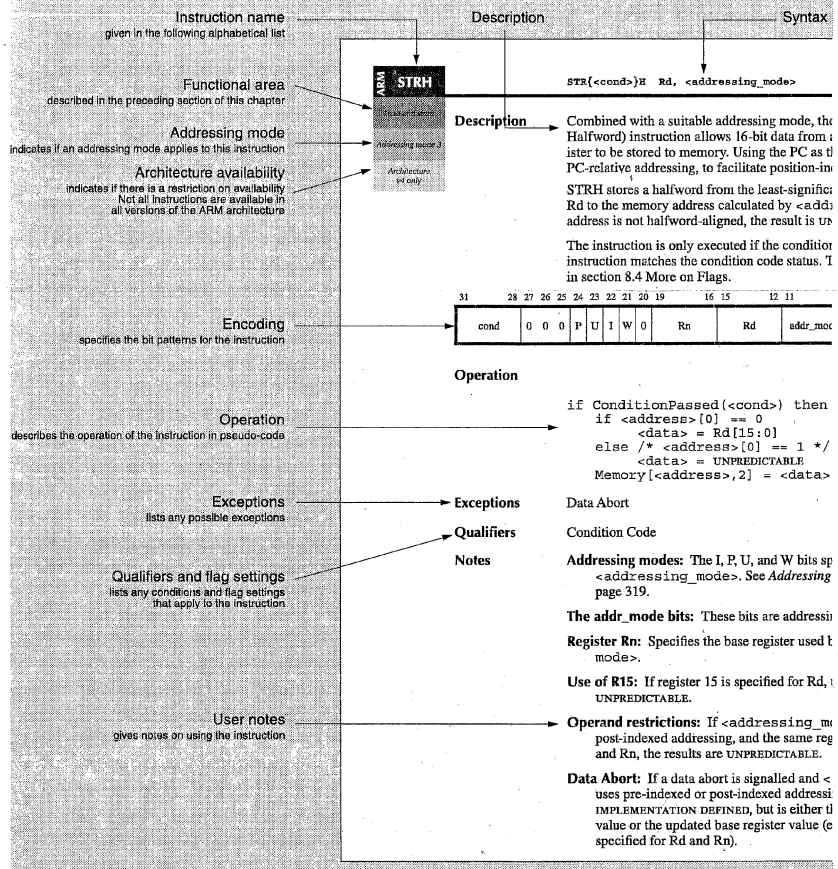
APPENDIX A

The ARM V4T Instruction Set



ADC{<cond>}{S} Rd, Rn, <shifter_operand>

Description

The ADC (Add with Carry) instruction adds the value of <shifter_operand> and the value of the Carry flag to the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

ADC is used to synthesize multi-word addition. If register pairs R0,R1 and R2,R3 hold 64-bit values (where R0 and R2 hold the least-significant words), the following instructions leave the 64-bit sum in R4,R5:

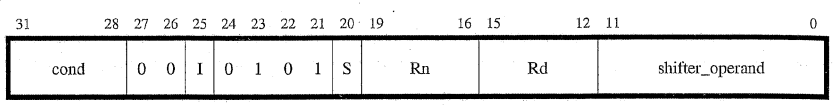
```
ADDS        R4,R0,R2
ADC         R5,R1,R3
```

The instruction:

```
ADCS R0,R0,R0
```

produces a single-bit Rotate Left with Extend operation (33-bit rotate though the Carry flag) on R0.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn + <shifter_operand> + C Flag
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + <shifter_operand> + C Flag)
        V Flag = OverflowFrom (Rn + <shifter_operand> + C Flag)
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

ADD{<cond>}{S} Rd, Rn, <shifter_operand>

Description

The ADD instruction adds the value of <shifter_operand> to the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

ADD is used to add two values together to produce a third.

To increment a register value (in Rx), use:

```
ADD Rx, Rx, #1
```

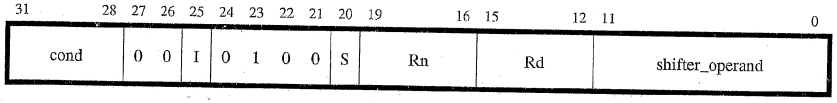
Constant multiplication (of Rx) by 2^n+1 (into Rd) can be performed with:

```
ADD Rd, Rx, Rx, LSL #n
```

To form a PC-relative address, use:

```
ADD Rs, PC, #offset
```

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn + <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + <shifter_operand>)
        V Flag = OverflowFrom (Rn + <shifter_operand>)
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

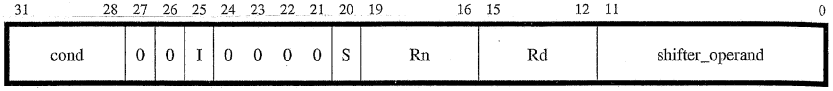
```
AND{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The AND instruction performs a bitwise AND of the value of register Rn with the value of `<shifter_operand>`, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

AND is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1's in the field to be extracted, and 0's elsewhere.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn AND <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.


```
B{L}{<cond>}  <target address>
```

Description

The B (Branch) and BL (Branch and Link) instructions provide both conditional and unconditional changes to program flow. The Branch and Link instruction is used to perform a subroutine call; the return from subroutine is achieved by copying the LR to the PC.

B and BL cause a branch to a target address. The branch target address is calculated by:

1 Shifting the 24-bit signed (two's complement) offset left two bits

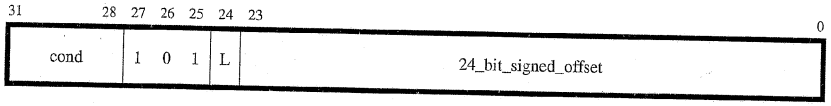
2 Sign-extending the result to 32 bits

3 Adding this to the contents of the PC (which contains the address of the branch instruction plus 8)

The instruction can therefore specify a branch of +/– 32 MB.

In the BL variant of the instruction, the L (link) bit is set, and the address of the instruction following the branch is copied into the link register (R14).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend(<24_bit_signed_offset>) << 2)
```

Exceptions None

Qualifiers Condition Code

L (Link) stores a return address in the LR (R14) register

Notes

Offset calculation: An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus eight (because the PC holds the address of the current instruction plus eight).

Memory bounds: Branching backward past location zero and forward over the end of the 32-bit address space is UNPREDICTABLE.

ARM BIC

Data processing

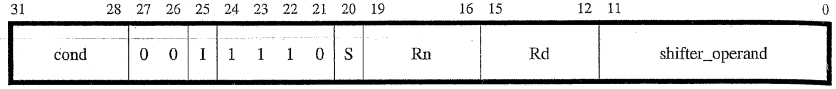
```
BIC{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The BIC (Bit Clear) instruction performs a bitwise AND of the value of register Rn with the complement of the value of <shifter_operand>, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

BIC can be used to clear selected bits in a register; for each bit, BIC with 1 will clear the bit, BIC with 0 will leave it unchanged.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn AND NOT <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

A

BX ARM

Branch

Architecture v4 only

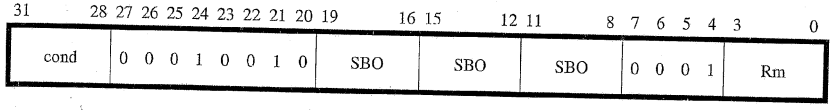
```
BX{<cond>}  Rm
```

Description

The BX (Branch and Exchange) instructions set is UNDEFINED on ARM Architecture Version 4. On ARM Architecture Version 4T, this instruction branches and selects the instruction set decoder to use to decode the instructions at the branch destination. The branch target address is the value of register Rm. The T flag is updated with bit 0 of the value of register Rm.

BX is used to branch between ARM code and THUMB code. On ARM Architecture 4, it causes an UNDEFINED instruction exception to allow the THUMB instruction set to be emulated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    T Flag = Rm[0]
    PC = Rm[31:1] << 1
```

Exceptions None

Operation Condition Code

Notes

Transferring to THUMB: When transferring to the THUMB instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC.

Transferring to ARM: When transferring to the ARM instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC. If bit[1] of Rm is set, the result is UNPREDICTABLE.

A

ARM CDP

Not in architecture v3

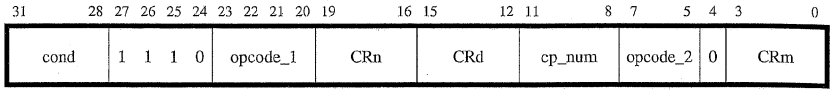
```
CDP{<cond>}  p<cp#>, <opcode_1>, CRd, CRn, CRm, <opcode_2>
```

Description

The CDP (Coprocessor Data Processing) instruction tells the coprocessor specified by `<cp#>` to perform an operation that is independent of ARM registers and memory. If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

CDP is used to initiate coprocessor instructions that do not operate on values in ARM registers or in main memory; for example, a floating-point multiply instruction for a floating-point accelerator coprocessor.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Coprocessor[<cp_num>] dependent operation
```

Exceptions Undefined Instruction

Qualifiers Condition Code

Notes **Coprocessor fields:** Only instruction bits[31:24], bits[11:8], and bit[4] are ARM architecturally defined. The remaining fields are recommendations for compatibility with ARM Development Systems.

A

CMN ARM

Data processing

Addressing mode 1

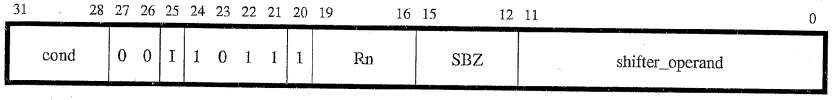
```
CMN{<cond>}  Rn, <shifter_operand>
```

Description

The CMN (Compare Negative) instruction compares an arithmetic value and the negative of an arithmetic value (an immediate or the value of a register) and sets the condition code flags so that subsequent instructions can be conditionally executed.

CMN performs a comparison by adding (or subtracting the negative of) the value of `<shifter_operand>` to (from) the value of register Rn, and updates the condition code flags (based on the result). The comparison is the subtraction of the negative of the second operand from the first operand (which is the same as adding the two operands).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <alu_out> = Rn + <shifter_operand>
    N Flag = <alu_out>[31]
    Z Flag = if <alu_out> == 0 then 1 else 0
    C Flag = CarryFrom(Rn + <shifter_operand>)
    V Flag = OverflowFrom (Rn + <shifter_operand>)
```

Exceptions None

Qualifiers Condition Code

Notes **Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

A

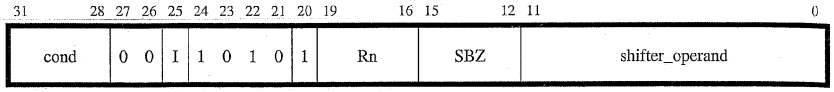
```
CMP{<cond>}  Rn, <shifter_operand>
```

Description

The CMP (Compare) instruction compares two arithmetic values and sets the condition code flags so that subsequent instructions can be conditionally executed. The comparison is a subtraction of the second operand from the first operand.

CMP performs a comparison by subtracting the value of `<shifter_operand>` from the value of register Rn and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <alu_out> = Rn - <shifter_operand>
    N Flag = <alu_out>[31]
    Z Flag = if <alu_out> == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - <shifter_operand>)
    V Flag = OverflowFrom (Rn - <shifter_operand>)
```

Exceptions None

Qualifiers Condition Code

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

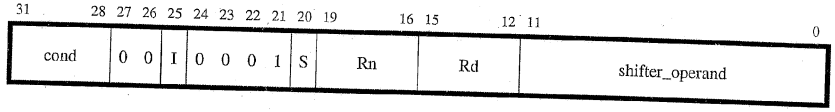

```
EOR{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The EOR (Exclusive OR) instruction performs a bitwise Exclusive OR of the value of register Rn with the value of `<shifter_operand>`, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

EOR can be used to invert selected bits in a register; for each bit, EOR with 1 will invert that bit, and EOR with 0 will leave it unchanged.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn EOR <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

ARM LDC

Addressing mode 5

Not in architecture v1

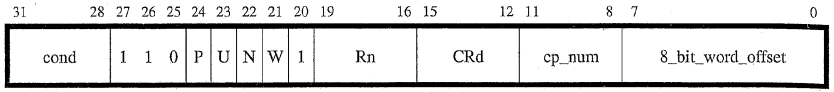
LDC{<cond>} p<cp_num>, CRd, <addressing_mode>

Description

The LDC (Load Coprocessor) instruction is useful to load coprocessor data from memory. The N bit could be used to distinguish between a single- and double-precision transfer for a floating-point load instruction.

