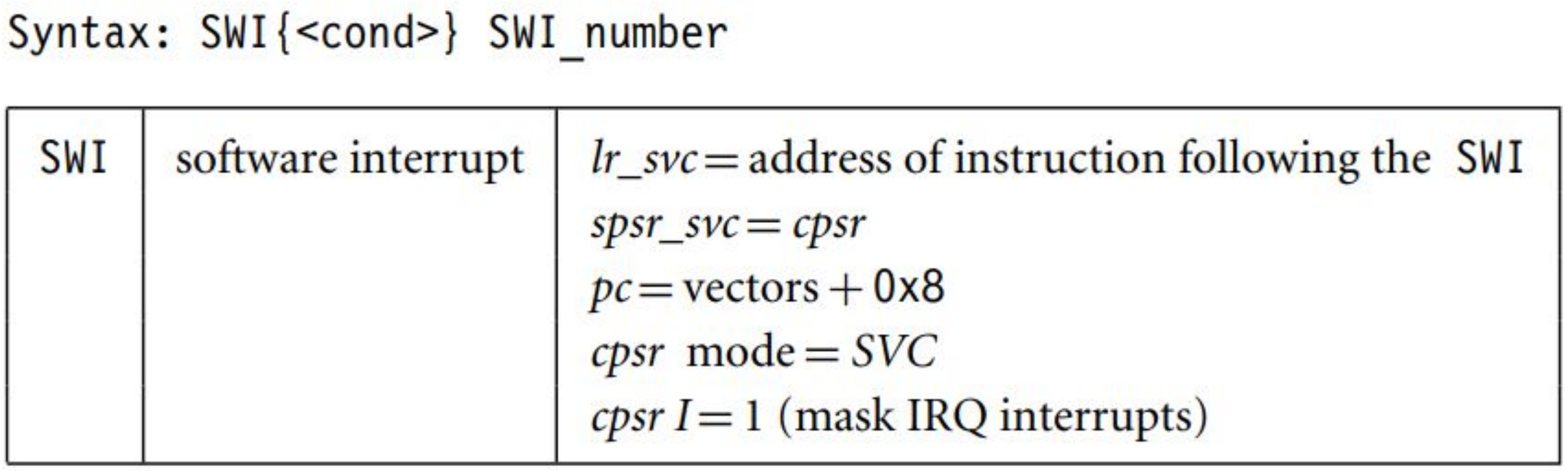
Swap



```
PRE     mem32[0x9000] = 0x12345678
         r0 = 0x00000000
         r1 = 0x11112222
         r2 = 0x00009000

         SWP    r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
         r0 = 0x12345678
         r1 = 0x11112222
         r2 = 0x00009000
```

Software interrupt instruction

- A software interrupt instruction causes a software interrupt execution.
- It provides a mechanism for applications to call Operating System routines.



```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	lr_svc = address of instruction following the SWI $spsr\_svc = cpsr$ pc = vectors + 0x8 $cpsr$ mode = SVC $cpsr\ I = 1$ (mask IRQ interrupts)

SWI Instruction

- When the processor executes a SWI instruction, it sets the PC to the offset 0x8 in the vector table.
- The instruction also forces the processor mode to SVC, which allows an os routine to be called in a privileged mode.
- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Program Status Register Instructions

MRS - transfers the contents of either the *cpsr or spsr* into a register

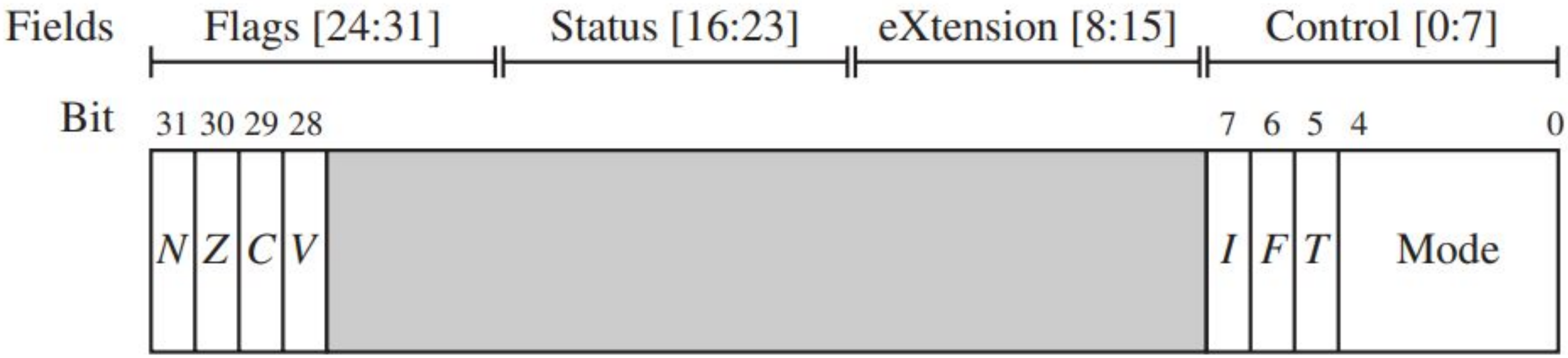
MSR - transfers the contents of a register into the *cpsr or spsr*

Read and write of cpsr or spsr - through MRS, MSR

It affect the fields (control, extension, status, flags).

Program Status Register Instructions

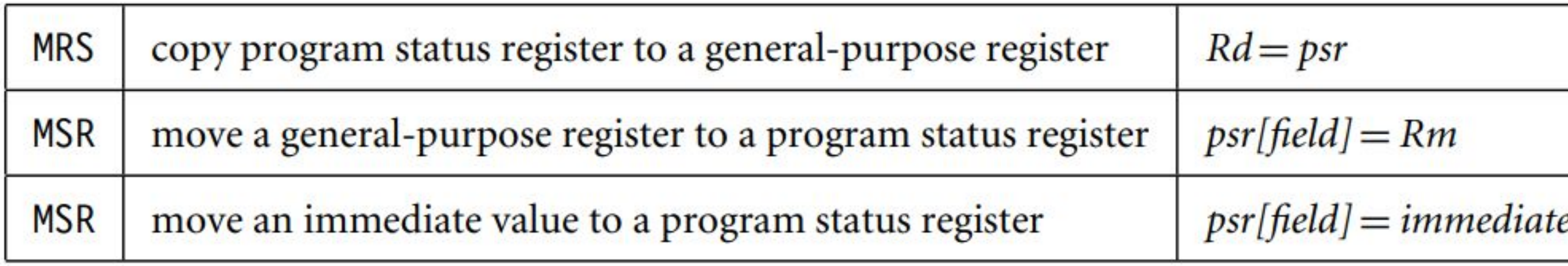
```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```



