UNIT-III

The ARM Instruction Set: Introduction, Exceptions, Conditional execution , Branch and Branch with Link (B, BL),Branch, Branch with Link and exchange (BX, BLX) , Software Interrupt (SWI) ,Data processing instructions, Multiply instructions, Single word and unsigned byte data transfer instructions , Half-word and signed byte data transfer instructions, Multiple register transfer instructions , Status register to general register transfer instructions ,General register to status register transfer instructions , Coprocessor instructions. Coprocessor data operations, Coprocessor data transfers, Coprocessor register transfers, Breakpoint instruction (BRK - architecture v5T only), unused instruction space, Memory faults, ARM architecture variants.

INTRODUCTION

Datatype

ARM processors support six data types:

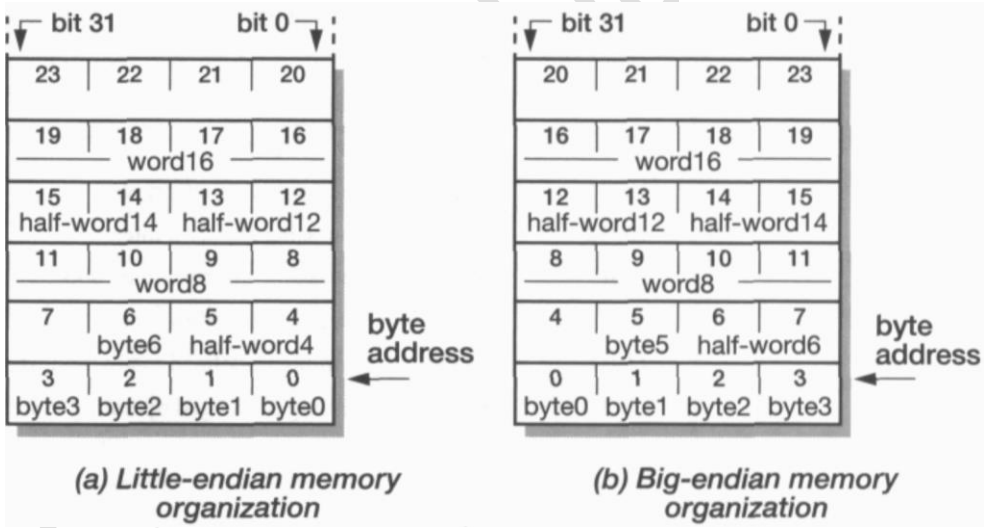
• 8-bit signed and unsigned bytes.

• 16-bit signed and unsigned half-words; these are aligned on 2-byte boundaries.

• 32-bit signed and unsigned words; these are aligned on 4-byte boundaries.

ARM instructions are all 32-bit words and must be word-aligned. Thumb instructions are half-words and must be aligned on 2-byte boundaries.

Memory Organisation



There are two ways to store words in a byte-addressed memory, depending on whether the least significant byte is stored at a lower or higher address than the next most significant byte. Default memory organization is Little endian.

Privileged modes

ARM has other **privileged** operating modes which are used to handle exceptions and supervisor calls (which are sometimes called **software interrupts).**

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

The current operating mode is defined by the bottom five bits of the CPSR. The privileged modes can be used to give a weaker level of protection that is useful for trapping errant software.
Each privileged mode (except system mode) has associated with it a Saved Program Status Register, or SPSR. This register is used to save the state of the CPSR (Current Program Status Register) when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed.

EXCEPTIONS

Exceptions are usually used to handle unexpected events which arise during the execution of a program, such as interrupts or memory faults, also cover software interrupts, undefined instruction traps, and the system reset
Three groups:
– Exceptions generated as the direct effect of execution an instruction (Ex: Software interrupts, undefined instructions, and prefetch abort)
– Exceptions generated as a side effect of an instruction (Data aborts)
– Exceptions generated externally (Reset, IRQ and FIQ)

Exception Entry
When an exception arises, ARM completes the current instruction as best it can (except that reset exception terminates the current instruction immediately) and then departs from the current instruction sequence to handle the exception which starts from a specific location (exception vector).
Processor performs the following sequence:
– Change to the operating mode corresponding to the particular exception
– Save the address of the instruction following the exception entry instruction in r14 of the new mode
– Save the old value of CPSR in the SPSR of the new mode
– Disable IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disable further faster interrupt by setting bit 6 of the CPSR
– Force the PC to begin execution at the relevant vector address

- Normally the vector address contains a branch to the relevant routine
- Two banked registers in each of the privilege modes are used to hold the return address and stack pointer.

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

Exception Return

Once the exception has been handled, the user task is normally resumed. The sequence is

– Any modified user registers must be restored from the handler's stack

– CPSR must be restored from the appropriate SPSR

– PC must be changed back to the relevant instruction address

The last two of these steps cannot be carried out independently. If the CPSR is restored first, the banked r14 holding the return address is no longer accessible; if the PC is restored first, the exception handler loses control of the instruction stream and cannot cause the restoration of the CPSR to take place.

Exception Priorities

Priority order

– Reset (highest priority)

– Data abort

– FIQ

– IRQ

– Prefetch abort

– SWI, undefined instruction

The most complex exception scenario is where an FIQ, an IRQ and a third exception (which is not Reset) happen simultaneously. FIQ has higher priority than IRQ and also masks it out, so the IRQ will be ignored until the FIQ handler explicitly enables IRQ or returns to the user code.