LDC loads memory data from the sequence of consecutive memory addresses calculated by <addressing_mode> to the coprocessor specified by <cp_num>. If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
          Coprocessor[<cp_num>] = Memory[<address>,4]
          <address> = <address> + 4
    assert <address> == <end_address>
```

Exceptions

Undefined Instruction; Data Abort

Qualifiers

Condition Code

Notes

Addressing mode: The P, U, and W bits specify the type of <addressing_mode>. See *Addressing Mode 5* starting on page 329.

The N bit: This bit is coprocessor-dependent. It can be used to distinguish between two sizes of data to transfer.

Register Rn: Specifies the base register used by <addressing_mode>.

Coprocessor fields: Only instruction bits[31:23], bits[21:16], and bits[11:0] are ARM architecturally defined. The remaining fields (bit[22] and bits[15:12]) are recommendations for compatibility with ARM Development Systems.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

Non-word-aligned addresses: Load coprocessor register instructions ignore the least-significant two bits of <address> (the words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

A

LDM (1)

Addressing mode 4

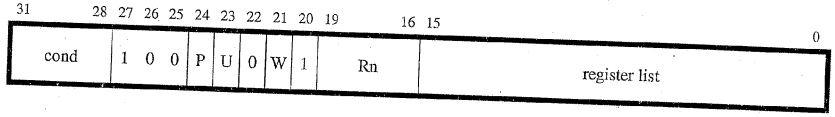
LDM{<cond>}<addressing_mode> Rn{!}, <registers>

Description

This form of the LDM (Load Multiple) instruction is useful as a block load instruction (combined with store multiple it allows efficient block copy) and for stack operations, including procedure exit, to restore saved registers, load the PC with the return address, and update the stack pointer.

In this case, LDM loads a non-empty subset (or possibly all) of the general-purpose registers from sequential memory locations. The registers are loaded in sequence, the lowest-numbered register first, from the lowest memory address (<start_addr>); the highest-numbered register last, from the highest memory address (<end_addr>). 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_addr>
    for i = 0 to 15
        if <register_list>[i] == 1
            Ri = Memory[<address>,4]
            <address> = <address> + 4
    assert <end_add> == <address> - 4
```

Exceptions

Data Abort

Qualifiers

Condition Code

! sets the W bit, causing base register update

Notes

Addressing mode: The P, U, and W bits specify the type of <addressing_mode>. See *Addressing Mode 4* starting on page 322.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: Using R15 as the base register Rn gives an UNPREDICTABLE result.

Operand restrictions: If the base register Rn is specified in <register_list>, and writeback is specified, the final value of Rn is UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> specifies writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if Rn is specified in <register_list>). If register 15 is specified in <register_list>, it must not be overwritten if a data abort occurs.

Non-word-aligned addresses: Load multiple instructions ignore the least-significant two bits of <address> (the words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

A

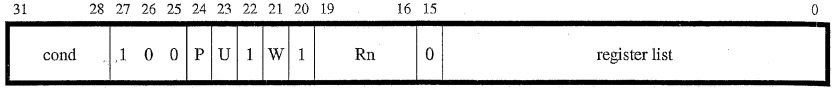
`LDM{<cond>}<addressing_mode>  Rn, <registers>^`

Description

This form of the LDM (Load Multiple) instruction loads user mode registers when the processor is in a privileged mode (useful when performing process swaps).

In this case, LDM instruction loads a non-empty subset (or possibly all except the PC) of the User mode general-purpose registers (which are also the system mode general-purpose registers) from sequential memory locations. The registers are loaded in sequence, the lowest-numbered register first, from the lowest memory address (`<start_address>`); the highest-numbered register last, from the highest memory address (`<end_address>`).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_address>
    for i = 0 to 14
          if <register_list>[i] == 1
                Ri_usr = Memory[<address>,4]
                <address> = <address> + 4
    assert <end_address> == <address> - 4
```

Exceptions

Data Abort

Qualifiers

Condition Code

Notes

Addressing mode: The P and U bits specify the type of `<addressing_mode>`. See *Addressing Mode 4* starting on page 322.

Banked registers: LDM must not be followed by an instruction which accesses banked registers (a following NOP is a good way to ensure this).

Writeback: Setting bit 21 (the W bit) has UNPREDICTABLE results.

User and System mode: LDM is UNPREDICTABLE in User mode or System mode.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified as the base register Rn, the result is UNPREDICTABLE.

Base register mode: The base register is read from the current processor mode registers, not the User mode registers.

Data Abort: If a data abort is signalled, the value left in Rn is the original base register value.

Non-word-aligned addresses: LDM instructions ignore the least-significant two bits of `<address>` (words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

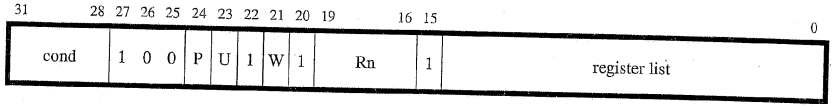
`LDM{<cond>}<addressing_mode>  Rn{!}, <registers_and_pc>^`

Description

This form of the LDM (Load Multiple) instruction is useful for returning from an exception, to restore saved registers, load the PC with the return address, update the stack pointer, and restore the CPSR from the SPSR.

In this case, LDM loads a non-empty subset (or possibly all) of the general-purpose registers and the PC from sequential memory locations. The registers are loaded in sequence, the lowest-numbered register first, from the lowest memory address; the highest-numbered register last, from the highest memory address. The SPSR of the current mode is copied to the CPSR.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_address>
    for i = 0 to 15
          if <register_list>[i] == 1
                Ri = Memory[<address>,4]
                <address> = <address> + 4
    assert <end_address> == <address> - 4
    CPSR = SPSR
```

Exceptions

Data Abort

Qualifiers

Condition Code

! sets the W bit, causing base register update

Notes

Addressing mode: The P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 4* starting on page 322.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: Using R15 as the base register Rn gives an UNPREDICTABLE result.

User and System mode: This instruction is UNPREDICTABLE in User or System mode.

Operand restrictions: If the base register Rn is specified in `<register_list>`, and writeback is specified, the final value of Rn is UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` specifies writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if Rn is specified in `<register_list>`). If register 15 is specified in `<register_list>`, it must not be overwritten if a data abort occurs.

Non-word-aligned addresses: Load multiple instructions ignore the least-significant two bits of `<address>` (the words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

`LDR{<cond>} Rd, <addressing_mode>`

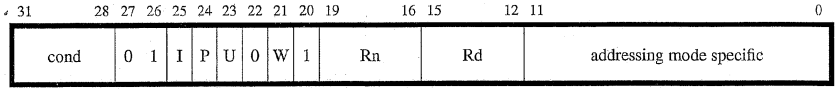
Description

Combined with a suitable addressing mode, the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address (precede the LDR instruction with `MOV LR, PC` to synthesize a branch and link).

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDR loads a word from the memory address calculated by `<addressing_mode>` and writes it to register Rd. If the address is not word-aligned, the loaded data is rotated so that the addressed byte occupies the least-significant byte of the register. If the PC is specified as register Rd, the instruction loads a branch to the address (data) into the PC.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if <address>[1:0] == 0b00
       Rd = Memory[<address>,4]
    else if <address>[1:0] == 0b01
       Rd = Memory[<address>,4] Rotate_Right 8
    else if <address>[1:0] == 0b10
       Rd = Memory[<address>,4] Rotate_Right 16
    else /* <address>[1:0] == 0b11 */
       Rd = Memory[<address>,4] Rotate_Right 24
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

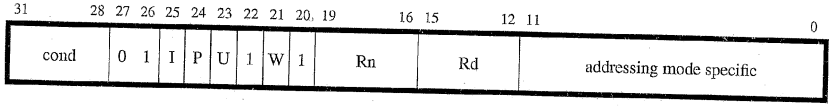
`LDR{<cond>}B Rd, <addressing_mode>`

Description

Combined with a suitable addressing mode, the LDRB (Load Register Byte) instruction allows 8-bit memory data to be loaded into a general-purpose register where it can be manipulated. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDRB loads a byte from the memory address calculated by `<addressing_mode>`, zero-extends the byte to a 32-bit word, and writes the word to register Rd.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Memory[<address>,1]
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd, and Rn, the results are UNPREDICTABLE.

Non-word-aligned addresses: Store Word instructions ignore the least-significant two bits of `<address>` (the words are not rotated as for load word).

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

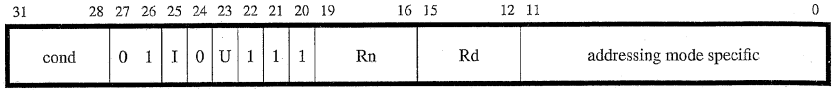
```
LDR{<cond>}BT  Rd, <post_indexed_addressing_mode>
```

Description

The LDRBT (Load Register Byte with Translation) instruction can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it has User mode privilege.

LDRBT loads a byte from the memory address calculated by `<post_indexed_addressing_mode>`, zero-extends the byte to a 32-bit word, and writes the word to register Rd. If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor was in User mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Memory[<address>,1]
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, and U bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<post_indexed_addressing_mode>`.

User mode: If this instruction is executed in User mode, an ordinary User mode access is performed.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).


```
LDR{<cond>}H  Rd, <addressing_mode>
```

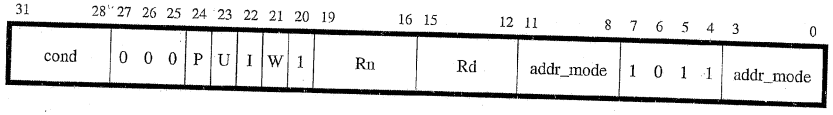
Description

Used with a suitable addressing mode, the LDRH (Load Register Halfword) instruction allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing to facilitate position-independent code.

LDRH loads a halfword from the memory address calculated by `<addressing_mode>`, zero-extends the halfword to a 32-bit word, and writes the word to register Rd. If the address is not halfword-aligned, the result is UNPREDICTABLE.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if <address>[0] == 0
        <data> = Memory[<address>,2]
    else /* <address>[0] == 1 */
        <data> = UNPREDICTABLE
    Rd = <data>
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 3* starting on page 315.

The addr_mode bits: These bits are addressing-mode specific.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing. and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word aligned addresses: If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.

`LDR{<cond>}SB  Rd, <addressing_mode>`

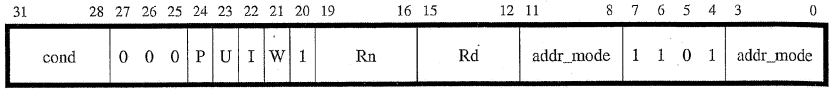
Description

Used with a suitable addressing mode, the LDRSB (Load Register Signed Byte) instruction allows 8-bit signed memory data to be loaded into a general-purpose register where it can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDRSB loads a byte from the memory address calculated by `<addressing_mode>`, sign extends the byte to a 32-bit word, and writes the word to register Rd.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <data> = Memory[<address>,1]
    Rd = SignExtend(<data>)
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 3* starting on page 315.

The addr_mode bits: These bits are addressing mode specific.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

`LDR{<cond>}SH  Rd, <addressing_mode>`

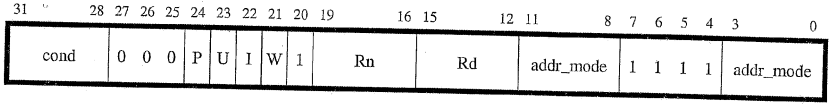
Description

Used with a suitable addressing mode, the LDRSH (Load Register Signed Halfword) instruction allows 16-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDRSH loads a halfword from the memory address calculated by `<addressing_mode>`, sign-extends the halfword to a 32-bit word, and writes the word to register Rd. If the address is not halfword-aligned, the result is UNPREDICTABLE.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if <address>[0] == 0
        <data> = Memory[<address>,2]
    else /* <address>[0] == 1 */
        <data> = UNPREDICTABLE
    Rd = SignExtend(<data>)
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 3* starting on page 315.

The addr_mode bits: These bits are addressing mode specific.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word aligned addresses: If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

ARM

LDRT

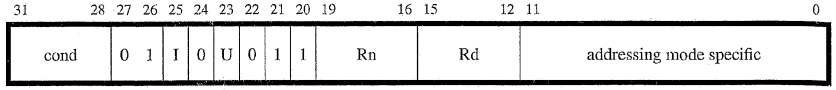
`LDR{<cond>}T Rd, <post_indexed_addressing_mode>`

Description

The LDRT (Load Register with Translation) instruction can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it has User mode privilege.

LDRT loads a word from the memory address and writes it to register Rd. If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor was in User mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if <address>[1:0] == 0b00
        Rd = Memory[<address>,4]
    else if <address>[1:0] == 0b01
        Rd = Memory[<address>,4] Rotate_Right 8
    else if <address>[1:0] == 0b10
        Rd = Memory[<address>,4] Rotate_Right 16
    else /* <address>[1:0] == 0b11 */
        Rd = Memory[<address>,4] Rotate_Right 24
```

Exceptions

Data Abort

Qualifiers

Condition Code

Notes

Addressing modes: The I, P, and U bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<post_indexed_addressing_mode>`.

User mode: If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions: If the same register is specified for Rd and Rn the results are UNPREDICTABLE.

Data Abort: If data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

A

MCR

MCR

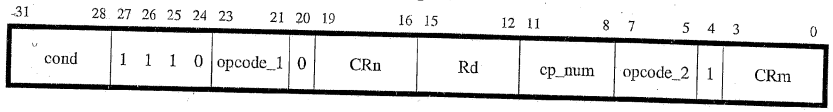
`MCR{<cond>} p<cp#>, <opcode_1>, Rd, CRn, CRm, <opcode_2>`

Description

The MCR (Move to Coprocessor from ARM Register) instruction is used to initiate coprocessor instructions that operate on values in ARM registers, for example, a fixed-point to floating-point conversion instruction for a floating-point accelerator coprocessor.

MCR passes the value of register Rd to the coprocessor specified by `<cp_num>`. If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Coprocessor[<cp_num>] = Rd
```

Exceptions

Undefined Instruction

Qualifiers

Condition Code

Notes

Coprocessor fields: Only instruction bits[31:24], bit[20], bits[15:8], and bit[4] are ARM architecturally defined. The remaining fields are recommendations for compatibility with ARM Development Systems.

