Comparison/test instructions

- CMP : compare
- CMN : negated compare
- TST : bit-wise test with a value
- TEQ : bit-wise test for two values
- These instructions set only the NZCV bits of CPSR.

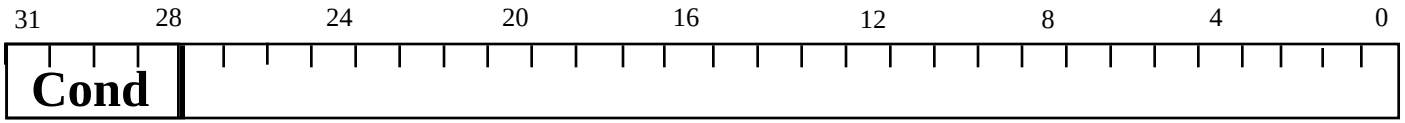
Conditional Execution

- 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.

Conditional Execution

- 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.

Comparison instructions

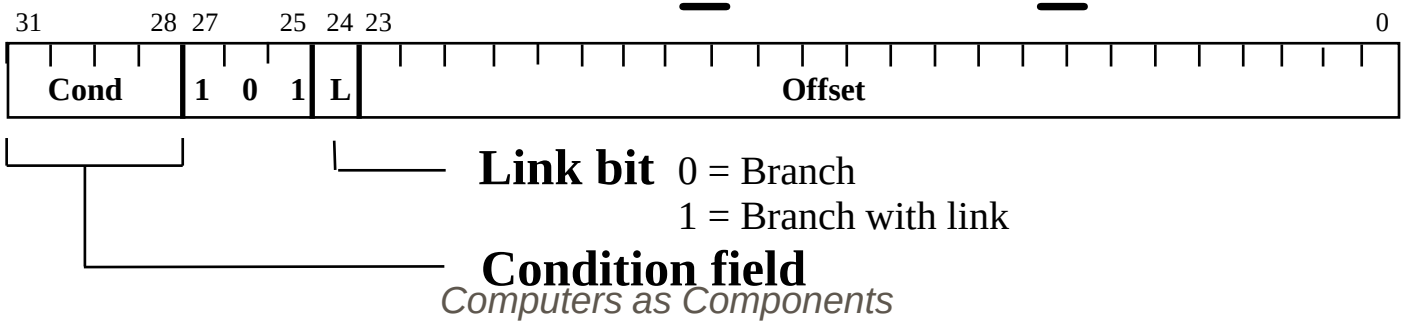
- CMP : compare two 32-bit integers
  - flags set as a result of Rn - N
  - CMP r0, r1, LSR#2; compare r0 with (r1/4)
    BHS label ; if r0 >= (r1/4) goto label
- CMN : compare negative
  - flags set as a result of Rn + N
  - CMN r0, #3; ; compare r0 with (-3)
    BLT label ; if r0 < (-3) goto label

Test instructions

- TST : bit-wise test of a 32-bit value
  - flags set as a result of Rn & N
  - TST r0,#0xFF ; test if the LSB 8 bits are 0
- TEQ : bit-wise test for two 32-bit values
  - flags set as a result of Rn ^ N
  - TEQ r0, #1 ; test to see if r0==1

Branch instructions (1)

- Used to call routines or changes the flow of execution
  - Subroutines, if-then-else structures, and loops
- Branch :
  - `B{<cond>} label`
- Branch with Link :
  - `BL{<cond>} sub_routine_label`



Branch instructions (2)

- 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 (3)

- 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 in the EX stage.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).

Branch instructions (4)

- 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.

Example: BL

- To return from subroutine, simply need to restore the PC from the LR:

```
    BL sub0          ;branch to subroutine
    CMP r1, #5       ;compare r1 with 5
    MOVEQ r1, #0
      …
Sub0
    <subroutine code >
    MOV pc, lr       ;return to (CMP r1, #5)
```

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.
    1. Load data values from memory into registers.
    2. Process data in registers using a number of data processing instructions which are not slowed down by memory access.
    3. Store results from registers out to memory.

Load / Store Instructions

- 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).

