

The ARM instruction set

Matteo SONZA REORDA
Dip. Automatica e Informatica
Politecnico di Torino



Data types

- ARM processors support six data types:
 - 8-bit signed and unsigned bytes
 - 16-bit signed and unsigned half-words; they must be aligned on 2-byte boundaries
 - 32-bit signed and unsigned words; they must be aligned on 4-byte boundaries
- All internal and memory operations are 32-bit wide

Little- and big-endian representation

- ARM processors may be configured to work in either modes
- We will adopt the little-endian representation

Privileged modes

- ARM processors may work
 - In the user mode
 - In a privileged mode
- Privileged modes are used to handle exceptions and supervisor calls (i.e., software interrupts)
- The current operating mode is defined by the lower 5 bits of the CPSR
- When a privileged mode is entered, the current value of the CPSR is save in the SPSR (Saved Program Status Register)

Operating modes

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Exceptions

- Exceptions include interrupts (from the outside), traps and supervisor calls
- They may be categorized in 3 groups:
 - Exceptions that are a direct effect of an instruction:
 - Software interrupts
 - Undefined instructions
 - Prefetch abort (i.e., memory fault during fetch)
 - Exceptions that are a side-effect of an instruction
 - Data aborts (i.e., memory fault during a load/store data access)
 - Exceptions generated externally
 - Reset
 - IRQ
 - FIQ

Exceptions management

- When an exception arises, the processor
 - Completes the current instruction (as best as it can)
 - Changes the operating mode to the appropriate one
 - Saves the address of the following instruction to r14
 - Saves the previous value of CPSR in SPSR
 - Disables interrupts on IRQs by setting bit 7 of CSR (and also fast interrupts, if the exception is a fast interrupt)
 - Forces the PC to a value between 00_{16} and $1C_{16}$, depending on the exception type

The vector address

- Locations from 00_{16} to $1C_{16}$ are called *vector address*, and usually contain branches to exception handlers
- The FIQ code, being the last, may start immediately

Exception vector addresses

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

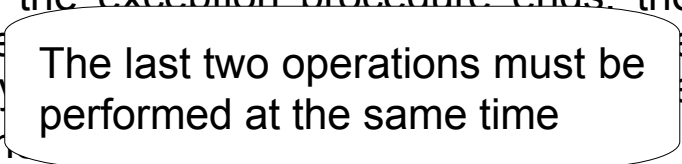
Privileged mode registers

- Each privileged mode owns two special registers, which are used to
 - Save the return address
 - Holds a stack pointer

Return from exceptions

- When the exception procedure ends, the handler code should restore the user state exactly as it was when the exception arose
- This means
 - Restoring the modified user registers
 - Restoring the CPSR from the appropriate SPSR
 - Restoring the PC to the next instruction in the user stream

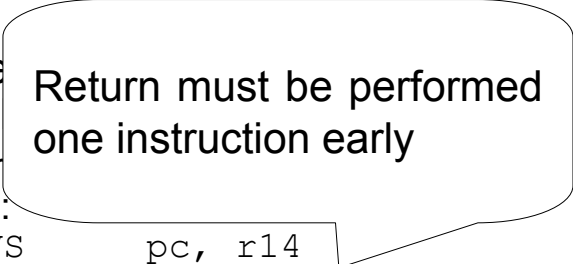
Return from exceptions

- When the exception procedure ends, the handler code should restore the user state exactly as it was when the exception arose
 - This means
 - Restoring the modified user registers
 - Restoring the CPSR from the appropriate SPSR
 - Restoring the PC to the next instruction in the user stream
- 
- The last two operations must be performed at the same time

Return from exceptions

- In the case the return address is in r14, the following instructions can be used
 - To return from a SWI or undefined instruction trap:
`MOVS pc, r14`
 - To return from an IRQ, FIQ, or prefetch abort:
`SUBS pc, r14, #4`
 - To return from a data abort to retry the data access:
`SUBS pc, r14, #8`