A

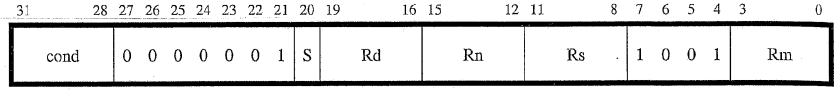
MLA{<cond>}{<S>} Rd, Rm, Rs, Rn

Description

The MLA (Multiply Accumulate) instruction multiplies signed or unsigned operands to produce a 32-bit result, which is then added to a third operand, and written to the destination register.

MLA multiplies the value of register Rm with the value of register Rs, adds the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = (Rm * Rs + Rn)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = unaffected
```

Exceptions

None

Qualifiers

Condition Code

S updates condition code flags N and Z

Notes

Use of R15: Specifying R15 for register Rd, Rm, Rs, or Rn has UNPREDICTABLE results.

Operand restriction: Specifying the same register for Rd and Rm has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned: Because the MLA instruction produces only the lower 32 bits of the 64-bit product, MLA gives the same answer for multiplication of both signed and unsigned numbers.

MOV{<cond>}{S} Rd, <shifter_operand>

Description

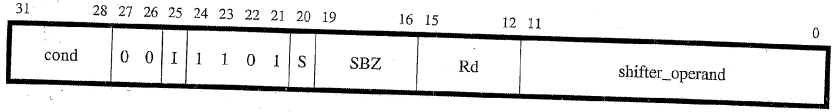
The MOV (Move) instruction is used to:

- Move a value from one register to another
- Put a constant value into a register
- Perform a shift without any other arithmetic or logical operation

When the PC is the destination of the instruction, a branch occurs, and MOV PC, LR can be used to return from a subroutine call (see the B and BL instructions) and to return from some types of exception.

MOV moves the value of <shifter_operand> to the destination register Rd, and optionally updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions

None

Qualifiers

Condition Code

S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

MRC

```
MRC{<cond>}  p<cp#>, <opcode_1>, Rd, CRn, CRm, <opcode_2>
```

Description

The MRC (Move to ARM Register from Coprocessor) instruction is used to initiate coprocessor instructions that return values to ARM registers, for example, a floating-point to fixed-point conversion instruction for a floating-point accelerator coprocessor.

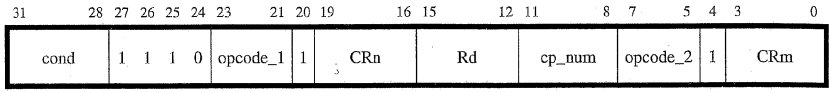
Specifying R15 as the destination register is useful for operations like a floating-point compare instruction.

MRC has two uses:

1. If Rd specifies register 15, the condition code flags bits are updated from the top four bits of the value from the coprocessor specified by <cp_num> (to allow conditional branching on the status of a coprocessor) and the other 28 bits are IGNORED.
2. Otherwise the instruction writes into register Rd a value from the coprocessor specified by <cp#>.

If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   if Rd == 15 then
        N flag = (value from Coprocessor[<cp_num>])[31]
        Z flag = (value from Coprocessor[<cp_num>])[30]
        C flag = (value from Coprocessor[<cp_num>])[29]
        V flag = (value from Coprocessor[<cp_num>])[28]
   else /* Rd != 15 */
        Rd = value from Coprocessor[<cp_num>]
```

Exceptions Undefined Instruction

Qualifiers Condition Code

Notes

Coprocessor fields: Only instruction bits[31:24], bit[20], bits[15:8], and bit[4] are ARM architecturally defined. The remaining fields are recommendations for compatibility with ARM Development Systems.

MRS

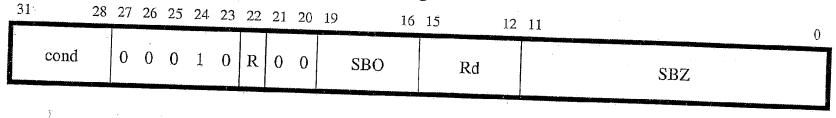
```
MRS{<cond>} Rd, CPSR
MRS{<cond>} Rd, SPSR
```

Description

The MRS instruction moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

The MRS moves the value of the CPSR, or the value of the SPSR corresponding to the current mode, to a general-purpose register.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   if R == 1 then
        Rd = SPSR
   else
        Rd = CPSR
```

Exceptions None

Qualifiers Condition Code

Notes

Opcode [11:0]: Execution of MRS instructions with any non-zero bits in opcode[11:0] is UNPREDICTABLE.

Opcode [19:16]: Execution of MRS instructions with any non-one bits in opcode[19:16] is UNPREDICTABLE.

User mode SPSR: Accessing the SPSR when in User mode or System mode is UNPREDICTABLE.

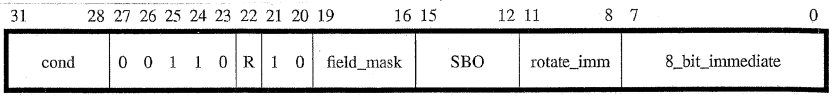
```
MSR{<cond>}  CPSR_f, #32bit immediate
MSR{<cond>}  CPSR_<fields>,Rm
MSR{<cond>}  SPSR_f, #32bit immediate
MSR{<cond>}  SPSR_<fields>, Rm
```

Description

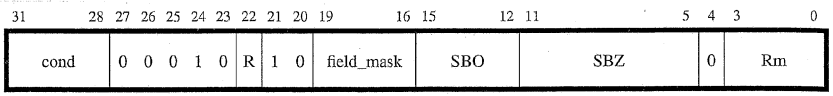
The MSR (Move to Status register from ARM Register) instruction transfers the value of a general-purpose register to the CPSR or the SPSR of the current mode. This is used to update the value of the condition code flags, interrupt enables, or the processor mode.

MSR moves the value of Rm or the value of the 32-bit immediate (encoded as an 8-bit value with rotate) to the CPSR or the SPSR corresponding to the current mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Immediate operand



Register operand

Operation

```
if ConditionPassed(<cond>) then
    if opcode[25] == 1
        <operand> = <8_bit_immediate> Rotate_Right (<rotate_imm> * 2)
    else /* opcode[25] == 0 */
        <operand>
    if R == 0 then
        if <field_mask>[0] == 1 and InAPrivilegedMode() then
            CPSR[7:0] = <operand>[7:0]
        if <field_mask>[1] == 1 and InAPrivilegedMode() then
            CPSR[15:8] = <operand>[15:8]
        if <field_mask>[2] == 1 and InAPrivilegedMode() then
            CPSR[23:16] = <operand>[23:16]
        if <field_mask>[3] == 1 then
            CPSR[31:24] = <operand>[31:24]
    else /* R == 1 */
        if <field_mask>[0] == 1 and CurrentModeHasSPSR() then
            SPSR[7:0] = <operand>[7:0]
        if <field_mask>[1] == 1 and CurrentModeHasSPSR() then
            SPSR[15:8] = <operand>[15:8]
        if <field_mask>[2] == 1 and CurrentModeHasSPSR() then
            SPSR[23:16] = <operand>[23:16]
        if <field_mask>[3] == 1 and CurrentModeHasSPSR() then
            SPSR[31:24] = <operand>[31:24]
```


Exceptions

None

Qualifiers

Condition Code

`<fields>` is one of

`_c`	sets the control field mask bit (bit 0)
`_x`	sets the extension field mask bit (bit 1)
`_s`	sets the status field mask bit (bit 2)
`_f`	sets the flags field mask bit (bit 3)

Notes

Immediate operand: The immediate form of this instruction can only be used to set the flag bits (PSR bits 31:24). Using the immediate form on any other fields has UNPREDICTABLE results.

PSR update: The value of a PSR must be updated by moving the PSR to a general-purpose register (using the MRS instruction), modifying the relevant bits of the general-purpose register, and restoring the updated general-purpose register value back into the PSR (using the MSR instruction).

User mode CPSR: Any writes to CPSR[23:0] in User mode are IGNORED (so that user mode programs cannot change to a privileged mode).

User mode SPSR: Accessing the SPSR when in User mode is UNPREDICTABLE.

System mode SPSR: Accessing the SPSR when in System mode is UNPREDICTABLE.

Deprecated field specification The CPSR, CPSR_flg, CPSR_ctl, CPSR_all, SPSR, SPSR_flg, SPSR_ctl, and SPSR_all forms of PSR field specification have been superseded by the `csxf` format shown above.

CPSR, SPSR, CPSR_all, and SPSR_all produce a field mask of 0b1001.
CPSR_flg and SPSR_flg produce a field mask of 0b1000.
CPSR_ctl and SPSR_ctl produce a field mask of 0b0001.

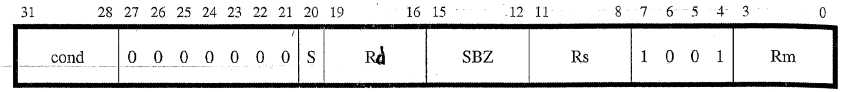
```
MUL{<cond>}{<S>}  Rd, Rm, Rs
```

Description

The MUL (Multiply) instruction is used to multiply signed or unsigned variables to produce a 32-bit result.

MUL multiplies the value of register Rm with the value of register Rs, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z

Notes

Use of R15: Specifying R15 for register Rd, Rm, or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for Rd and Rm has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned: Because the MUL instruction produces only the lower 32 bits of the 64-bit product, MUL gives the same answer for multiplication of both signed and unsigned numbers.


```
MVN{<cond>}{S}  Rd, <shifter_operand>
```

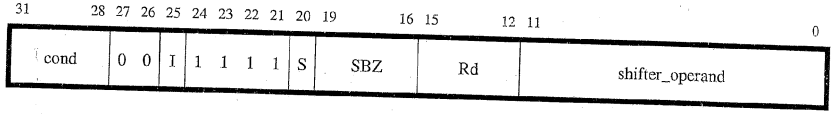
Description

The MVN (Move Negative) instruction is used to:

- Write a negative value into a register
- Form a bit mask
- Take the one's complement of a value

MVN moves the logical one's compliment of the value of `<shifter_operand>` to the destination register Rd, and optionally updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = NOT <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

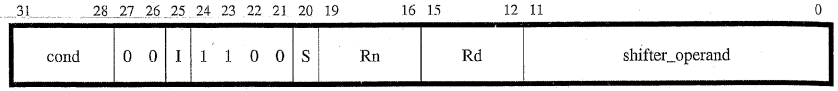
```
ORR{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The ORR (Logical OR) instruction can be used to set selected bits in a register; for each bit, OR with 1 will set the bit, OR with 0 will leave it unchanged.

ORR performs a bitwise (inclusive) OR of the value of register Rn with the value of `<shifter_operand>`, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   Rd = Rn OR <shifter_operand>
   if S == 1 and Rd == R15 then
        CPSR = SPSR
   else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = <shifter_carry_out>
        V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code
S updates condition code flags N, Z, and C

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.


```
RSB{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The RSB (Reverse Subtract) instruction subtracts the value of register Rn from the value of `<shifter_operand>`, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

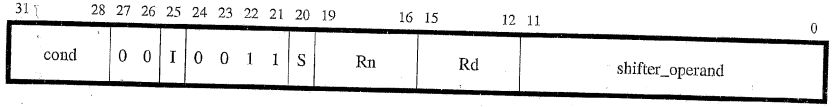
The following instruction stores the negative (two's complement) of Rx in Rd:

```
RSB Rd, Rx, #0
```

Constant multiplication (of Rx) by 2^n-1 (into Rd) can be performed with:

```
RSB Rd, Rx, Rx, LSL #n
```

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   Rd = <shifter_operand> - Rn
   if S == 1 and Rd == R15 then
        CPSR = SPSR
   else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(<shifter_operand> - Rn)
        V Flag = OverflowFrom (<shifter_operand> - Rn)
```

Exceptions None

Qualifiers Condition Code
S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

RSC{<cond>}{S} Rd, Rn, <shifter_operand>

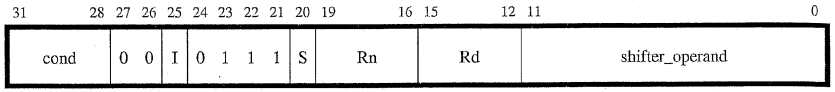
Description

The RSC (Reverse Subtract with Carry) instruction subtracts the value of register Rn and the value of NOT (Carry Flag) from the value of <shifter_operand>, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

To negate the 64-bit value in R0, R1, use the following sequence (R0 holds the least-significant word) and store the result in R2,R3:

```
RSBS        R2,R0,#0
RSC         R3,R1,#0
```

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = <shifter_operand> - Rn - NOT(C Flag)
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(<shifter_operand> - Rn - NOT(C Flag))
        V Flag = OverflowFrom (<shifter_operand> - Rn - NOT(C Flag))
```

Exceptions

None

Qualifiers

Condition Code
S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

SBC{<cond>}{S} Rd, Rn, <shifter_operand>

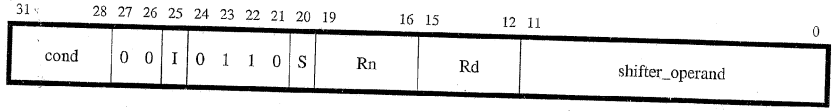
Description

The SBC (Subtract with Carry) instruction is used to synthesize multi-word subtraction. If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least-significant words), the following instructions leave the 64-bit difference in R4,R5:

```
SUBS        R4,R0,R2
SBC         R5,R1,R3
```

SBC subtracts the value of <shifter_operand> and the value of NOT (Carry Flag) from the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn - <shifter_operand> - NOT(C Flag)
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - <shifter_operand> - NOT(C Flag))
        V Flag = OverflowFrom (Rn - <shifter_operand> - NOT(C Flag))
```

Exceptions

None

Qualifiers

Condition Code

S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

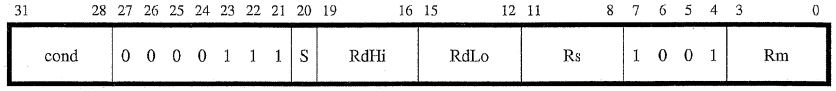
`SMLAL{<cond>}{<S>} RdLo, RdHi, Rm, Rs`

Description

The SMLAL (Signed Multiply Accumulate Long) instruction multiplies signed variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

SMLAL multiplies the signed value of register Rm with the signed value of register Rs to produce a 64-bit result. The lower 32 bits of the result are added to RdLo and stored in RdLo; the upper 32 bits, and the carry from the addition to RdLo, are added to RdHi and stored in RdHi. The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = UNPREDICTABLE
```

Exceptions None

Qualifiers

Condition Code
S updates condition code flags N, Z

Notes

Use of R15: Specifying R15 for register RdHi, RdLo, Rm, or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results. Specifying the same register for RdLo and Rm has UNPREDICTABLE results. Specifying the same register for RdHi and RdLo has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

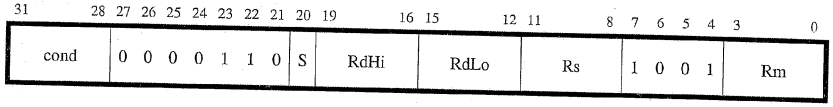
`SMULL{<cond>}{<S>} RdLo, RdHi, Rm, Rs`

Description

The SMULL (Signed Multiply Long) instruction multiplies signed variables to produce a 64-bit result in two general-purpose registers.

SMULL multiplies the signed value of register Rm with the signed value of register Rs to produce a 64-bit result. The upper 32 bits of the result are stored in RdHi; the lower 32 bits are stored in RdLo. The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    RdHi = (Rm * Rs)[63:32]
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = UNPREDICTABLE
```

Exceptions None

Qualifiers

Condition Code
S updates condition code flags N, Z

Notes

Use of R15: Specifying R15 for register RdHi, RdLo, Rm, or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results. Specifying the same register for RdLo and Rm has UNPREDICTABLE results. Specifying the same register for RdHi and RdLo has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

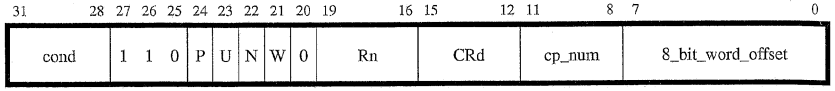
```
STC{<cond>}  p<cp_num>, CRd, <addressing_mode>
```

Description

The STC (Store Coprocessor) instruction is useful for storing coprocessor data to memory. The N bit could be used to distinguish between a single- and double-precision transfer for a floating-point store instruction.

STC stores data from the coprocessor specified by `<cp_num>` to the sequence of consecutive memory addresses calculated by `<addressing_mode>`. If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
         Memory[<address>,4] = value from Coprocessor[<cp_num>]
         <address> = <address> + 4
    assert <address> == <end_address>
```

Exceptions Undefined Instruction; Data Abort

Qualifiers Condition Code

Notes

Addressing mode: The P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 5* starting on page 329.

The N bit: This bit is coprocessor-dependent. It can be used to distinguish between two sizes of data to transfer.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Coprocessor fields: Only instruction bits[31:23], bits[21:16], and bits[11:0] are ARM architecturally defined. The remaining fields (bit[22] and bits[15:12]) are recommendations for compatibility with ARM Development Systems.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

Non-word-aligned addresses: Store coprocessor register instructions ignore the least-significant two bits of `<address>` (the words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

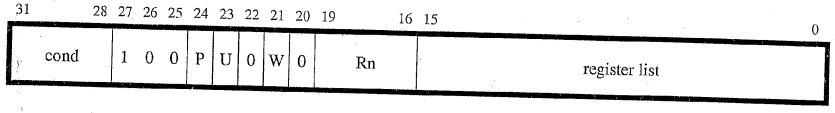

```
STM{<cond>}<addressing_mode>  Rn{!}, <registers>
```

Description

The STM (Store Multiple) instruction is useful as a block store instruction (combined with load multiple it allows efficient block copy) and for stack operations, including procedure entry to save general-purpose registers and the return address, and for updating the stack pointer.

STM stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations. The registers are stored in sequence, the lowest-numbered register first, to the lowest memory address (`<start_address>`); the highest-numbered register last, to the highest memory address (`<end_address>`).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_address>
    for i = 0 to 15
         if <register_list>[i] == 1
              Memory[<address>,4] = Ri
              <address> = <address> + 4
    assert <end_address> == <address> - 4
```

Exceptions Data Abort

Qualifiers Condition Code
! sets the W bit, causing base register update

Notes

Addressing mode: The P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 4* starting on page 322.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified as the base register Rn, the result is UNPREDICTABLE. If register 15 is specified in `<register_list>`, the value stored is IMPLEMENTATION DEFINED.

Operand restrictions: If Rn is specified in `<register_list>`, and writeback is specified, the stored value of Rn is UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` specifies writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

Non-word-aligned addresses: STM instructions ignore the least-significant two bits of `<address>` (words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

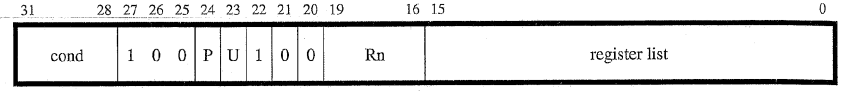
```
STM{<cond>}<addressing_mode>  Rn{!}, <registers>^
```

Description

The STM (Store Multiple) instruction is used to store the user mode registers when the processor is in a privileged mode (useful when performing process swaps).

This form of STM stores a subset (or possibly all) of the user mode general-purpose registers (which are also the system mode general-purpose registers) to sequential memory locations. The registers are stored in sequence, the lowest-numbered register first, to the lowest memory address (<start_addr>); the highest-numbered register last, to the highest memory address (<end_addr>).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <address> = <start_addr>
    for i = 0 to 15
        if <register_list>[i] == 1
            Memory[<address>,4] = Ri_usr
            <address> = <address> + 4
    assert <end_addr> == <address> - 4
```

Exceptions

Data Abort

Qualifiers

Condition Code

Notes

Addressing mode: The P and W bits specity the type of <addressing_mode>. See *Addressing Mode 4* starting on page 322.

Banked registers: This instruction must not be followed by an instruction which accesses banked registers (a following NOP is a good way to ensure this).

Writeback: Setting bit 21 (the W bit) has UNPREDICTABLE results.

User and System mode: This instruction is UNPREDICTABLE in User or System mode.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified as the base register Rn, the result is UNPREDICTABLE. If register 15 is specified in <register_list>, the value stored is IMPLEMENTATION DEFINED.

Base register mode: The base register is read from the current processor mode registers, not the user mode registers.

Data Abort: If a data abort is signalled, the value left in Rn is the original base register value.

Non-word-aligned addresses: Load multiple instructions ignore the least-significant two bits of <address> (the words are not rotated as for load word).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

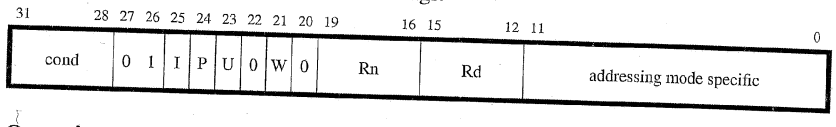

```
STR{<cond>}  Rd, <addressing_mode>
```

Description

Combined with a suitable addressing mode, the STR (Store Register) instruction stores 32-bit data from a general purpose register into memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

STR stores a word from register Rd to the memory address calculated by <addressing_mode>.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Memory[<address>,4] = Rd
```

Exceptions

Data Abort

Qualifiers

Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of <addressing_mode>. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified for Rd, the value stored is IMPLEMENTATION DEFINED.

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

```
STR{<cond>}B  Rd, <addressing_mode>
```

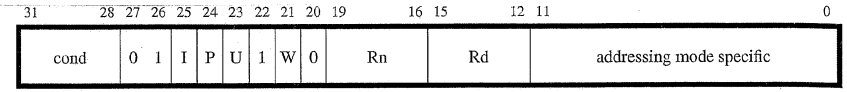
Description

Combined with a suitable addressing mode, the STRB (Store Register Byte) writes the least-significant byte of a general-purpose register to memory.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

STRB stores a byte from the least-significant byte of register Rd to the memory address calculated by `<addressing_mode>`.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   Memory[<address>,1] = Rd[7:0]
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

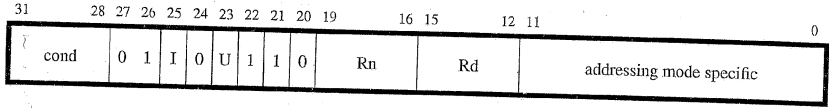

```
STR{<cond>}BT  Rd, <post_indexed_addressing_mode>
```

Description

The STRBT (Store Register Byte with Translation) instruction can be used by a (privileged) exception handler that is emulating a memory access instruction which would normally execute in User mode. The access is restricted as if it has User mode privilege.

STRBT stores a byte from the least-significant byte of register Rd to the memory address calculated by `<post_indexed_addressing_mode>`. If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor were in user mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   Memory[<address>,1] = Rd[7:0]
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, and U bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<post_indexed_addressing_mode>`.

User mode: If this instruction is executed in User mode, an ordinary user mode access is performed.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

STRH

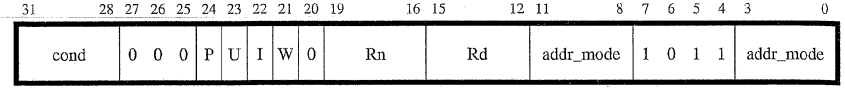
```
STR{<cond>}H  Rd, <addressing_mode>
```

Description

Combined with a suitable addressing mode, the STRH (Store Register Halfword) instruction allows 16-bit data from a general-purpose register to be stored to memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

STRH stores a halfword from the least-significant halfword of register Rd to the memory address calculated by `<addressing_mode>`. If the address is not halfword-aligned, the result is UNPREDICTABLE.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    if <address>[0] == 0
        <data> = Rd[15:0]
    else /* <address>[0] == 1 */
        <data> = UNPREDICTABLE
    Memory[<address>,2] = <data>
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, U, and W bits specify the type of `<addressing_mode>`. See *Addressing Mode 3* starting on page 315.

The addr_mode bits: These bits are addressing-mode specific.

Register Rn: Specifies the base register used by `<addressing_mode>`.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word aligned addresses: If the store address is not halfword-aligned, the stored value is UNPREDICTABLE.

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.

STRT

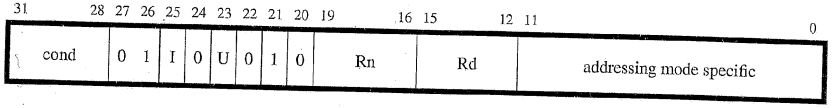
```
STR{<cond>}T  Rd, <post_indexed_addressing_mode>
```

Description

The STRT (Store Register with Translation) instruction can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it has User mode privilege.

STRT stores a word from register Rd to the memory address calculated by `<post_indexed_addressing_mode>`. If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor was in User mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Memory[<address>,4] = Rd
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Addressing modes: The I, P, and U bits specify the type of `<addressing_mode>`. See *Addressing Mode 2* starting on page 304.

Register Rn: Specifies the base register used by `<post_indexed_addressing_mode>`.

User mode: If this instruction is executed in User mode, an ordinary User mode access is performed.

Use of R15: If register 15 is specified for Rd, the value stored is IMPLEMENTATION DEFINED.

Operand restrictions: If the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

```
SUB{<cond>}{S}  Rd, Rn, <shifter_operand>
```

Description

The SUB (Subtract) instruction is used to subtract one value from another to produce a third. To decrement a register value (in Rx) use:

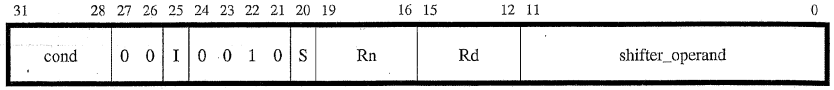
```
SUB Rx, Rx, #1
```

SUB subtracts the value of `<shifter_operand>` from the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

SUBS is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a `CMP Rx, #0`.

Use `SUBS PC, LR, #4` to return from an interrupt.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    Rd = Rn - <shifter_operand>
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - <shifter_operand>)
        V Flag = OverflowFrom (Rn - <shifter_operand>)
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N, Z, C, and V

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 320.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.


```
SWI{<cond>}  <24_bit_immediate>
```

Description

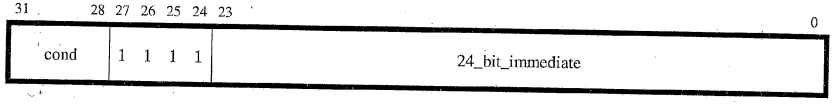
The SWI (Software Interrupt) instruction causes a SWI exception, see section 11.8.5 SWIs.