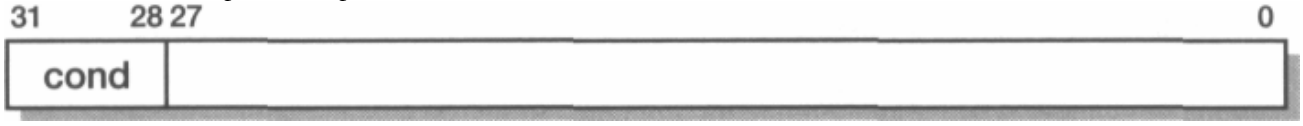
If the third exception is a data abort, the processor will enter the data abort handler and then immediately enter the FIQ handler, since data abort entry does not mask FIQs out. The data abort is 'remembered' in the return path and will be processed when the FIQ handler returns

Address Exceptions

It includes the use of all of the first eight word locations in memory as exception vector addresses apart from address 0x00000014.These traps were referred to as 'address exceptions'.

Conditional Execution

A unusual feature of the ARM instruction set is that every instruction is conditionally executed. The condition field occupies the top four bits of the 32-bit instruction field:



The ARM condition code field.

Each of the 16 values of the condition field causes the instruction to be executed or skipped according to the values of the N, Z, C and V flags in the CPSR. The condition is specified with a two-letter suffix, such as EQ or CC, appended to the mnemonic. The condition is tested against the current processor flags and if not met the instruction is treated as a no-op. This feature often removes the need to branch, avoiding pipeline stalls and increasing speed.

The Condition Codes:

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	Any
1111	NV	Never	None

The 'always' condition (AL) may be omitted since it is the default condition that is assumed if no other condition is specified. The 'never' condition (NV) should not be used.

Branch,Branch with link Instructions

Branch and Branch with Link instructions cause a switch in the sequence of instruction execution. The ARM normally executes instructions from sequential word addresses in memory, using conditional execution to skip over individual instructions where required. Whenever the program deviates from sequential execution a control flow instruction is used to modify the program counter.
BL is used to perform a subroutine call, with the return being caused by copying the link register back into the PC.

Both forms of the instruction may be executed conditionally or unconditionally.

Syntax

B{*cond*} label

BL{*cond*} label

Unconditional jump: Ex:

```
      B      LABEL
```

```
                        ; unconditional jump ..
          ..
  LABEL   ..            ; .. to here
```

To execute a loop ten times:

```
          MOV     r0, #10 ; initialize loop counter
  LOOP    ..
          SUBS    r0, #1  ; decrement counter setting CCs
          BNE     LOOP    ; if counter <> 0 repeat loop..
          ..              ; .. else drop through
```

To call a subroutine:

```
          ..
          BL      SUB     ; branch and link to subroutine SUB
          ..              ; return to here
          ..
  SUB     ..              ; subroutine entry point
          MOV     PC, r14 ; return
```

Conditional subroutine call:

```
..
CMP     r0, #5   ; if r0 < 5
BLLT    SUB1     ; then call SUB1
BLGE    SUB2     ; else call SUB2
..
```

Branch with Exchange,Branch link with exchange(BX, and BLX)

1: BX | BLX Rm
2: BLX <target address>

where:

'<target address>' is normally a label in the assembler code

• If Rm[0] is 1, the processor switches to execute Thumb instructions
• If Rm[0] is 0, the processor continues executing ARM instructions

These instructions are available on ARM chips which support the Thumb (16-bit) instruction set, and are a mechanism for switching the processor to execute Thumb instructions or for returning symmetrically to ARM and Thumb calling routines. A similar Thumb instruction causes the processor to switch back to 32-bit ARM instructions.
Example:

An unconditional jump:

```
        BX      r0      ; branch to address in r0,
                        ; enter Thumb state if r0[0] = 1
```

A call to a Thumb subroutine:

```
        CODE32          ; ARM code follows
        ..
        BLX     TSUB    ; call Thumb subroutine
         ..
         CODE16             ; start of Thumb code
TSUB     ..                 ; Thumb subroutine
         BX      r14        ; return to ARM code
```

Software Interrupt (SWI)

The software interrupt instruction is used for calls to the operating system and is often called a 'supervisor call'. It puts the processor into supervisor mode .

```
        SWI{<cond>}     <24-bit immediate>
```

To output the character 'A':

```
        MOV     r0, #'A'            ; get 'A' into r0..
        SWI     SWI_WriteC          ; .. and print it
```

A subroutine to output a text string following the call:

```
        ..
        BL      STROUT              ; output following message
        =       "Hello World",&0a,&0d,0
        ..                          ; return to here
        ..
STROUT  LDRB    r0, [r14], #1       ; get character
        CMP     r0, #0              ; check for end marker
```

The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code.
If the condition is passed the instruction enters supervisor mode using the standard ARM exception entry sequence. In detail, the processor actions are:
1. Save the address of the instruction after the SWI in r14_svc.
2. Save the CPSR in SPSR_svc.
3. Enter supervisor mode and disable IRQs (but not FIQs) by setting CPSR[4:0] to 10011_2 and CPSR[7] to l.
4. Set the PC to 08]6 and begin executing the instructions there.

