ARM Thumb

Thumb Registers

ARM Thumb interworking

Thumb Instructions

Thumb Instruction Set

16-bit instruction set

Compared to ARM, Thumb has

- high performance if the processor data bus is 16-bit
- low performance if the data bus is 32 bit

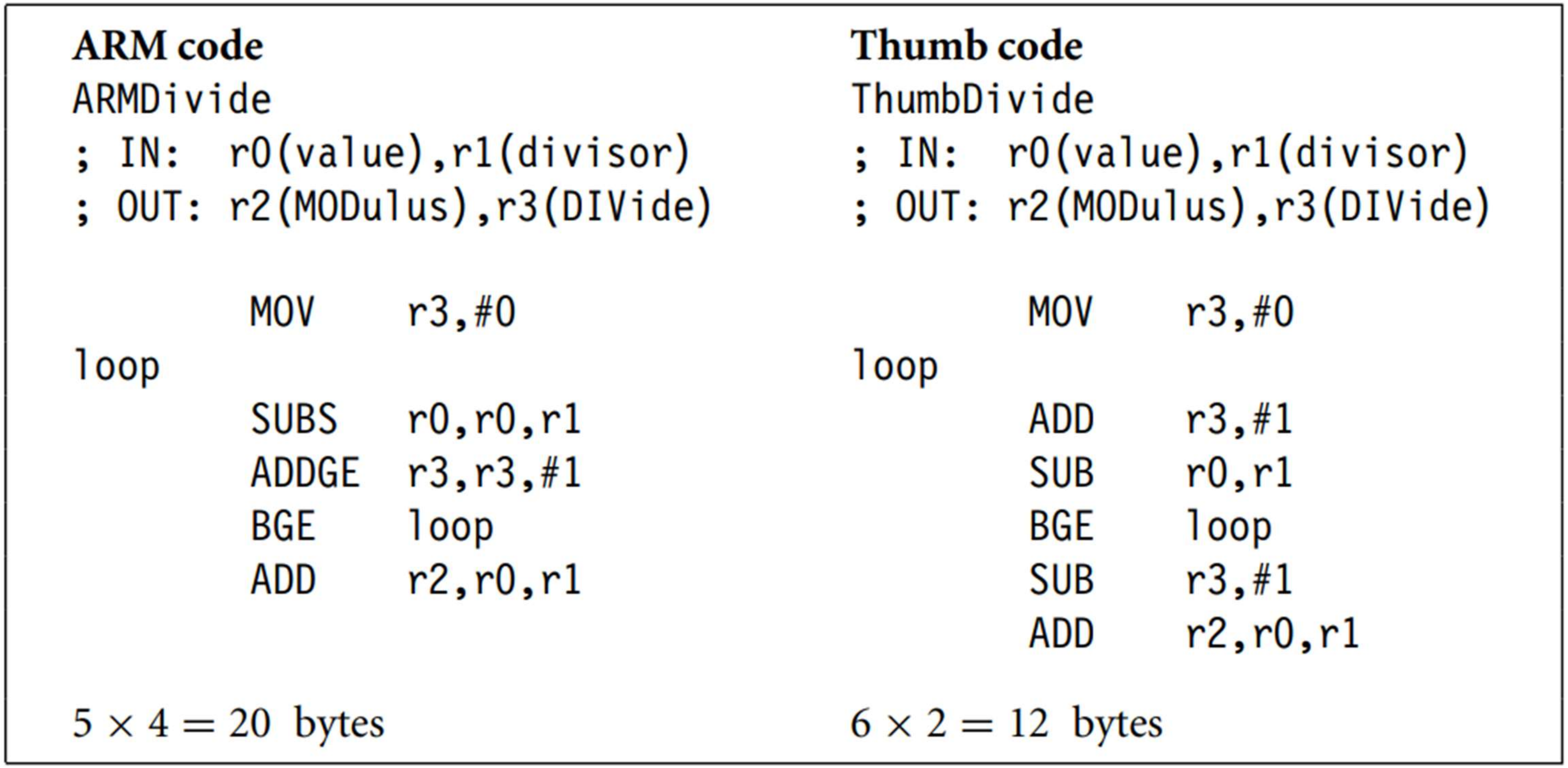
Use Thumb for memory constrained systems

High code density - memory space occupied by an executable program

Thumb implementation takes up 30% less memory

Thumb

- Thumb implementation uses more instructions yet memory required is less
- Code density was the main driving force for the Thumb instruction set



ARM code

```
ARMDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        SUBS    r0,r0,r1
        ADDGE   r3,r3,#1
        BGE     loop
        ADD     r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

```
ThumbDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        ADD     r3,#1
        SUB     r0,r1
        BGE     loop
        SUB     r3,#1
        ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Code density.