Return from exceptions

- In the case the return address is in r14, the following instructions can be used
 - To return from a SWI or undefined instruction trap:
`MOVS pc, r14`
 - To return from an IRQ, FIQ, or prefetch abort:
`SUBS pc, r14, #4`
 - To return from a data abort to retry the data access:
`SUBS pc, r14, #8`
- 

Return from exceptions

- In the case of the following exceptions, the return must be performed two instructions early:
 - To return from a trap:
`MOVS pc, r14`
 - To return from an I/Q, or prefetch abort:
`SUBS pc, r14, #4`
 - To return from a data abort to retry the data access:
`SUBS pc, r14, #8`

Return from exceptions

- In the case the return address is in the stack, the following instruction can be used

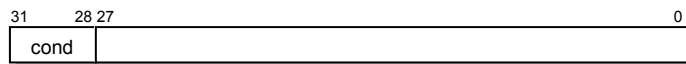
```
LDMFD r13!, {r0-r3,pc}^
```


Exception priorities

- If multiple exceptions arise at the same time, the following priorities are used
 - Reset (highest priority)
 - Data abort
 - FIQ
 - IRQ
 - Prefetch abort
 - SWI and undefined instruction

Conditional execution

- The 4 most significant bits in each instruction code specify a possible condition



Condition codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

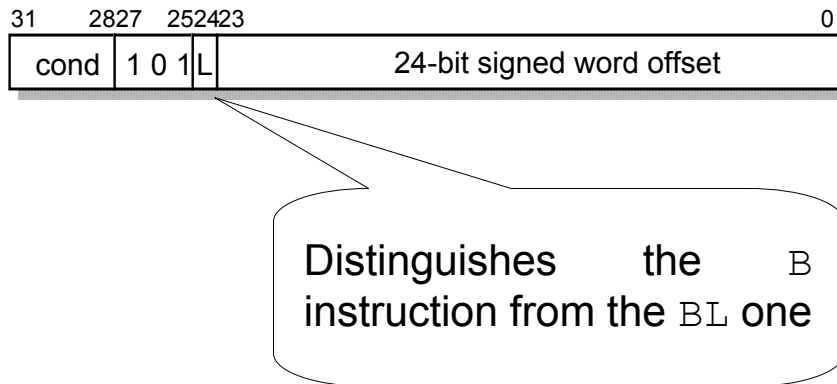
Branch (B) and Branch with Link (BL)

- Assembler format

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

- The `<target address>` is normally a label, that the assembler transforms into a signed 24-bit constant to be added to the address of the branch instruction
- The branch range is therefore +/-32 Mbytes
- The `BL` instruction also moves the address of the following instruction to `r14`

Binary encoding



Branch and eXchange (BX)

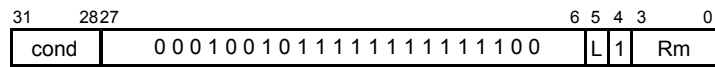
- Forces the processor to execute a jump to a Thumb instruction
- The target may be specified as a register or as an offset
- In the latter case the instruction can not be conditioned

Branch with Link and eXchange (BLX)

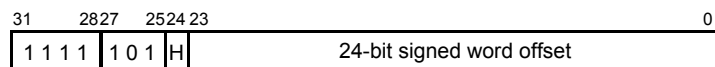
- Forces the processor to execute a jump to a Thumb procedure
- The target may be specified as a register or as an offset
- In the latter case the instruction can not be conditioned

Binary encoding

(1) BLX Rm



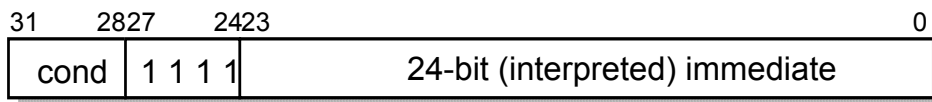
(2) BLX label



SoftWare Interrupt (SWI)

- Is used to call the operating system
- Puts the processor into supervisor mode
- Begins executing instructions from address 08_{16} , where a possible Operating System is possibly be stored