To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc.

Data processing instructions

The ARM data processing instructions are used to modify data values in registers.The operations that are supported include arithmetic and bit-wise logical combinations of 32-bit data types. The ARM data processing instructions employ a 3-address format, which means that the two source operands and the destination register are specified independently. One source operand is always a register; the second may be a register, a shifted register or an immediate value. The shift applied to the second operand, if it is a register, may be a logical or arithmetic shift or a rotate, and it may be by an amount specified either as an immediate quantity or by a fourth register. The operations that may be specified are listed below. When the instruction does not require all the available operands (for instance MOV ignores Rn and CMP ignores Rd) the unused register field should be set to zero. The assembler will do this automatically.

ARM data processing instructions.

Opcode 124:21)	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd:=Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn 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 Sccon	RnANDOp2
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 OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd:=RnANDNOTOp2
1111	MVN	Move negated	Rd:=NOTOp2

These instructions allow direct control of whether or not the processor's condition codes are affected by their execution through the S bit (bit 20). When clear, the condition codes will be unchanged; when set (and Rd is not r15; see below):

• The N flag is set if the result is negative, otherwise it is cleared (that is, N equals bit 31 of the result).
• The Z flag is set if the result is zero, otherwise it is cleared.
• The C flag is set to the carry-out from the ALU when the operation is arithmetic (ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN) or to the carry-out from the shifter otherwise. If no shift is required, C is preserved.
• The V flag is preserved in non-arithmetic operations. It is set in an arithmetic operation if there is an overflow from bit 30 to bit 31 and cleared if no overflow occurs. It has significance only when an arithmetic operation has operands that are viewed as 2's complement signed values, and indicates a result that is out of range.
These instructions may be used to multiply a register by a small constant. Example is given below:

A subroutine to multiply r0 by 10:

```
                MOV  R0,  #3
                BL    TIMES10
                ……..
TIMES10         MOV  R0, R0, LSL   #1        ;R0=R0 x 2^1
                ADD   R0, R0, R0, LSL  #2    ;R0=R0 + R0 x 2^2
                MOV  PC, R14
```

Multiply instructions

ARM multiply instructions produce the product of two 32-bit binary numbers held in registers. The result of multiplying two 32-bit binary numbers is a 64-bit product. Some forms of the instruction, available only on certain versions of the processor, store the full result into two independently specified registers; other forms store only the least significant 32 bits into a single register. In all cases there is a multiply-accumulate variant that adds the product to a running total and both signed and unsigned operands may be used. The least significant 32 bits of the result are the same for signed and

unsigned operands, so there is no need for separate signed and unsigned versions of the 32-bit result instructions.
The S bit controls the setting of the condition codes as with the other data processing instructions. When it is set in the instruction:
• The N flag is set to the value of bit 31 of Rd for the variants which produce a 32-bit result, and bit 31 of RdHi for the long forms.
• The Z flag is set if Rd or RdHi and RdLo are zero.
• The C flag is set to a meaningless value.
• The V flag is unchanged.
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

• 'RdHi : RdLo' is the 64-bit number formed by concatenating RdHi (the most significant 32 bits) and RdLo (the least significant 32 bits).'[31:0]' selects only the least significant 32 bits of the result.
• Simple assignment is denoted by ':='.
• Accumulation (adding the right-hand side to the left) is denoted by '+='.

Instructions that produce the least significant 32 bits of the product:

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

The following instructions produce the full 64-bit result:
<mul>{<cond>}{S}RdHi, RdLo, Rm, Rs where <mul> is
one of the 64-bit multiply types (UMULL, UMLAL, SMULL, SMLAL).

To form a scalar product of two vectors:

```
        MOV     rll , #20           ; initialize loop counter
        MOV     rlO , #0            ;initialize total
LOOP    LDR     r0 , [r8] , #4      ;get first component..
        LDR     r1 , [r9] , #4 .    ; .and second
        MLA     rlO , r0 , r1 , rlO ; accumulate product
        SUBS    rll , rll , #1      ;decrement loop counter
        BNE     LOOP
```

Count leading zeros (CLZ - architecture v5T only)

This instruction is only available on ARM processors that support architecture v5T. It is useful for renormalizing numbers, and performs its functions far more efficiently than can be achieved using other ARM instructions.

The instruction sets Rd to the number of the bit position of the most significant 1 in Rm. If Rm is zero Rd will be set to 32.