psr byte fields.

Program Status Register Instructions

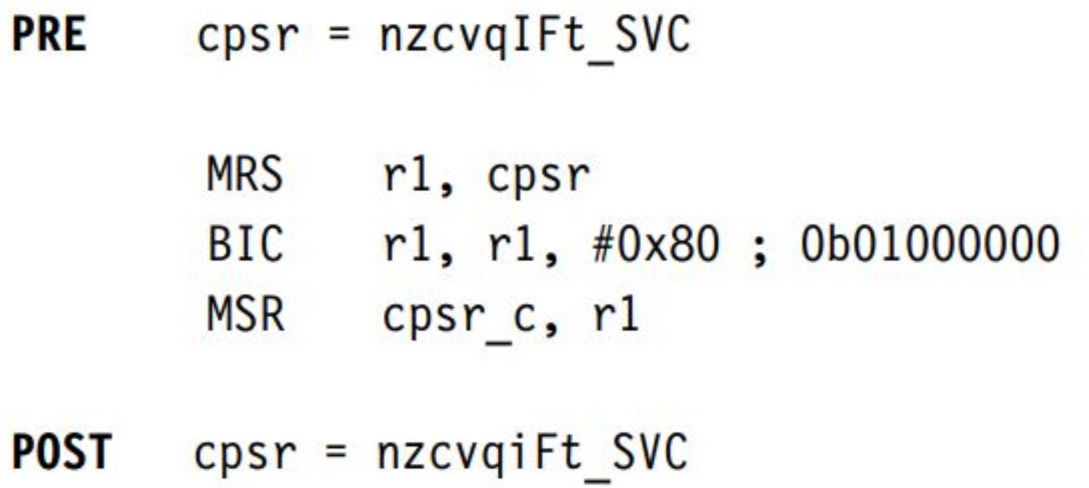
c field controls the interrupt masks, Thumb state, Processor mode.



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

Program Status Register Instructions

- enable IRQ interrupts by clearing the I mask
- use both MRS and MSR instructions to read from and write to the cpsr
- MSR - copy cpsr into r1
- BIC - clears bit 7 of r1
- r1 is copied back to cpsr



```
PRE     cpsr = nzcvqIFt_SVC

        MRS    r1, cpsr
        BIC    r1, r1, #0x80 ; 0b01000000
        MSR    cpsr_c, r1

POST    cpsr = nzcvqiFt_SVC
```

CONDITIONAL EXECUTION

- Most ARM instructions are conditionally executed
- Instructions only executes if the condition code flags pass a given condition or test
- Performance and code density can increased by using conditional execution instructions
- condition field - 2 letter mnemonic - appended to the instruction mnemonic
- default mnemonic is AL - always execute

CONDITIONAL EXECUTION

Conditional execution

- reduces the number of branches
- reduces the number of pipeline flushes
- improves the performance of the executed code

CE depends upon 2 components

- condition fields - located in the instruction
- condition flags - located in the cpsr

CONDITIONAL EXECUTION

- ADD instruction with condition EQ appended
- It is only executed when zero flag in the cpsr is set to 1

r0 = r1 + r2 if zero flag is set

```
ADDEQ r0, r1, r2
```

- only comparison instructions and data processing instructions with the S suffix appended to the mnemonic update the condition flags in the cpsr

CONDITIONAL EXECUTION

Advantage of conditional execution

c code

```
while (a!=b)
{
  if (a>b) a -= b; else b -= a;
}
```

ARM assembler code

```
            ; Greatest Common Divisor Algorithm
gcd
            CMP     r1, r2
            BEQ     complete
            BLT     lessthan
            SUB     r1, r1, r2
            B       gcd

lessthan
            SUB     r2, r2, r1
            B       gcd

complete
...
```

with conditional execution

```
gcd
        CMP r1, r2
        SUBGT r1, r1, r2
        SUBLT r2, r2, r1
        BNE gcd
```

ARM instructions can be conditionally executed.

It reduces the number of instructions required to perform a specific algorithm.