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

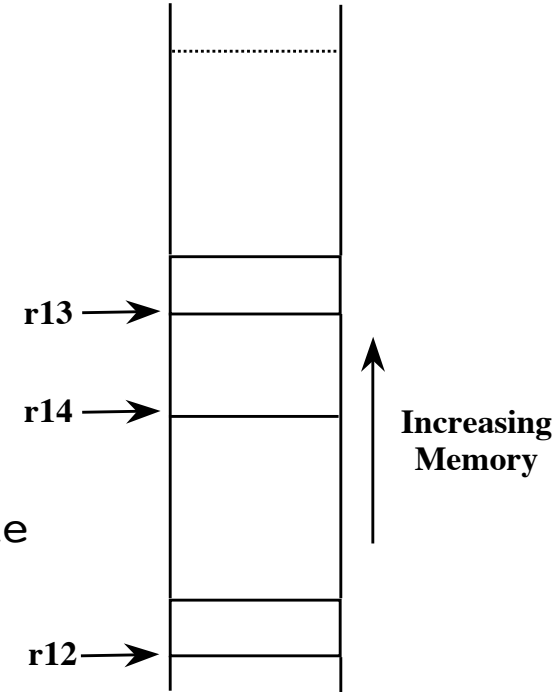
Direct functionality of Block Data Transfer

* **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

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
```

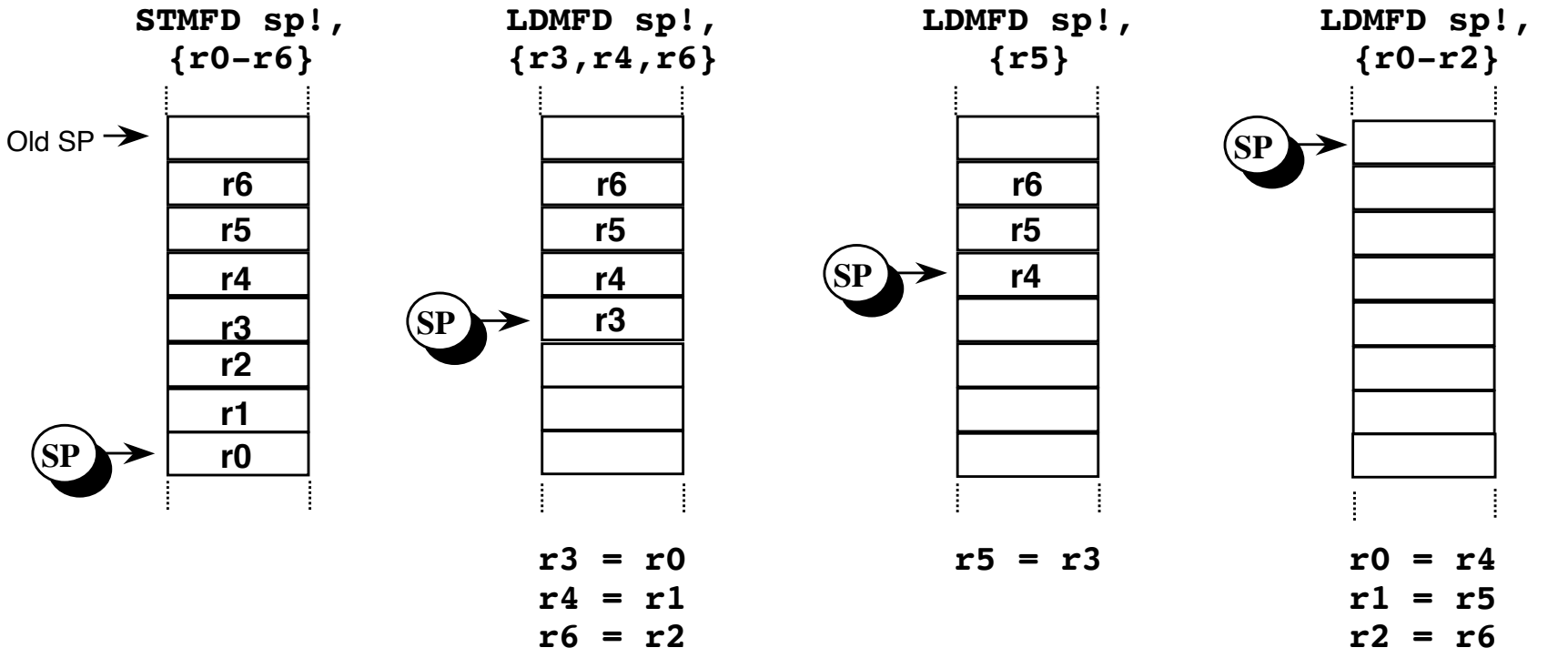


- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

Quiz #5

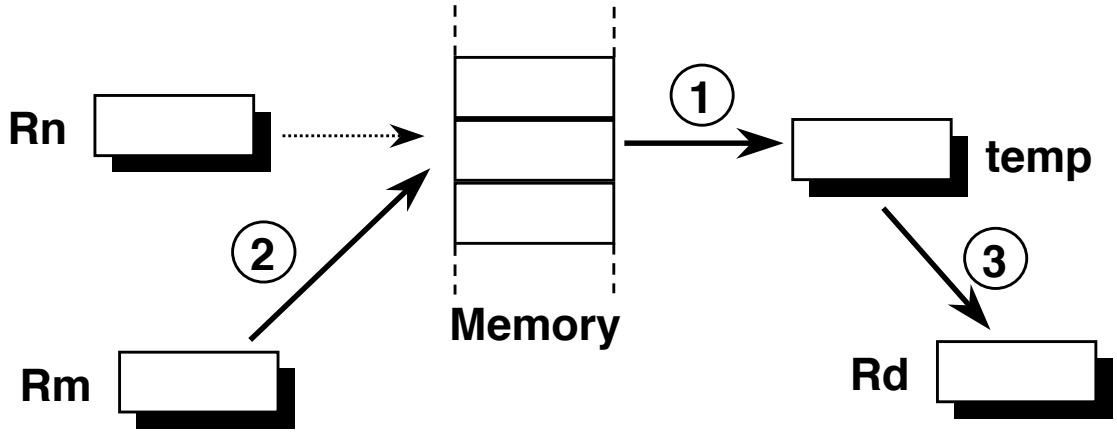
* **The contents of registers r0 to r6 need to be swapped around thus:**
  - r0 moved into r3
  - r1 moved into r4
  - r2 moved into r6
  - r3 moved into r5
  - r4 moved into r0
  - r5 moved into r1
  - r6 moved into r2
* **Write a segment of code that uses full descending stack operations to carry this out, and hence requires no use of any other registers for temporary storage.**

Quiz #5 - Sample Solution



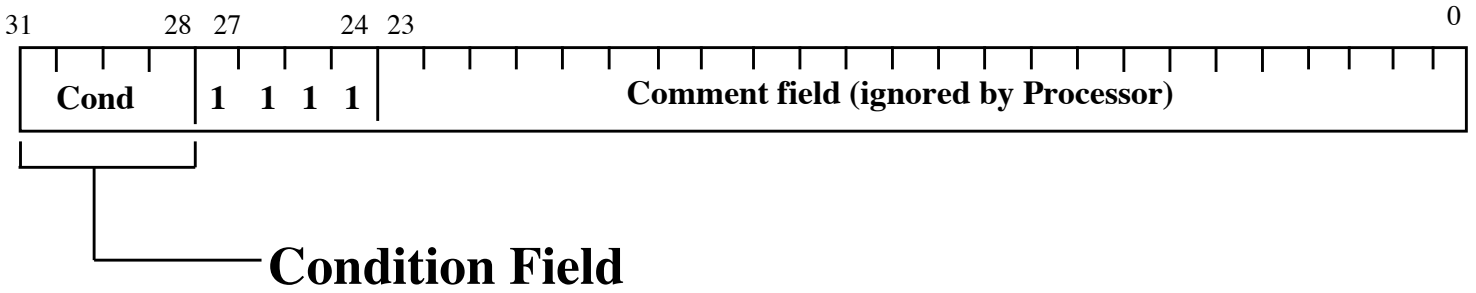