Load/store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register : `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`

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

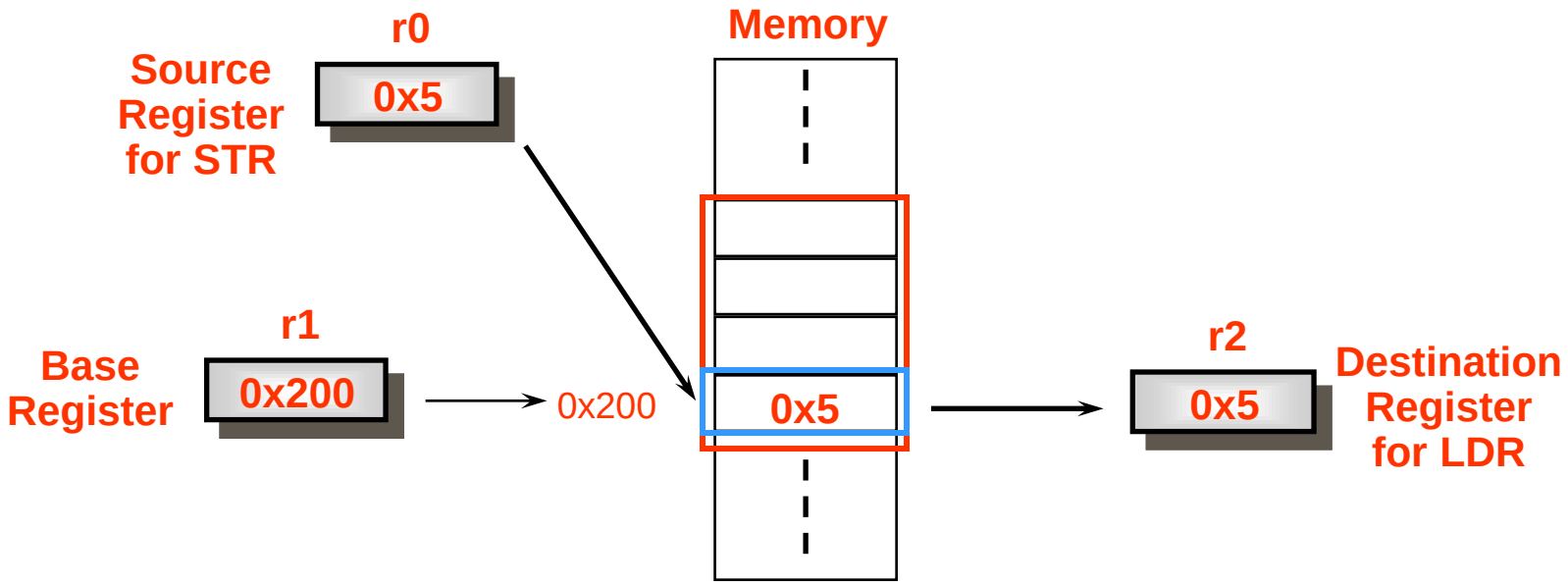
Single register data transfer

- 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 : 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: Base Register




```
str r0, [r1]
ldr r2, [r1]
```

