



The ARM Instruction Set



The ARM Instruction Set


The ARM Instruction Set - ARM University Program - V1.0



Main features of the ARM Instruction Set

- * All instructions are 32 bits long.
- * Most instructions execute in a single cycle.
- * Every instruction can be conditionally executed.
- * A load/store architecture
 - Data processing instructions act only on registers
 - Three operand format
 - Combined ALU and shifter for high speed bit manipulation
 - Specific memory access instructions with powerful auto-indexing addressing modes.
 - 32 bit and 8 bit data types
 - and also 16 bit data types on ARM Architecture v4.
 - Flexible multiple register load and store instructions
- * Instruction set extension via coprocessors


The ARM Instruction Set - ARM University Program - V1.0



Processor Modes

- * The ARM has six operating modes:
 - *User* (unprivileged mode under which most tasks run)
 - *FIQ* (entered when a high priority (fast) interrupt is raised)
 - *IRQ* (entered when a low priority (normal) interrupt is raised)
 - *Supervisor* (entered on reset and when a Software Interrupt instruction is executed)
 - *Abort* (used to handle memory access violations)
 - *Undef* (used to handle undefined instructions)
- * ARM Architecture Version 4 adds a seventh mode:
 - *System* (privileged mode using the same registers as user mode)


The ARM Instruction Set - ARM University Program - V1.0



The Registers

- * ARM has 37 registers in total, all of which are 32-bits long.
 - 1 dedicated program counter
 - 1 dedicated current program status register
 - 5 dedicated saved program status registers
 - 30 general purpose registers
- * However these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access
 - a particular set of r0-r12 registers
 - a particular r13 (the stack pointer) and r14 (link register)
 - r15 (the program counter)
 - cpsr (the current program status register)
- and privileged modes can also access
 - a particular spsr (saved program status register)

The ARM Instruction Set - ARM University Program - V1.0



Register Organisation


General registers and Program Counter

User/System	FIQ/32	Supervisor/32	Abort/32	IRQ/32	Undef/32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13(spsr)	r13_fiq	r13_spsr	r13_spsr	r13_spsr	r13_spsr
r14(lr)	r14_fiq	r14_spsr	r14_spsr	r14_spsr	r14_spsr
r15(pc)	r15_fiq	r15_spsr	r15_spsr	r15_spsr	r15_spsr

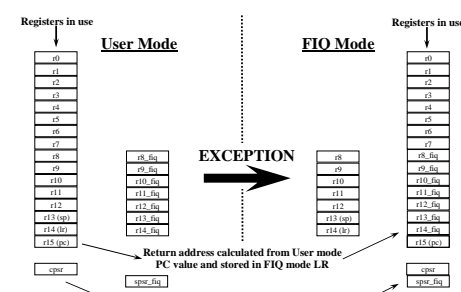
Program Status Registers

	cpsr	spsr	spsr_fiq	spsr_svr	spsr_und	spsr_irq	spsr_fiq
cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
spsr	spsr	spsr	spsr	spsr	spsr	spsr	spsr
spsr_fiq	spsr_fiq	spsr_fiq	spsr_fiq	spsr_fiq	spsr_fiq	spsr_fiq	spsr_fiq
spsr_svr	spsr_svr	spsr_svr	spsr_svr	spsr_svr	spsr_svr	spsr_svr	spsr_svr
spsr_und	spsr_und	spsr_und	spsr_und	spsr_und	spsr_und	spsr_und	spsr_und
spsr_irq	spsr_irq	spsr_irq	spsr_irq	spsr_irq	spsr_irq	spsr_irq	spsr_irq


The ARM Instruction Set - ARM University Program - V1.0



Register Example: User to FIQ Mode



The ARM Instruction Set - ARM University Program - V1.0



The ARM Instruction Set

Accessing Registers using ARM Instructions

- * **No breakdown of currently accessible registers.**
 - All instructions can access r0-r14 directly.
 - Most instructions also allow use of the PC.
- * **Specific instructions to allow access to CPSR and SPSR.**
- * **Note : When in a privileged mode, it is also possible to load / store the (banked out) user mode registers to or from memory.**
 - See later for details.

