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.

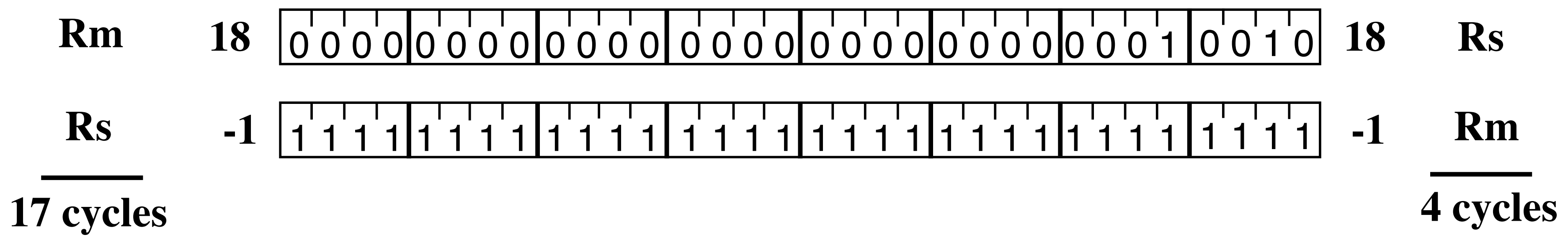
Multiplication Instructions

* **The Basic ARM provides two multiplication instructions.**
* **Multiply**
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
* **Multiply Accumulate - does addition for free**
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
* **Restrictions on use:**
  - Rd and Rm cannot be the same register
    - Can be avoid by swapping Rm and Rs around. This works because multiplication is commutative.
  - Cannot use PC.

  These will be picked up by the assembler if overlooked.
* **Operands can be considered signed or unsigned**
  - Up to user to interpret correctly.

Multiplication Implementation

* **The ARM makes use of Booth’s Algorithm to perform integer multiplication.**
* **On non-M ARMs this operates on 2 bits of Rs at a time.**
  - For each pair of bits this takes 1 cycle (plus 1 cycle to start with).
  - However when there are no more 1’s left in Rs, the multiplication will early-terminate.
* **Example: Multiply 18 and -1 : Rd = Rm * Rs**



* **Note: Compiler does not use early termination criteria to decide on which order to place operands.**

Extended Multiply Instructions

* **M variants of ARM cores contain extended multiplication hardware. This provides three enhancements:**

- An *8 bit Booth's Algorithm* is used
  - Multiplication is carried out faster (maximum for standard instructions is now 5 cycles).
- *Early termination method improved* so that now completes multiplication when all remaining bit sets contain
  - all zeroes (as with non-M ARMs), or
  - all ones.

  Thus the previous example would early terminate in 2 cycles in both cases.
- *64 bit results* can now be produced from two 32bit operands
  - Higher accuracy.
  - Pair of registers used to store result.

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.

Quiz #3

1. Specify instructions which will implement the following:

a) r0 = 16

b) r1 = r0 * 4

c) r0 = r1 / 16 (r1 signed 2's comp.)

d) r1 = r2 * 7

2. What will the following instructions do?

a) ADDS r0, r1, r1, LSL #2

b) RSB r2, r1, #0

3. What does the following instruction sequence do?

```
ADD r0, r1, r1, LSL #1
SUB r0, r0, r1, LSL #4
ADD r0, r0, r1, LSL #7
```

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).