Binary encoding



- The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code

SWI: operation sequence

- Save the address of the following instruction in `r14_svc`
- Save the CPSR in `SPSR_svc`
- Enter supervisor mode and disable IRQs (not FIQs) by setting `CPSR[4:0]` to 10011_2 and `CPSR[7]` to 1
- Set the PC to 08_{16}

Examples

- To output the character 'A':

```
...  
MOV r0, #'A'  
SWI SWI_WriteC
```

- To finish executing the user program and return to the monitor program:

```
...  
SWI SWI_Exit
```

Example

- A subroutine to output a text string following the call:

```
...
BL          STROUT
=           "Hello World", &0a, &0d, 0
...
STROUT     LDRB          r0, [r14], #1
           CMP          r0, #0
           SWINE        SWI_WriteC
           BNE          STROUT
           ADD          r14, #3      ; align to
           BIC          r14, #3      ; next word
           MOV          pc, r14      ; return
```

Data processing instructions

- They are used to modify data values in registers using arithmetic and bit-wise operations
- The first operand is always a register Rn
- The second operand op2 may be
 - A register
 - A shifted register
 - An immediate value
- The result is stored in a register Rd

Data processing instructions

- They are registers operations
- The first operand op1 is a register.
- The second operand op2 may be
 - A register
 - A shifted register
 - An immediate value
- The result is stored in a register Rd

The shift may be logical, arithmetic or rotate.

The shift amount may be specified either through an immediate or by a further register.

Data processing instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	$Rd := Rn \text{ AND } Op2$
0001	EOR	Logical bit-wise exclusive OR	$Rd := Rn \text{ EOR } Op2$
0010	SUB	Subtract	$Rd := Rn - Op2$
0011	RSB	Reverse subtract	$Rd := Op2 - Rn$
0100	ADD	Add	$Rd := Rn + Op2$
0101	ADC	Add with carry	$Rd := Rn + Op2 + C$
0110	SBC	Subtract with carry	$Rd := Rn - Op2 + C - 1$
0111	RSC	Reverse subtract with carry	$Rd := Op2 - Rn + C - 1$
1000	TST	Test	Sec on Rn AND Op2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	$Rd := Rn \text{ OR } Op2$
1101	MOV	Move	$Rd := Op2$
1110	BIC	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
1111	MVN	Move negated	$Rd := \text{NOT } Op2$

Assembler format

```
<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>  
    <op>{<cond>}{S} Rd, Rn, Rm, {<shift>}
```

Assembler format

```
<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>  
    <op>{<cond>}{S} Rd, Rn, Rm, {<shift>}
```



Omitted for MOV and MVN

Assembler format

`<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>`
`<op>{<cond>}{S} Rd, Rn, Rm, {<shift>}`

Omitted for CMP, CMN,
TST, TEQ

Assembler format

`<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>`
`<op>{<cond>}{S} Rd, Rn, Rm, {<shift>}`

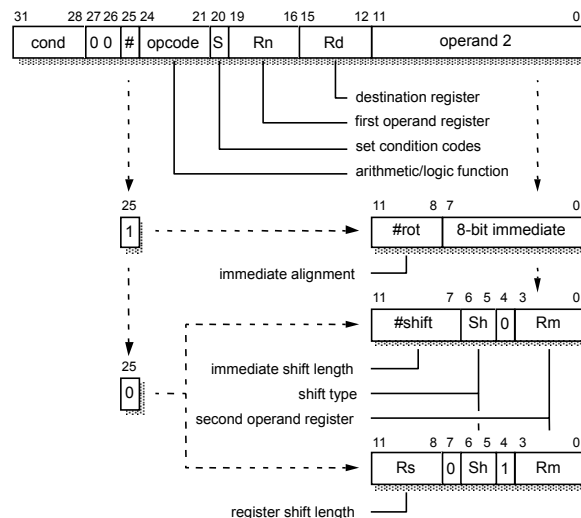
Specifies

- the shift type, i.e., LSL, LSR, ASL, ASR, ROR, or RRX
- the shift amount, i.e., a 5-bit immediate `#<#shift>` or a register `Rs`

Condition codes

- These instructions may set or not the condition code, depending on whether the S character appears or not