The SWI instruction is used as an operating system service call. It can be used in two ways:

- To use the 24-bit immediate value to indicate the OS service that is required
- To ignore the 24-bit field and indicate the service required with a general-purpose register

An SWI exception is generated, which is handled by an operating system to provide the requested service.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    R14_svc = address of SWI instruction + 4
    SPSR_svc = CPSR
    CPSR[5:0] = 0b010011; enter Supervisor mode
    CPSR[7] = 1; disable IRQ
    PC = 0x08
```

Exceptions None

Qualifiers Condition Code

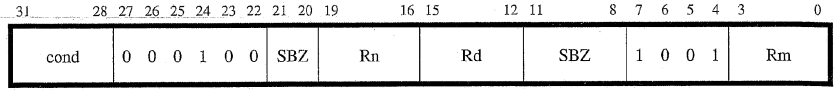
SWP{<cond>} Rd, Rm, [Rn]

Description

The SWP (Swap) instruction 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 Rn, this instruction swaps the value of the register and the value at the memory address.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   <temp> = Memory[Rn,4]
   Memory[Rn,4] = Rm
   Rd = <temp>
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Non-word-aligned addresses: If the address is not word-aligned, the loaded value is rotated right by 8 times the value of `<address>[1:0]`.

Use of R15: If register 15 is specified for Rd, Rn, or Rm, the result is UNPREDICTABLE.

Operand restrictions: If the same register is specified as Rn and Rm, or Rn and Rd, the result is UNPREDICTABLE.

Data Abort: If a data abort is signalled on either the load access or the store access (or both), the loaded value is not written to Rd.

Alignment: If an implementation includes a System Control Coprocessor, and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

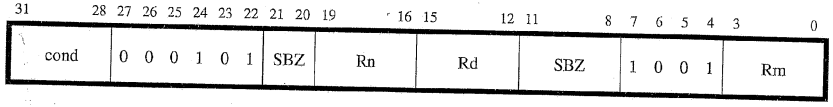
SWP{<cond>}B Rd, Rm, [Rn]

Description

The SWPB (Swap Byte) instruction swaps a byte between registers and memory.

SWPB loads a byte from the memory address given by the value of register Rn. The value of the least-significant byte of register Rm is stored to the memory address given by Rn, and the original loaded value is zero-extended to a 32-bit word, and the word is written to register Rd. If the same register is specified for Rd and Rn, this instruction swaps the value of the least-significant byte of the register and the byte value at the memory address.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
   <temp> = Memory[Rn,1]
   Memory[Rn,1] = Rm[7:0]
   Rd = <temp>
```

Exceptions Data Abort

Qualifiers Condition Code

Notes

Use of R15: If register 15 is specified for Rd, Rn, or Rm, the result is UNPREDICTABLE.

Operand restrictions: If the same register is specified as Rn and Rm, or Rn and Rd, the result is UNPREDICTABLE.

Data Abort: If a data abort is signalled on either the load access or the store access (or both), the loaded value is not written to Rd.

TEQ

```
TEQ{<cond>}  Rn, <shifter_operand>
```

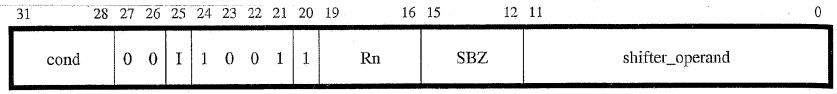
Description

The TEQ (Test Equivalence) instruction is used to test if two values are equal, without affecting the V flag (as CMP does). TEQ is also useful for testing if two values have the same sign.

The comparison is the Logical Exclusive OR of the two operands.

TEQ performs a comparison by logically Exclusive ORing the value of register Rn with the value of `<shifter_operand>`, and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <alu_out> = Rn EOR <shifter_operand>
    N Flag = <alu_out>[31]
    Z Flag = if <alu_out> == 0 then 1 else 0
    C Flag = <shifter_carry_out>
    V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

TST

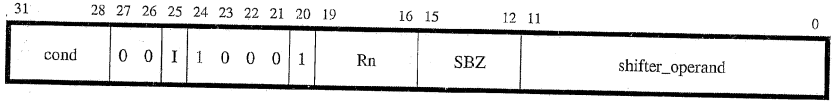
```
TST{<cond>}  Rn, <shifter_operand>
```

Description

The TST (Test) instruction is used to determine if many bits of a register are all clear, or if at least one bit of a register is set. The comparison is a logical AND of the two operands.

TST performs a comparison by logically ANDing the value of register Rn with the value of `<shifter_operand>`, and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    <alu_out> = Rn AND <shifter_operand>
    N Flag = <alu_out>[31]
    Z Flag = if <alu_out> == 0 then 1 else 0
    C Flag = <shifter_carry_out>
    V Flag = unaffected
```

Exceptions None

Qualifiers Condition Code

Notes

Shifter operand: The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 290.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

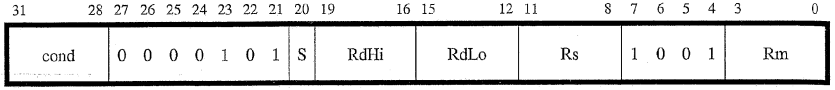
`UMLAL{<cond>}{<S>}  RdLo, RdHi, Rm, Rs`

Description

The UMLAL (Unsigned Multiply Accumulate Long) instruction multiplies unsigned variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers.The result is written back to the two destination general-purpose registers.

UMLAL multiplies the unsigned value of register Rm with the unsigned value of register Rs to produce a 64-bit result. The lower 32 bits of the result are added to RdLo and stored in RdLo; the upper 32 bits and the carry from the addition to RdLo are added to RdHi and stored in RdHi. The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = UNPREDICTABLE
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N and Z

Notes

Use of R15: Specifying R15 for register RdHi, RdLo, Rm, or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results. Specifying the same register for RdLo and Rm has UNPREDICTABLE results. Specifying the same register for RdHi and RdLo has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

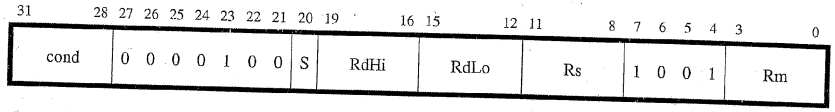
`UMULL{<cond>}{<S>}  RdLo, RdHi, Rm, Rs`

Description

The UMULL (Unsigned Multiply Long) instruction multiplies unsigned variables to produce a 64-bit result in two general-purpose registers.

UMULL multiplies the unsigned value of register Rm with the unsigned value of register Rs to produce a 64-bit result. The upper 32 bits of the result are stored in RdHi; the lower 32 bits are stored in RdLo. The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. See section 8.4 More on Flags.



Operation

```
if ConditionPassed(<cond>) then
    RdHi = (Rm * Rs)[63:32]
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = UNPREDICTABLE
        V Flag = UNPREDICTABLE
```

Exceptions None

Qualifiers Condition Code

S updates condition code flags N and Z

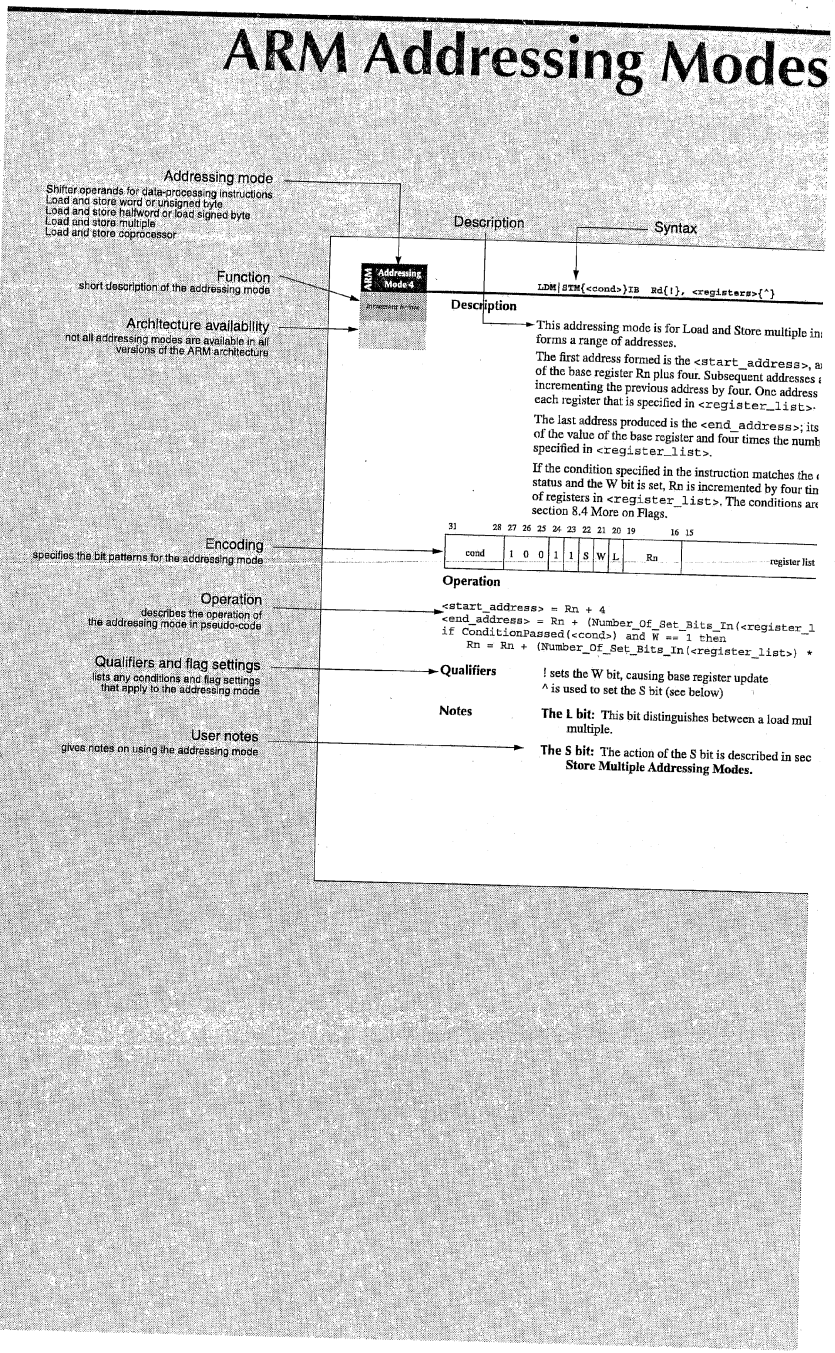
Notes

Use of R15: Specifying R15 for register RdHi, RdLo, Rm, or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results. Specifying the same register for RdLo and Rm has UNPREDICTABLE results. Specifying the same register for RdHi and RdLo has UNPREDICTABLE results.

Early termination: If the multiplier implementation supports early termination, it must be implemented on the value of the Rs operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

ARM Addressing Modes

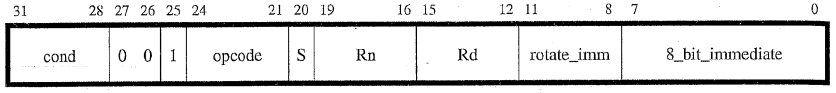


A

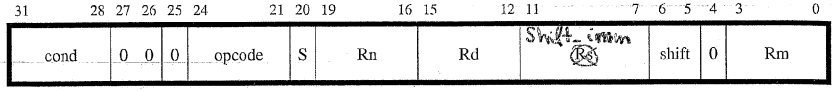
Data-processing Operands

```
<opcode>{<cond>}{S} {Rd}, {Rn}, <shifter_operand>
```

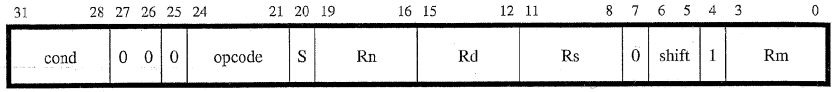
32-bit immediate



Immediate shifts



Register shifts



Description

<opcode>	Describes the operation of the instruction.
S bit	Indicates that the instruction updates the condition codes.
Rd	Specifies the destination register.
Rn	Specifies the first source operand register.
Bits[11:0]	The fields within bits[11:0] are collectively called a `<shifter_operand>`. This is described below.
Bit 25	Is referred to as the I bit, and is used to distinguish between an immediate `<shifter_operand>` and a register-based `<shifter_operand>`.

The shifter operand

As well as producing `<shifter_operand>`, the shifter produces a carry-out which some instructions write into the Carry Flag.

The shifter operand takes one of three basic formats:

- Immediate operand value
- Register operand value
- Shifted register operand value

Format 1: Immediate operand value

An immediate operand value is formed by rotating an 8-bit constant (in a 32-bit word) by an even number of bits (0,2,4,8...26,28,30). Thus, each instruction contains an 8-bit constant and a 4-bit rotate to be applied to that constant.

Valid constants are:

```
0xff,0x104,0xff0,0xff00,0xff000,0xff000000,0xf000000f
```

Invalid constants are:

```
0x101,0x102,0xff1,0xff04,0xff003,0xffffffff,0xf000001f
```

For example:

```
MOV   R0, #0            ; Move zero to R0
ADD   R3, R3, #1        ; Add one to the value of register 3
CMP   R7, #1000         ; Compare value of R7 with 1000
BIC   R9, R8, #0xff00   ; Clear bits 8-15 of R8 and store in R9
```

Format 2: Register operand value

A register operand value is simply the value of a register. The value of the register is used directly as the operand to the data-processing instruction.

For example:

```
MOV   R2, R0        ; Move the value of R0 to R2
ADD   R4, R3, R2    ; Add R2 to R3, store result in R4
CMP   R7, R8        ; Compare the value of R7 and R8
```

Format 3: Shifted register operand value

A shifted register operand value is the value of a register, shifted (or rotated) before it is used as the data-processing operand. There are five types of shift:

ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROR	Rotate right
RRX	Rotate right with extend

The number of bits to shift by is specified either as an immediate or as the value of the register.

For example:

```
MOV   R2, R0, LSL #2        ; Shift R0 left by 2, store in R2
                            ; (R2=R0x4)
ADD   R9, R5, R5, LSL #3    ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB   R9, R5, R5, LSL #3    ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB   R10, R9, R8, LSR #4   ; R10 = R9 - R8 / 16
MOV   R12, R4, ROR R3       ; R12 = R4 rotated right by value of R3
```

The default shifter operand

The default register operand (register Rm specified with no shift) uses the form register shift left by immediate, with the immediate set to zero.

Shifter operands

The 11 types of <shifter_operand> are described on the following pages:

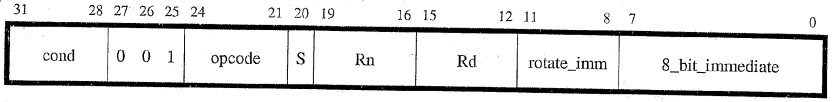
A

#<immediate>

Description

The <shifter_operand> value is formed by rotating (to the right) an 8-bit immediate value to any even bit position in a 32-bit word. If the rotate immediate is zero, the carry-out from the shifter is the value of the C flag, otherwise, it is set to bit 31 of the value of <shifter_operand>.

This data-processing operand provides a constant (defined in the instruction) operand to a data-processing instruction.



Operation

```
<shifter_operand> = <8_bit_immediate> Rotate_Right
(<rotate_imm> * 2)
if <rotate_imm> == 0 then
   <shifter_carry_out> = C flag
else /* <rotate_imm> > 31 */
   <shifter_carry_out> = <shifter_operand>[31]
```

Notes

Legitimate immediates: Not all 32-bit immediates are legitimate; only those that can be formed by rotating an 8-bit immediate right by an even amount are valid 32-bit immediates for this format.

Alternative assembly specification: The 32-bit immediate can also be specified by:

```
#<8_bit_immediate>, <rotate_amount>
```

where:

```
<rotate_amount> = <rotate_imm> << 1
```

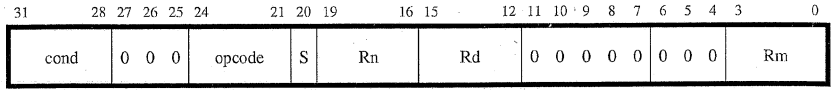
A

Rm

Description

This data-processing operand provides the value of a register directly.

This is an instruction operand produced by the value of register Rm. The carry-out from the shifter is the C flag.



Operation

```
<shifter_operand> = Rm
<shifter_carry_out> = C Flag
```

Notes

Encoding: This instruction is encoded as a Logical shift left by immediate (see page 295, **Rm, LSL #<shift_imm>**) with a shift of zero (`<shift_imm> == 0`).

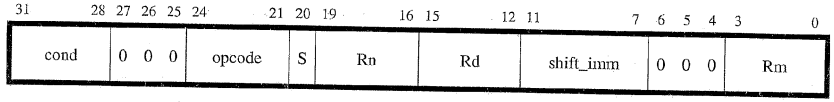
Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

Rm, LSL #<shift_imm>

Description

This data-processing operand is used to provide either the value of a register directly (lone register operand (see page 294, **Rm**), or the value of a register shifted left (multiplied by a constant power of two).

This instruction operand is produced by the value of register Rm, logically shifted left by an immediate value in the range 0 to 31. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, or the C flag if no shift is specified (lone register operand, see page 294, **Rm**).



Operation

```
if <shift_imm> == 0 then /* Register Operand */
   <shifter_operand> = Rm
   <shifter_carry_out> = C Flag
else /* <shift_imm> > 0 */
   <shifter_operand> = Rm Logical_Shift_Left <shift_imm>
   <shifter_carry_out> = Rm[32 - <shift_imm>]
```

Notes

Default shift: If the value of `<shift_imm> == 0`, the operand may be written as just Rm (see page 294, **Rm**).

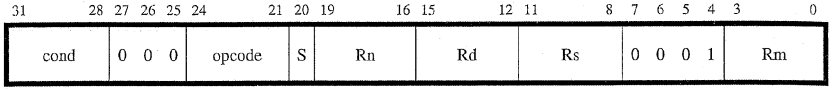
Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

`Rm, LSL Rs`

Description

This data-processing operand is used to provide the value of a register multiplied by a variable power of two.

It is produced by the value of register Rm, logically shifted left by the value in the least-significant byte of register Rs. Zeros are inserted into the vacated bit positions.



Operation

```
if Rs[7:0] == 0 then
   <shifter_operand> = Rm
   <shifter_carry_out> = C Flag
else if Rs[7:0] < 32 then
   <shifter_operand> = Rm Logical_Shift_Left Rs[7:0]
   <shifter_carry_out> = Rm[32 - Rs[7:0]]
else if Rs[7:0] == 32 then
   <shifter_operand> = 0
   <shifter_carry_out> = Rm[0]
else /* Rs[7:0] > 32 */
   <shifter_operand> = 0
   <shifter_carry_out> = 0
```

Notes

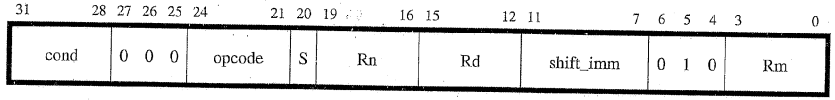
Use of R15: Specifying R15 as register Rm, register Rn, or register Rs has UNPREDICTABLE results.

`Rm, LSR #<shift_imm>`

Description

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a constant power of two).

It is produced by the value of register Rm logically shifted right by an immediate value in the range 1 to 32. Zeros are inserted into the vacated bit positions. A shift by 32 is encoded by `<shift_imm> = 0`.



Operation

```
if <shift_imm> == 0 then
   <shifter_operand> = 0
   <shifter_carry_out> = Rm[31]
else /* <shift_imm> > 0 */
   <shifter_operand> = Rm Logical_Shift_Right <shift_imm>
   <shifter_carry_out> = Rm[<shift_imm> - 1]
```

Notes

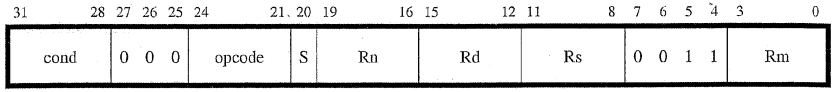
Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

`Rm, LSR Rs`

Description

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a variable power of two).

It is produced by the value of register Rm logically shifted right by the value in the least-significant byte of register Rs. Zeros are inserted into the vacated bit positions.



Operation

```
if Rs[7:0] == 0 then
    <shifter_operand> = Rm
    <shifter_carry_out> = C Flag
else if Rs[7:0] < 32 then
    <shifter_operand> = Rm Logical_Shift_Right Rs[7:0]
    <shifter_carry_out> = Rm[Rs[7:0] - 1]
else if Rs[7:0] == 32 then
    <shifter_operand> = 0
    <shifter_carry_out> = Rm[31]
else /* Rs[7:0] > 32 */
    <shifter_operand> = 0
    <shifter_carry_out> = 0
```

Notes

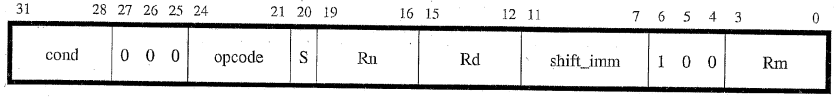
Use of R15: Specifying R15 as register Rm, register Rn, or register Rs has UNPREDICTABLE results.

`Rm, ASR #<shift_imm>`

Description

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a constant power of two).

It is produced by the value of register Rm arithmetically shifted right by an immediate value in the range 1 to 32. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions. A shift by 32 is encoded by `<shift_imm> = 0`.



Operation

```
if <shift_imm> == 0 then
    if Rm[31] == 0 then
        <shifter_operand> = 0
        <shifter_carry_out> = Rm[31]
    else /* Rm[31] == 1 */
        <shifter_operand> = 0xffffffff
        <shifter_carry_out> = Rm[31]
else /* <shift_imm> > 0 */
    <shifter_operand> = Rm Arithmetic_Shift_Right
<shift_imm>
    <shifter_carry_out> = Rm[<shift_imm> - 1]
```

Notes

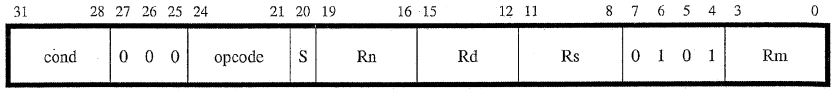
Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

`Rm, ASR Rs`

Description

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a variable power of two).

It is produced by the value of register Rm arithmetically shifted right by the value in the least-significant byte of register Rs. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions.



Operation

```
if Rs[7:0] == 0 then
    <shifter_operand> = Rm
    <shifter_carry_out> = C Flag
else if Rs[7:0] < 32 then
    <shifter_operand> = Rm Arithmetic_Shift_Right Rs[7:0]
    <shifter_carry_out> = Rm[Rs[7:0] - 1]
else /* Rs[7:0] >= 32 */
    if Rm[31] == 0 then
        <shifter_operand> = 0
        <shifter_carry_out> = Rm[31]
    else /* Rm[31] == 1 */
        <shifter_operand> = 0xffffffff
        <shifter_carry_out> = Rm[31]
```

Notes

Use of R15: Specifying R15 as register Rm, register Rn, or register Rs has UNPREDICTABLE results.

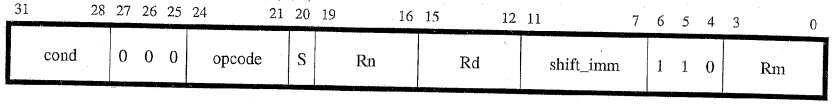
`Rm, ROR #<shift_imm>`

Description

This data-processing operand is used to provide the value of a register rotated by a constant value.

An instruction operand produced by the value of register Rm rotated right by an immediate value in the range 1 to 31. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left.

When `<shift_imm> = 0`, a Rotate right with extend operation is performed; see page 303, **`Rm, RRX`**.



Operation

```
if <shift_imm> == 0 then
    See page 303, Rm, RRX
else /* <shift_imm> > 0 */
    <shifter_operand> = Rm Rotate_Right <shift_imm>
    <shifter_carry_out> = Rm[<shift_imm> - 1]
```

Notes

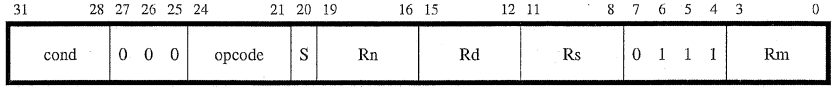
Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

`Rm, ROR Rs`

Description

This data-processing operand is used to provide the value of a register rotated by a variable value.

It is produced by the value of register Rm rotated right by the value in the least-significant byte of register Rs. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left.



Operation

```
if Rs[7:0] == 0 then
    <shifter_operand> = Rm
    <shifter_carry_out> = C Flag
else if Rs[4:0] == 0 then
    <shifter_operand> = Rm
    <shifter_carry_out> = Rm[31]
else /* Rs[4:0] > 0 */
    <shifter_operand> = Rm Rotate_Right Rs[4:0]
    <shifter_carry_out> = Rm[Rs[4:0] - 1]
```

Notes

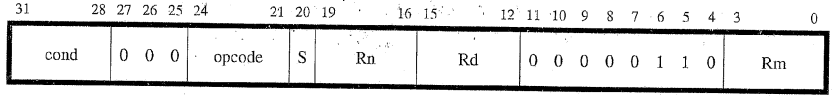
Use of R15: Specifying R15 as register Rm, register Rn, or register Rs has UNPREDICTABLE results.

`Rm, RRX`

Description

This data-processing operand can be used to perform a 33-bit rotate right using the Carry Flag as the 33rd bit.

It is produced by the value of register Rm shifted right by one bit, with the Carry Flag replacing the vacated bit position.



Operation