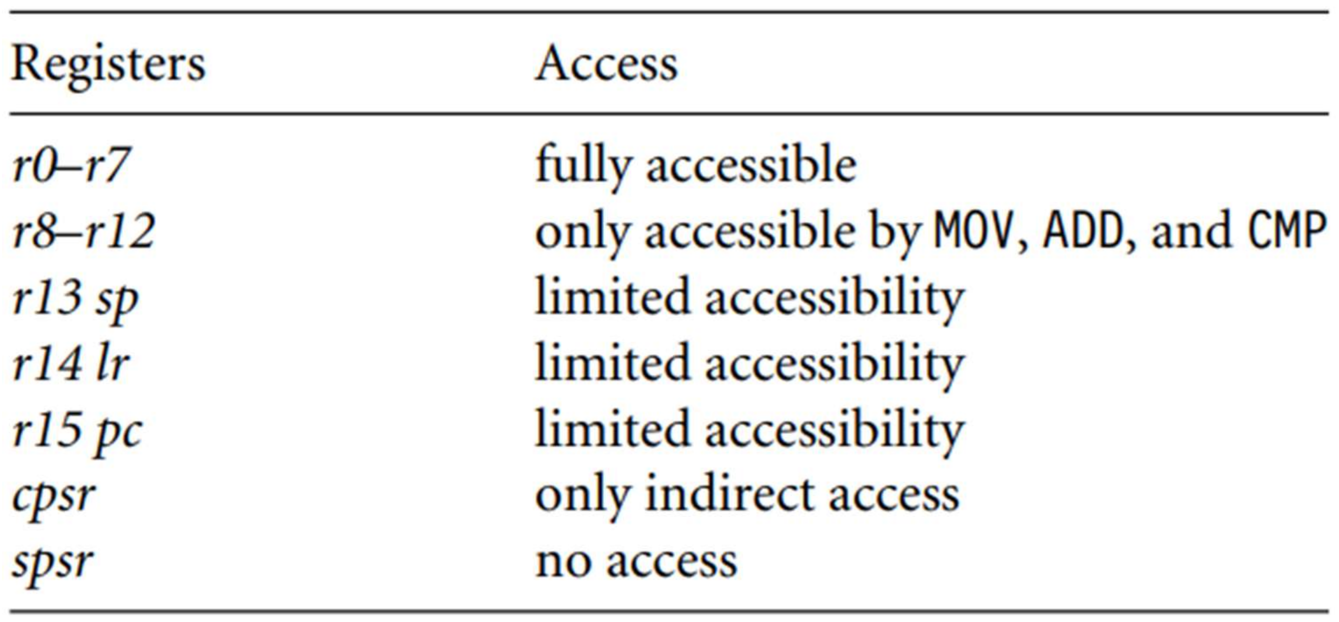
Thumb instruction set.

Mnemonics	THUMB ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
ASR	v1	arithmetic shift right
B	v1	branch relative
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v2	breakpoint instructions
BL	v1	relative branch with link
BLX	v2	branch with link and exchange
BX	v1	branch with exchange
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit integers
EOR	v1	logical exclusive OR of two 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1	load a single value from a virtual address in memory
LSL	v1	logical shift left
LSR	v1	logical shift right
MOV	v1	move a 32-bit value into a register
MUL	v1	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
NEG	v1	negate a 32-bit value
ORR	v1	logical bitwise OR of two 32-bit values
POP	v1	pops multiple registers from the stack
PUSH	v1	pushes multiple registers to the stack
ROR	v1	rotate right a 32-bit value
SBC	v1	subtract with carry a 32-bit value
STM	v1	store multiple 32-bit registers to memory
STR	v1	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
TST	v1	test bits of a 32-bit value

Thumb Register

- Thumb can access only low registers - r0 - r7
- r8 - r12 - accessible only to MOV, ADD, CMP
- CMP and all data processing instructions operating on low registers update the condition flags in cpsr
- There is no direct access to the cpsr or spsr
- There are no MRS and MSR equivalent instructions
- No coprocessor instructions in the Thumb state

Thumb Register



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 -

- method of linking ARM and Thumb code together for both assembly and C/C++
- handles the transition between the two states
- Veneer - extra code - needed to carry out the transition
- ATPCS - defines the ARM and Thumb procedure call standards

ARM - THUMB INTERWORKING

- From an ARM routine, if Thumb routine is called, a state change happens and it is set in T bit of cpsr.
- BX and BLX instructions cause a switch between ARM and Thumb state.
- Instead of calling the routine directly, the linker calls the veneer, which switches to Thumb state using the BX instruction.

ARM - THUMB INTERWORKING

- BX and BLX instructions version - ARM and Thumb
- ARM BX enters Thumb state only if bit 0 of the address in Rn is set to 1.
- otherwise it enters ARM state.

```
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 - THUMB INTERWORKING

ARM BX can be conditionally executed.

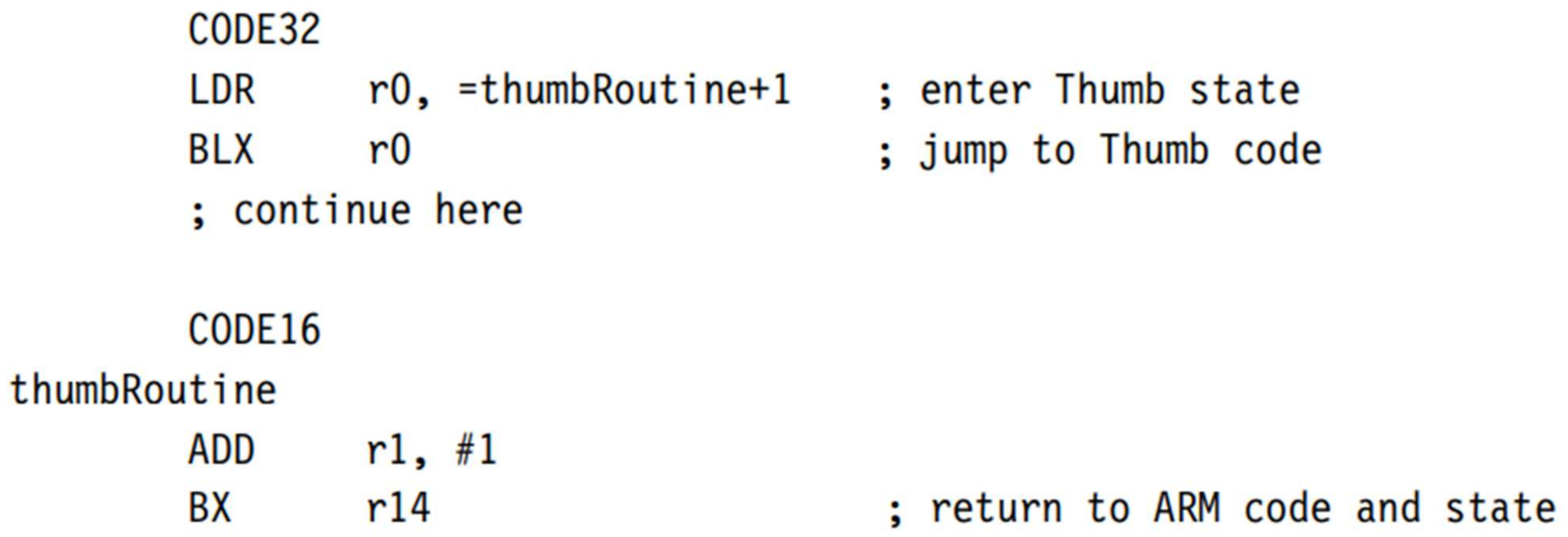
However, Thumb BX cannot be conditionally executed.

Branch address into Thumb has the lowest bit set (Sets T bit in cpsr).

Prior to Branch, code sets the return address using MOV instruction.