Binary code



Examples

```
ADD    r5, r1, r3    ; r5:=r1+r3
SUBS   r2, r2, #1     ; r2:=r2-1, set cc
ADD    r0, r0, r0, LSL #2
                                ; multiply r0 by 5
```

Example

- A subroutine to multiply r0 by 10:

```
MOV    r0, #3
BL     TIMES10 ; subroutine call
...
TIMES10 MOV r0, r0, LSL #1      ; × 2
        ADD r0, r0, r0, LSL #2  ; × 5
        MOV pc, r14 ; return
```

Example

- To add a 64-bit integer in r0, r1 to one in r2, r3:

```
ADDS    r2, r2, r0
```

```
ADC     r3, r3, r1
```

Use of r15

- The PC may specified as destination register, resulting in a branch or return from subroutine
- If the S bit is set the SPSR of the current mode is copied in the CPSR
- In this way the Pc and CPSR are modified atomically

Multiply instructions

- ARM processors may include a set of multiply instructions

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	$Rd := (Rm * Rs) [31:0]$
001	MLA	Multiply-accumulate (32-bit result)	$Rd := (Rm * Rs + Rn) [31:0]$
100	UMULL	Unsigned multiply long	$RdHi:RdLo := Rm * Rs$
101	UMLAL	Unsigned multiply-accumulate long	$RdHi:RdLo += Rm * Rs$
110	SMULL	Signed multiply long	$RdHi:RdLo := Rm * Rs$
111	SMLAL	Signed multiply-accumulate long	$RdHi:RdLo += Rm * Rs$

Multiply instructions

- ARM processors may include a set of multiply instructions

Selects only the least significant 32 bits of the result

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	$Rd := (Rm * Rs) [31:0]$
001	MLA	Multiply-accumulate (32-bit result)	$Rd := (Rm * Rs + Rn) [31:0]$
100	UMULL	Unsigned multiply long	$RdHi:RdLo := Rm * Rs$
101	UMLAL	Unsigned multiply-accumulate long	$RdHi:RdLo += Rm * Rs$
110	SMULL	Signed multiply long	$RdHi:RdLo := Rm * Rs$
111	SMLAL	Signed multiply-accumulate long	$RdHi:RdLo += Rm * Rs$

It is the concatenation of two 32-bit registers $RdHi$ and $RdLo$

Accumulation

Multiply instructions

- ARM processor multiply instructions (UMULL, UMULAL, SMULL, SMLAL) are only supported by ARM7 versions with an M in their name (e.g., ARM7DM)

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	$Rd := (Rm * Rs) [31:0]$
001	MLA	Multiply-accumulate (32-bit result)	$Rd := (Rm * Rs + Rn) [31:0]$
100	UMULL	Unsigned multiply long	$RdHi:RdLo := Rm * Rs$
101	UMULAL	Unsigned multiply-accumulate long	$RdHi:RdLo += Rm * Rs$
110	SMULL	Signed multiply long	$RdHi:RdLo := Rm * Rs$
111	SMLAL	Signed multiply-accumulate long	$RdHi:RdLo += Rm * Rs$

Early ARM only supported 32-bit multiply instructions (MUL and MLA)

Assembler format

- Instructions that produce a 32-bit result:

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

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

- Instructions that produce a 64-bit result:

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

UMULL, UMULAL,
SMULL, SMLAL

Example

- To form a scalar product of two vectors (pointed to by r8 and r9, respectively):

```
MOV    r11, #20    ; initialize loop counter
MOV    r10, #0     ; initialize total
LOOP   LDR    r0, [r8], #4    ; first component
        LDR    r1, [r9], #4    ; second component
        MLA    r10, r0, r1, r10    ; accumulate
        SUBS   r11, r11, #1    ; update loop counter
        BNE    LOOP
```

Binary encoding

31	28	27	24	23	21	20	19	16	15	12	11	8	7	4	3	0
cond				0 0 0 0		mul		S	Rd/RdHi	Rn/RdLo		Rs		1 0 0 1		Rm

Count leading zeros (CLZ)

