ARM

Advanced RISC Machines

The ARM Instruction Set

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

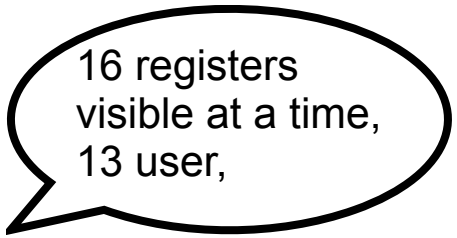
      and also 16 bit data types on ARM Architecture v4.
    - Flexible multiple register load and store instructions
* **Instruction set extension via coprocessors**

* 16 registers (32 bits)
* PC is a regular register
  => can read/write with regular instructions

The Registers

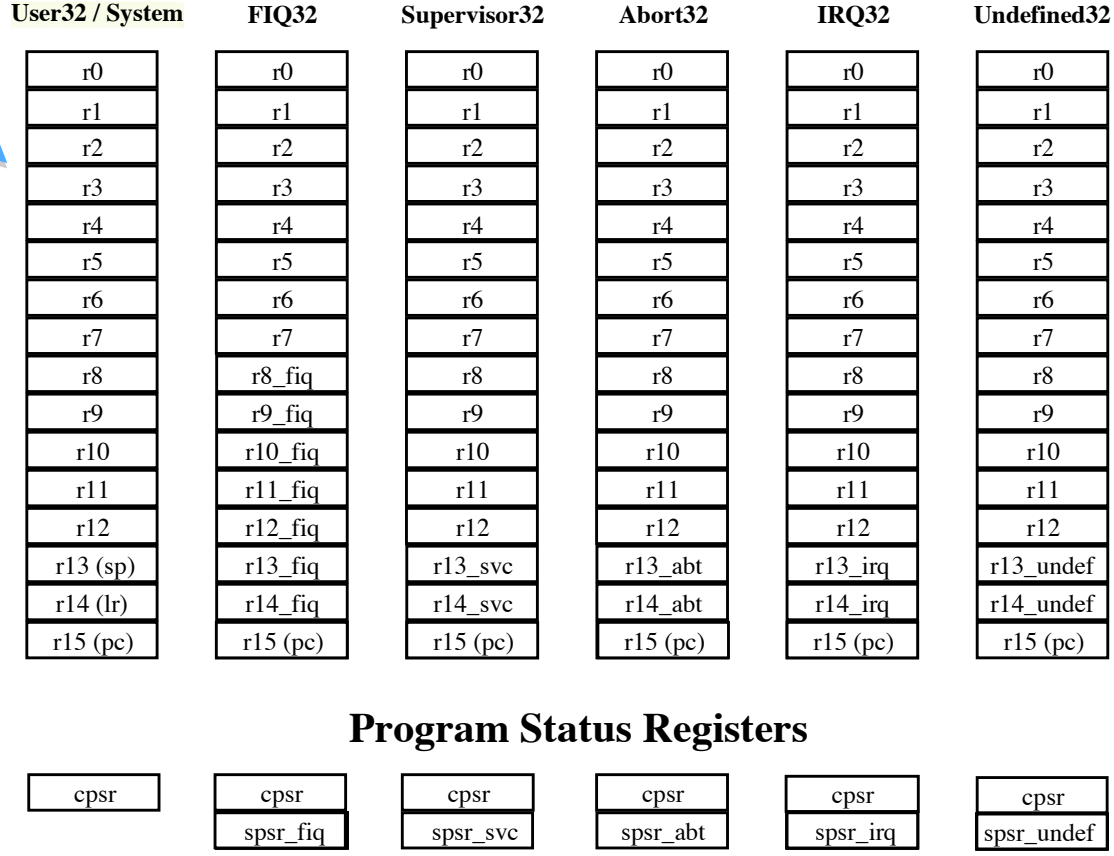
* **ARM has 37 registers in total, all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers
* **However these are arranged into several banks, with the accessible bank 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)