```
; 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
```

BX to BLX - Return address is set in the lr



```
        CODE32
        LDR     r0, =thumbRoutine+1    ; enter Thumb state
        BLX     r0                     ; jump to Thumb code
        ; continue here

        CODE16
thumbRoutine
        ADD     r1, #1
        BX      r14                     ; return to ARM code and state
```

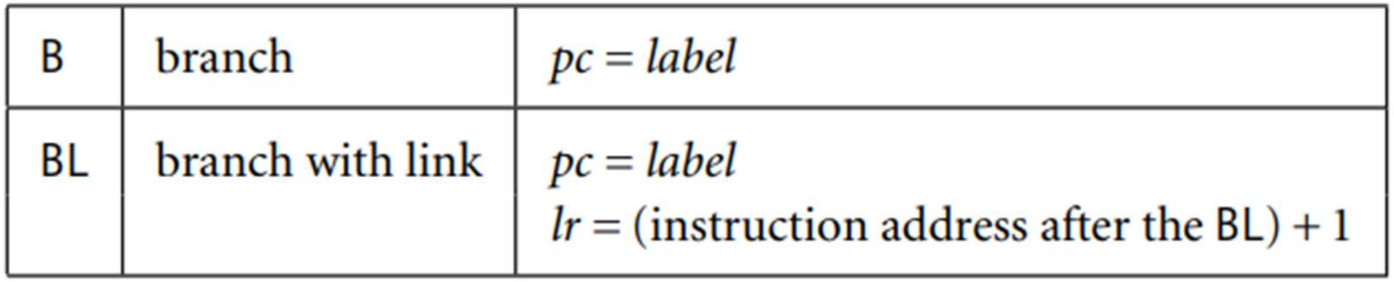
BRANCH INSTRUCTION

1. Similar to ARM version and conditionally executed. Branch range is limited to a signed 8-bit immediate.
2. Conditional part is removed and expands the effective branch range to a signed 11-bit immediate
3. The conditional branch instruction is the only conditionally executed instruction in Thumb state.

BRANCH INSTRUCTION

- BL instruction is not conditionally executed and has an approximate range of +/- 4 MB.
- This range is possible because BL instructions are translated into a pair of 16-bit Thumb instructions.