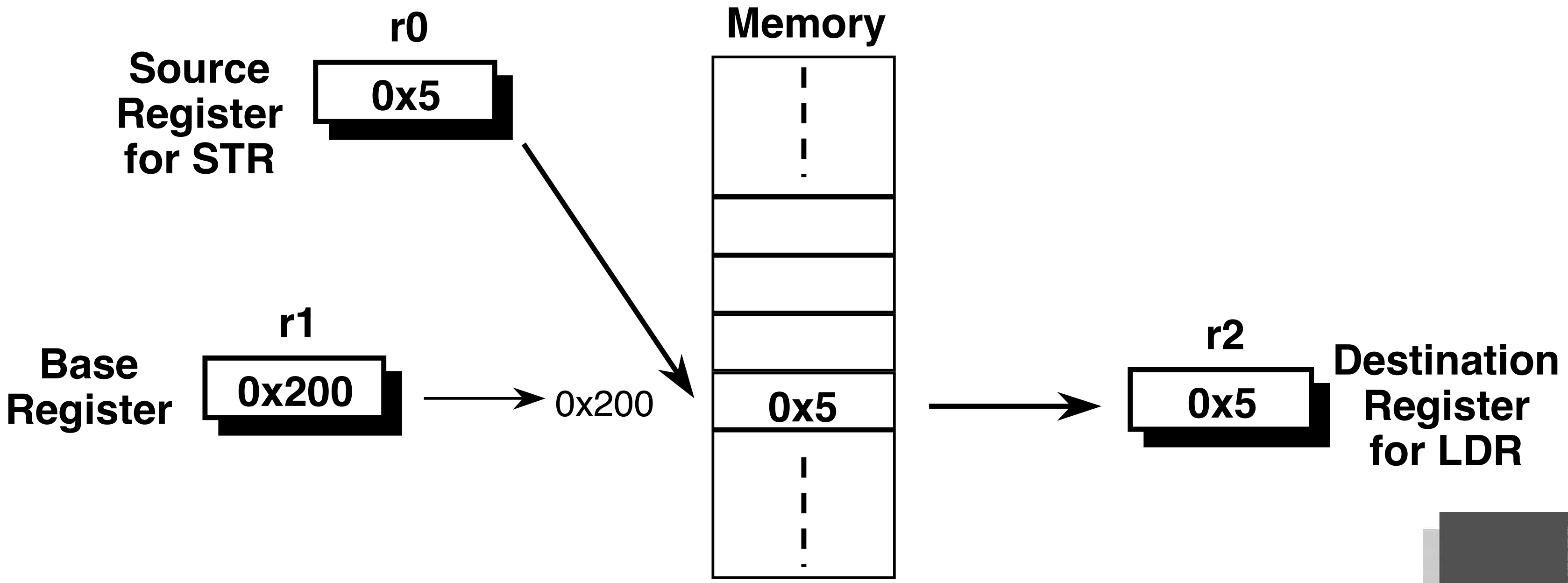
Single register data transfer

* **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:**

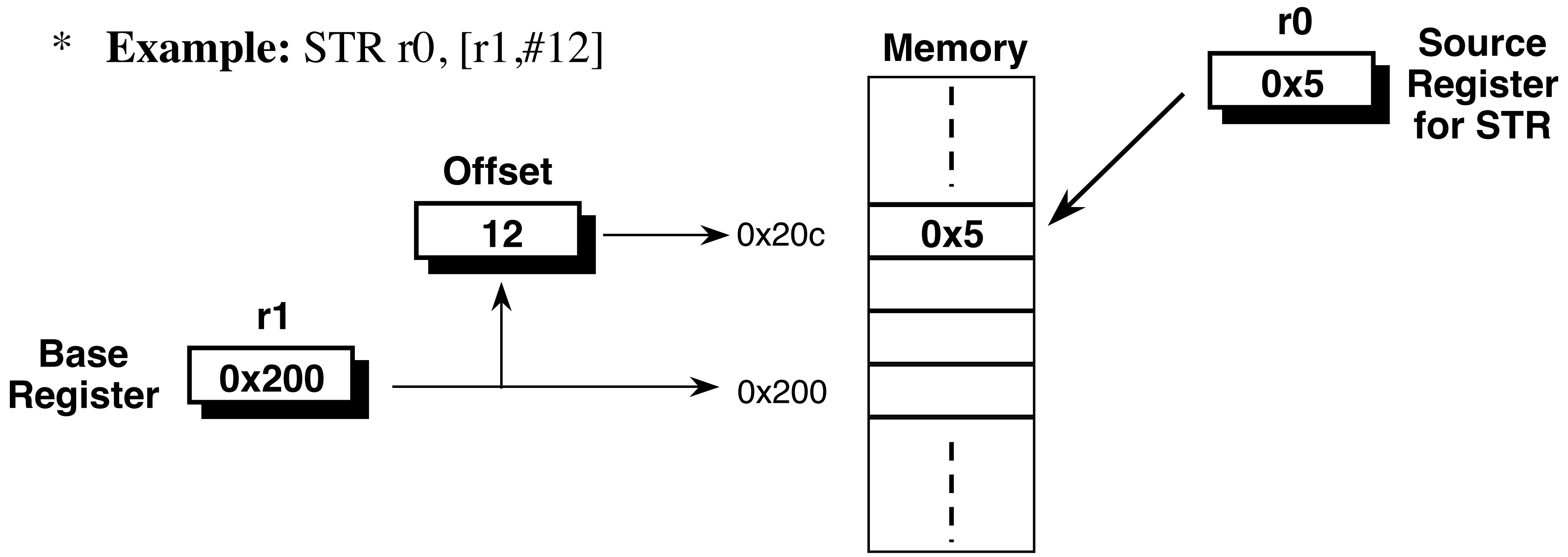
- Prefix the offset value or register with '**+**' (default) or '**-**'.