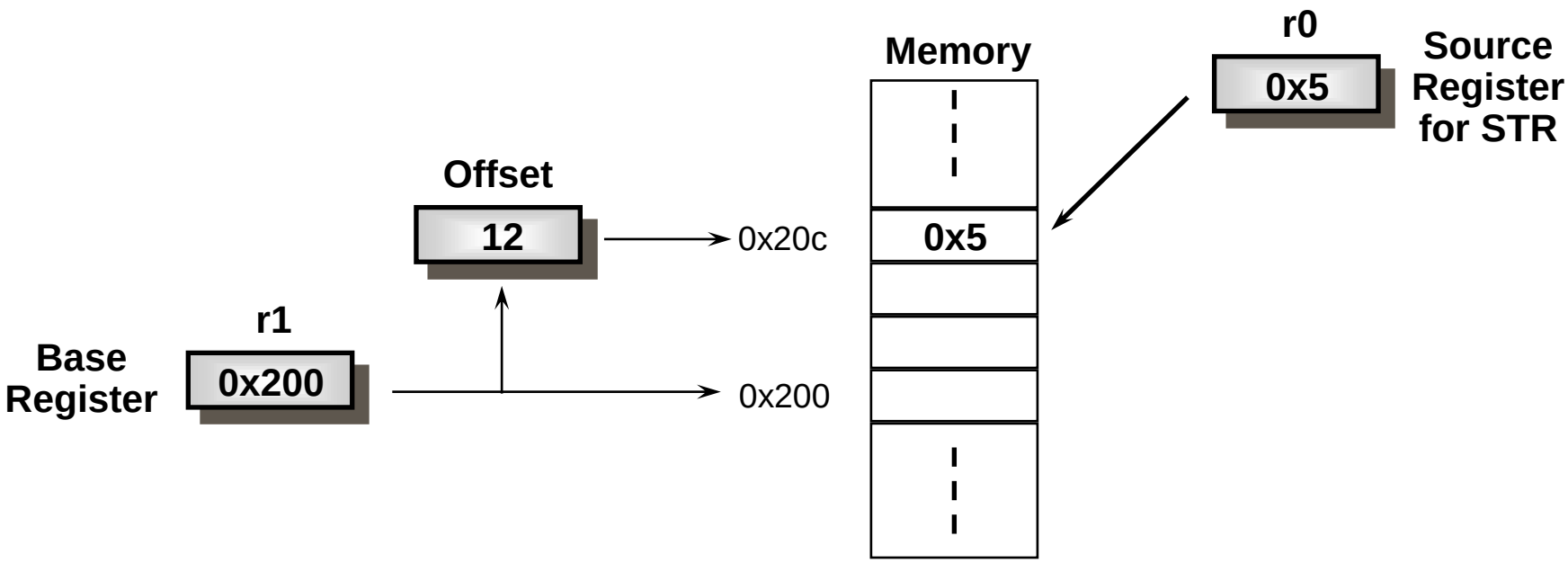
```
strb r0, [r1]
ldrb r2, [r1]
```

Load and Store : Offsets from the Base Register

- Preindex with writeback: data = mem[base+offset]
  - Base address register: base + offset
  - Example: ldr r0, [r1,#4]!
- Preindex: data = mem[base+offset]
  - Base address register: notupdated
  - Example: ldr r0, [r1,#4]
- Postindex: data = mem[base]
  - Base address register: base + offset
  - Example: ldr r0, [r1],#4