The ARM Instruction Set - ARM University Program - V1.0



7

The Program Status Registers (CPSR and SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

- * **Condition Code Flags**
 - N = Negative result from ALU flag.
 - Z = Zero result from ALU flag.
 - C = ALU operation Carried out
 - V = ALU operation oVerflowed
- * **Interrupt Disable bits.**
 - I = 1, disables the IRQ.
 - F = 1, disables the FIQ.
- * **T Bit (Architecture v4T only)**
 - T = 0, Processor in ARM state
 - T = 1, Processor in Thumb state
- * **Mode Bits**
 - M[4:0] define the processor mode.

The ARM Instruction Set - ARM University Program - V1.0



8

Condition Flags

Flag	Logical Instruction	Arithmetic Instruction
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

The ARM Instruction Set - ARM University Program - V1.0



9

The Program Counter (R15)

- * **When the processor is executing in ARM state:**
 - All instructions are 32 bits in length
 - All instructions must be word aligned
 - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- * **R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.**
- * **Thus to return from a linked branch**
 - MOV r15,r14
- or
 - MOV pc,lr

The ARM Instruction Set - ARM University Program - V1.0



10

Exception Handling and the Vector Table

- * **When an exception occurs, the core:**
 - Copies CPSR into SPSR_<mode>
 - Sets appropriate CPSR bits
 - If core implements ARM Architecture 4T and is currently in Thumb state, then
 - ARM state is entered.
 - Mode field bits
 - Interrupt disable flags if appropriate.
 - Maps in appropriate banked registers
 - Stores the "return address" in LR_<mode>
 - Sets PC to vector address
- * **To return, exception handler needs to:**
 - Restore CPSR from SPSR_<mode>
 - Restore PC from LR_<mode>

0x00000000	Reset
0x00000004	Undefined Instruction
0x00000008	Software Interrupt
0x0000000C	Prefetch Abort
0x00000010	Data Abort
0x00000014	Reserved
0x00000018	IRQ
0x0000001C	FIQ
-----	-----

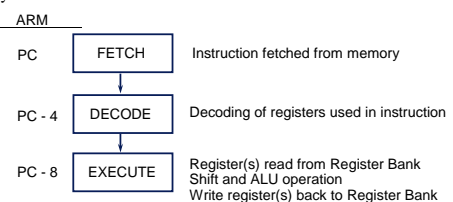
The ARM Instruction Set - ARM University Program - V1.0



11

The Instruction Pipeline

- * **The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.**
 - Allows several operations to be undertaken simultaneously, rather than serially.



- * **Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.**

The ARM Instruction Set - ARM University Program - V1.0



12

The ARM Instruction Set

ARM Instruction Set Format

31										28-7										16-5										8-1										0										Instruction type
Cond	0	0	0	0	0	0	0	0	0	Opcode	S	Rn	Rd	Operand2										Data processing / PSR Transfer																										
Cond	0	0	0	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	Multiply																														
Cond	0	0	0	0	0	1	1	1	1	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm	Long Multiply (v3M / v4 only)																														
Cond	0	0	0	0	1	0	0	0	0	E	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm	Swap																										
Cond	0	1	1	1	1	1	1	1	1	B	M	L	Rn	Rd	Offset										Load/Store Byte/Word																									
Cond	1	0	0	0	0	1	1	1	1	S	M	L	Rn	Register List										Load/Store Multiple																										
Cond	0	0	0	0	1	1	1	1	1	T	M	L	Rn	Rd	Offset1	1	S	H	1	Offset2	Halfword transfer: Immediate offset (v4 only)																													
Cond	0	0	0	0	1	1	1	1	1	H	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm	Halfword transfer: Register offset (v4 only)																									
Cond	1	0	1	1	1	1	1	1	1	Offset										Branch																														
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	Branch Exchange (v4T only)																								
Cond	1	1	1	1	1	1	1	1	1	T	M	L	Rn	CRd	CPNum	Offset										Coprocessor data transfer																								
Cond	1	1	1	1	1	1	1	1	1	0	Op1	CRn	CRd	CPNum	Op2	0	CRm	Coprocessor data operation																																
Cond	1	1	1	1	1	1	1	1	1	0	Op1	L	CRn	Rd	CPNum	Op2	1	CRm	Coprocessor register transfer																															
Cond	1	1	1	1	1	1	1	1	1	SWI Number										Software interrupt																														