* **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*.

Load and Store Word or Byte: Pre-indexed Addressing

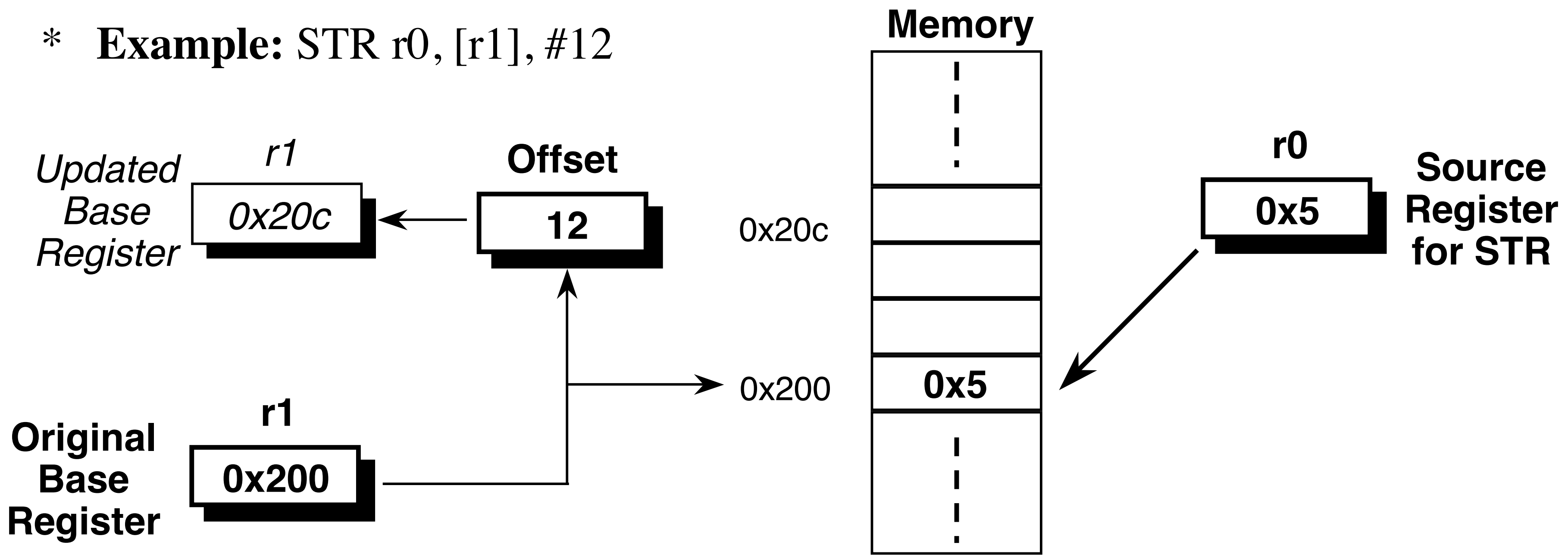
* **Example:** STR r0, [r1,#12]



* **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]

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-incremenet base register to 0x20c by multiplying this by 4:**
  - STR r0, [r1], r2, LSL #2