- The first instruction in the pair holds the high part of the branch offset and the second the low part.
- These instructions must be used as a pair.

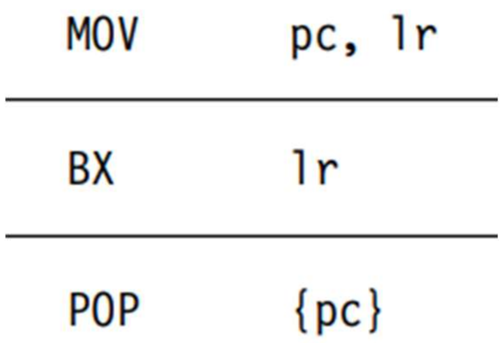
```
Syntax: B<cond> label
        B label
        BL label
```



B	branch	$pc = label$
BL	branch with link	$pc = label$ $lr =$ (instruction address after the BL) + 1

BRANCH INSTRUCTION

The following instructions are used to return from a subroutine call.



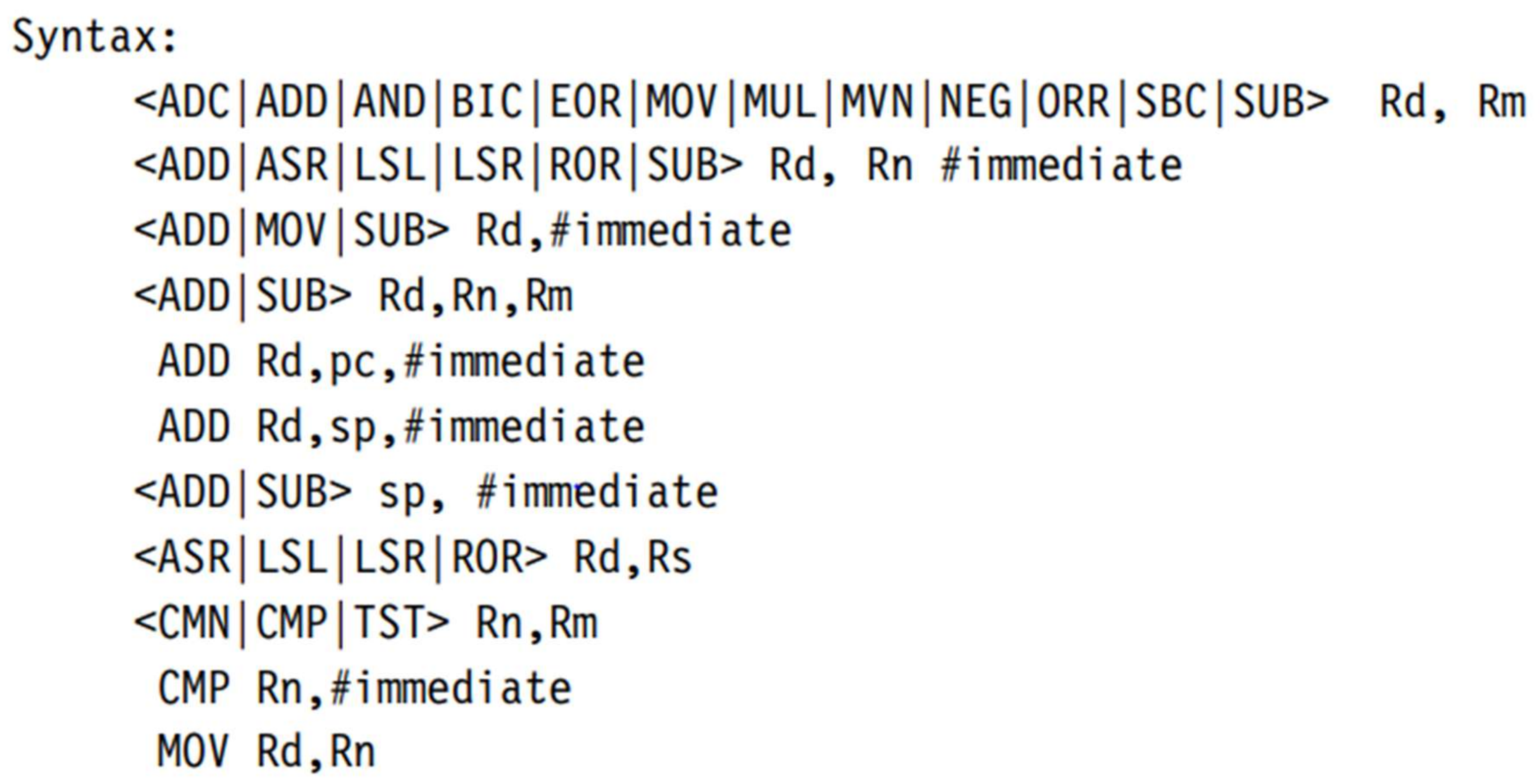
MOV	pc, lr
BX	lr
POP	{pc}

Data Processing Instructions

- Manipulates data within registers
- move instructions
- arithmetic instructions
- shifts
- logical instructions
- comparison instructions
- multiply instructions

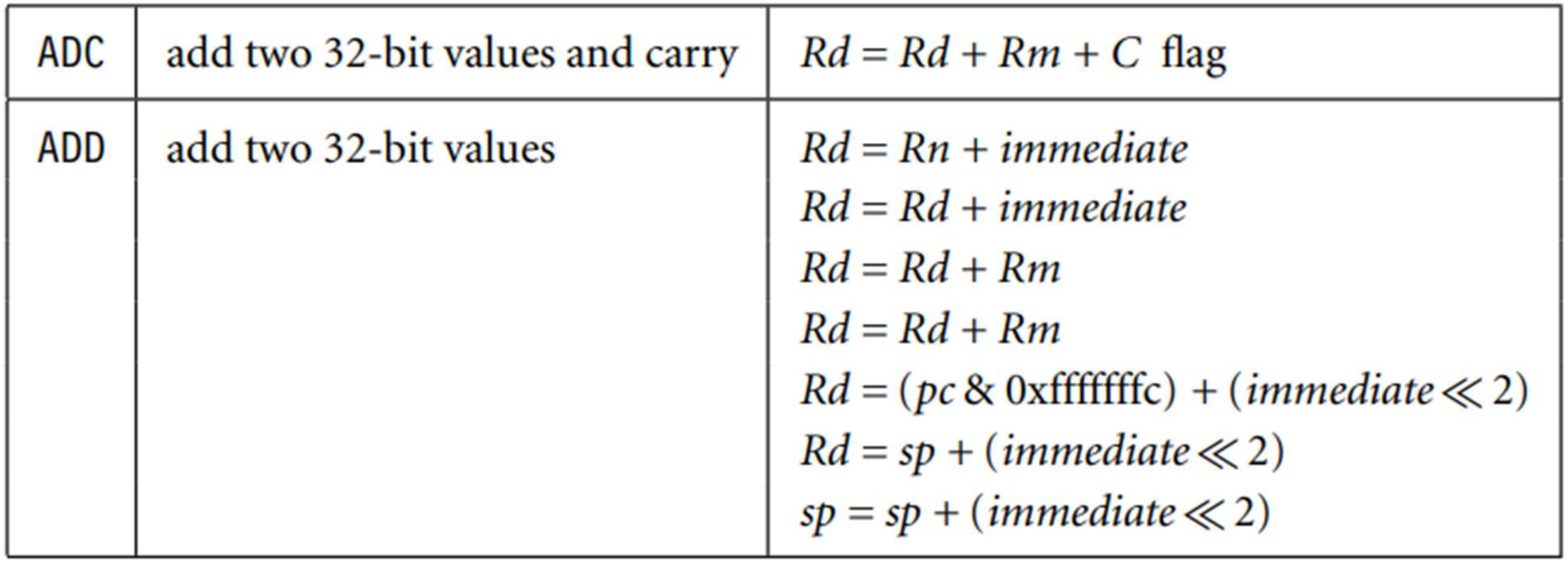
The Thumb data processing instructions are a subset of the ARM data processing instructions.

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
```