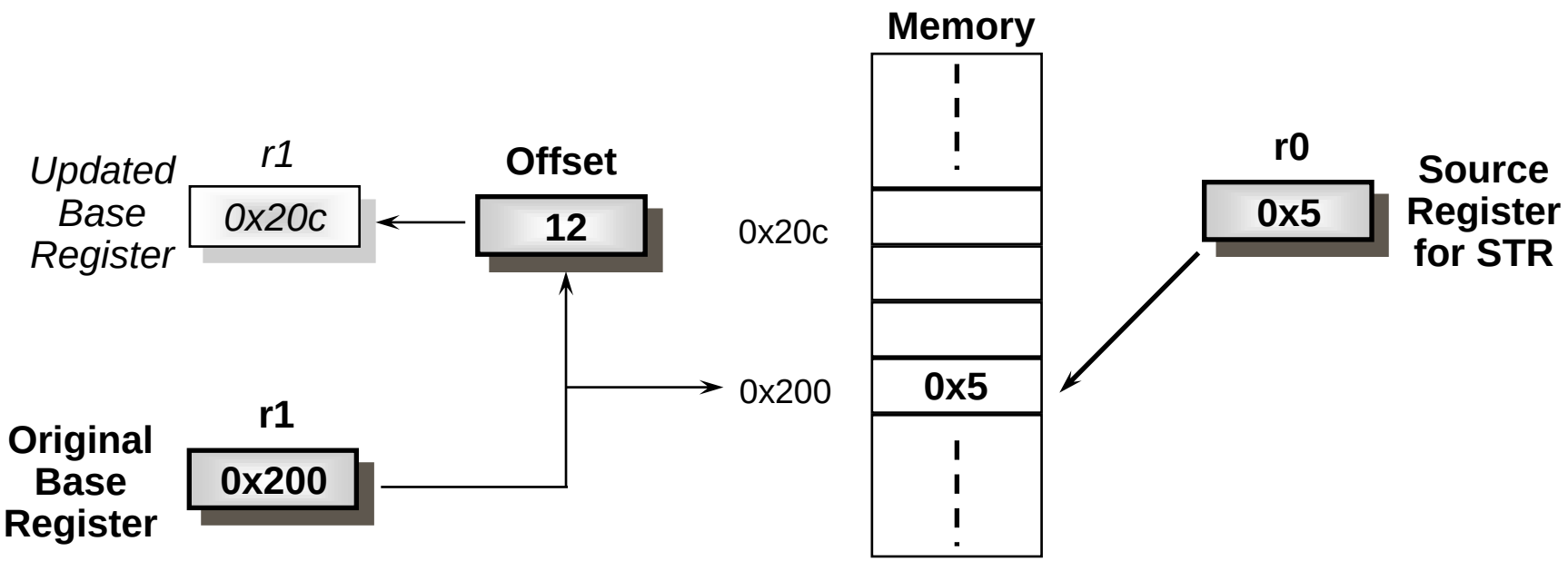
Load and Store : Pre-indexed Addressing

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



Load and Store : Post-indexed Addressing

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

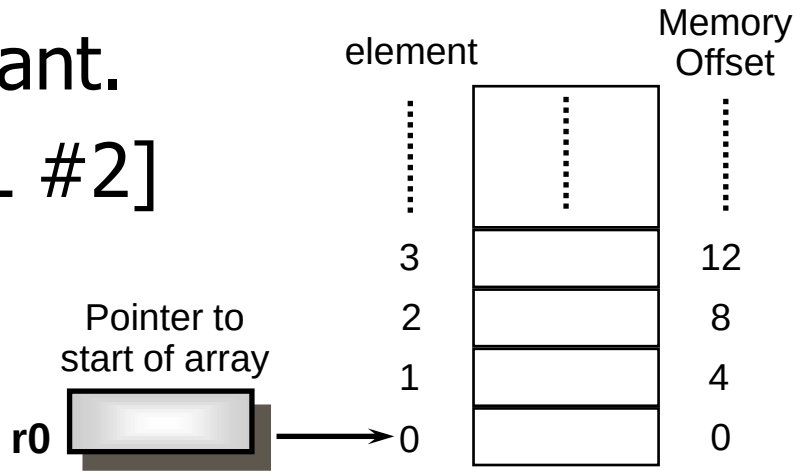


Load and Store : Post-indexed Addressing

- 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

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]