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>

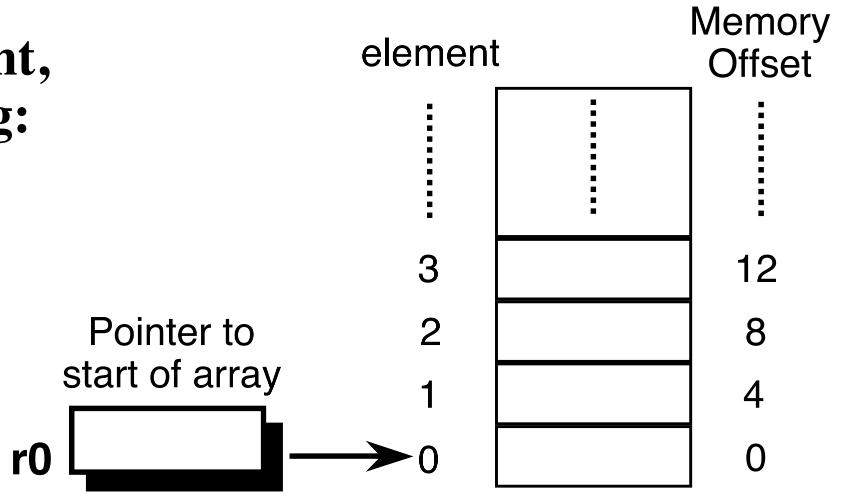
* **When used in a privileged mode, this does the load/store with user mode privilege.**
    - Normally used by an exception handler that is emulating a memory access instruction that would normally execute in user mode.

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.

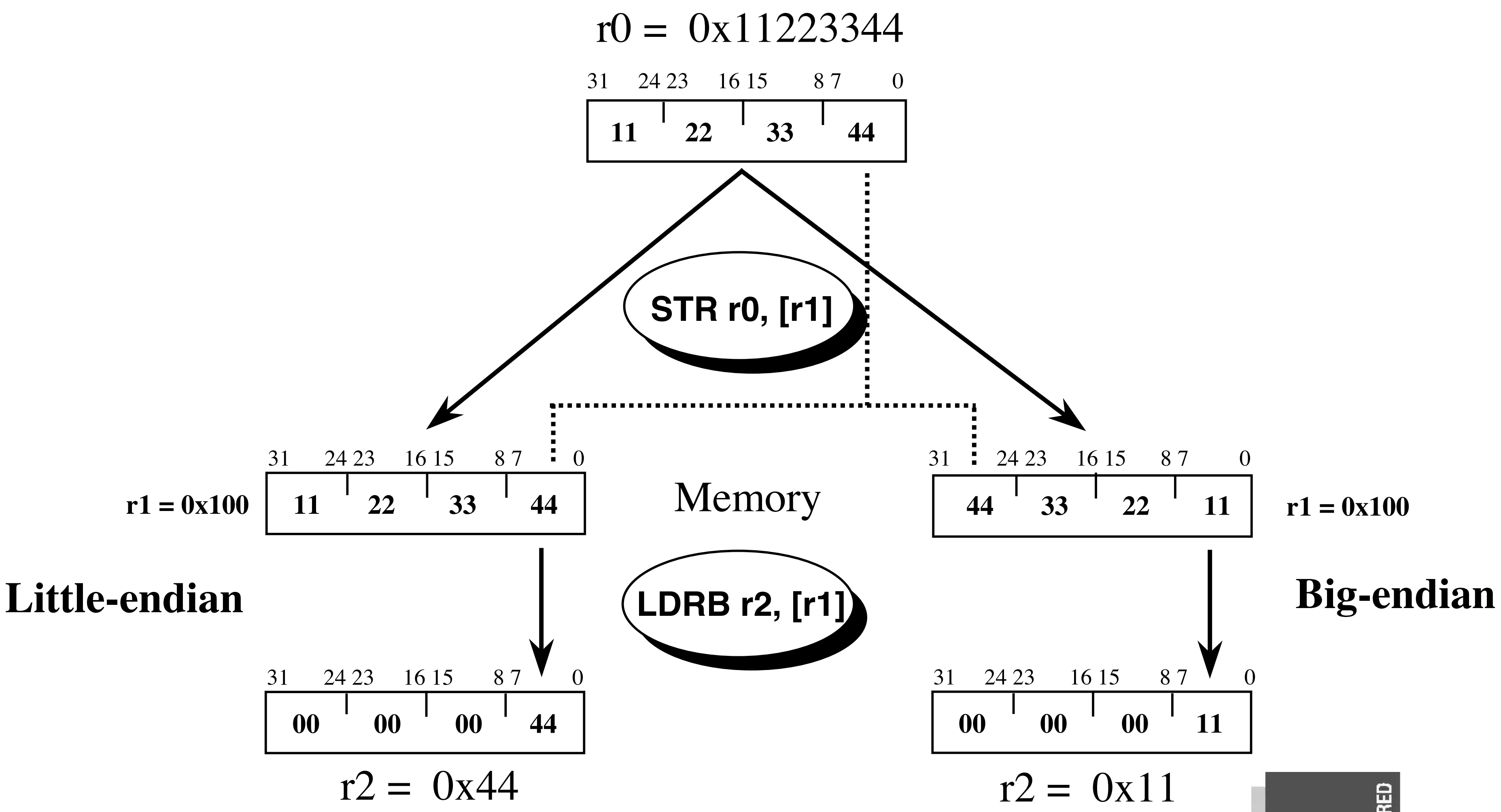
Offsets for Halfword and Signed Halfword / Byte Access

* **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.