Data Processing Instructions



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

AND	logical bitwise AND of two 32-bit values	$Rd = Rd \;\&\; Rm$
ASR	arithmetic shift right	$Rd = Rm \gg immediate,$ C flag $= Rm[immediate - 1]$ $Rd = Rd \gg Rs$, C flag $= Rd[Rs - 1]$
BIC	logical bit clear (AND NOT) of two 32-bit values	$Rd = Rd$ AND NOT(Rm)
CMN	compare negative two 32-bit values	$Rn + Rm$ *sets flags*
CMP	compare two 32-bit integers	$Rn - immediate$ *sets flags* $Rn - Rm$ *sets flags*
EOR	logical exclusive OR of two 32-bit values	$Rd = Rd$ EOR Rm
LSL	logical shift left	$Rd = Rm \ll immediate,$ C flag $= Rm[32 - immediate]$ $Rd = Rd \ll Rs$, C flag $= Rd[32 - Rs]$
LSR	logical shift right	$Rd = Rm \gg immediate,$ C flag $= Rd[immediate - 1]$ $Rd = Rd \gg Rs$, C flag $= Rd[Rs - 1]$
MOV	move a 32-bit value into a register	$Rd = immediate$ $Rd = Rn$ $Rd = Rm$
MUL	multiply two 32-bit values	$Rd = (Rm * Rd)[31{:}0]$
MVN	move the logical NOT of a 32-bit value into a register	$Rd =$ NOT(Rm)
NEG	negate a 32-bit value	$Rd = 0 - Rm$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rd$ OR Rm
ROR	rotate right a 32-bit value	$Rd = Rd$ RIGHT_ROTATE Rs, C flag $= Rd[Rs{-}1]$
SBC	subtract with carry a 32-bit value	$Rd = Rd - Rm -$ NOT(C flag)
SUB	subtract two 32-bit values	$Rd = Rn - immediate$ $Rd = Rd - immediate$ $Rd = Rn - Rm$ $sp = sp - (immediate \ll 2)$
TST	test bits of a 32-bit value	Rn AND Rm *sets flags*

- Thumb data processing instructions operate on low registers and update the cpsr.
- The exceptions are: Operating on high registers r8-r14 and the pc.
- These instructions, except for CMP, do not update the condition flags in the cpsr when using the high registers.
- The CMP instruction always updates the cpsr.

Thumb Add Instruction

- Barrel Shift operations are separate instructions (ASR, LSL, LSR, and ROR).
- Logical shift left to multiply register r2 by 2.

```
PRE     cpsr = nzcvIFT_SVC
        r1 = 0x80000000
        r2 = 0x10000000

        ADD     r0, r1, r2

POST    r0 = 0x90000000
        cpsr = NzcvIFT_SVC
```

Single-Register Load Store Instructions

It uses two pre-indexed addressing modes.