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



MOV     r0, #&100
CLZ     r1, r0              ; r1 := 8
```

Single Word and Unsigned Byte Data Transfer Instructions

These instructions are the most flexible way to transfer single bytes or words of data between ARM's registers and memory. Transferring large blocks of data is usually better done using the multiple register transfer instructions, and recent ARM processors also support instructions for transferring half-words and signed bytes.
Provided that a register has been initialized to point somewhere near (usually within 4 Kbytes of) the required memory address, these instructions provide an efficient load and store mechanism with a relatively rich set of addressing modes which includes immediate and register offsets, auto-indexing and PC-relative.
A pre-indexed (P = 1) addressing mode uses the computed address for the load or store operation, and then, when write-back is requested (W =1), updates the base register to the computed value.
A post-indexed (P = 0) addressing mode uses the unmodified base register for the transfer and then updates the base register to the computed address irrespective of the W bit.

The pre-indexed form of the instruction:
LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}
The post-indexed form:
LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>
A useful PC-relative form that leaves the assembler to do all the work:
LDR|STR{<Cond>}{B} Rd, LABEL

Example

```
        LDR     r1, UARTADD         ; UART address into r1
        STRB    r0, [r1]            ; store data to UART
        ..
UARTADD &       &1000000            ; address literal
```

Half-word and Signed Byte Data Transfer Instructions

These instructions are not supported by some early ARM processors. The addressing modes available with these instructions are a subset of those available with the unsigned byte and word forms.
These instructions are very similar to the word and unsigned byte forms described in the previous section, but here the immediate offset is limited to eight bits and the scaled register offset is no longer available.

S	H	Data type
1	0	Signed byte
0	1	Unsigned half-word
1	1	Signed half-word

Pre-indexed form

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

Post-indexed form

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

Where <offset> is # +/-<8-bit immediate> or +/- Rm and H|SH|SB selects the data type otherwise the assembler format is for words and unsigned byte transfer

Example

Expand an array of signed half-words into an array of words

```
            ADR     r1, ARRAY1      ;half-word array start
            ADR     r2, ARRAY2      ;word array start
            ADR     r3, ENDARR1     ;ARRAY1 end + 2
Loop        LDRSH   r0, [r1], #2    ;get signed half-word
            STR     r0, [r2], #4    ;save word
            CMP     r1, r3          ;check for end of array
            BLT     Loop            ;if not finished, loop
```

Multiple Register Transfer

The ARM multiple register transfer instructions allow any subset (or all) of the 16 registers visible in the current operating mode to be loaded from or stored to memory.

A form of the instruction also allows the operating system to load or store the user-mode registers to save or restore the user process state, and another form allows the CPSR to be restored from the SPSR as part of a return from an exception handler.

These instructions are used on procedure entry and return to save and restore workspace registers and are useful for high-bandwidth memory block copy routines.

The normal form of the instruction is:

LDM | STM {<cond>} {B} <add mode> Rn{!}, <register>

– <add mode> specifies one of the addressing modes

– '!': auto-indexing

–<registers> a list of registers, e.g., {r0, r3-r7, pc}

In non-user mode, the CPSR may be restored by

LDM {<cond>} <add mode> Rn{!}, <registers + PC>
The register list must contain the PC.

In non-user mode, the user registers may be saved or restored by
LDM | STM {<cond>} <add mode> Rn, <registers - PC>
Here the register list must not contain PC and write-back is not allowed.

Example:
To save 3 work registers and the return address upon entering a subroutine (assume r13 has been initialized for use as a stack pointer)

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

ToRestore the work registers and return

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

Swap Memory and Register Instructions

Swap instructions combine a load and a store of a word or an unsigned byte in a single instruction. Normally the two transfers are combined into an atomic memory operation that cannot be split by an external memory access (for instance from a DMA controller), and therefore the instruction can be used as the basis of a semaphore mechanism to give mutually exclusive access to data structures that are shared between multiple processes, processors, or a processor and a DMA controller. These instructions are little used outside their role in the construction of semaphores.

The instruction loads the word (B = 0) or unsigned byte (B = 1) at the memory location addressed by Rn into Rd, and stores the same data type from Rm into the same memory location. Rd and Rm may be the same register (but should both be distinct from Rn), in which case the register and memory values are exchanged. The ARM executes separate memory read and then memory write cycles, but asserts a 'lock' signal to indicate to the memory system that the two cycles should not be separated.

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

Example

```
ADR    r0, SEMAPHORE
SWPB   r1, r1, [r0]          ;exchange byte
```

1. The PC should not be used as any of the registers in this instruction.
2. The base register (Rn) should not be the same as either the source (Rm) or the destination (Rd) register.

Status Register to General Register Transfer instructions

When it is necessary to save or modify the contents of the CPSR or the SPSR of the current mode, those contents must first be transferred into a general register, the selected bits modified and then the value returned to the status register. These instructions perform the first step in this sequence.

MRS{<cond>} Rd, CPSR | SPSR

The CPSR(R=0) or the current mode SPSR(R=1) is copied into the destination register(R_d). All 32 bits are copied.

Example