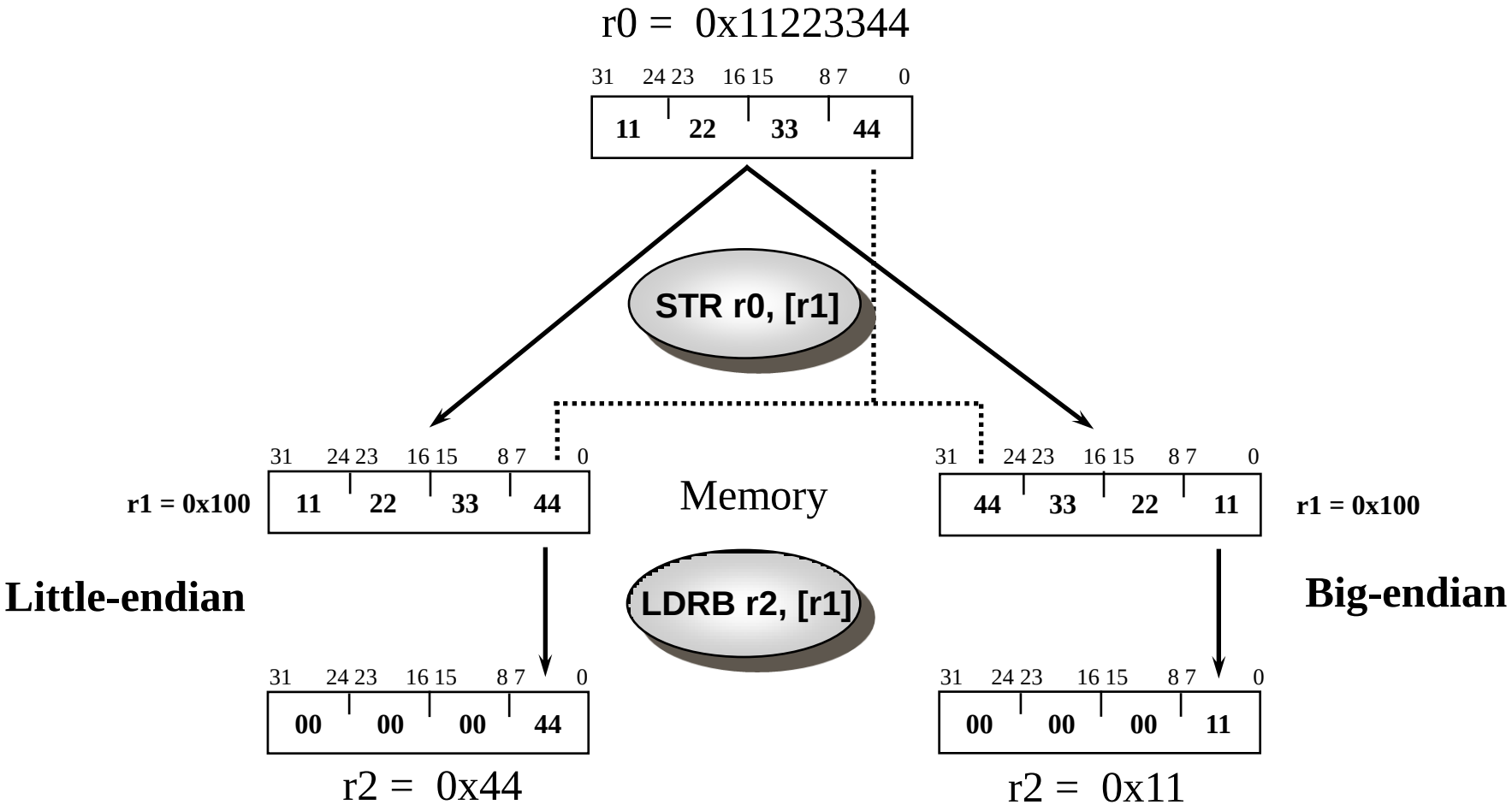
Example Usage of Addressing Modes

- 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.

Load and Store Byte: Effect of endianess

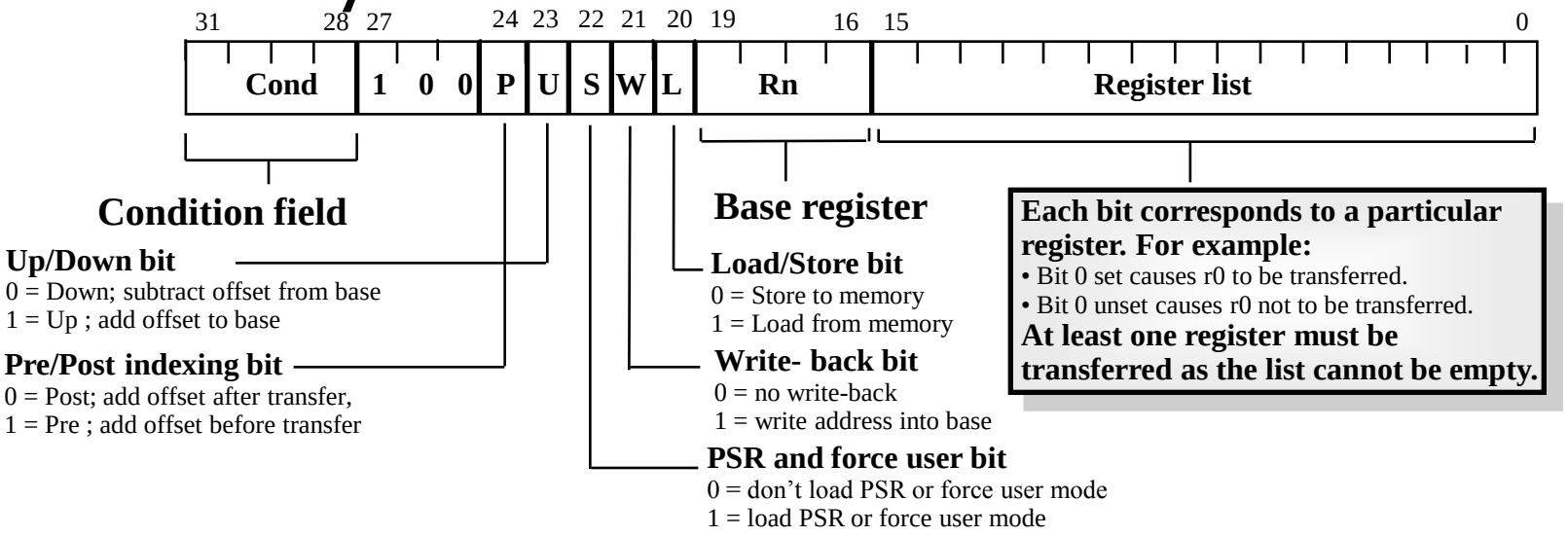
- The ARM can be set up to access its data in either little or big endian format.
- Little endian:
  - LSB byte of a word is stored in ***bits 0-7*** of an addressed word.
- Big endian:
  - LSB byte of a word is stored in ***bits 24-31*** of an addressed word.

Endianess Example



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.



Block Data Transfer (2)

- 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 previledged mode (postfix instruction with a '^').
    - LDR{<cond>}{add. mode>} Rn{!}, <regs>{^}
    - STR{<cond>}{add. mode>} Rn{!}, <regs>{^}

Block Data Transfer (3)

- 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.
  - IA (increment after)
  - IB (increment before)
  - DA (decrement after)
  - DB (decrement before)

Block Data Transfer (4)

- 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.