Swap and Swap Byte Instructions

* **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.**

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 mechansim, 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.

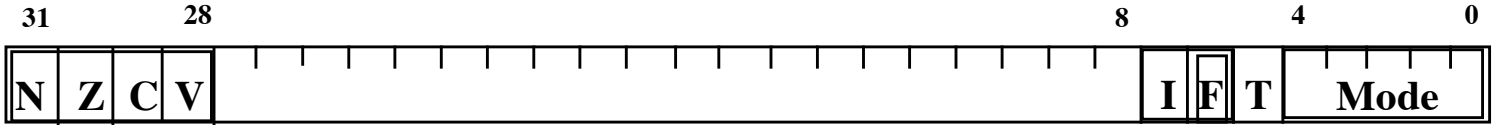
PSR Transfer Instructions

* **MRS and MSR allow contents of CPSR/SPSR to be transferred from appropriate status register to a general purpose register.**
  - All of status register, or just the flags, can be transferred.
* **Syntax:**
  - `MRS{<cond>} Rd,<psr>  ; Rd = <psr>`
  - `MSR{<cond>} <psr>,Rm  ; <psr> = Rm`
  - `MSR{<cond>} <psrf>,Rm  ; <psrf> = Rm`

  where
  - `<psr>  = CPSR, CPSR_all, SPSR or SPSR_all`
  - `<psrf> = CPSR_flg or SPSR_flg`
* **Also an immediate form**
  - `MSR{<cond>} <psrf>,#Immediate`
  - This immediate must be a 32-bit immediate, of which the 4 most significant bits are written to the flag bits.

