SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A’ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ARM-Thumb Instruction Set

Dr.G.Arthy
Assistant Professor
Department of EEE
SNS College of Engineering

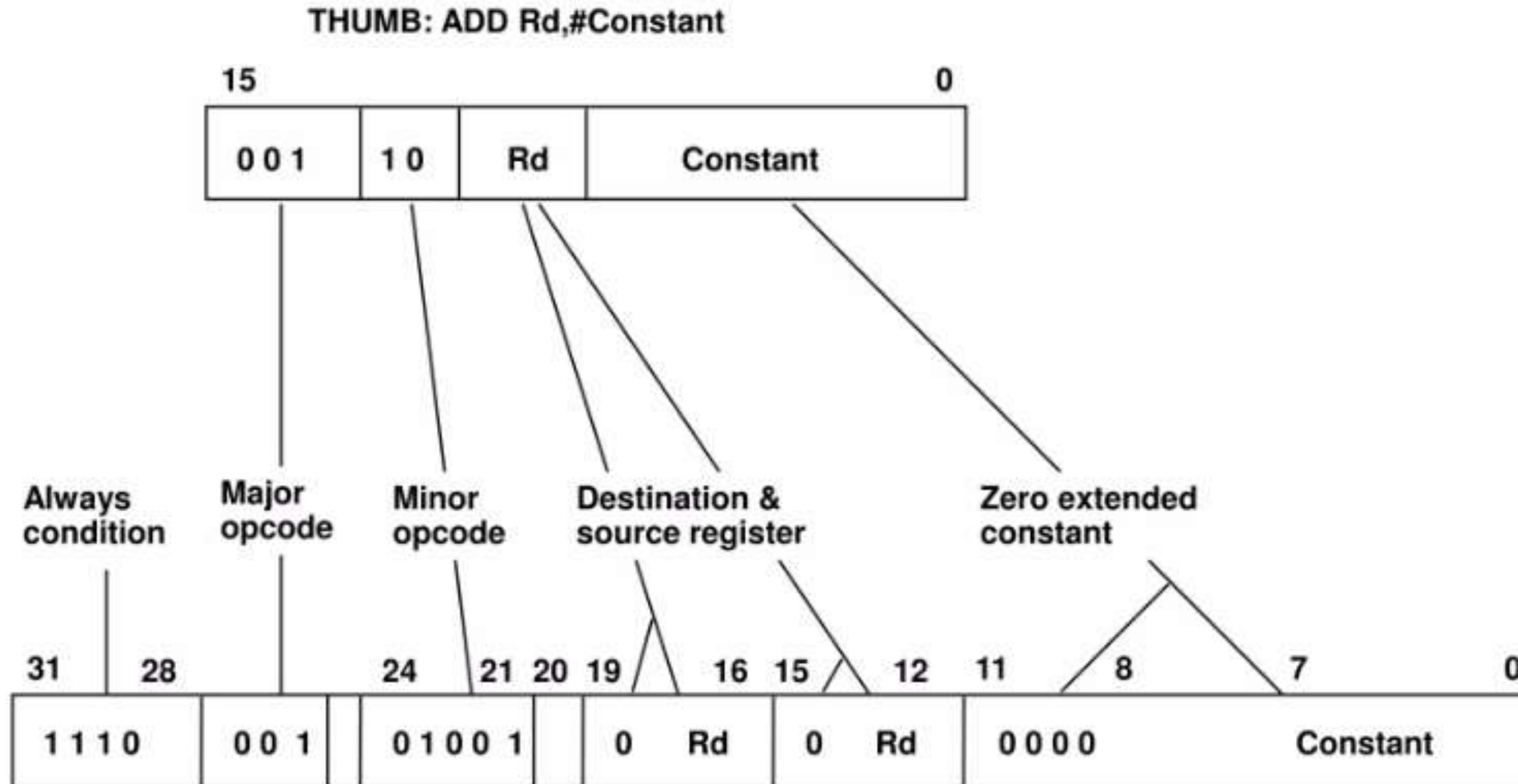
Thumb Instruction Set Advantages

- All instructions are exactly 16 bits long to improve code density over other 32-bit architectures
- The Thumb architecture still uses a 32-bit core, with:
  - 32-bit address space
  - 32-bit registers
  - 32-bit shifter and ALU
  - 32-bit memory transfer
- Gives....
  - Long branch range
  - Powerful arithmetic operations
  - Large address space

How Does Thumb Work ?