```
MRS   r0, CPSR
MRS   r3, SPSR
```

General Register to Status Register Transfer instructions

When it is necessary to save or modify the contents of the CPSR or the SPSR of the current mode, those contents must first be transferred into a general register, the selected bits modified and then the value returned to the status register. These instructions perform the last step in this sequence.

The operand, which may be a register (Rm) or a rotated 8-bit immediate (specified in the same way as the immediate form of operand2 in the data processing instructions), is moved under a field mask to the CPSR (R = 0) or current mode SPSR (R = I). The field mask controls the update of the four byte fields within the PSR register. Instruction bit 16 determines whether PSR[7:0] is updated, bit 17 controls PSR[15:8], bit 18 controls PSR[23:16] and bit 19 controls PSR[31:24]. When an immediate operand is used only the flags (PSR[31:24]) may be selected for update. (These are the only bits that may be updated by user-mode code.)

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

MSR{<cond>} CPSR_<field> | SPSR_<field>,Rm

Where <field> is one of

- c – the control field PSR[7:0]
- x – the extension field PSR[15:8]
- s – the status field PSR[23:16]
- f – the flag field PSR[31:24]

Example

– To set N, Z, C, V flags

- MSR CPSR_f, #&f0000000 ;set all the flags

ARM coprocessor instructions

The ARM architecture supports a general mechanism for extending the instruction set through the addition of coprocessors. The most common use of a coprocessor is the system coprocessor used to control on-chip functions such as the cache and memory management unit on the ARM720. A floating-point ARM coprocessor has also been developed, and application-specific coprocessors are a possibility. ARM coprocessors have their own private register sets and their state is controlled by instructions that mirror the instructions that control ARM registers. The ARM has sole responsibility for control flow, so the coprocessor instructions are concerned with data processing and data transfer. Following RISC load-store architectural principles, these categories are cleanly separated. The instruction formats reflect this:

• Coprocessor data operations are completely internal to the coprocessor and cause a state change in the coprocessor registers. An example would be floating-point addition, where two registers in the floating-point coprocessor are added together and the result placed into a third register.

• Coprocessor data transfer instructions load or store the values in coprocessor registers from or to memory. Since coprocessors may support their own data types, the number of words transferred for

each register is coprocessor dependent. The ARM generates the memory address, but the coprocessor controls the number of
words transferred. A coprocessor may perform some type conversion as part of the transfer (for instance the floating-point coprocessor converts all loaded values into its 80-bit internal representation).
• In addition to the above, it is sometimes useful to move values between ARM and coprocessor registers. Again taking the floating-point coprocessor as an illustration, a 'FIX' instruction takes a floating-point value from a coprocessor register, converts it to an integer, and moves the integer into an ARM register. A
floating-point comparison produces a result which is often needed to affect control flow, so the result of the compare must be moved to the ARM CPSR.

Coprocessor data operations

These instructions are used to control internal operations on data in coprocessor registers. The standard format follows the 3-address form of ARM's integer data processing instructions, but other interpretations of all the coprocessor fields are possible.
The ARM offers this instruction to any coprocessors that may be present. If it is accepted by one of them the ARM proceeds to the next instruction; if it is not accepted the ARM takes the undefined instruction trap.
Normally the coprocessor identified with the coprocessor number CP# will accept the instruction and perform the operation denned by the Copl and Cop2 fields, using CRn and CRm as the source operands and placing the result in CRd.

CDP{<cond>} <CP#>, <Copl>, CRd, CRn, CRm{, <Cop2>}

Example:

CDP p2, 3, CO, Cl, C2

CDPEQ p3, 6, Cl, C5, C7, 4

Coprocessor data transfers

The coprocessor data transfer instructions are similar to the immediate offset forms of the word and unsigned byte data transfer instructions described earlier, but with the offset limited to eight bits rather than 12.Auto-indexed forms are available, with pre- and post-indexed addressing.

The instruction is offered to any coprocessors which may be present; if none accepts it ARM takes the undefined instruction trap and may use software to emulate the coprocessor. Normally the coprocessor with coprocessor number CP#, if present, will accept the instruction.

The address calculation takes place within the ARM, using an ARM base register (Rn) and an 8-bit immediate offset which is scaled to a word offset by shifting it left two bit positions. The addressing mode and auto-indexing are controlled in the same way as the ARM word and unsigned byte transfer instructions. This defines the first transfer address; subsequent words are transferred to or from incrementing word addresses.

The data is supplied by or received into a coprocessor register (CRd), with the number of words transferred being controlled by the coprocessor and the N bit selecting one of two possible lengths.

The pre-indexed form:

LDC|STC{<cond>}{L} <CP#>, CRd, [Rn, <offset>]

{ ! } **The post-indexed form:**

LDCISTC{<cond>}{L} <CP#>, CRd, [Rn], <offset>

In both cases LDC selects a load from memory into the coprocessor register, STC selects a store from the coprocessor register into memory. The L flag, if present, selects the long data type (N= 1). <of fset> is # + /-<8-bit immediate>

LDC p6, CO, [r1]

STCEQL p5, Cl, [r0] , #4

During the data transfer the ARM will not respond to interrupt requests, so coprocessor designers should be careful not to compromise the system interrupt response time by allow ing very long data transfers. Limiting the maximum transfer length to 16 words will ensure that coprocessor data transfers take no longer than worst-case load and store multiple register instructions.

Coprocessor register transfers

These instructions allow an integer generated in a coprocessor to be transferred directly into a ARM register or the ARM condition code flags. Typical uses are:

• A floating-point FIX operation which returns the integer to an ARM register;

• A floating-point comparison which returns the result of the comparison directly to the ARM condition code flags where it can determine the control flow;