Using MRS and MSR

* **Currently reserved bits, may be used in future, therefore:**
  - they must be preserved when altering PSR
  - the value they return must not be relied upon when testing other bits.



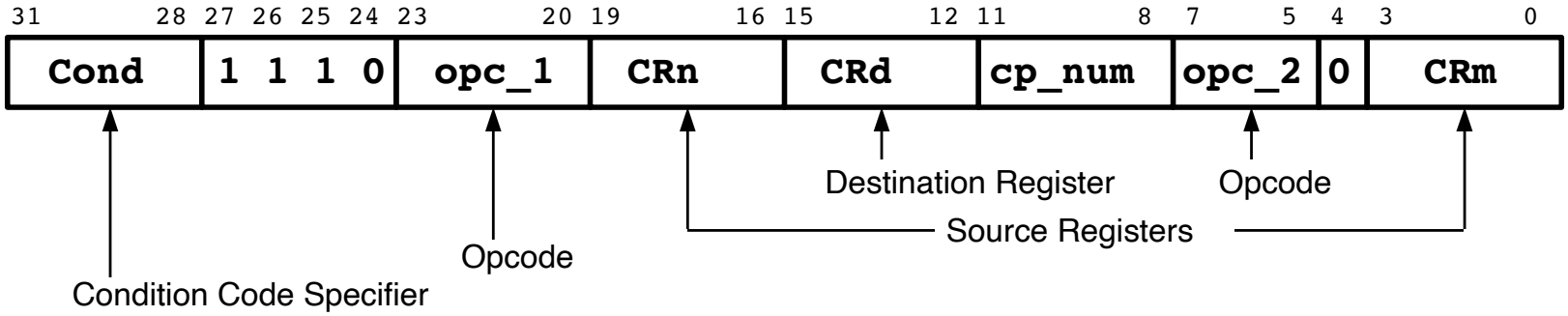
* **Thus read-modify-write strategy must be followed when modifying any PSR:**
  - Transfer PSR to register using MRS
  - Modify relevant bits
  - Transfer updated value back to PSR using MSR
* **Note:**
  - In User Mode, all bits can be read but only the flag bits can be written to.

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)

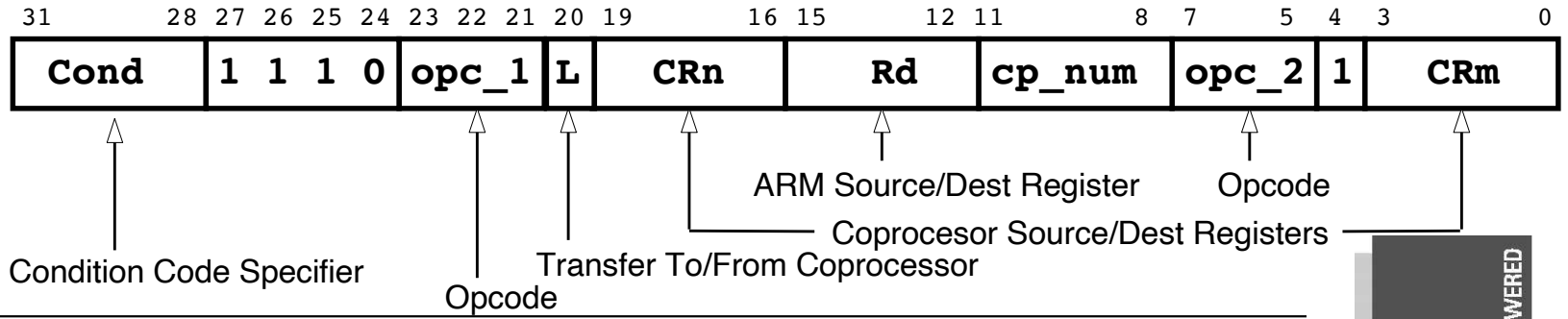
Coprocessor Data Processing

* **This instruction initiates a coprocessor operation**
* **The operation is performed only on internal coprocessor state**
  - For example, a Floating point multiply, which multiplies the contents of two registers and stores the result in a third register
* **Syntax:**
  - `CDP{<cond>} <cp_num>,<opc_1>,CRd,CRn,CRm,{<opc_2>}`



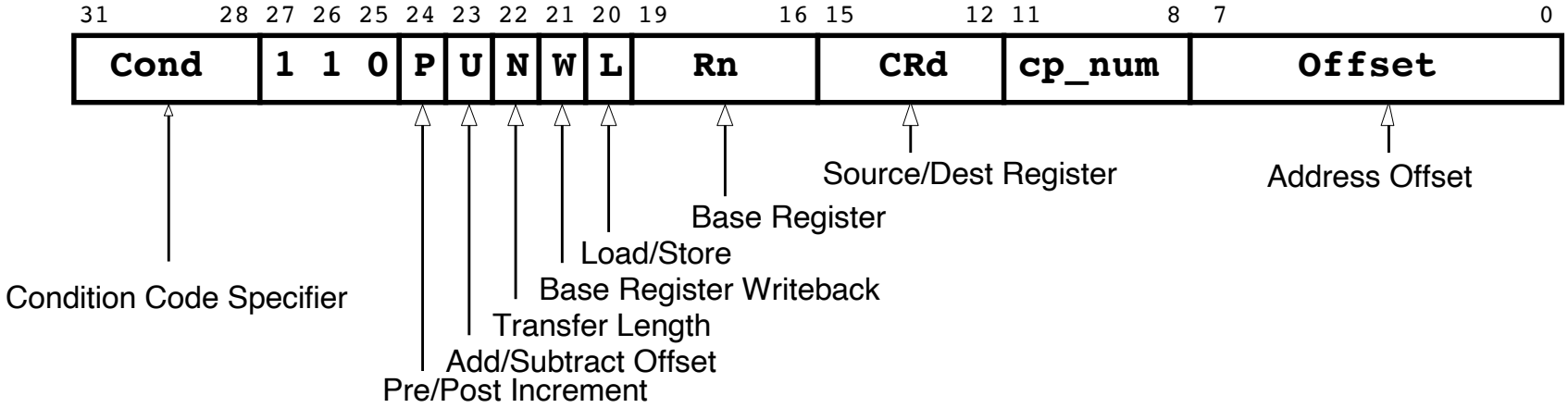
Coprocessor Register Transfers

* **These two instructions move data between ARM registers and coprocessor registers**
  - MRC : Move to Register from Coprocessor
  - MCR : Move to Coprocessor from Register
* **An operation may also be performed on the data as it is transferred**
  - For example a Floating Point Convert to Integer instruction can be implemented as a register transfer to ARM that also converts the data from floating point format to integer format.
* **Syntax**
  - `<MRC|MCR>{<cond>} <cp_num>,<opc_1>,Rd,CRn,CRm,<opc_2>`



Coprocessor Memory Transfers (1)

* Load from memory to coprocessor registers
* Store to memory from coprocessor registers.



Coprocessor Memory Transfers (2)