- **The Thumb instruction set is a subset of the ARM instruction set, optimized for code density.**
- **Almost every Thumb instructions have an ARM instructions equivalent:**
  - ADD Rd, #Offset8 <> ADDS Rd, Rd, #Offset8
- **Inline expansion of Thumb Instruction to ARM Instruction**
  - Real time decompression
  - Thumb instructions are not actually executed on the core
- **The core needs to know whether it is reading Thumb instructions or ARM instructions.**
  - Core has two execution states - ARM and Thumb
  - Core does not have a mixed 16 and 32 bit instruction set.

Thumb Instruction Set Decompression



Branch Instructions

- **Thumb supports four types of branch instruction:**
  - an unconditional branch that allows a forward or backward branch of up to 2Kbytes
  - a conditional branch to allow forward and backward branches of up to 256 bytes
  - a branch with link is supported with a pair of instructions that allow forward and backwards branches of up to 4Mbytes
  - a branch and exchange instruction branches to an address in a register and optionally switches to ARM code execution

- **List of branch instructions**
  - B conditional branch
  - B unconditional branch
  - BL Branch with link
  - BX Branch and exchange instruction set

Data Processing Instructions

- Thumb data-processing instructions are a subset of the ARM data-processing instructions
  - All Thumb data-processing instructions set the condition codes

- List of data-processing instructions
  - ADC, Add with Carry
  - ADD, Add
  - AND, Logical AND
  - ASR, Arithmetic shift right
  - BIC, Bit clear
  - CMN, Compare negative
  - CMP, Compare
  - EOR, Exclusive OR
  - LSL, Logical shift left
  - LSR, Logical shift right
  - MOV, Move
  - MUL, Multiply
  - MVN, Move NOT
  - NEG, Negate
  - ORR, Logical OR
  - ROR, Rotate Right
  - SBC, Subtract with Carry
  - SUB, Subtract
  - TST, Test

Load and Store Register Instructions

- **Thumb supports 8 types of load and store register instructions**

- **List of load and store register instructions**

– LDR	Load word
– LDRB	Load unsigned byte
– LDRH	Load unsigned halfword
– LDRSB	Load signed byte
– LDRSH	Load signed halfword
– STR	Store word
– STRB	Store byte
– STRH	Store halfword

Load and Store Multiple Instructions

- **Thumb supports four types of load and store multiple instructions**
- **Two (a load and store) are designed to support block copy**
- **The other two instructions (called PUSH and POP) implement a full descending stack, and the stack pointer is used as the base register**
- **List of load and store multiple instructions**
  - LDM Load multiple
  - POP Pop multiple
  - PUSH Push multiple
  - STM Store multiple

Thumb Register Usage

- In thumb state we can not access all registers directly.
- Summary of Thumb register usage.

Registers	Access
r0–r7	fully accessible
r8–r12	only accessible by MOV, ADD, and CMP
r13 sp	limited accessibility
r14 lr	limited accessibility
r15 pc	limited accessibility
cpsr	only indirect access
spsr	no access

ARM-Thumb Interworking

- *ARM-Thumb interworking is the name given to the method of linking ARM and Thumb* code together for both assembly and C/C++. It handles the transition between the two states.
- To call a Thumb routine from an ARM routine, the core has to change state. This state change is shown in the *T bit of the cpsr.*
- *The BX and BLX branch instructions cause a switch* between ARM and Thumb state while branching to a routine.

- Syntax:
  - BX Rm
  - BLX Rm | label

BX	Thumb version branch exchange	$pc = Rn$ & 0xfffffffe $T = Rn[0]$
BLX	Thumb version of the branch exchange with link	lr = (instruction address after the BLX) + 1 $pc = label$, $T = 0$ $pc = Rm$ & 0xfffffffe, $T = Rm[0]$

```
; ARM code
        CODE32                          ; word aligned
        LDR     r0, =thumbCode+1        ; +1 to enter Thumb state
        MOV     lr, pc                  ; set the return address
        BX      r0                      ; branch to Thumb code & mode
        ; continue here
; Thumb code
        CODE16                          ; halfword aligned
thumbCode
        ADD   r1, #1
        BX    lr                        ; return to ARM code & state
```

Data Processing Instructions

- The data processing instructions manipulate data within registers. They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions, and multiply instructions. The Thumb data processing instructions are a subset of the ARM data processing instructions.

