# Assembly Programming Overview (1A)

Young Won Lim
1/1/20

Copyright (c) 2014 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

# Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

# Data Processing Instruction Rules

All 32-bit operands
- Come from registers
- Are specified as literals in the instruction itself

The 32-bit result, if any
- Is placed in a register

The 64-bit result from long multiply instructions

3-address format
- Each of the operand and the result register are independently specified

# Arithmetic Operations

```
ADD   r0, r1, r2    ;   r0  :=  r1 + r2
ADC   r0, r1, r2    ;   r0  :=  r1 + r2 + C
SUB   r0, r1, r2    ;   r0  :=  r1 – r2
SBC   r0, r1, r2    ;   r0  :=  r1 – r2 + C – 1
RSB   r0, r1, r2    ;   r0  :=  r2 – r1
RSC   r0, r1, r2    ;   r0  :=  r2 – r1 + C – 1
```

# Bit-wise Logical Operations

```
AND    r0, r1, r2        ;    r0   :=   r1 and r2
ORR    r0, r1, r2        ;    r0   :=   r1 or r2
EOR    r0, r1, r2        ;    r0   :=   r1 xor r2
BIC    r0, r1, r2        ;    r0   :=   r1 and not r2
```

# Register Movement Operations

```
MOV    r0, r2        ;    r0   :=   r2
MVN    r0, r2        ;    r0   :=   not r2
```

# Immediate Operands

```
ADD     r3, r3, #1          ;    r3   := r3 + 1
AND     r8, r7, #&ff        ;    r8   := r7[7:0]
```

ff = 0000 0000 0000 0000 0000 0000 1111 1111
& bit-wise and operation with r7

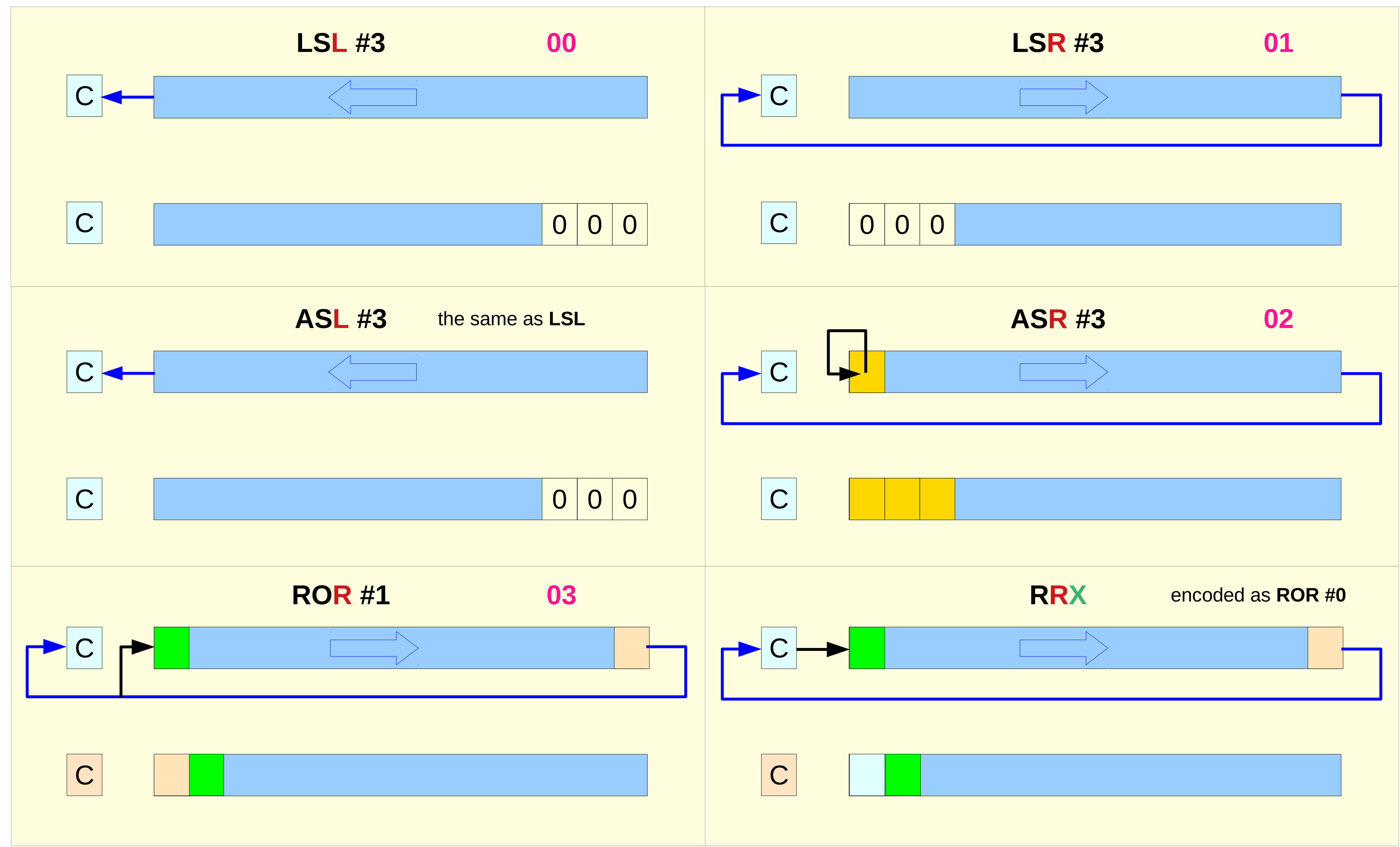
Most valid immediate value
Immediate = (0 → 255) x 2^{2n},  n : [0, 12]