Effect of endianess

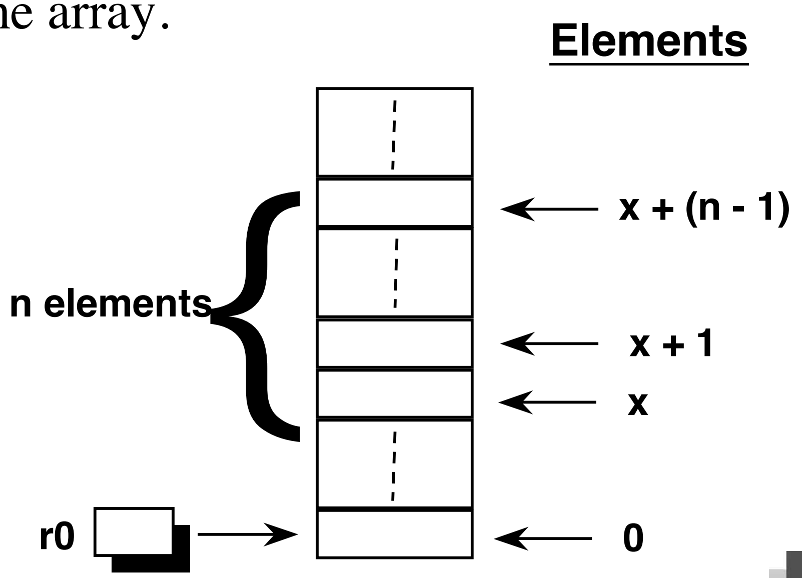
* **The ARM can be set up to access its data in either little or big endian format.**
* **Little endian:**
  - Least significant byte of a word is stored in ***bits 0-7*** of an addressed word.
* **Big endian:**
  - Least significant byte of a word is stored in ***bits 24-31*** of an addressed word.
* **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 endianess of the system involved.

Endianess Example



Quiz #4

* **Write a segment of code that add together elements x to x+(n-1) of an array, where the element x=0 is the first element of the array.**
* **Each element of the array is word sized (ie. 32 bits).**
* **The segment should use post-indexed addressing.**
* **At the start of your segments, you should assume that:**
    - r0 points to the start of the array.
    - r1 = x
    - r2 = n