- Syntax:
- <ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB> Rd, Rm
- <ADD|ASR|LSL|LSR|ROR|SUB> Rd, Rn #immediate
- <ADD|MOV|SUB> Rd,#immediate
- <ADD|SUB> Rd,Rn,Rm
- ADD Rd,pc,#immediate
- ADD Rd,sp,#immediate
- <ADD|SUB> sp, #immediate
- <ASR|LSL|LSR|ROR> Rd,Rs
- <CMN|CMP|TST> Rn,Rm
- CMP Rn,#immediate
- MOV Rd,Rn

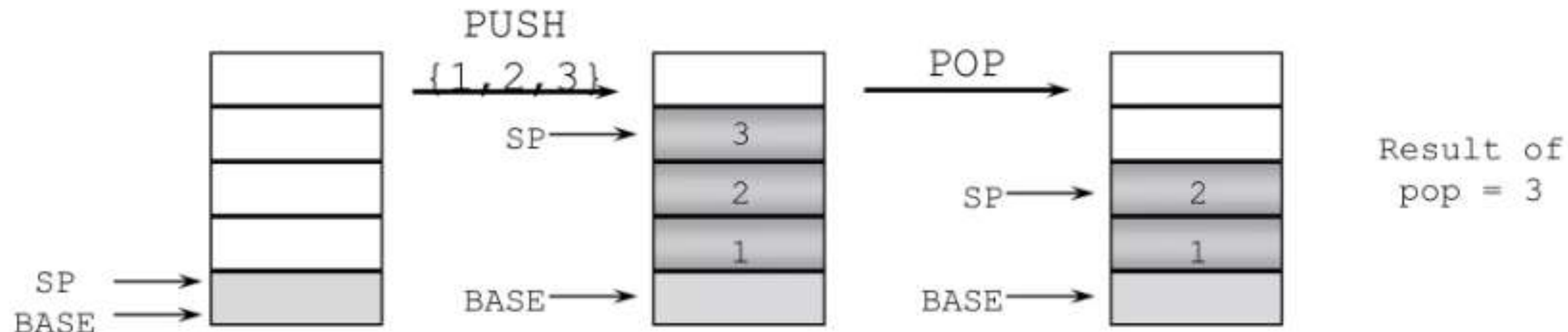
ADC	add two 32-bit values and carry	$Rd = Rd + Rm + C$ flag
ADD	add two 32-bit values	$Rd = Rn + immediate$ $Rd = Rd + immediate$ $Rd = Rd + Rm$ $Rd = Rd + Rm$ $Rd = (pc \& 0xfffffffc) + (immediate \ll 2)$ $Rd = sp + (immediate \ll 2)$ $sp = sp + (immediate \ll 2)$

This example shows a simple Thumb ADD instruction. It takes two low registers *r1 and r2* and adds them together. The result is then placed into register *r0, overwriting the original* contents. The *cpsr is also updated.*

PRE cpsr = nzcvIFT_SVC
r1 = 0x80000000
r2 = 0x10000000
ADD r0, r1, r2
POST r0 = 0x90000000
cpsr = **NzcvIFT_SVC**