- offset by register
- offset by immediate

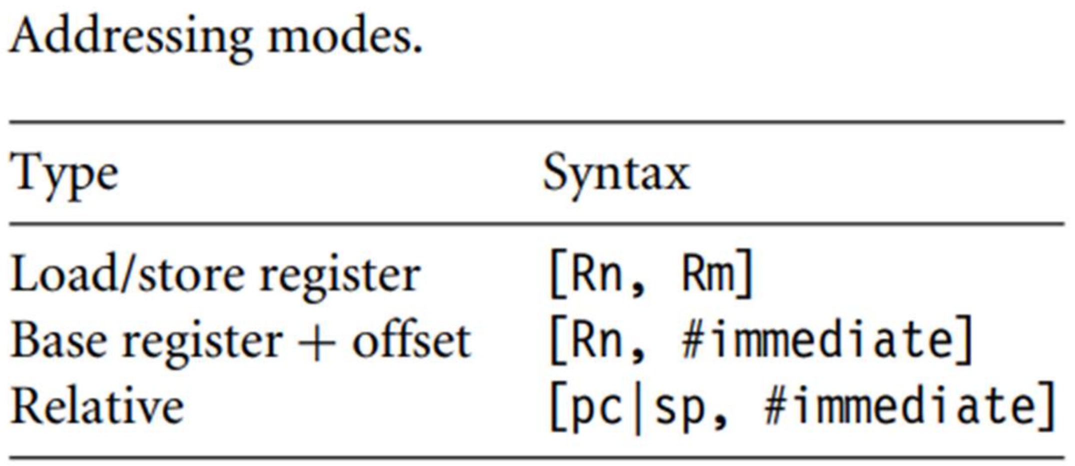
```
Syntax: <LDR|STR>{<B|H>} Rd, [Rn,#immediate]
        LDR{<H|SB|SH>} Rd,[Rn,Rm]
        STR{<B|H>} Rd,[Rn,Rm]
        LDR Rd,[pc,#immediate]
        <LDR|STR> Rd,[sp,#immediate]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend(mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend(mem16[address])*

Single-Register Load Store Instructions

- The offset by register uses a base register Rn plus the register offset Rm.
- The second uses the same base register Rn plus a 5-bit immediate or a value dependent on the data size.
- The 5-bit offset encoded in the instruction is multiplied by one for byte accesses, two for 16-bit accesses, and four for 32-bit accesses.

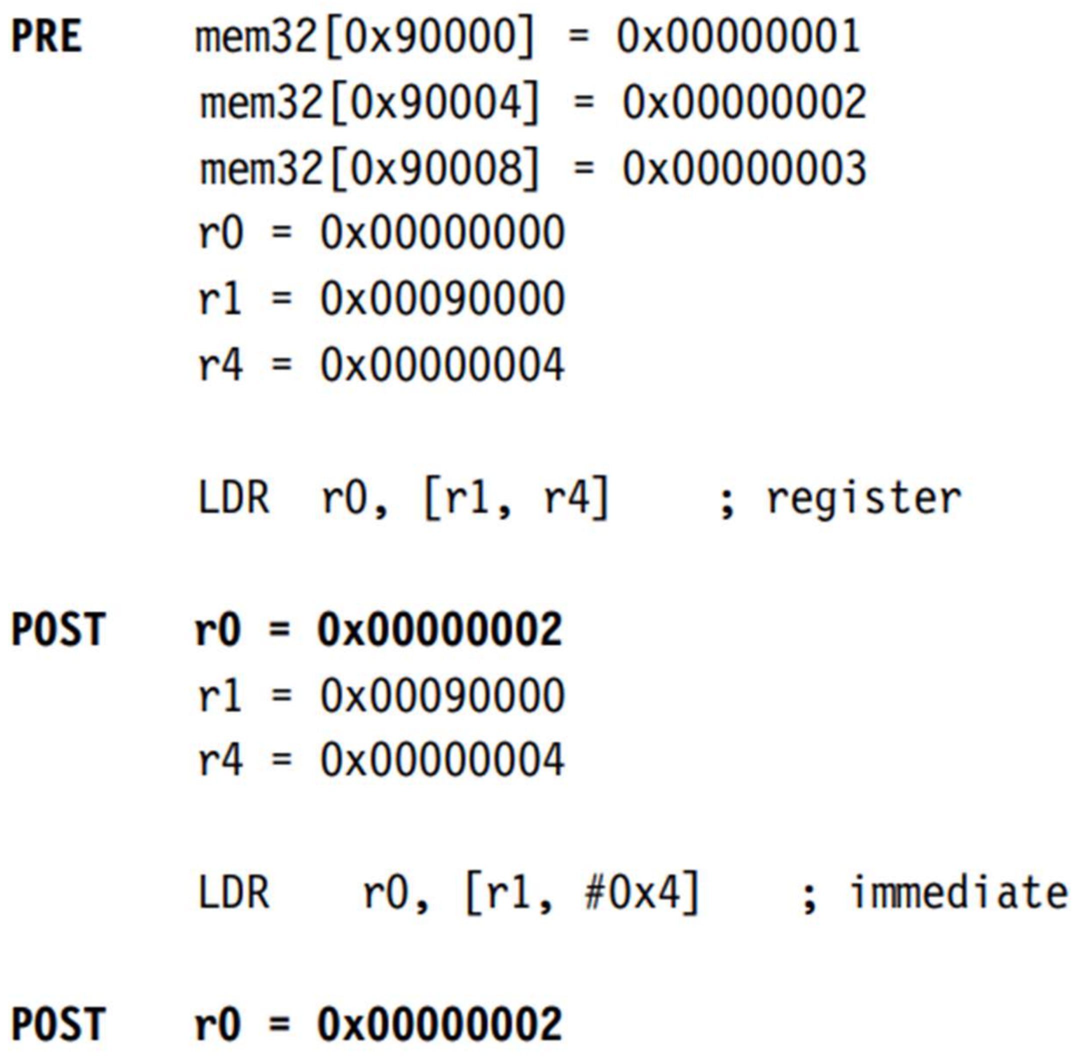
Addressing modes



Addressing modes.

Type	Syntax
Load/store register	`[Rn, Rm]`
Base register + offset	`[Rn, #immediate]`
Relative	`[pc|sp, #immediate]`

Single-Register Load Instructions



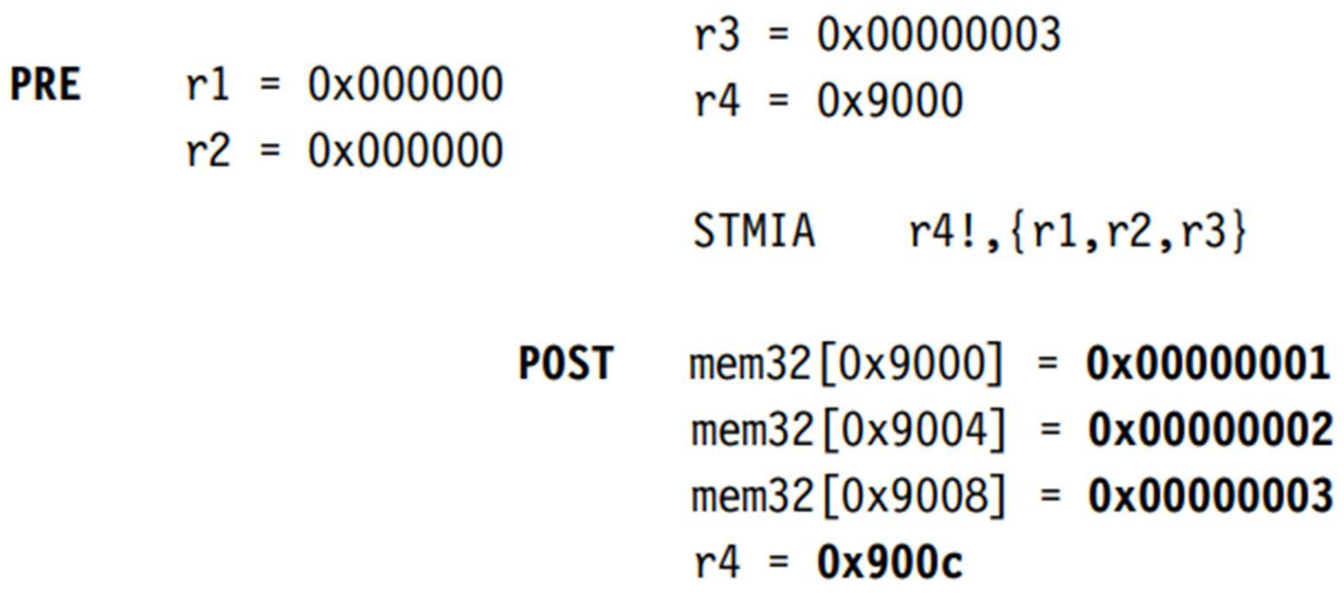
```
PRE     mem32[0x90000] = 0x00000001
        mem32[0x90004] = 0x00000002
        mem32[0x90008] = 0x00000003
        r0 = 0x00000000
        r1 = 0x00090000
        r4 = 0x00000004