• A FLOAT operation which takes an integer value from an ARM register and sends it to the coprocessor where it is converted to floating-point representation and placed in a coprocessor register.

The system control coprocessors used to control the cache and memory management functions on the more complex ARM CPUs generally use these instructions to access and modify the on-chip control registers.

The instruction is offered to any coprocessors present; normally the coprocessor with coprocessor number CP# will accept the instruction. If no coprocessor accepts the instruction ARM raises an undefined instruction trap. If a coprocessor accepts a load from coprocessor instruction, it will normally perform an operation defined by Copl and Cop2 on source operands CRn and CRm and return a 32-bit integer result to the ARM which will place it in Rd.

If a coprocessor accepts a store to coprocessor instruction, it will accept a 32-bit integer from the ARM register Rd and do something with it.

If the PC is specified as the destination register Rd in a load from coprocessor instruction, the top four bits of the 32-bit integer generated by the coprocessor are placed into the N, Z, C and V flags in the CPSR.

Move to ARM register from coprocessor:

MRC{<cond>} <CP#>, <Copl>, Rd, CRn, CRm{,<Cop2>}

Move to coprocessor from ARM register:

MCR{<cond>} <CP#>, <Copl>, Rd, CRn, CRm{,<Cop2>}

MCR p14, 3, r0, Cl, C2

MRCCS p2, 4, r3, C3, C4, *6*

1. The Copl, CRn, Cop2 and CRm fields are interpreted by the coprocessor.

2. Where the coprocessor must perform some internal work to prepare a 32-bit value for transfer to the ARM (for example, a floating-point FIX operation has to con vert the floating-point value into its equivalent fixed-point value), this must take place before the coprocessor commits to the transfer. Therefore it will often be necessary for the coprocessor handshake to 'busy-wait' while the data is prepared. The ARM can take interrupts during the busy-wait period, and if it does get interrupted it will break off from the handshake to service the interrupt. It will probably retry the coprocessor instruction when it returns from the interrupt service routine, but it may not; the interrupt may cause a task switch, for example. In either case, the coprocessor must give consistent results. Therefore the preparation work carried out before the handshake commit phase must not change the coprocessor's visible state.
3. Transfers from the ARM to the coprocessor are generally simpler since any data conversion work can take place in the coprocessor after the transfer has completed.

Breakpoint instruction (BKPT - architecture v5T only)

Breakpoint instructions are used for software debugging purposes; they cause the processor to break from normal instruction execution and enter appropriate debugging procedures.
This instruction causes the processor to take a prefetch abort when the debug hardware unit is configured appropriately.

Assembler format: BKPT
Example: BKPT

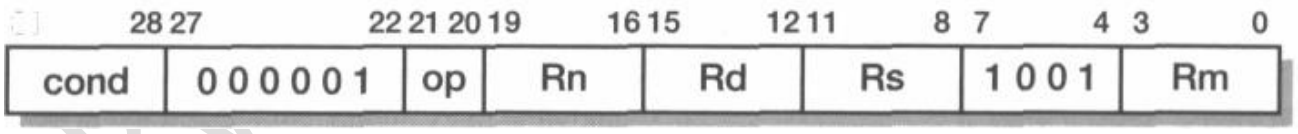
1. Only processors that implement ARM architecture v5T support the BRK instruct tion
2. BRK instructions are unconditional - the condition field must contain the 'ALWAYS' code.

Unused instruction space

Not all of the 2^{32} instruction bit encodings have been assigned meanings; the encodings that have not been used so far are available for future instruction set extensions. The unused instruction encodings each fall into particular gaps left in the used encodings, and their likely future use can be inferred from where they lie.

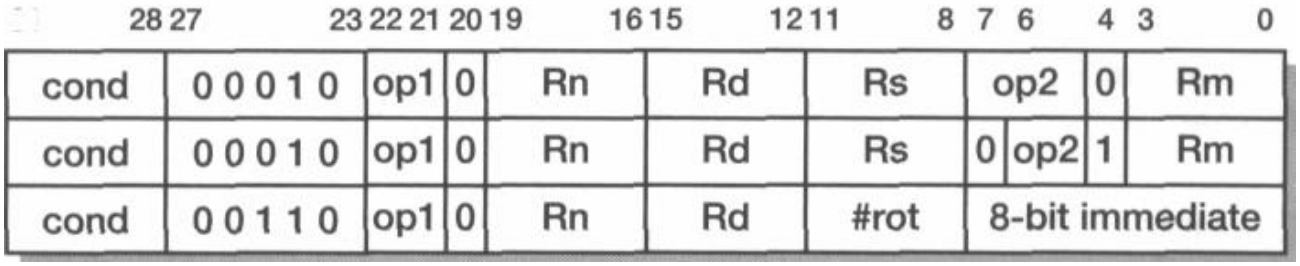
Unused arithmetic instructions

These instructions look very like the multiply instructions. This would be a likely encoding, for example, for an integer divide instruction.



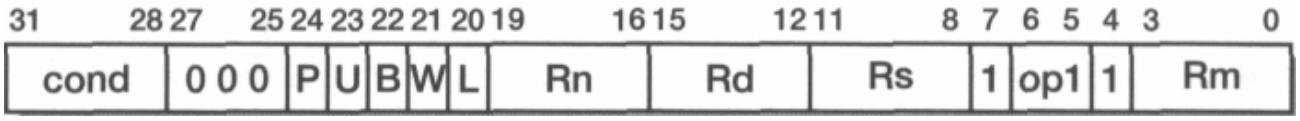
Unused control instructions

These instructions include the branch and exchange instructions. The gaps here could *be* used to encode other instructions that affect the processor operating mode.



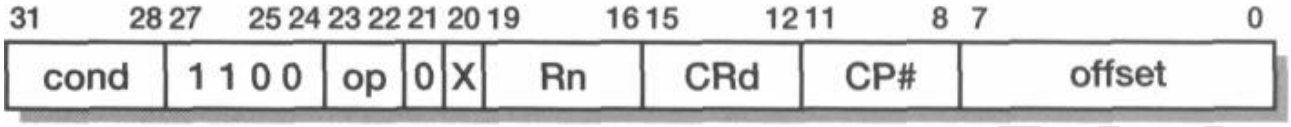
Unused load/store instructions

There are unused encodings in the areas occupied by the swap instructions. These are likely to be used to support additional data transfer instructions, should these be required in the future.



Unused Coprocessor instructions

The following instruction format is similar to the coprocessor data transfer instruction and is likely to be used to support any additional coprocessor instructions that may be required:



Undefined instruction space

The largest area of undefined instructions looks like the word and unsigned byte data transfer instruction. However the future options on this space are being kept completely open.

Memory faults

ARM processors allow the memory system (or, more usually, the memory management unit) to fault on any memory access. What this means is that instead of returning the requested value from memory, the memory system returns a signal that indicates that the memory access has failed to complete correctly. The processor will then enter an exception handler and the system software will attempt to recover from the problem. The most common sources of a memory fault in a general-purpose machine are:

Page Absent

• The addressed memory location has been paged out to disk.

In a virtual memory system infrequently used pages are held on disk. An attempt to access instructions or data on such a page will fail, causing the MMU to abort the access. The system software must identify the cause of the abort, fetch the required page into memory from disk, change the translation tables in the MMU accordingly and retry the aborted access. Since fetching a page from disk is a slow process, the operating system will often switch out the faulted process and schedule another task while the transfer takes place.

Page protected

- The addressed memory location is temporarily inaccessible.

When a page is loaded into memory, the operating system may initially make it read only. An attempt to write the page will fault, alerting the operating system to the fact that the page has been modified and must be saved when it is swapped out to disk again. Some operating systems will periodically make pages inaccessible in order to generate statistics about their use for the paging algorithm.

Soft memory errors

- A soft error has been detected in the memory.

A large memory system has a not-insignificant error rate due to alpha particle radiation changing the state of a dynamic RAM storage cell. Where the memory system has simple error detection (such as a parity check) the fault is not recoverable so the faulting process must be terminated. Where the