The ARM Instruction Set - ARM University Program - V1.0



13

Conditional Execution

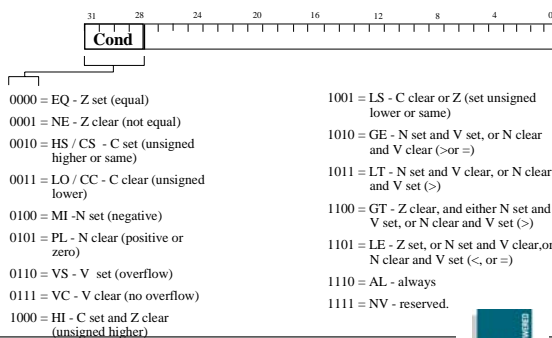
- * Most instruction sets only allow branches to be executed conditionally.
- * However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
 - All instructions contain a condition field which determines whether the CPU will execute them.
 - Non-executed instructions soak up 1 cycle.
 - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- * This removes the need for many branches, which stall the pipeline (3 cycles to refill).
 - Allows very dense in-line code, without branches.
 - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The ARM Instruction Set - ARM University Program - V1.0



14

The Condition Field



The ARM Instruction Set - ARM University Program - V1.0



15

Using and updating the Condition Field

- * To execute an instruction conditionally, simply postfix it with the appropriate condition:
 - For example an add instruction takes the form:
 - ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)
 - To execute this only if the zero flag is set:
 - ADDEQ r0,r1,r2 ; If zero flag set then... ; ... r0 = r1 + r2
- * By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".
 - For example to add two numbers and set the condition flags:
 - ADDS r0,r1,r2 ; r0 = r1 + r2 ; ... and set flags

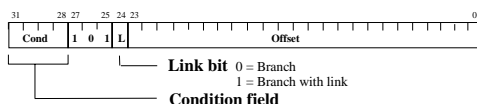
The ARM Instruction Set - ARM University Program - V1.0



16

Branch instructions (1)

- * Branch : B<cond> label
- * Branch with Link : BL<cond> sub_routine_label



- * The offset for branch instructions is calculated by the assembler:
 - By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
 - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word - aligned) and stored into the instruction encoding.
 - This gives a range of ± 32 Mbytes.

The ARM Instruction Set - ARM University Program - V1.0



17

Branch instructions (2)

- * When executing the instruction, the processor:
 - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- * Execution then continues from the new PC, once the pipeline has been refilled.
- * The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.
 - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- * To return from subroutine, simply need to restore the PC from the LR:
 - MOV pc, lr
 - Again, pipeline has to refill before execution continues.
- * The "Branch" instruction does not affect LR.
- * Note: Architecture 4T offers a further ARM branch instruction, BX
 - See Thumb Instruction Set Module for details.

The ARM Instruction Set - ARM University Program - V1.0



18

The ARM Instruction Set

Data processing Instructions

- * **Largest family of ARM instructions, all sharing the same instruction format.**
- * **Contains:**
 - Arithmetic operations
 - Comparisons (no results - just set condition codes)
 - Logical operations
 - Data movement between registers
- * **Remember, this is a load / store architecture**
 - These instructions only work on registers, *NOT* memory.
- * **They each perform a specific operation on one or two operands.**
 - First operand always a register - Rn
 - Second operand sent to the ALU via barrel shifter.
- * **We will examine the barrel shifter shortly.**

The ARM Instruction Set - ARM University Program - V1.0



19

Arithmetic Operations

- * **Operations are:**
 - ADD operand1 + operand2
 - ADC operand1 + operand2 + carry
 - SUB operand1 - operand2
 - SBC operand1 - operand2 + carry - 1
 - RSB operand2 - operand1
 - RSC operand2 - operand1 + carry - 1
- * **Syntax:**
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- * **Examples**
 - ADD r0, r1, r2
 - SUBGT r3, r3, #1
 - RSBLES r4, r5, #5

The ARM Instruction Set - ARM University Program - V1.0



20

Comparisons

- * **The only effect of the comparisons is to**
 - **UPDATE THE CONDITION FLAGS.** Thus no need to set S bit.
- * **Operations are:**
 - CMP operand1 - operand2, but result not written
 - CMN operand1 + operand2, but result not written
 - TST operand1 AND operand2, but result not written
 - TEQ operand1 EOR operand2, but result not written
- * **Syntax:**
 - <Operation>{<cond>} Rn, Operand2
- * **Examples:**
 - CMP r0, r1
 - TSTEQ r2, #5