Stacks

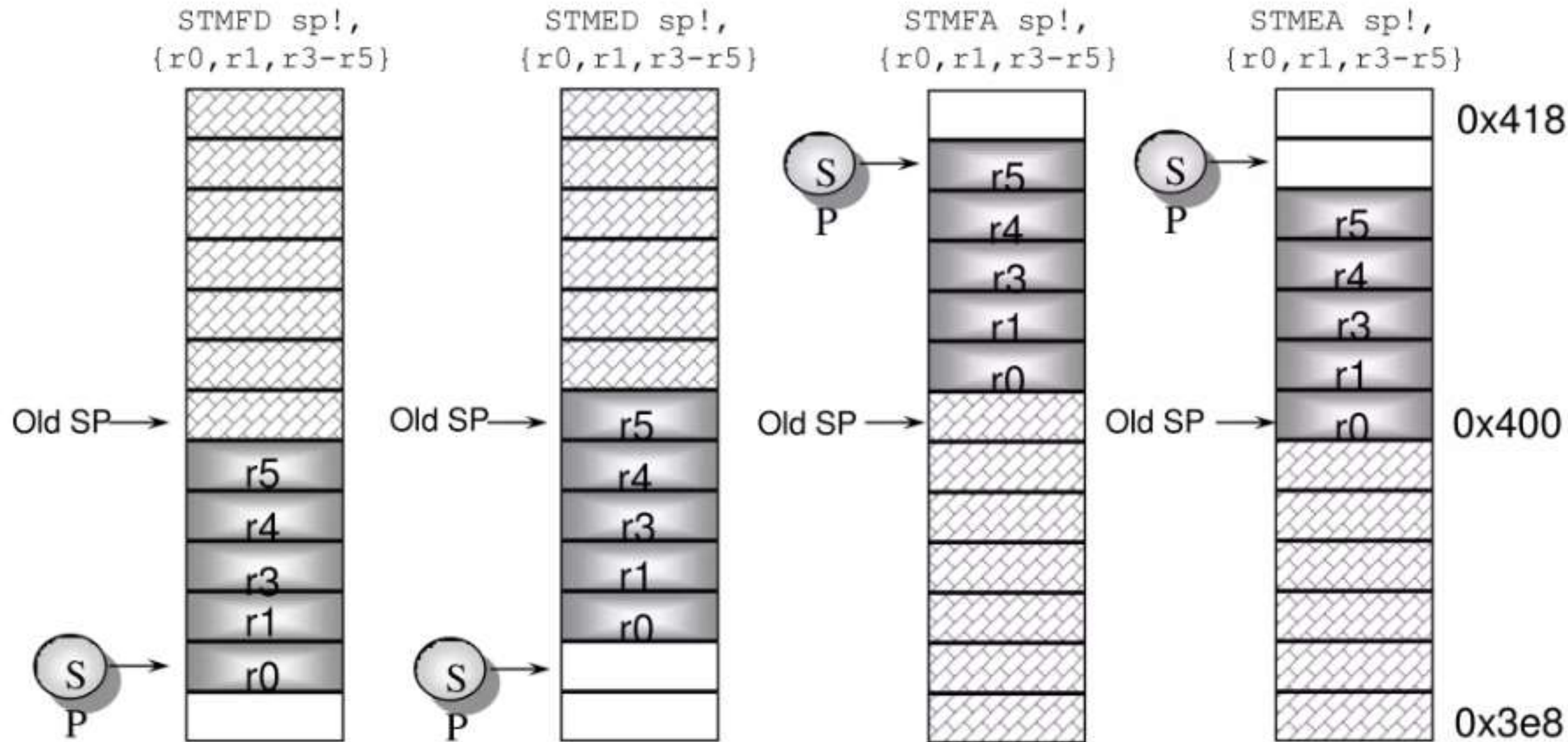
- **A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.**
- **Two pointers define the current limits of the stack.**
  - A base pointer
    - used to point to the "bottom" of the stack (the first location).
  - A stack pointer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



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).

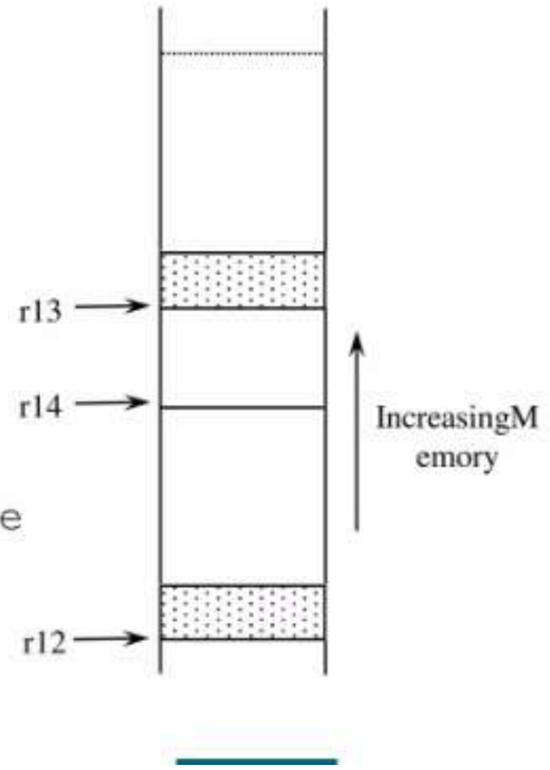
Direct functionality of Block Data Transfer

* **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.