Example 1

- Pre
  - mem32[0x80018] = 0x03, mem32[0x80014] = 0x02
    mem32[0x80010] = 0x01, r0=0x00080010
    r1=0x00000000, r2=0x00000000, r3=0x00000000

- ldmia r0!, {r1-r3}

- Post
  - r0=0x0008001c, r1=0x00000001, r2=0x00000002,
    r3=0x00000003

Example 2

- Pre
  - r0=0x00009000, r1=0x00000009, r2=0x00000008, r3=0x00000007
- stmib r0!, {r1-r3};

  mov r1, #1

  mov r1, #2

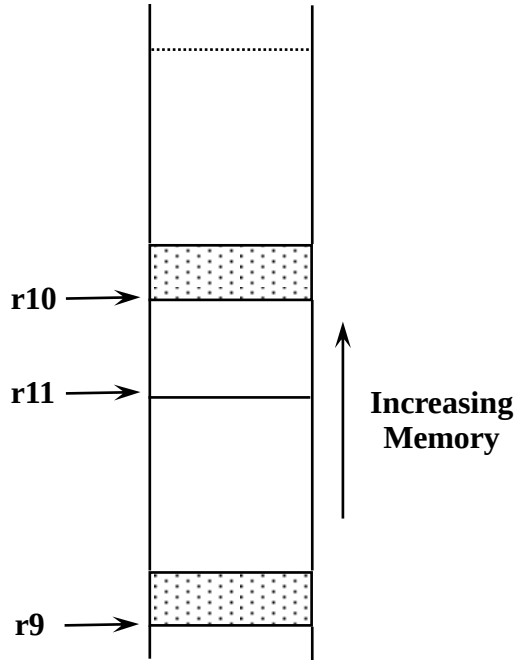
  mov r1, #3
- Post
  - r0=0x0000900c, r1=0x00000001, r2=0x00000002, r3=0x00000003

Example 2

- Pre
  - r0=0x0000900c, r1=0x00000001, r2=0x00000002, r3=0x00000003
- ldmda r0!, {r1-r3};
- Post
  - r0=0x00009000, r1=0x00000009, r2=0x00000008, r3=0x00000009
- The stmib instruction stores the values of registers {r1-r3} to memory. We then corrupt the registers {r1-r3}. The ldmda reloads the original values and restores the base pointer r0.

Example 3: Block memory copy

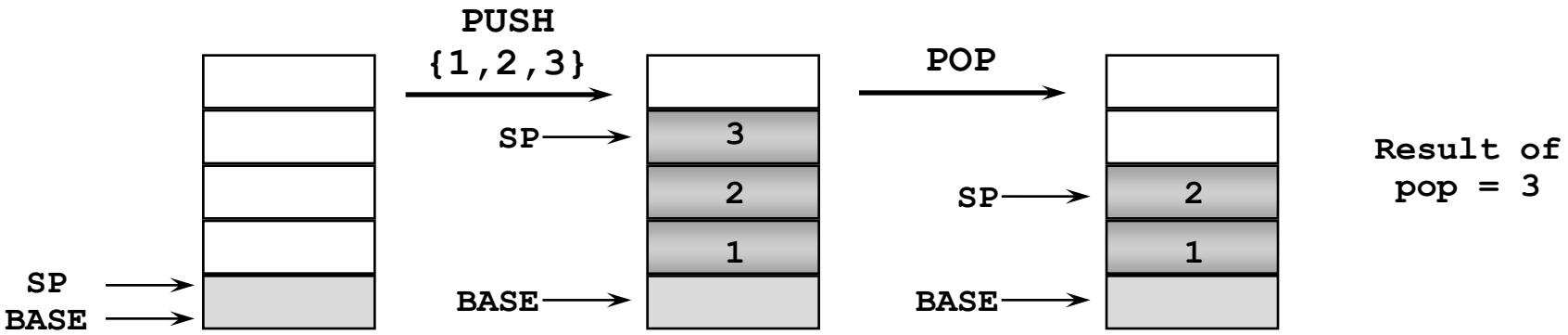
```
loop
  ldmia r9!, {r0-r7}    ; load 32 bytes from source
  stmia r10!, {r0-r7}   ; and store them to destination
  ; have reached to the end
  cmp r9 r11
  bne loop
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source
```



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.

Stacks



- 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.