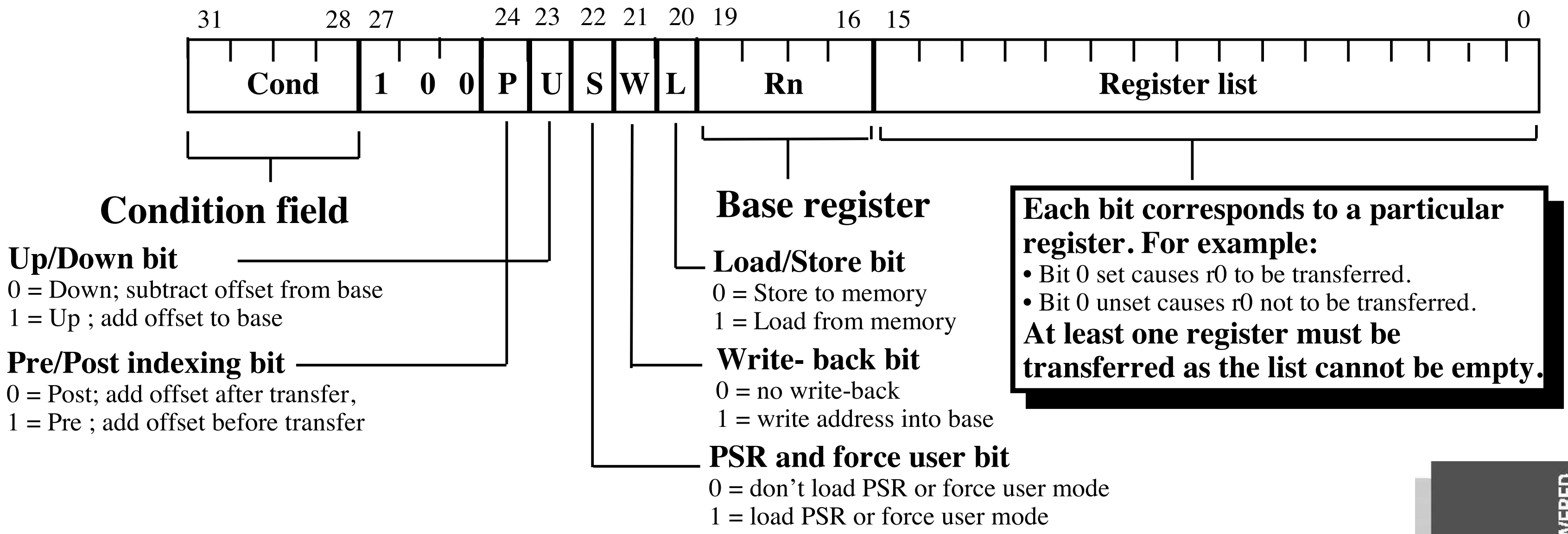


Quiz #4 - Sample Solution

```
    ADD r0, r0, r1, LSL#2       ; Set r0 to address of element x
    ADD r2, r0, r2, LSL#2       ; Set r2 to address of element n+1
    MOV r1, #0                  ; Initialise counter
loop
    LDR r3, [r0], #4            ; Access element and move to next
    ADD r1, r1, r3              ; Add contents to counter
    CMP r0, r2                  ; Have we reached element x+n?
    BLT loop                    ; If not - repeat for
                                ;       next element
    ; on exit sum contained in r1
```

Block Data Transfer (1)

* **The Load and Store Multiple instructions (LDM / STM) allow betweeen 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 priviledged mode (postfix instruction with a '^').



31
28
27
24
23
22
21
20
19
16
15
0
Cond
1 0 0
P
U
S
W
L
Rn
Register list
Condition field
Base register
Up/Down bit
0 = Down; subtract offset from base
1 = Up ; add offset to base
Pre/Post indexing bit
0 = Post; add offset after transfer,
1 = Pre ; add offset before transfer
Load/Store bit
0 = Store to memory
1 = Load from memory
Write- back bit
0 = no write-back
1 = write address into base
PSR and force user bit
0 = don't load PSR or force user mode
1 = load PSR or force user mode

Each bit corresponds to a particular register. For example:
- Bit 0 set causes r0 to be transferred.
- Bit 0 unset causes r0 not to be transferred.

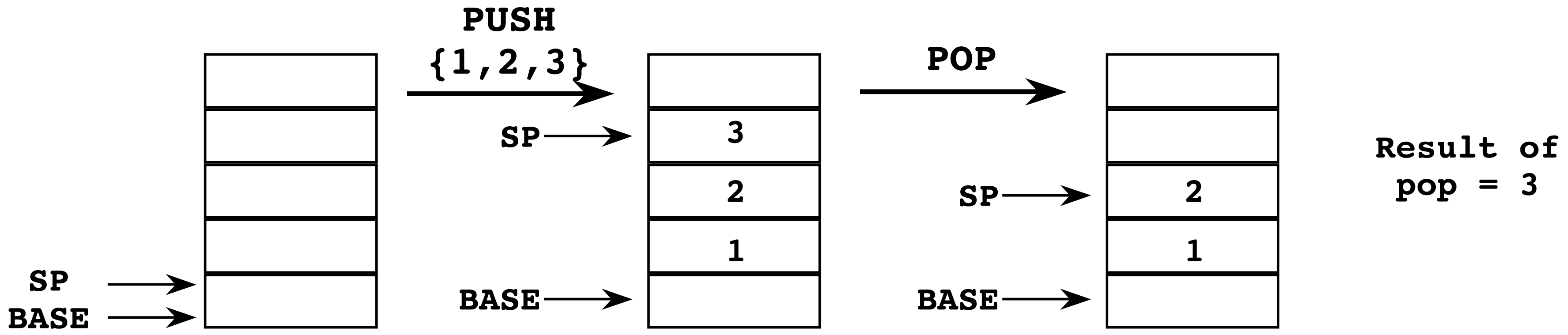
At least one register must be transferred as the list cannot be empty.