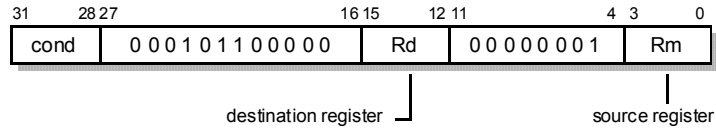
- This instructions is only available in v5T architectures
- It sets Rd to the number of zeros at more significant bit positions than the most significant 1 in Rm
- If Rm is zero Rd is set to 32
- Assembly format:

`CLZ{<cond>} Rd, Rm`

Example

```
MOV    r0, #&100
CLZ    r1, r0          ; r1:=23
```

Binary encoding



Data transfer instructions

- They can be categorized into
 - Single word and unsigned byte data transfer
 - Half word and signed byte data transfer
 - Multiple register transfer

Single word and unsigned byte data transfer instructions

- They allow transferring data from ARM registers to memory and viceversa

Assembler format

- Pre-indexed form

`LDR|STR{<cond>}{B} Rd, [Rn, <offset>] {!}`

- Post-indexed form

`LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>`

- PC-relative form

`LDR|STR{<cond>}{B} Rd, LABEL`

Assembler format

- Pre-indexed form

LDR|STR{<cond>}{B} Rd, [Rn, <offset>] {!}

- Post-indexed form

LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>

- PC-relative form

LDR|STR{<cond>} Rd, LABEL

Selects an unsigned
byte transfer

Assembler format

- Pre-indexed form

LDR|STR{<cond>}{B} Rd, [Rn, <offset>] {!}

- Post-indexed form

LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>

- PC-relative form

LDR|STR{<cond>} Rd, LABEL

May be

- #+/-<12-bit immediate>

or

- +/-Rm {, shift}, where register
specified shift amounts are not available

Assembler format

- Pre-indexed form

LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}

- Post-indexed form

LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>

- PC-relative form

LDR|STR{<cond>}{B} Rd, LABEL



Selects auto-indexing

Assembler format

- Pre-indexed form

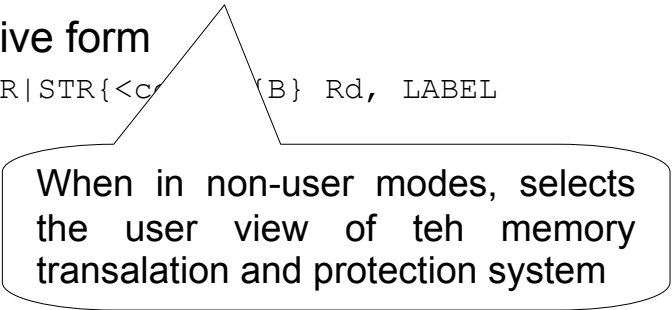
LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}

- Post-indexed form

LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>

- PC-relative form

LDR|STR{<cond>}{B} Rd, LABEL



When in non-user modes, selects the user view of the memory translation and protection system

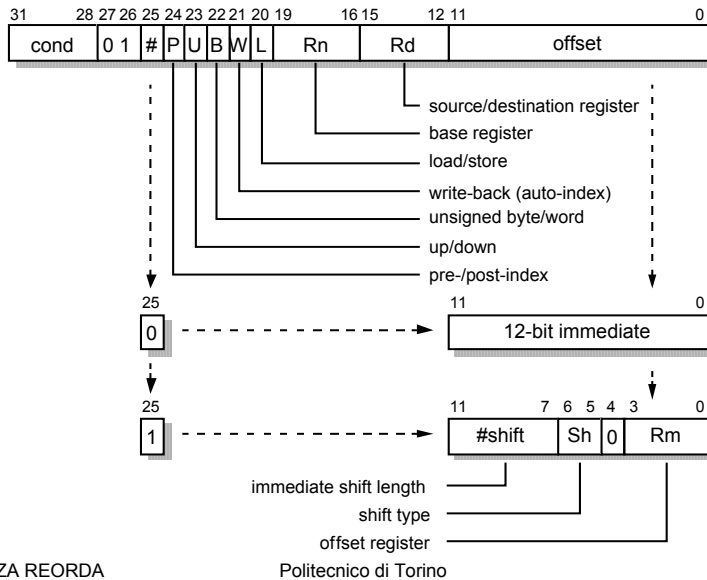
Unsigned byte transfer

- From memory to register:
 - The byte is zero extended to 32 bits
- From register to memory:
 - The bottom 8 bits of the register are stored in the addressed location