The ARM Instruction Set - ARM University Program - V1.0



21

Logical Operations

- * **Operations are:**
 - AND operand1 AND operand2
 - EOR operand1 EOR operand2
 - ORR operand1 OR operand2
 - BIC operand1 AND NOT operand2 [ie bit clear]
- * **Syntax:**
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- * **Examples:**
 - AND r0, r1, r2
 - BICEQ r2, r3, #7
 - EORS r1,r3,r0

The ARM Instruction Set - ARM University Program - V1.0



22

Data Movement

- * **Operations are:**
 - MOV operand2
 - MVN NOT operand2
- Note that these make no use of operand1.**
- * **Syntax:**
 - <Operation>{<cond>}{S} Rd, Operand2
- * **Examples:**
 - MOV r0, r1
 - MOVS r2, #10
 - MVNEQ r1,#0

The ARM Instruction Set - ARM University Program - V1.0



23

The Barrel Shifter

- * **The ARM doesn't have actual shift instructions.**
- * **Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.**
- * **So what operations does the barrel shifter support?**

The ARM Instruction Set - ARM University Program - V1.0

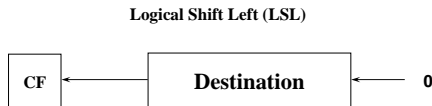


24

The ARM Instruction Set

Barrel Shifter - Left Shift

- * Shifts left by the specified amount (multiplies by powers of two) e.g. LSL #5 = multiply by 32



The ARM Instruction Set - ARM University Program - V1.0

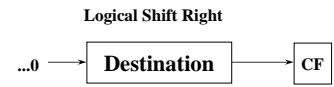


25

Barrel Shifter - Right Shifts

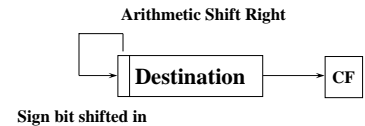
Logical Shift Right

- Shifts right by the specified amount (divides by powers of two) e.g. LSR #5 = divide by 32



Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g. ASR #5 = divide by 32



The ARM Instruction Set - ARM University Program - V1.0

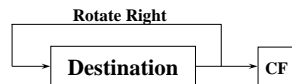


26

Barrel Shifter - Rotations

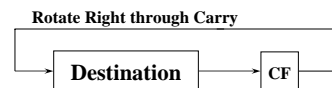
Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB. e.g. ROR #5
- Note the last bit rotated is also used as the Carry Out.



Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.

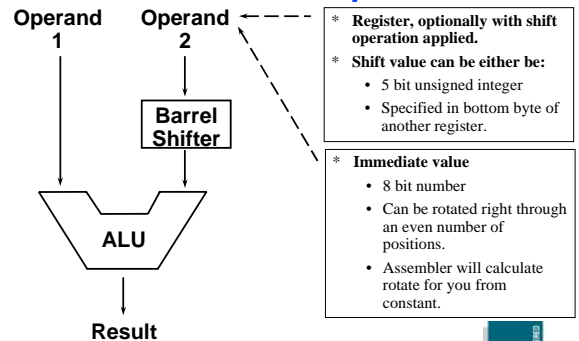


The ARM Instruction Set - ARM University Program - V1.0



27

Using the Barrel Shifter: The Second Operand



The ARM Instruction Set - ARM University Program - V1.0



28

Second Operand : Shifted Register

- * The amount by which the register is to be shifted is contained in either:
 - the immediate 5-bit field in the instruction
 - *NO OVERHEAD*
 - Shift is done for free - executes in single cycle.
 - the bottom byte of a register (not PC)
 - Then takes extra cycle to execute
 - ARM doesn't have enough read ports to read 3 registers at once.
 - Then same as on other processors where shift is separate instruction.
- * If no shift is specified then a default shift is applied: LSL #0
 - i.e. barrel shifter has no effect on value in register.