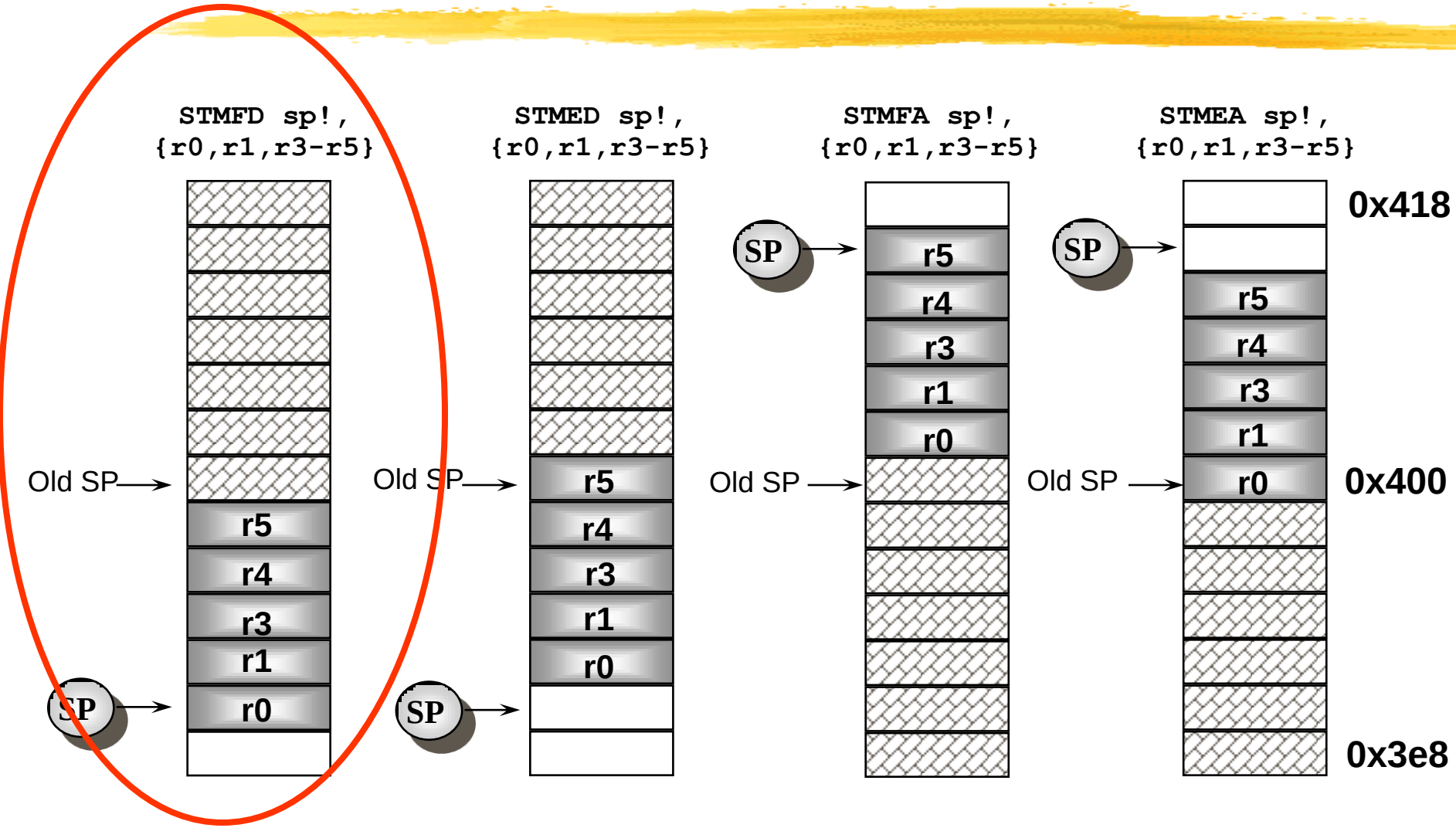
Stack Operation

- 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)

Stack Operation

- The stack type to be used is given by the postfix to the instruction:
  - Push / Pop
  - 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



Stacks and Subroutines (1)

- 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 :