Example

```
LDR r1, UARTADD ; UART address into r1
STRB    r0, [r1]      ; store data to UART
...
UARTADD & 0x1000000    ; address literal
```

Binary encoding



Matteo SONZA REORDA

Politecnico di Torino

61

Half-word and signed byte data transfer instructions

- These instructions are not supported by some early ARM processors
- They are very similar to the word and unsigned byte forms, but
 - The immediate offset is limited to 8 bits
 - The scaled register offset is not available

Assembler formats

- Pre-indexed form

LDR|STR{<cond>}H|SH|SB Rd, [Rn, <offset>]{!}

- Post-indexed form

LDR|STR{<cond>}H|SH|SB Rd, [Rn], <offset>

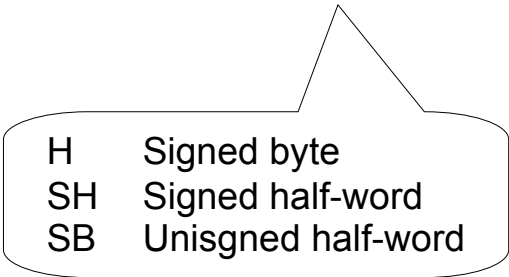
Assembler formats

- Pre-indexed form

LDR|STR{<cond>}H|SH|SB Rd, [Rn, <offset>]{!}

- Post-indexed form

LDR|STR{<cond>}H|SH|SB Rd, [Rn], <offset>



H	Signed byte
SH	Signed half-word
SB	Unisgned half-word

Example

- To expand an array of signed half-words into an array of words:

```
ADR  r1, ARRAY1
ADR  r2, ARRAY2
ADR  r3, ENDARR1
LOOP LDRSHr0, [r1], #2    ; get signed half-word
STR  r0, [r2], #4        ; save word
CMP  r1, r3
BLT  LOOP
```

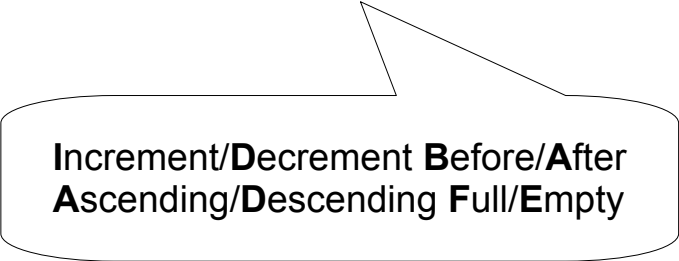
Multiple register transfer instructions

- They allow any subset of the registers visible in the current operating mode to be loaded from or stored to memory

Assembler format

- Normal form

`LDM|STM{<cond>}<add mode> Rn{!}, <registers>`



**Increment/Decrement Before/After
Ascending/Descending Full/Empty**

Assembler format

- The following additional form exists

`LDM|STM{<cond>}<add mode> Rn{!}, <registers+pc>^`

- In this case the two following operations are atomically performed
 - The SPSR of the current mode is copied into the CPSR
 - The PC is restored

Example

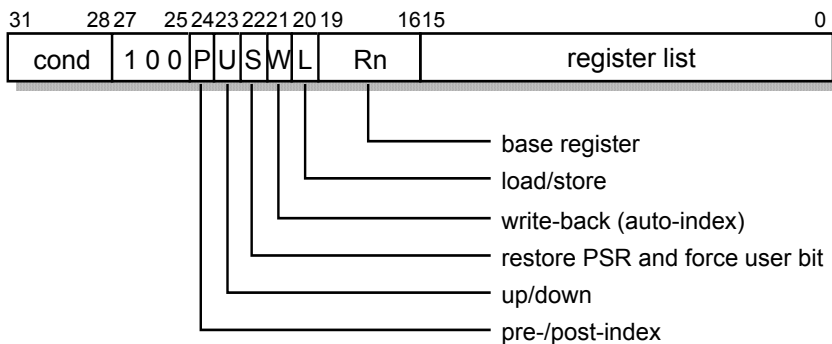
- To save 3 work registers and the return address in the stack (e.g., upon entering a subroutine):

```
STMFD    r13!, {r0-r2, r14}
```