        LDR  r0, [r1, r4]     ; register

POST    r0 = 0x00000002
        r1 = 0x00090000
        r4 = 0x00000004

        LDR    r0, [r1, #0x4]    ; immediate

POST    r0 = 0x00000002
```

- Both instructions carry out the same operation.
- Second LDR uses a fixed offset
- first one depends on the value in r4

Multiple register load-store instructions

- The Thumb versions of the load-store multiple instructions are reduced forms of the ARM load-store multiple instructions.
- They only support the increment after addressing mode.
- N - number of registers in the list of registers.
- These instructions always update the base register Rn after execution.
- The base register and list of registers are limited to the low registers r0-r7.

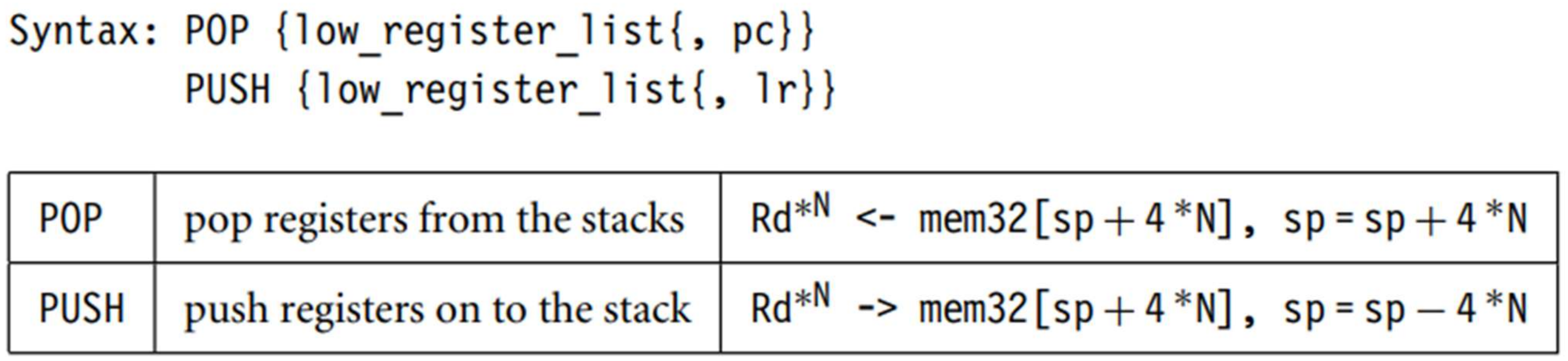
- This example saves registers r1 to r3 to memory address 0x9000 to 0x900c
- It also updates base register r4
- The update character ! is not an option, unlike with the ARM instruction set



```
PRE     r1 = 0x000000
        r2 = 0x000000
        r3 = 0x00000003
        r4 = 0x9000

        STMIA    r4!,{r1,r2,r3}

POST    mem32[0x9000] = 0x00000001
        mem32[0x9004] = 0x00000002
        mem32[0x9008] = 0x00000003
        r4 = 0x900c
```

Stack instructions

- The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional push and pop.
- No stack pointer in the instruction.
- Stack pointer is fixed as register r13 in Thumb operations, SP is automatically updated.
- The list of registers is limited to low registers r0-r7.

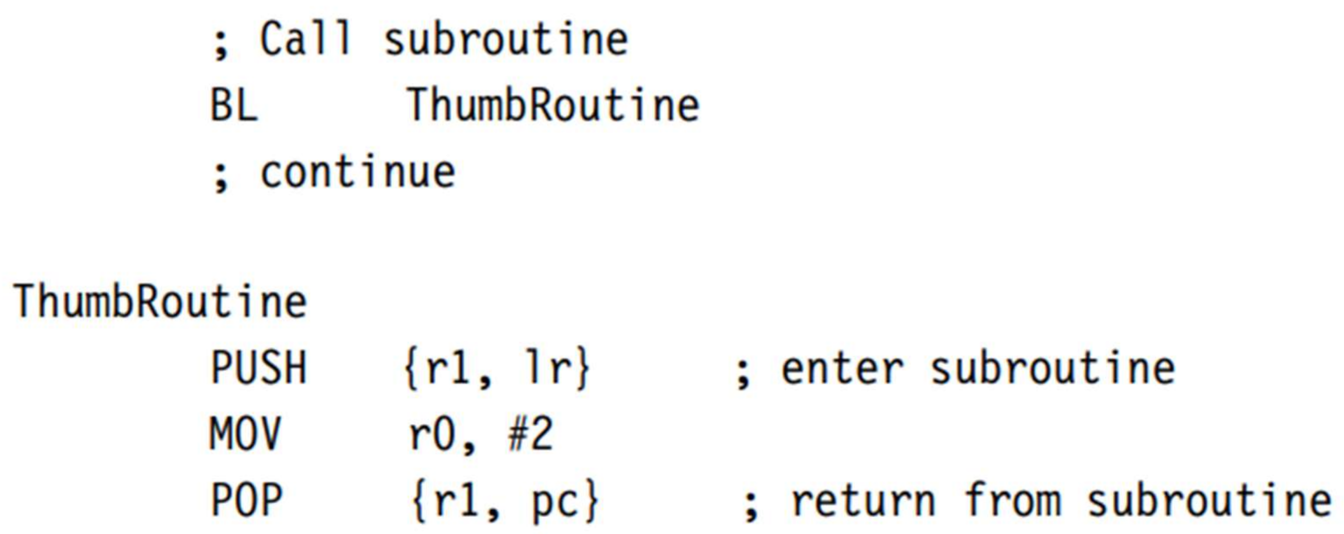


```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

POP	pop registers from the stacks	Rd^{*N} <- mem32[sp + 4 *N], sp = sp + 4 *N
PUSH	push registers on to the stack	Rd^{*N} -> mem32[sp + 4 *N], sp = sp − 4 *N

Stack instructions

- PUSH register list also can include the link register lr.
- POP register list can include pc.
- This provides support for subroutine entry and exit.
- The subroutine Thumbroutine is called using a branch with link instruction.
- The link register lr is pushed onto the stack with register r1.
- Upon return, register r1 is popped off the stack, as well as the return address being loaded into pc. This returns from the subroutine