Register Organisation

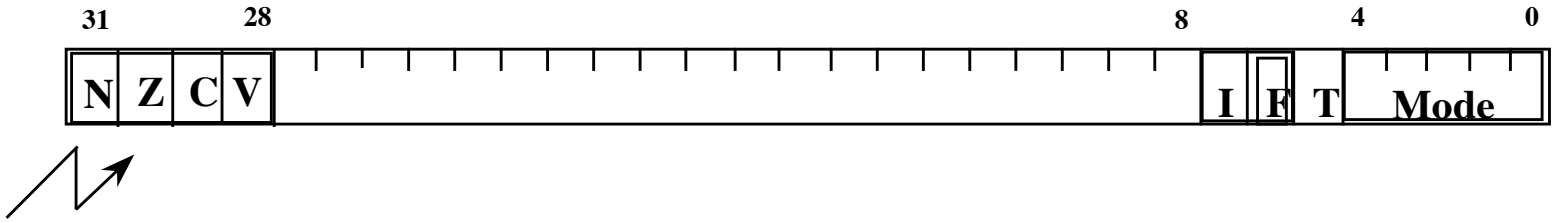
User mode
RF view



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 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 = **N**egative result from ALU flag.
  Z = **Z**ero result from ALU flag.
  C = ALU operation **C**arried out
  V = ALU operation o**V**erflowed

* **Mode Bits**

  M[4:0] define the processor mode.

* **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

Condition Flags

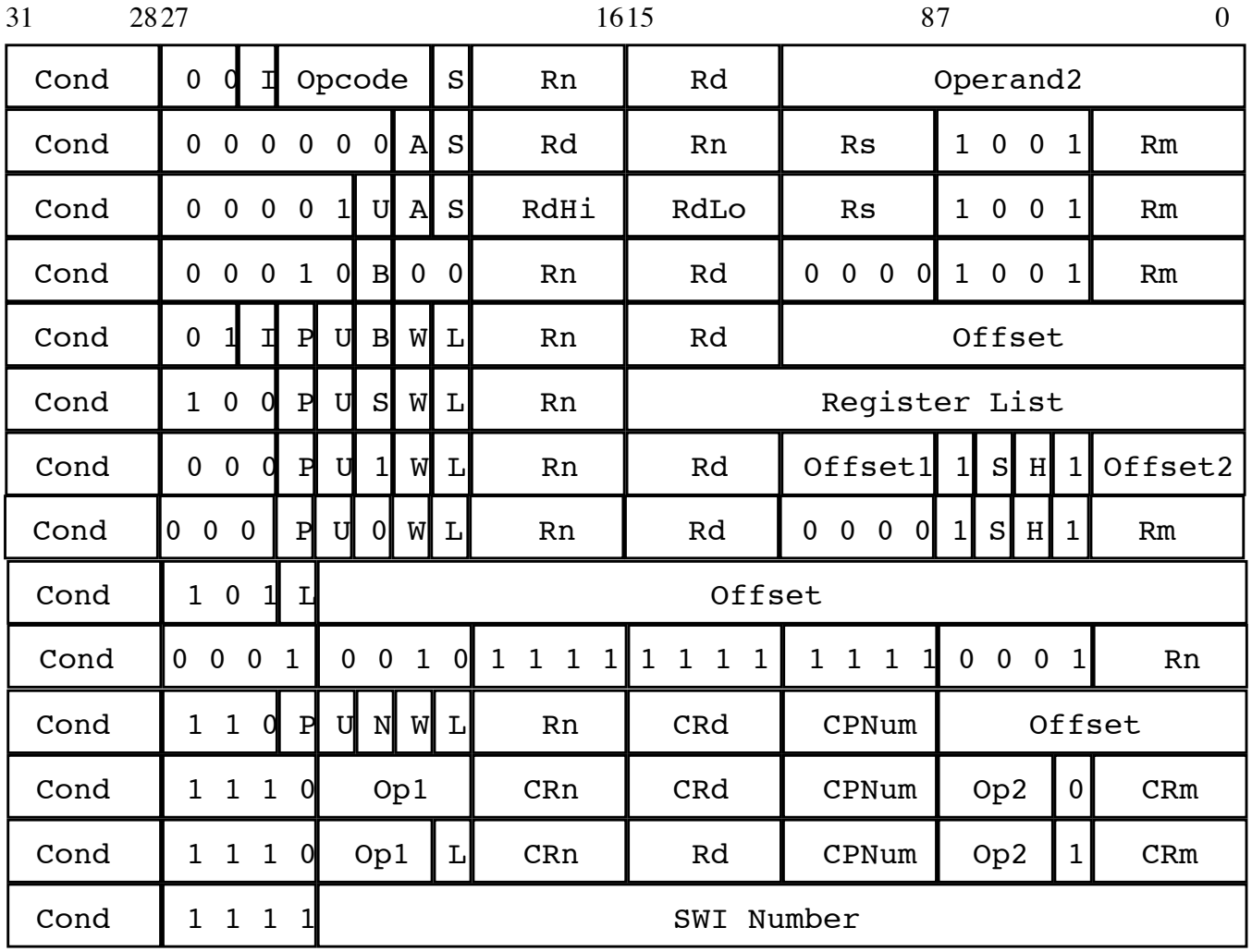
	Logical Instruction	Arithmetic Instruction
Flag		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater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