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.

Stacks

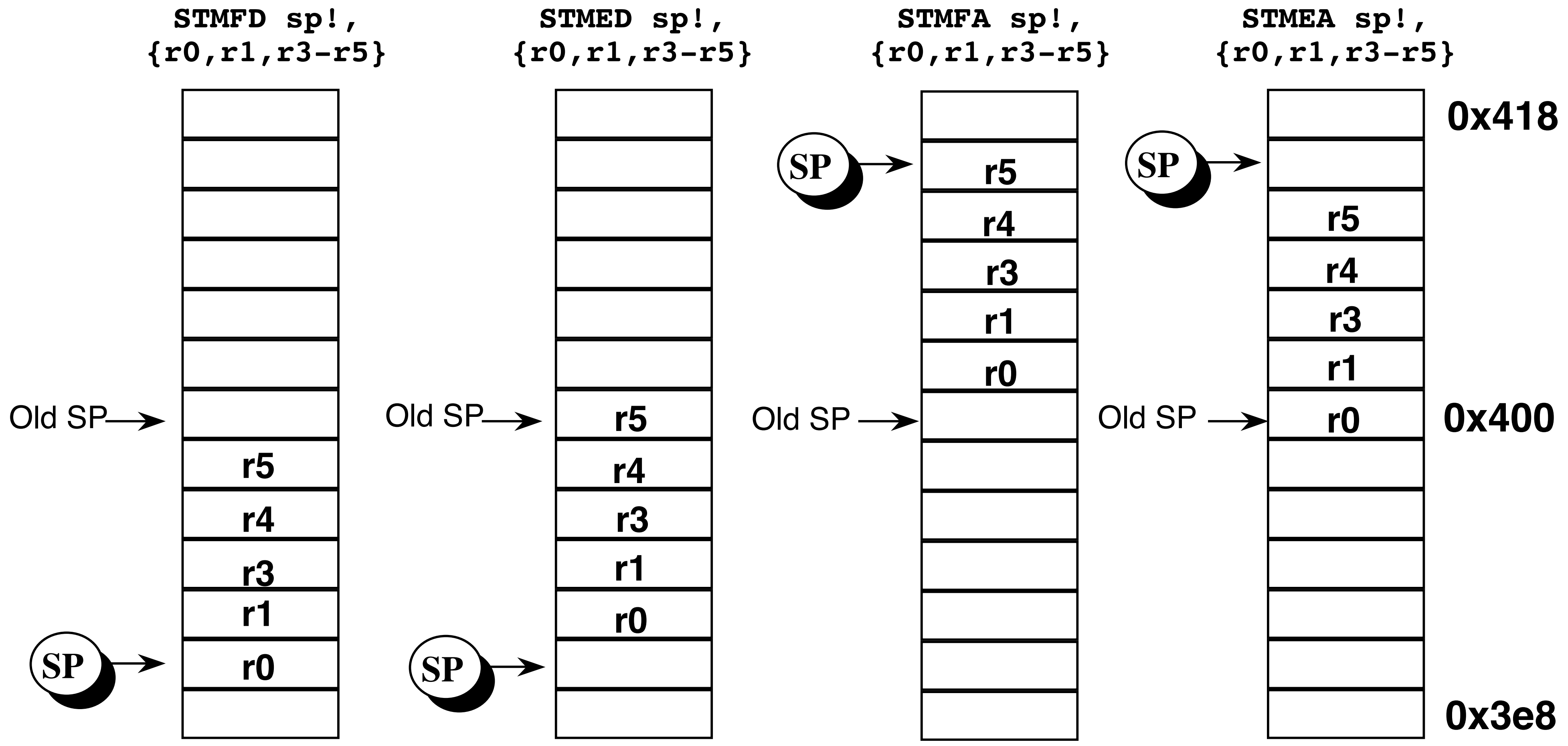
* **A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.**
* **Two pointers define the current limits of the stack.**
    - A base pointer
        - used to point to the "bottom" of the stack (the first location).
    - A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



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.**

Stack Examples


STMFD sp!, {r0,r1,r3-r5}
STMED sp!, {r0,r1,r3-r5}
STMFA sp!, {r0,r1,r3-r5}
STMEA sp!, {r0,r1,r3-r5}
0x418
0x400
0x3e8
Old SP
SP
r5
r4
r3
r1
r0