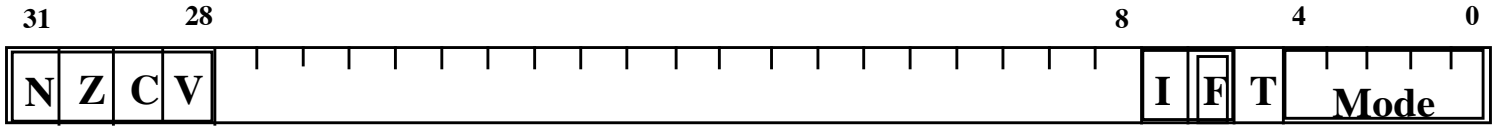
* **Syntax of these is similar to word transfers between ARM and memory:**
  - `<LDC|STC>{<cond>}{<L>} <cp_num>,CRd,<address>`
    - PC relative offset generated if possible, else causes an error.
  - `<LDC|STC>{<cond>}{<L>} <cp_num>,CRd,<[Rn,offset]{!}>`
    - Pre-indexed form, with optional writeback of the base register
  - `<LDC|STC>{<cond>}{<L>} <cp_num>,CRd,<[Rn],offset>`
    - Post-indexed form

  where

  - <L> when present causes a "long" transfer to be performed (N=1) else causes a "short" transfer to be performed (N=0).
    - Effect of this is coprocessor dependant.

Quiz #6

* **Write a short code segment that performs a mode change by modifying the contents of the CPSR**
  - The mode you should change to is user mode which has the value 0x10.
  - This assumes that the current mode is a priveleged mode such as supervisor mode.
  - This would happen for instance when the processor is reset - reset code would be run in supervisor mode which would then need to switch to user mode before calling the main routine in your application.
  - You will need to use MSR and MRS, plus 2 logical operations.



Quiz #6 - Sample Solution

* **Set up useful constants:**

```
mmask   EQU     0x1f            ; mask to clear mode bits
userm   EQU     0x10            ; user mode value
```

* **Start off here in supervisor mode.**

```
        MRS     r0, cpsr        ; take a copy of the CPSR
        BIC     r0,r0,#mmask    ; clear the mode bits
        ORR     r0,r0,#userm    ; select new mode
        MSR     cpsr, r0        ; write back the modified
                                ; CPSR
```

* **End up here in user mode.**

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**