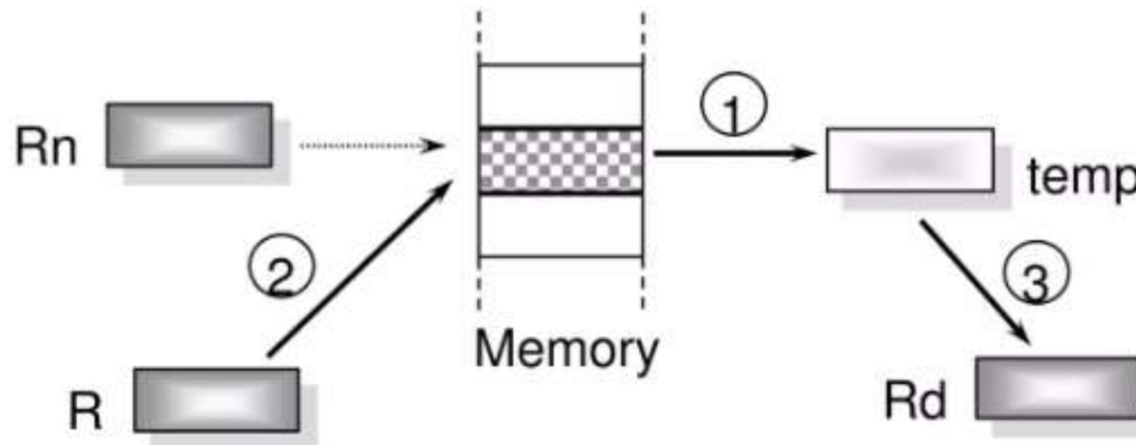
```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11}  ; load 48 bytes
        STMIA   r13!, {r0-r11}  ; and store them
        CMP     r12, r14        ; check for the end
        BNE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



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

Swap and Swap Byte Instructions

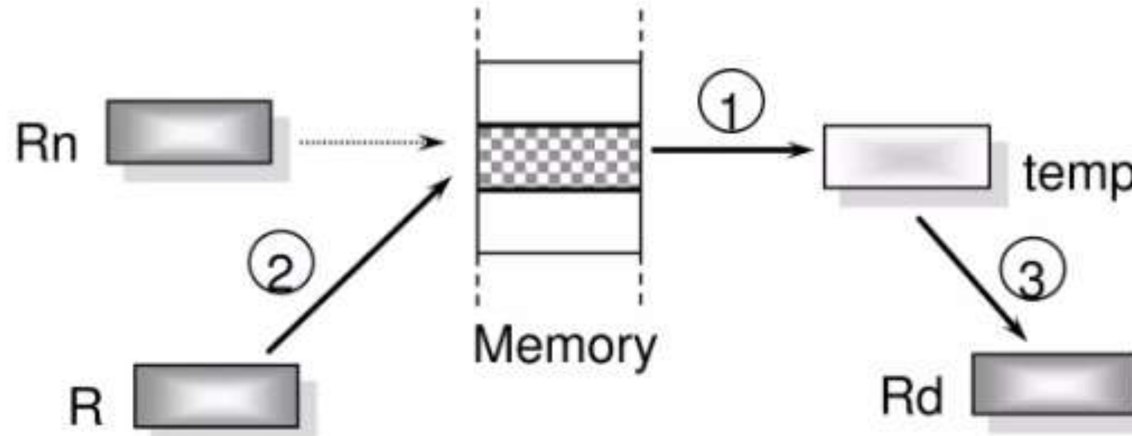
* **Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.**
* **Syntax:**
  - SWP{<cond>}{B} Rd, Rm, [Rn]



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

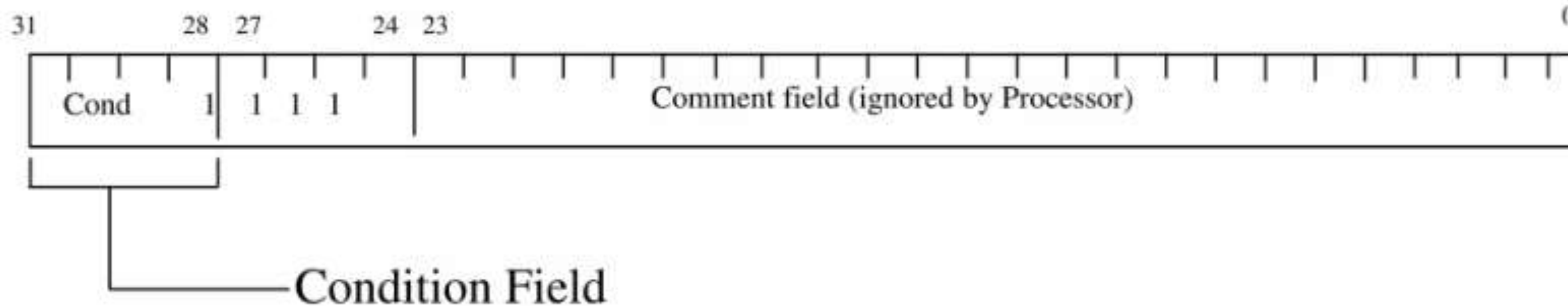
Swap and Swap Byte Instructions

* 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.

Software Interrupt Instruction

- The Thumb software interrupt (SWI) instruction causes a software interrupt exception. If any interrupt or exception flag is raised in Thumb state,the processor automatically reverts back to ARM state to handle the exception
- Syntax: SWI immediate

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts) *cpsr T* = 0 (ARM state)

ASSESSMENT

1) What is the advantage of using Thumb instruction?

2)How many branch instruction does Thumb support?

Thank you