The ARM Instruction Set - ARM University Program - V1.0



29

Second Operand : Using a Shifted Register

- * Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- * A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
 - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- * Example: $r0 = r1 * 5$

$$= r1 + (r1 * 4)$$

$$\Leftrightarrow \text{ADD } r0, r1, r1, \text{LSL } \#2$$
- * Example: $r2 = r3 * 105$

$$= r3 * 15 * 7$$

$$= r3 * (16 - 1) * (8 - 1)$$

$$\Leftrightarrow \text{RSB } r2, r3, r3, \text{LSL } \#4; r2 = r3 * 15$$

$$\Leftrightarrow \text{RSB } r2, r2, r2, \text{LSL } \#3; r2 = r2 * 7$$

The ARM Instruction Set - ARM University Program - V1.0



30

The ARM Instruction Set

Multiply-Long and Multiply-Accumulate Long

- * **Instructions are**
 - MULL which gives $RdHi, RdLo := Rm * Rs$
 - MLAL which gives $RdHi, RdLo := (Rm * Rs) + RdHi, RdLo$
- * **However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)**
 - Need to specify whether operands are signed or unsigned
- * **Therefore syntax of new instructions are:**
 - UMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - UMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
- * **Not generated by the compiler.**

Warning : Unpredictable on non-M ARMs.

The ARM Instruction Set - ARM University Program - V1.0



37

Load / Store Instructions

- * **The ARM is a Load / Store Architecture:**
 - Does not support memory to memory data processing operations.
 - Must move data values into registers before using them.
- * **This might sound inefficient, but in practice isn't:**
 - Load data values from memory into registers.
 - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
 - Store results from registers out to memory.
- * **The ARM has three sets of instructions which interact with main memory. These are:**
 - Single register data transfer (LDR / STR).
 - Block data transfer (LDM/STM).
 - Single Data Swap (SWP).

The ARM Instruction Set - ARM University Program - V1.0



38

Single register data transfer

- * **The basic load and store instructions are:**
 - Load and Store Word or Byte
 - LDR / STR / LDRB / STRB
- * **ARM Architecture Version 4 also adds support for halfwords and signed data.**
 - Load and Store Halfword
 - LDRH / STRH
 - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
 - LDRSB / LDRSH
- * **All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.**
 - e.g. LDREQB
- * **Syntax:**
 - <LDR/STR>{<cond>}{<size>} Rd, <address>

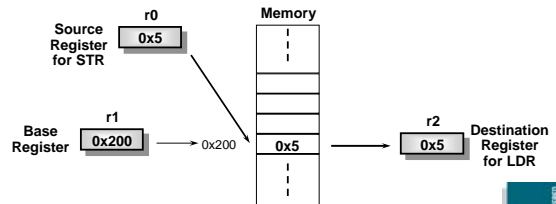
The ARM Instruction Set - ARM University Program - V1.0



39

Load and Store Word or Byte: Base Register

- * **The memory location to be accessed is held in a base register**
 - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
 - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.



The ARM Instruction Set - ARM University Program - V1.0



40

Load and Store Word or Byte: Offsets from the Base Register

- * **As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.**
- * **This offset can be**
 - An unsigned 12bit immediate value (ie 0 - 4095 bytes).
 - A register, optionally shifted by an immediate value
- * **This can be either added or subtracted from the base register:**
 - Prefix the offset value or register with '+' (default) or '-'.
- * **This offset can be applied:**
 - before the transfer is made: **Pre-indexed addressing**
 - optionally auto-incrementing the base register, by postfixing the instruction with an '!'.
 - after the transfer is made: **Post-indexed addressing**
 - causing the base register to be auto-incremented.

The ARM Instruction Set - ARM University Program - V1.0



41

Load and Store Word or Byte: Pre-indexed Addressing

- * **Example: STR r0, [r1, #12]**
-
- * **To store to location 0x1f4 instead use: STR r0, [r1, #-12]**
 - * **To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!**
 - * **If r2 contains 3, access 0x20c by multiplying this by 4:**
 - STR r0, [r1, r2, LSL #2]