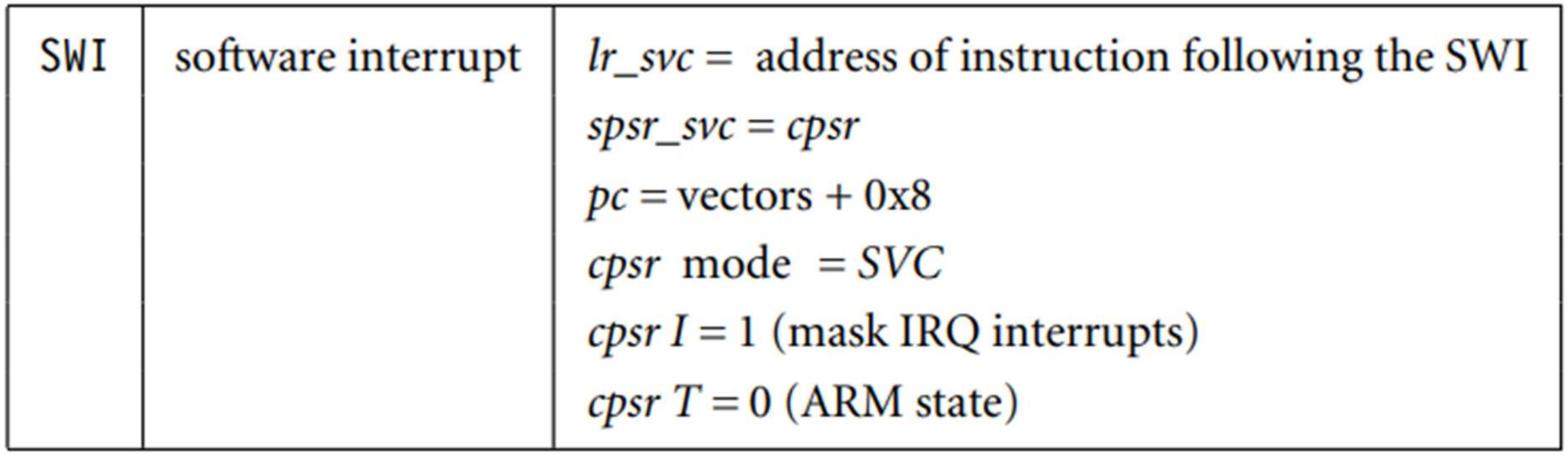


```
        ; Call subroutine
        BL      ThumbRoutine
        ; continue

ThumbRoutine
        PUSH    {r1, lr}      ; enter subroutine
        MOV     r0, #2
        POP     {r1, pc}      ; return from subroutine
```

Software interrupt instruction

- The Thumb software interrupt 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.
- The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent.
- SWI number is limited to the range 0 - 255 and it is not conditionally executed.

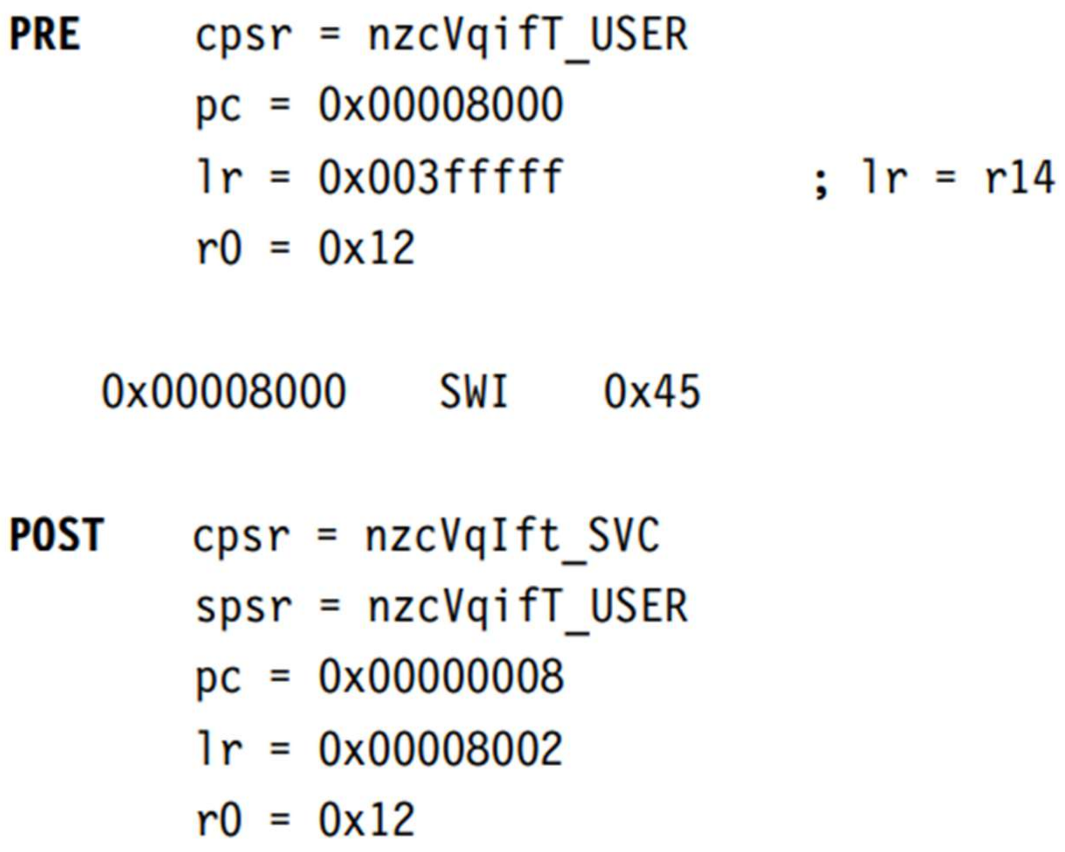
```
Syntax: SWI immediate
```



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)

Thumb SWI instruction

The processor goes from Thumb state to ARM state after execution.



```
PRE     cpsr = nzcVqifT_USER
        pc = 0x00008000
        lr = 0x003fffff              ; lr = r14
        r0 = 0x12

    0x00008000    SWI    0x45

POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqifT_USER
        pc = 0x00000008
        lr = 0x00008002
        r0 = 0x12
```