ARM Instruction Set Format



Instruction type

Data processing / PSR Transfer

Multiply

Long Multiply **(v3M / v4 only)**

Swap

Load/Store Byte/Word

Load/Store Multiple

Halfword transfer : Immediate offset **(v4 only)**

Halfword transfer: Register offset **(v4 only)**

Branch

Branch Exchange **(v4T only)**

Coprocessor data transfer

Coprocessor data operation

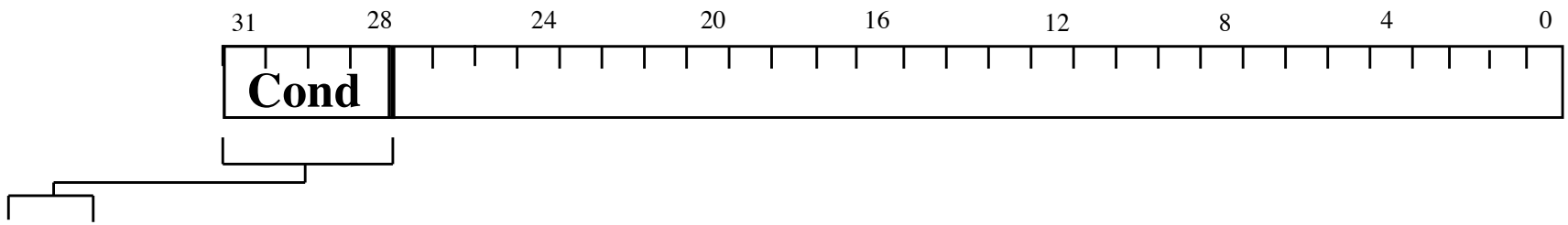
Coprocessor register transfer

Software interrupt

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 Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

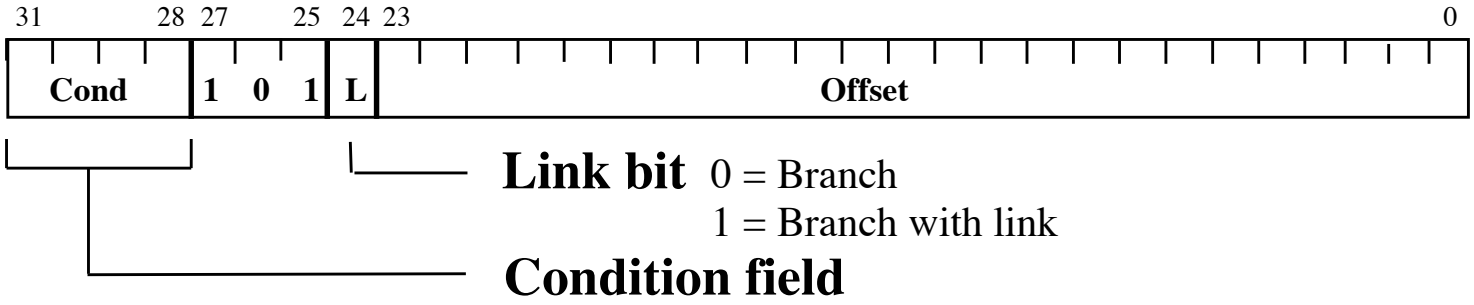
1111 = NV - reserved.

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`

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.

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.