The ARM Instruction Set - ARM University Program - V1.0

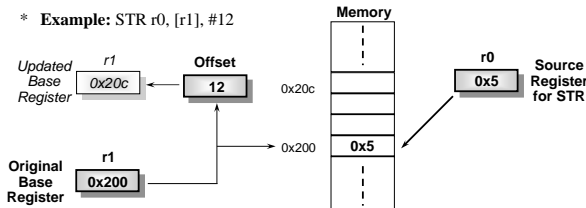


42

The ARM Instruction Set

Load and Store Word or Byte: Post-indexed Addressing

* Example: STR r0, [r1], #12



* To auto-increment the base register to location 0x1f4 instead use:

- STR r0, [r1], #-12

* If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:

- STR r0, [r1], r2, LSL #2

The ARM Instruction Set - ARM University Program - V1.0



43

Load and Stores with User Mode Privilege

* When using post-indexed addressing, there is a further form of Load/Store Word/Byte:

- <LDR|STR>{<cond>}{B}T Rd, <post_indexed_address>

* When used in a privileged mode, this does the load/store with user mode privilege.

- Normally used by an exception handler that is emulating a memory access instruction that would normally execute in user mode.

The ARM Instruction Set - ARM University Program - V1.0



44

Example Usage of Addressing Modes

* Imagine an array, the first element of which is pointed to by the contents of r0.

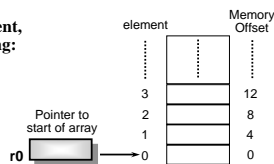
* If we want to access a particular element, then we can use pre-indexed addressing:

- r1 is element we want.
- LDR r2, [r0, r1, LSL #2]

* If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:

- r1 is address of current element (initially equal to r0).
- LDR r2, [r1], #4

Use a further register to store the address of final element, so that the loop can be correctly terminated.



The ARM Instruction Set - ARM University Program - V1.0



45

Offsets for Halfword and Signed Halfword / Byte Access

* The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.

* However the actual offset formats are more constrained:

- The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.
- The register form cannot have a shift applied to it.

The ARM Instruction Set - ARM University Program - V1.0



46

Effect of endianness

* The ARM can be set up to access its data in either little or big endian format.

* **Little endian:**

- Least significant byte of a word is stored in *bits 0-7* of an addressed word.

* **Big endian:**

- Least significant byte of a word is stored in *bits 24-31* of an addressed word.

* This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).

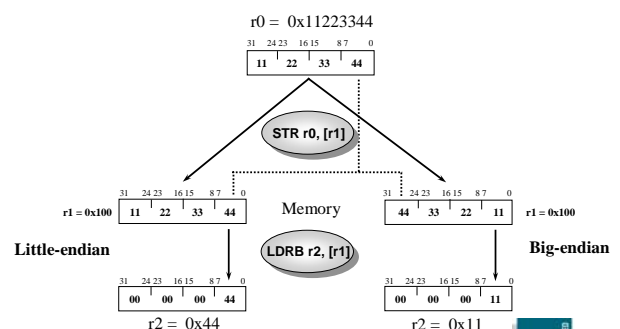
- Which byte / halfword is accessed will depend on the endianness of the system involved.

The ARM Instruction Set - ARM University Program - V1.0



47

Endianness Example



The ARM Instruction Set - ARM University Program - V1.0

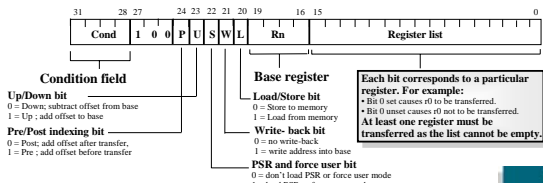


48

The ARM Instruction Set

Block Data Transfer (1)

- * The Load and Store Multiple instructions (LDM / STM) allow between 1 and 16 registers to be transferred to or from memory.
- * The transferred registers can be either:
 - Any subset of the current bank of registers (default).
 - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a 'A').



The ARM Instruction Set - ARM University Program - V1.0

49

Block Data Transfer (2)

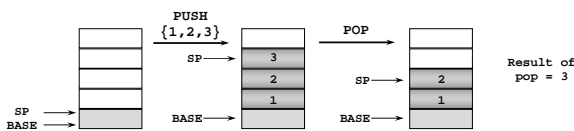
- * Base register used to determine where memory access should occur.
 - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
 - Base register can be optionally updated following the transfer (by appending it with an '!').
 - Lowest register number is always transferred to/from lowest memory location accessed.
- * These instructions are very efficient for
 - Saving and restoring context
 - For this useful to view memory as a stack.
 - Moving large blocks of data around memory
 - For this useful to directly represent functionality of the instructions.