- To restore the work registers and return :

```
LDMFD    r13!, {r0-r2, r14}
```

Binary encoding



Swap memory and register instruction (SWP)

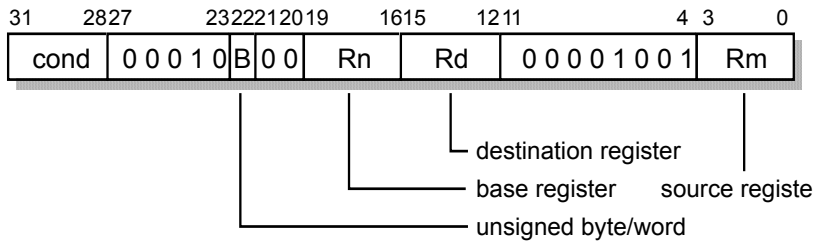
- It combines a load and store of a word or unsigned byte in a single atomic instruction
- In this way the two transfers can not be split and a semaphore mechanism can be implemented
- Assembler format:

`SWP{<cond>}{B} Rd, Rm, {Rn}`

Example

```
ADR    r0, SEMAPHORE
SWPB   r1, r1, [r0]
```

Binary encoding



Status register to general register transfer instructions

- They move the CPSR or SPSR of the current mode in a general register
- **Assembler format:**

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

- **Examples:**

```
MRS    r0, CPSR    ; move the CPSR to r0
MRS    r3, SPSR    ; move the SPSR to r3
```

General register to status register transfer instructions

- They load the CPSR or SPSR of the current mode with the value of a general register
- **Assembler format:**

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

where <field> can be either **c** (for the control fields) or **f** (for the flags field)

Examples

- To set the N, Z, C and V flags:

```
MSR      CPSR_f, #&f0000000 ; set the flags
```

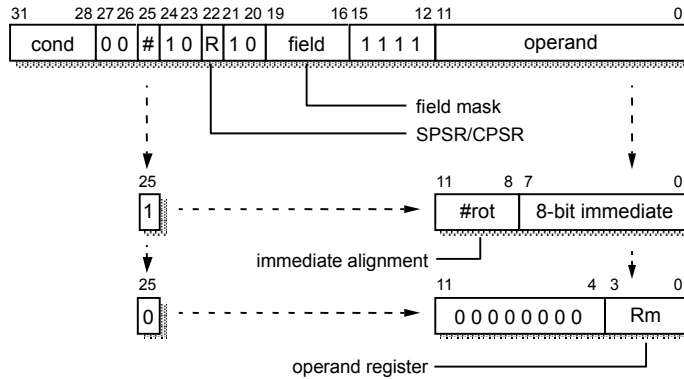
- To set just the C flag, preserving N, Z and V:

```
MRS      r0, CPSR ; move the CPSR to r0
```

```
ORR      r0, r0, #&20000000 ; set bit 29
```

```
MSR      CPSR_f, r0 ; move back to CPSR
```

Binary encoding



Coprocessor instructions

- They belong to three groups:
 - Coprocessor data operations
 - Coprocessor data transfers
 - Coprocessor register transfers

Breakpoint instruction

- They are only supported by the v5T architecture
- They are used for software debug purposes

Unused instruction space

- Not all the available 2^{32} possible combinations do correspond to an instruction
- Some of them are reserved for future use
- ARM processors trigger an undefined instruction trap if they decode such instruction codes

Memory faults

- The most common sources for memory faults are
 - Page absent
 - Page protected
 - Soft memory error.
- In all cases, the processor triggers an exception, and tries to block the execution of the current instruction rolling back to the state existing before the execution began.