```
<shifter_operand> = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
<shifter_carry_out> = Rm[0]
```

Notes

Encoding: The instruction encoding is in the space that would be used for `ROR #0`.

Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

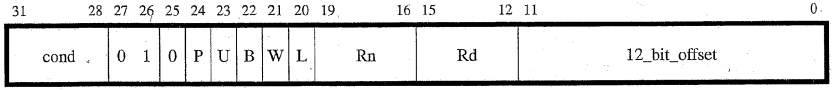
ADC instruction: A rotate right with extend can be performed with an ADC instruction.

Load and Store Word or Unsigned Byte Addressing Modes

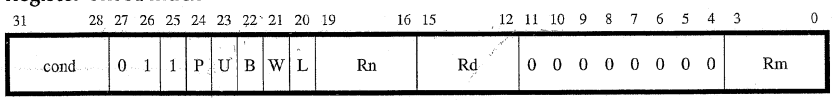
There are nine addressing modes used to calculate the address for a load and store word or unsigned byte instruction. Each addressing mode is described in detail on the following pages.

A

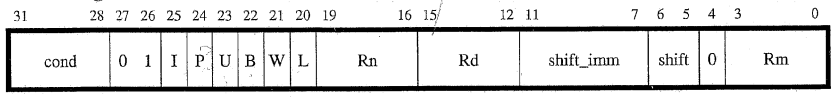
Immediate offset/index



Register offset/index



Scaled register offset/index



Notes

The P bit: Pre/Post-indexing:

Pre-indexing (P==1) indicates the offset is applied to the base register, and the result is used as the address.

Post-indexing (P==0) indicates the base register value is used for the address; the offset is then applied to the base register and written back to the base register.

The U bit: Indicates whether the offset is added to the base (U == 1) or subtracted from the base (U == 0).

The B bit: This bit distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.

The W bit: This bit has two meanings:

if P == 1	if W == 1, the calculated address will be written back to the base register. (If W == 0, the base register will not be updated.)
if P == 0	if W == 1, the current access is treated (by the protection and memory system) as a User mode access. (If W == 0, a normal access is performed.)

The L bit: This bit distinguishes between a Load (L == 1) and a Store (L == 0).

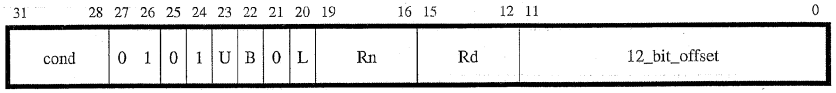
A

[Rn, #+/-<12_bit_offset]

Description

This addressing mode is useful for accessing structure (record) fields, and accessing parameters and local variables in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.



Operation

```
if U == 1 then
   <address> = Rn + <12_bit_offset>
else /* U == 0 */
   <address> = Rn - <12_bit_offset>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

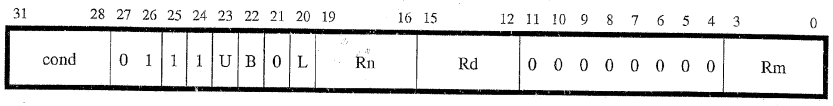
Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

[Rn, +/- Rm]

Description

This addressing mode is used for pointer + offset arithmetic, and accessing a single element of an array.

It calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.



Operation

```
if U == 1 then
   <address> = Rn + Rm
else /* U == 0 */
   <address> = Rn - Rm
```

Notes

Encoding: This addressing mode is encoded as an LSL scaled register offset, scaled by zero.

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

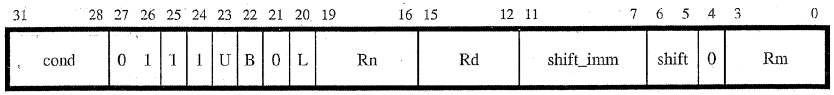
Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

```
[Rn, +/-Rm, LSL #<shift_imm>]
[Rn, +/-Rm, LSR #<shift_imm>]
[Rn, +/-Rm, ASR #<shift_imm>]
[Rn, +/-Rm, ROR #<shift_imm>]
[Rn, +/-Rm, RRX]
```

Description

These addressing modes are used for accessing a single element of an array of values larger than a byte.

They calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.



Operation

```
case <shift> of
    00 /* LSL */
        <index> = Rm Logical_Shift_Left <shift_imm>
    01 /* LSR */
        <index> = Rm Logical_Shift_Right <shift_imm>
    10 /* ASR */
        <index> = Rm Arithmetic_Shift_Right <shift_imm>
    11 /* ROR or RRX */
        if <shift_imm> == 0 then /* RRX */
            <index> = (C Flag Logical_Shift_Left 31)
                                OR (Rm Logical_Shift_Right 1)
        else /* ROR */
            <index> = Rm Rotate_Right <shift_imm>
endcase
if U == 1 then
    <address> = Rn + <index>
else /* U == 0 */
    <address> = Rn - <index>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

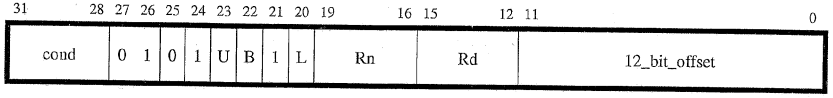

```
[Rn, #+/-<12_bit_offset>]!
```

Description

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
if U == 1 then
    <address> = Rn + <12_bit_offset>
else /* if U == 0 */
    <address> = Rn - <12_bit_offset>
if ConditionPassed(<cond>) then
    Rn = <address>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

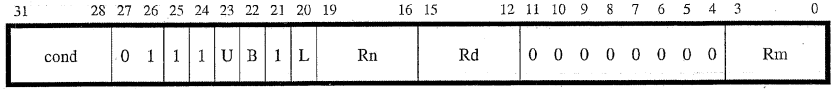
Use of R15: Specifying R15 as register Rn has UNPREDICTABLE results.

[Rn, +/- Rm]!

Description

This addressing mode calculates an address by adding or subtracting the value of an index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
if U == 1 then
    <address> = Rn + Rm
else /* U == 0 */
    <address> = Rn - Rm
if ConditionPassed(<cond>) then
   Rn = <address>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

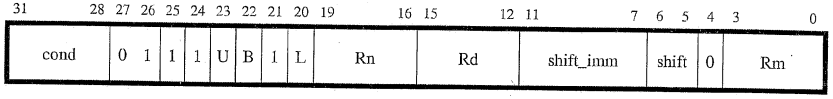
Operand Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

[Rn, +/-Rm, LSL #<shift_imm>]!
[Rn, +/-Rm, LSR #<shift_imm>]!
[Rn, +/-Rm, ASR #<shift_imm>]!
[Rn, +/-Rm, ROR #<shift_imm>]!
[Rn, +/-Rm, RRX]!

Description

These five addressing modes calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
case <shift> of
   00 /* LSL */
        <index> = Rm Logical_Shift_Left <shift_imm>
   01 /* LSR */
        <index> = Rm Logical_Shift_Right <shift_imm>
   10 /* ASR */
        <index> = Rm Arithmetic_Shift_Right <shift_imm>
   11 /* ROR or RRX */
        if <shift_imm> == 0 then /* RRX */
             <index> = (C Flag Logical_Shift_Left 31)
             OR (Rm Logical_Shift_Right 1)
        else /* ROR */
             <index> = Rm Rotate_Right <shift_imm>
endcase
if U == 1 then
   <address> = Rn + <index>
else /* U == 0 */
   <address> = Rn - <index>
if ConditionPassed(<cond>) then
   Rn = <address>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

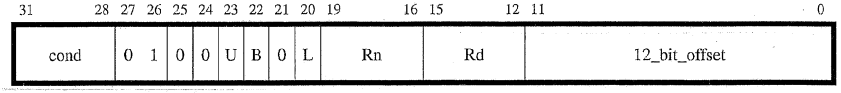
`[Rn], #+/-<12_bit_offset>`

Description

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

It calculates an address from the value of base register Rn.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
<address> = Rn
if ConditionPassed(<cond>) then
   if U == 1 then
        Rn = Rn + <12_bit_offset>
   else /* U == 0 */
        Rn = Rn - <12_bit_offset>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

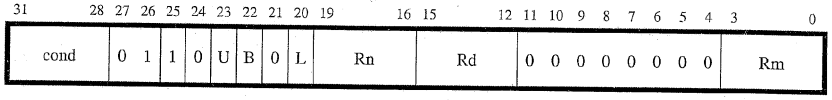
Use of R15: Specifying R15 as register Rn has UNPREDICTABLE results.

`[Rn], +/- Rm`

Description

This addressing mode calculates its address from the value of base register Rn.

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
<address> = Rn
if ConditionPassed(<cond>) then
   if U == 1 then
        Rn = Rn + Rm
   else /* U == 0 */
        Rn = Rn - Rm
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15: Specifying R15 as register Rn or Rm has UNPREDICTABLE results.

Operand Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

```
[Rn], +/-Rm, LSL #<shift_imm>
[Rn], +/-Rm, LSR #<shift_imm>
[Rn], +/-Rm, ASR #<shift_imm>
[Rn], +/-Rm, ROR #<shift_imm>
[Rn], +/-Rm, RRX
```

Description

If the condition specified in the instruction matches the condition code status, the shifted or rotated value of index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.

Operation

```
<address> = Rn
case <shift> of
    00 /* LSL */
        <index> = Rm Logical_Shift_Left <shift_imm>
    01 /* LSR */
        <index> = Rm Logical_Shift_Right <shift_imm>
    10 /* ASR */
        <index> = Rm Arithmetic_Shift_Right <shift_imm>
    11 /* ROR or RRX */
        if <shift_imm> == 0 then /* RRX */
            <index> = (C Flag Logical_Shift_Left 31)
            OR (Rm Logical_Shift_Right 1)
        else /* ROR */
            <index> = Rm Rotate_Right <shift_imm>
endcase
if ConditionPassed(<cond>) then
    if U == 1 then
        Rn = Rn + <index>
    else /* U == 0 */
        Rn = Rn - <index>
```

Notes

The B bit: This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

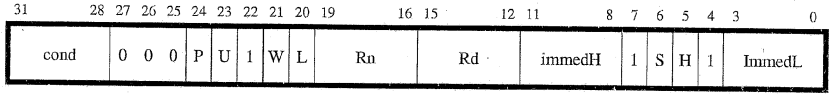
Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

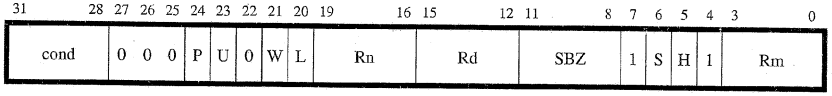
Load and Store Halfword or Load Signed Byte Addressing Modes

There are six addressing modes which are used to calculate the address for a load and store (signed or unsigned) halfword or load signed byte instructions.

Immediate offset/index



Register offset/index



Notes

The P bit: Pre-indexing (P==1) indicates the offset is applied to the base register, and the result is used as the address.
Post-indexing (P==0) indicates the base register value is used for the address; the offset is then applied to the base register and written back to the base register.

The U bit: Indicates whether the offset is added to the base (U==1) or subtracted from the base (U==0).

The W bit: If P is set, W indicates that the calculated address will be written back to the base register; if P is clear, the W bit must be clear or the instruction is UNPREDICTABLE.

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

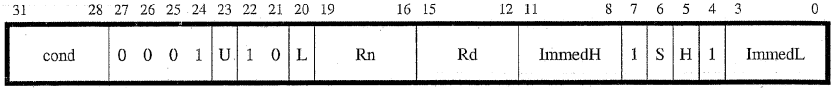
The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

`[Rn, #+/-<8_bit_offset>]`

Description

This addressing mode is used for accessing structure (record) fields, and accessing parameters and local variables in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.



Operation

```
<8_bit_offset> = (<immedH> << 4) OR <immedL>
if U == 1 then
   <address> = Rn + <8_bit_offset>
else /* U == 0 */
   <address> = Rn - <8_bit_offset>
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

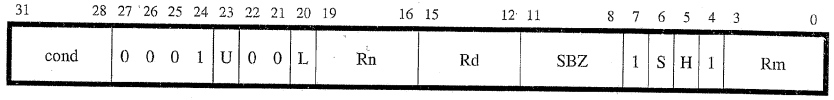
Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

`[Rn, +/- Rm]`

Description

This addressing mode is useful for pointer + offset arithmetic, and for accessing a single element of an array.

It calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.



Operation

```
if U == 1 then
   <address> = Rn + Rm
else /* U == 0 */
   <address> = Rn - Rm
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

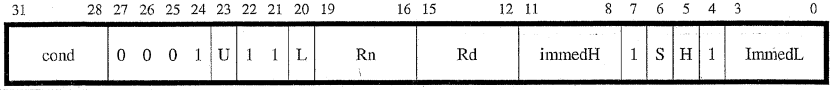
```
[Rn, #+/-<8_bit_offset>]!
```

Description

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
<8_bit_offset> = (<immedH> << 4) OR <immedL>
if U == 1 then
    <address> = Rn + <8_bit_offset>
else /* U == 0 */
    <address> = Rn - <8_bit_offset>
if ConditionPassed(<cond>) then
    Rn = <address>
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

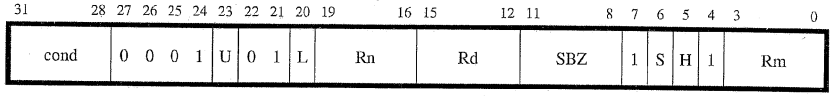
Use of R15: Specifying R15 as register Rn has UNPREDICTABLE results.


```
[Rn, +/- Rm]!
```

Description

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register R n.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
if U == 1 then
    <address> = Rn + Rm
else /* U == 0 */
    <address> = Rn - Rm
if ConditionPassed(<cond>) then
    Rn = <address>
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

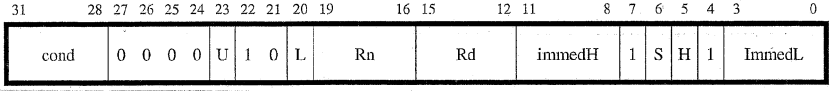
```
[Rn, #+/-<8_bit_offset>]!
```

Description

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

It calculates an address from the value of base register Rn.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
<address> = Rn
<8_bit_offset> = (<immedH> << 4) OR <immedL>
if ConditionPassed(<cond>) then
   if U == 1 then
        Rn = Rn + <8_bit_offset>
   else /* U == 0 */
        Rn = Rn - <8_bit_offset>
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

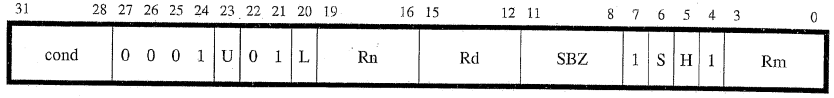
The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Use of R15: Specifying R15 as register Rn has UNPREDICTABLE results.