memory system has full error check and correct (ECC) hardware the processor will usually be unaware of the error, though a fault could still be generated in order that the operating system can accumulate statistics on the memory error rate. In the intermediate case, where the memory has a hardware error detector but relies on software error correction, the fault, correct and retry sequence is followed.

Embedded systems

In a typical small embedded ARM application, a hard disk is usually unavailable, and in any case paging to disk is usually incompatible with the real-time constraints that the system must meet. Furthermore the memory system is usually small (a few megabytes at most, comprising a handful of memory chips) so the soft error rate is negligible and error detection is rarely incorporated. Therefore many embedded systems will not use memory faults at all.

A typical use in an embedded system might be to store a library of routines in compressed form in ROM and to use the virtual memory technique to trap calls to individual routines, expanding them as required into RAM for execution. The benefit of storing them in compressed form is the reduction in the size and cost of the ROM; the penalty is the time taken for decompression. An additional use in an embedded system might be to offer some protection for processes running under a real-time operating system.

Memory faults

The ARM handles memory faults detected during instruction fetches *(prefetch aborts)* and those detected during data transfers (*data aborts)* separately.

Prefetch aborts

If an instruction fetch faults, the memory system raises the abort signal (a dedicated input to the processor) and returns a meaningless instruction word. Internally ARM puts the meaningless instruction into the instruction pipeline along with the abort flag, and then continues with business as usual until the instruction enters the decode stage, whereupon the abort flag overrides the instruction and causes the decoder to generate an exception entry sequence using the prefetch abort vector. If the aborted instruction does not get executed, for instance because it was fetched immediately after a branch instruction that was ultimately taken, then no exception is raised and the fault is ignored.

Data aborts

Memory faults which arise during an access to memory for a data value are far more complex to handle. The memory system need not differentiate the instruction and data cases; it simply raises the abort input when it sees an address that it can't handle. The processor has to work much harder in response to a data abort, however, since this is a problem with the instruction that is currently executing whereas a prefetch abort is a problem with an instruction that has not yet entered decode. Since the objective, in some cases, is to retry the instruction when the cause of the fault has been resolved, the instruction should do its best to ensure that its state (that is, its register values) after an abort is unchanged from its state before it started executing. Failing that, it should at least ensure that enough state can be recovered so that after the instruction has been executed a second time its state is the same as it would have been had the instruction completed the first time.

LDM data abort

Let us consider a load multiple instruction with 16 registers in the register list using r0 as the base register. Initially the addresses are fine, so the loading begins. The first data value to arrive overwrites r0, then successive registers get written until the final address (destined for the PC) crosses a page boundary and faults. Most of the processor state has been lost. Fortunately the processor has kept a copy of the base register value (possibly after auto-indexing) in a dark corner while the instruction was proceeding, so the last act of the instruction, when it should have been changing the PC to the new value had the PC access not faulted, is instead to copy this preserved value back into the base register.

So we have preserved the PC and the (modified) base register, but we have overwritten several other registers in the meantime. The base register modification can be reversed by software, since we can inspect the instruction and determine the number of registers in the list and the addressing mode. The overwritten registers are exactly those that we will load again with the correct values when we retry the instruction.

The requirement to recover from data aborts was added fairly late in the development of the first ARM processor chip. Up to that point, the various load and store multiple addressing modes had been implemented starting from the base address and incrementing up or decrementing down memory according to the mode. The chip was therefore designed with an address incrementer/decrementer unit.When it became clear that support for virtual memory was needed it was rapidly seen that the decrementing mode made abort recovery much harder, since the PC could be overwritten before the abort was signalled. Therefore the implementation was changed always to increment the address. The memory addresses used were the same, and the mapping of register to memory was unchanged); just the order of the transfer was changed to lowest address first, PC last.This change was implemented too late to affect the layout of the address generating logic, so the first ARM silicon has an address incrementer/decrementer hard-wired always to increment. Needless to say, this redundancy was not carried forward to subsequent implementations.

Abort timing

The earlier a processor gets an indication of a fault from the memory system, the better placed it is to preserve state. The earlier a processor requires the fault signal, the harder the memory system is to design. There is therefore a tension between the architectural simplicity of the processor's fault handling and the engineering efficiency of the memory system.

In order to ease the constraints on the cache and MMU designs, later ARMs were redesigned to allow aborts to be flagged at the end of the cycle, with a similar timing to the read data. The compromise that had to be accepted was that now the processor state has changed further so there is more work for the abort recovery software to do.

ARM data aborts

The state of the ARM after a data abort depends on the particular processor and, with some processors, on the early/late abort configuration:

• In all cases the PC is preserved (so on data abort exception entry r14_abt con tains the address of the faulting instruction plus eight bytes).

• The base register will either be unmodified, or will contain a value modified by auto-indexing (it will not be overwritten by a loaded value).

• Other load destination registers may have been overwritten, but the correct value will be loaded when the instruction is retried.

Because the base register may be modified by auto-indexing, certain auto-indexing modes should be avoided. For example:

```
LDR r0, [r1], rl
```

This instruction uses r1 as the address for the load, then uses post-indexing to add r1 to itself, losing the top bit in the process. If, following a data abort, only the modified value of r1 is available, it is not possible to recover the original transfer address. In general, using the same register for the base and the index in an addressing mode should be avoided.

ARM architecture variants

The ARM architecture has undergone a number of revisions in the course of its development. The various architecture versions are described below.

Version 1
ARM architecture version 1 describes the first ARM processor, developed at Acorn Computers Limited between 1983 and 1985. These first ARM chips supported only 26-bit addressing and had no multiply or coprocessor support. Their only use in a product was in the ARM second processor attachments to the BBC microcomputer; these were made in very small numbers, but established the ARM as the first commercially exploited single-chip RISC microprocessor. They were also used internally within Acorn in prototypes of the Archimedes personal workstation.

Version 2
The ARM2 chip was sold in volume in the Acorn Archimedes and A3000 products. It was still a 26-bit address machine, but included the 32-bit result multiply instructions and coprocessor support. ARM2 employs the architecture that ARM Limited now calls ARM architecture version 2.

Version 2a
The ARM3 chip was the first ARM with an on-chip cache. The architecture was very similar to version 2, but added the atomic load and store (SWP) instruction and introduced the use of coprocessor 15 as the system control coprocessor to manage the cache.

Version 3
The first ARM processor designed by ARM Limited following their establishment as a separate company in 1990 was the ARM6, sold as a macrocell, a stand-alone processor (the ARM60) and as an integrated CPU with an on-chip cache, MMU and write buffer (the ARM600, and the ARM610 used in the Apple Newton). The ARM6 introduced ARM architecture version 3, which had 32-bit addressing, separate CPSR and SPSRs, and added the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode.
ARM architecture version 3 is backwards compatible with version 2a, allowing either hard-wired 26-bit operation or process-by-process mixed 26- and 32-bit operation.

Version 3G
ARM architecture version 3G is version 3 without backwards compatibility to version 2a.

Version 3M
ARM architecture version 3M introduces the signed and unsigned multiply and multiply-accumulate instructions that generate the full 64-bit result.

Version 4
Version 4 of the architecture adds the signed and unsigned half-word and signed byte load and store instructions and reserves some of the S WI space for architecturally defined operations. The system mode (a privileged mode that uses the user registers) is introduced, and several unused corners of the instruction space are trapped as undefined instructions. At this stage those uses of r15 which yielded 'pc + 12' in earlier ARMs are declared to give unpredictable results (so architecture version 4 compliant implementations need not reproduce the 'pc + 12' behaviour). This is the first architecture version to have a full formal definition.

Version 4T
The 16-bit Thumb compressed form of the instruction set is introduced in version 4T of the architecture.

Version 5T
Version 5T of the ARM architecture has been introduced recently, and at the time of writing is supported only by ARM10 processors (and these will soon support v5TE). It is a superset of architecture version 4T, adding the BLX, CLZ and BRK instructions.

Version 5TE
Version 5TE adds the signal processing instruction set extensions.