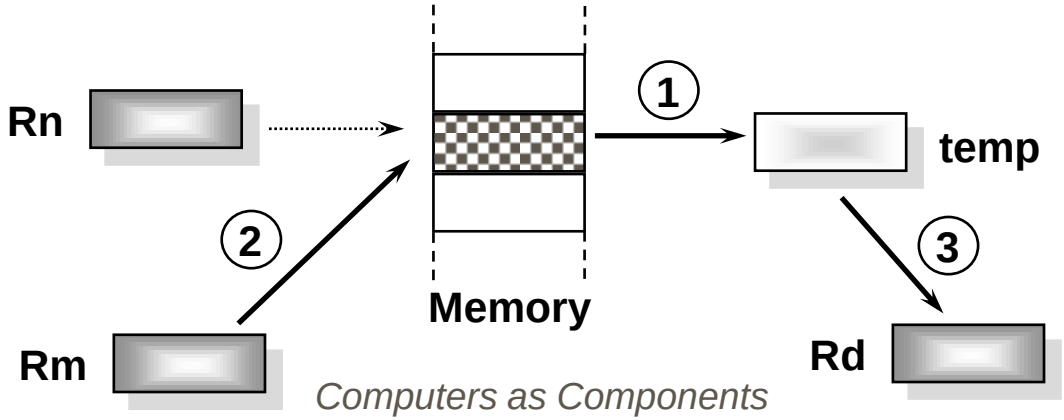
```
STMFD sp!,{r0-r12, lr}   ; stack all registers
.........                ; and the return address
.........
LDMFD sp!,{r0-r12, pc}   ; load all the registers
                         ; and return automatically
```

Stacks and Subroutines (2)

- 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 priviledged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

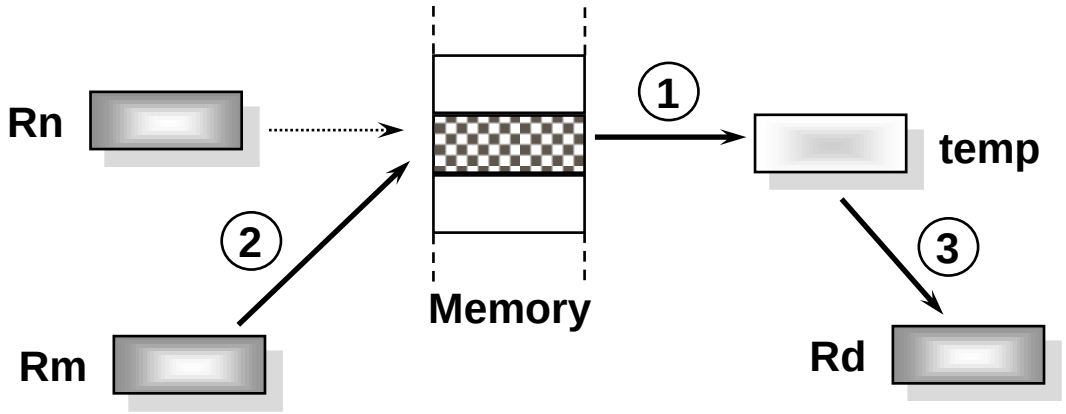
Swap and Swap Byte

- 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]



Swap and Swap Byte

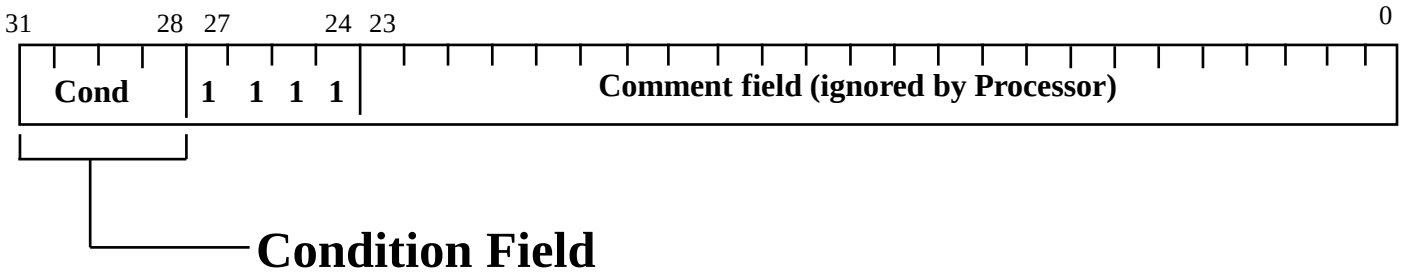
- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.



Example: Swap

- Pre
  - mem32[0x9000] = 0x12345678,
    r0=0x00000000, r1=0x11112222,
    r2=0x00009000
- swp r0, r1, [r2]
- Post
  - Mem32[0x9000]= 0x11112222
    r0=0x12345678, r1=0x00000001,
    r2=0x00009000,

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.

Software Interrupt (SWI)

- 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.