```
[Rn, +/- Rm]!
```

Description

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in section 8.4 More on Flags.



Operation

```
<address> = Rn
if ConditionPassed(<cond>) then
   if U == 1 then
        Rn = Rn + Rm
   else /* U == 0 */
        Rn = Rn - Rm
```

Notes

The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Load and Store Multiple Addressing Modes

Load Multiple instructions load a subset (possibly all) of the general-purpose registers from memory. Store Multiple instructions store a subset (possibly all) of the general purpose registers to memory. These instructions have a single instruction format.

Load and Store Multiple addressing modes produce a sequential range of addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.

There are four Load and Store Multiple addressing modes:

Increment After page 323

```
LDM|STM{<cond>}IA  Rn{!}, <registers>{^}
```

Increment Before page 324

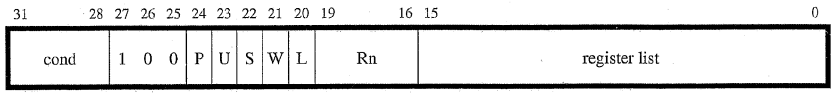
```
LDM|STM{<cond>}IB  Rn{!}, <registers>{^}
```

Decrement After page 325

```
LDM|STM{<cond>}DA  Rn{!}, <registers>{^}
```

Decrement Before page 326

```
LDM|STM{<cond>}DB  Rn{!}, <registers>{^}
```



Notes

The register list: The `<register_list>` has 1 bit for each general-purpose register; bit 0 is for register zero, bit 15 is for register 15 (the PC). The `<register_list>` is specified in the instruction mnemonic using a comma-separated list of registers, surrounded by brackets. If no bits are set, the result is UNPREDICTABLE.

The U bit: Indicates that the transfer is made upward (U==1) or downward (U==0) from base register.

The P bit: Pre-indexing or post-indexing:

- P==1 indicates that each address in the range is incremented (U==1) or decremented (U==0) before it is used to access memory.
- P==0 indicates that each address in the range is incremented (U==1) or decremented (U==0) after it is used to access memory.

The W bit: Indicates that the base register will be updated after the transfer. The base register is incremented (U==1) or decremented (U==0) by four times the number of registers in the register list.

The S bit: For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred and not the registers of the current mode.

The L bit: Distinguishes between Load (L==1) and Store (L==0) instructions.


```
LDM|STM{<cond>}IA  Rd{!}, <registers>{^}
```

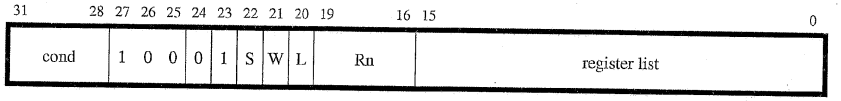
Description

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 `<register_list>`.

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 `<register_list>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<register_list>`. The conditions are defined in section 8.4 More on Flags.



Operation

```
<start_address> = Rn
<end_address> = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4) - 4
if ConditionPassed(<cond>) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4)
```

Qualifiers

! sets the W bit, causing base register update
^ is used to set the S bit (see below)

Notes

The L bit: This bit distinguishes between a load multiple and a store multiple.

The S bit: The action of the S bit is described in section **Load and Store Multiple Addressing Modes.**

`LDM|STM{<cond>}IB  Rd{!}, <registers>{^}`

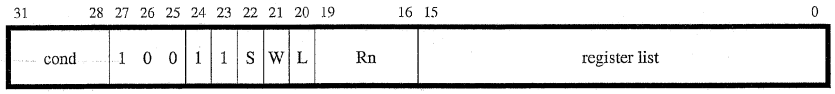
Description

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 plus four. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<register_list>`.

The last address produced is the `<end_address>`; its value is the sum of the value of the base register and four times the number of registers specified in `<register_list>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<register_list>`. The conditions are defined in section 8.4 More on Flags.



Operation

```
<start_address> = Rn + 4
<end_address> = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4)
if ConditionPassed(<cond>) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4)
```

Qualifiers

! sets the W bit, causing base register update
^ is used to set the S bit (see below)

Notes

The L bit: This bit distinguishes between a load multiple and a store multiple.

The S bit: The action of the S bit is described in section **Load and Store Multiple Addressing Modes.**

`LDM|STM{<cond>}DA  Rd{!}, <registers>{^}`

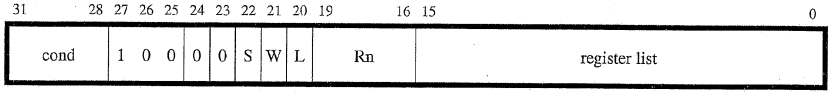
Description

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 `<register_list>`, plus 4. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<register_list>`.

The last address produced is the `<end_address>`; its value is the value of base register Rn.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<register_list>`. The conditions are defined in section 8.4 More on Flags.



Operation

```
<start_address> = Rn - (Number_Of_Set_Bits_In(<register_list>) * 4) + 4
<end_address> = Rn
if ConditionPassed(<cond>) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(<register_list>) * 4)
```

Qualifiers

! sets the W bit, causing base register update
^ is used to set the S bit (see below)

Notes

The L bit: This bit distinguishes between a load multiple and a store multiple.

The S bit: The action of the S bit is described in section **Load and Store Multiple Addressing Modes.**

```
LDM|STM{<cond>}DB  Rd{!}, <registers>{^}
```

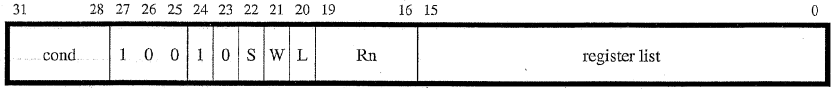
Description

This addressing mode is for Load and Store multiple instructions which form 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 `<register_list>`. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<register_list>`.

The last address produced is the `<end_address>`; its value is the value of base register Rn minus four.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<register_list>`. The conditions are defined in section 8.4 More on Flags.



Operation

```
<start_address> = Rn - (Number_Of_Set_Bits_In(<register_list>) * 4)
<end_address> = Rn - 4
if ConditionPassed(<cond>) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(<register_list>) * 4)
```

Qualifiers

! sets the W bit, causing base register update
^ is used to set the S bit (see below)

Notes

The L bit: This bit distinguishes between a load multiple and a store multiple.

The S bit: The action of the S bit is described in section **Load and Store Multiple Addressing Modes**.

Load and Store Multiple Addressing Modes (Alternative names)

Block data transfer

The four addressing mode names given in **Load and Store Halfword or Load Signed Byte Addressing Modes** (IA, IB, DA, DB) are most useful when a load and store multiple instruction is being used for block data transfer, as it is likely that the Load Multiple and Store Multiple will have the same addressing mode, so that the data is stored in the same way that it was loaded.

However, if Load Multiple and Store Multiple are being used to access a stack, the data will not be loaded with the same addressing mode that was used to store the data, because the load (pop) and store (push) operations must adjust the stack in opposite directions.

Stack operations

Load Multiple and Store Multiple addressing modes may be specified with an alternative syntax, which is more applicable to stack operations. Two attributes are used to describe the stack.

Full or Empty

Full	is defined to have the stack pointer pointing to the last used (full) location in the stack
Empty	is defined to have the stack pointer pointing to the first unused (empty) location in the stack

Ascending or Descending

Descending	grows toward decreasing memory address (toward the bottom of memory)
Ascending	grows toward increasing memory address (toward the top of memory)

This allows four types of stack to be defined:

1. Full Descending (FD)
2. Empty Descending (ED)
3. Full Ascending (FA)
4. Empty Ascending (EA)

Table A-1 shows the relationship between the four types of stack, the four types of addressing mode shown above, and the L, U, and P bits in the instruction format.

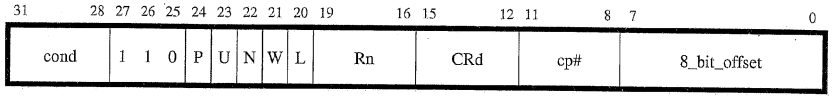
Instruction	Addressing Mode	Stack Type	L bit	P bit	U bit
LDM (Load)	IA (Increment After)	FD (Full Descending)	1	0	1
STM (Store)	IA (Increment After)	EA (Empty Ascending)	0	0	1
LDM (Load)	IB (Increment Before)	ED (Empty Descending)	1	1	1
STM (Store)	IB (Increment Before)	FA (Full Ascending)	0	1	1
LDM (Load)	DA (Decrement After)	FA (Full Ascending)	1	0	0
STM (Store)	DA (Decrement After)	ED (Empty Descending)	0	0	0
LDM (Load)	DB (Decrement Before)	EA (Empty Ascending)	1	1	0
STM (Store)	DB (Decrement Before)	FD (Full Descending)	0	1	0

Table A-1: ***LDM/STM addressing modes***

Load and Store Coprocessor Addressing Modes

There are three addressing modes which are used to calculate the address of a load or store coprocessor instruction:

1. Immediate offset page **[Rn, #+/-(<8_bit_offset>*4)]**
   `<opcode>{<cond>}{L} p<cp#>,CRd,[Rn,#+/-(<8_bit_offset>*4)]`
2. Immediate pre-indexed page **[Rn, #+/-(<8_bit_offset>*4)]!**
   `<opcode>{<cond>}{L} p<cp#>, CRd, [Rn, #+/-(<8_bit_offset>*4)]!`
3. Immediate post-indexed page **[Rn], #+/-(<8_bit_offset>*4)**
   `<opcode>{<cond>}{L} p<cp#>, CRd, [Rn], #+/-(<8_bit_offset>*4)`



Notes

The P bit: Pre-indexing (P==1) or post-indexing (P==0):

(P==1) indicates that the offset is added to the base register, and the result is used as the address.

(P==0) indicates that the base register value is used for the address; the offset is then added to the base register and written back to the base register (because W will equal 1, see below).

The U bit: Indicates that the offset is added to the base (U==1) or that the offset is subtracted from the base (U==0).

The N bit: The meaning of this bit is coprocessor-dependent; its recommended use is to distinguish between different-sized values to be transferred.

The W bit: This indicates that the calculated address will be written back to the base register. If P is 0, W must equal 1 or the result is UNPREDICTABLE.

The L bit: Distinguishes between Load (L==1) and Store (L==0) instructions.

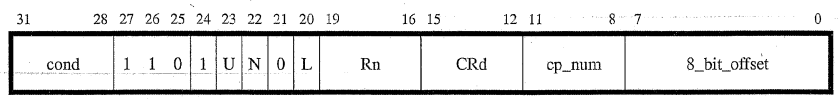
```
[Rn, #+/-(<8_bit_offset>*4)]
```

Description

This addressing mode produces a sequence of consecutive addresses.

The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.



Operation

```
if ConditionPassed(<cond>) then
    if U == 1 then
        <address> = Rn + <8_bit_offset> * 4
    else /* U == 0 */
        <address> = Rn - <8_bit_offset> * 4
    <start_address> = <address>
    while (NotFinished(coprocessor[<cp_num>]))
        <address> = <address> + 4
    <end_address> = <address>
```

Notes

The N bit: This bit is coprocessor-dependent.

The L bit: Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

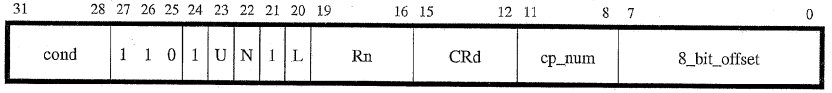

```
[Rn, #+/-(<8_bit_offset>*4)]!
```

Description

This addressing mode produces a sequence of consecutive addresses.

The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. The first address is written back to the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.



Operation

```
if ConditionPassed(<cond>) then
    if U == 1 then
        Rn = Rn + <8_bit_offset> * 4
    else /* U == 0 */
        Rn = Rn - <8_bit_offset> * 4
    <start_address> = Rn
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
        <address> = <address> + 4
    <end_address> = <address>
```

Notes

The N bit: This bit is coprocessor-dependent.

The L bit: Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

`[Rn], #+/-(<8_bit_offset>*4)`

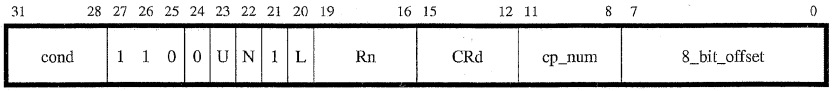
Description

This addressing mode produces a sequence of consecutive addresses.

The first address is the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The base register Rn is updated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn.

The coprocessor must not request a transfer of more than 16 words.



Operation

```
if ConditionPassed(<cond>) then
    <start_address> = Rn
    if U == 1 then
        Rn = Rn + <8_bit_offset> * 4
    else /* U == 0 */
        Rn = Rn - <8_bit_offset> * 4
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
        <address> = <address> + 4
    <end_address> = <address>
```

Notes

The N bit: This bit is coprocessor-dependent.

The L bit: Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15: If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

The W bit: If bit 21 (the Writeback bit) is not set, the result is UNPREDICTABLE.