The ARM Instruction Set - ARM University Program - V1.0

50

Stacks

- * A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.
- * Two pointers define the current limits of the stack.
 - A base pointer
 - used to point to the "bottom" of the stack (the first location).
 - A stack pointer
 - used to point the current "top" of the stack.



The ARM Instruction Set - ARM University Program - V1.0

51

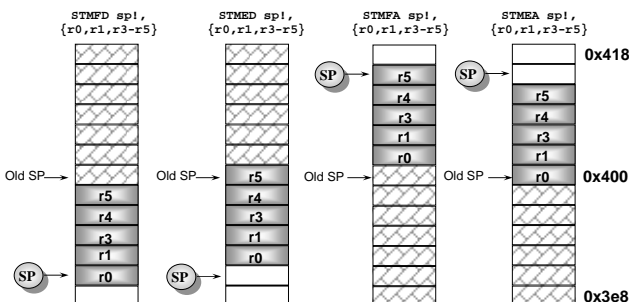
Stack Operation

- * Traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory.
- * The value of the stack pointer can either:
 - Point to the last occupied address (Full stack)
 - and so needs pre-decrementing (ie before the push)
 - Point to the next occupied address (Empty stack)
 - and so needs post-decrementing (ie after the push)
- * The stack type to be used is given by the postfix to the instruction:
 - STMFD / LDMFD : Full Descending stack
 - STMFA / LDMFA : Full Ascending stack
 - STMED / LDMED : Empty Descending stack
 - STMEA / LDMEA : Empty Ascending stack
- * Note: ARM Compiler will always use a Full descending stack.

The ARM Instruction Set - ARM University Program - V1.0

52

Stack Examples



The ARM Instruction Set - ARM University Program - V1.0

53

Stacks and Subroutines

- * One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller:

```
STMFD sp!,{r0-r12, lr}      ; stack all registers
.....                      ; and the return address
.....
LDMPD sp!,{r0-r12, pc}     ; load all the registers
.....                      ; and return automatically
```

- * See the chapter on the ARM Procedure Call Standard in the SDT Reference Manual for further details of register usage within subroutines.
- * If the pop instruction also had the 'S' bit set (using '^') then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

The ARM Instruction Set - ARM University Program - V1.0

54

The ARM Instruction Set

Direct functionality of Block Data Transfer

- * When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:
 - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- * In order to do this, LDM / STM support a further syntax in addition to the stack one:
 - STMIA / LDMIA : Increment After
 - STMIB / LDMIB : Increment Before
 - STMDA / LDMDA : Decrement After
 - STMDB / LDMDB : Decrement Before

The ARM Instruction Set - ARM University Program - V1.0



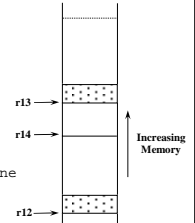
55

Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```

; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop  LDMIA  r12!, {r0-r11} ; load 48 bytes
      STMIA  r13!, {r0-r11} ; and store them
      CMP   r12, r14      ; check for the end
      BNE  loop          ; and loop until done
    
```



- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

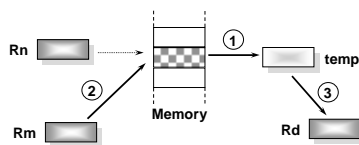
The ARM Instruction Set - ARM University Program - V1.0



56

Swap and Swap Byte Instructions

- * Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- * Syntax:
 - SWP{<cond>} {B} Rd, Rm, [Rn]



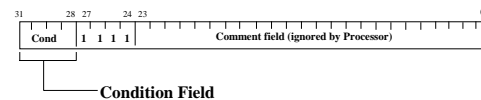
- * Thus to implement an actual swap of contents make Rd = Rm.
- * The compiler cannot produce this instruction.

The ARM Instruction Set - ARM University Program - V1.0



57

Software Interrupt (SWI)



- * In effect, a SWI is a user-defined instruction.
- * It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- * The handler can then examine the comment field of the instruction to decide what operation has been requested.
- * By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- * See Exception Handling Module for further details.

The ARM Instruction Set - ARM University Program - V1.0



58