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 instruction 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.**

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

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

Loading full 32 bit constants

* **Although the MOV/MVN mechansim will load a large range of constants into a register, sometimes this mechansim will not generate the required constant.**
* **Therefore, the assembler also provides a method which will load *ANY* 32 bit constant:**
  - `LDR rd,=numeric constant`
* **If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated.**
* **Otherwise, the assembler will produce an LDR instruction with a PC-relative address to read the constant from a literal pool.**
  - `LDR r0,=0x42 ; generates MOV r0,#0x42`
  - `LDR r0,=0x55555555 ; generate LDR r0,[pc, offset to lit pool]`
* **As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.**

Load / Store Instructions

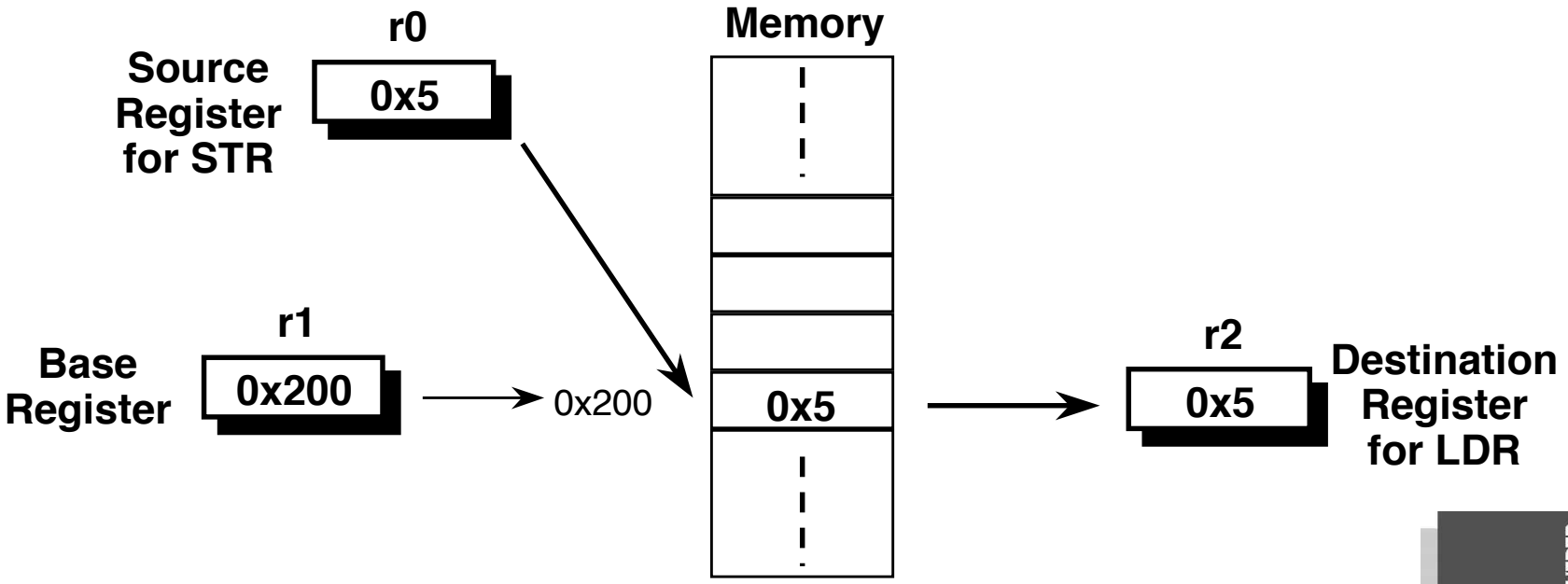
* **The ARM is a Load / Store Architecture:**
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
* **This might sound inefficient, but in practice isn't:**
  - Load data values from memory into registers.
  - Process data in registers 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).

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>

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.



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*.

Coprocessors

* **The ARM architecture supports 16 coprocessors**
* **Each coprocessor instruction set occupies part of the ARM instruction set.**
* **There are three types of coprocessor instruction**
  - Coprocessor data processing
  - Coprocessor (to/from ARM) register transfers
  - Coprocessor memory transfers (load and store to/from memory)
* **Assembler macros can be used to transform custom coprocessor mneumonics into the generic mneumonics understood by the processor.**
* **A coprocessor may be implemented**
  - in hardware
  - in software (via the undefined instruction exception)
  - in both (common cases in hardware, the rest in software)