# Shifted Register Operands

ADD r3, r2, r1 , LSL #3 ; r3 := r2 + 8*r1

| | | |
|---|---|---|
| LSL | Logical Shift Left | |
| LSR | Logical Shift Right) | |
| ASL | Arithmetic Shift Left) | |
| ASR | Arithmetic Shift Right) | Sign |
| ROR | Rotate Right by 0 to 32 | C flag |
| ROX | Rotate Right Extended by 1 | C flag |

# Shift Examples



# CPSR – ALU flags

**N** Negative flag – the last ALU operation yields a negative result

**Z** Zero flag – the last ALU operation yields a zero result

**C** Carry flag – the last ALU operation yields a carry out

**V** Overflow flag – the last ALU operation yields a overflowed result

**Conditional Branches**

**Conditional Execution**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **N** | **Z** | **C** | **V** | | | | | | | | | | | | | | | | | | | | | **I** | **F** | **T** | **mode** | | | | |

**Current Program Status Register (CPSR)**

# Comparison Operations

```
CMP    r1, r2        ;    set cc on r1 – r2
CMN    r1, r2        ;    set cc on r1 + r2
TST    r1, r2        ;    set cc on r1 and r2
TEQ    r1, r2        ;    set cc on r1 xor r2
```

The comparison instructions only set cc

# Setting the Condition Codes

All other data processing instructions
must make explicit request

S: Set condition codes

```
ADDS    r2, r2, r0          ; 32-bit carry out → C flag
ADC     r3, r3, r1          ; and added into high word
                            ; Add with Carry
```

# Multiplies

```
MUL     r4, r3, r2              ; r4  := (r3 x r2)[31:0]
                                ; only the least significant 32-bit
```

restriction :
no immediate operand is allowed
the result register must differ from the first source register

If the S bit set  (MULS)
    the V flag is preserved and
    the C flag is rendered meaningless

# Conditional Branches

```
        MOV     r0, #0
LOOP    ...
        ADD     r0, r0, #1
        CMP     r0, #10      ; set cc on r0 – #10
        BNE     LOOP         ; branch if not equal to zero (Z=0)
```

# Conditional Execution

```
        CMP     r0, #5      ; set cc on r0 – #5
        BEQ     Bypass      ; branch if Z=1
        ADD     r1, r1, r0
        SUB     r1, r1, r2
Bypass  …

        CMP     r0, #5      ; set cc on r0 – #5
        ADDNE   r1, r1, r0  ; add operation if Z=0
        SUBNE   r1, r1, r2  ; add operation if Z=0

                            ; no change in Z flag after CMP
```

# Branch and link instructions

| | | | | |
|---|---|---|---|---|
| | BL | SUBR | ; save the old pc | R14 ← PC<br>implicitly saved |
| | … | | | |
| SUBR | … | | | |
| | … | | | |
| | MOV | pc, r14 | ; restore the old pc | PC ← R14<br>must save explicitly |

; Link Register **R14**

# Nested subroutine calls

```
        BL      SUB1              ; R14 ← PC1
        …
SUB1    STMFD   r13!, {r0-r4,r14} ; save r14 (the first PC saved)
        BL      SUB2              ; save the second PC  R14 ← PC2
SUB2    ...
```

# STM : Store Multiple Data

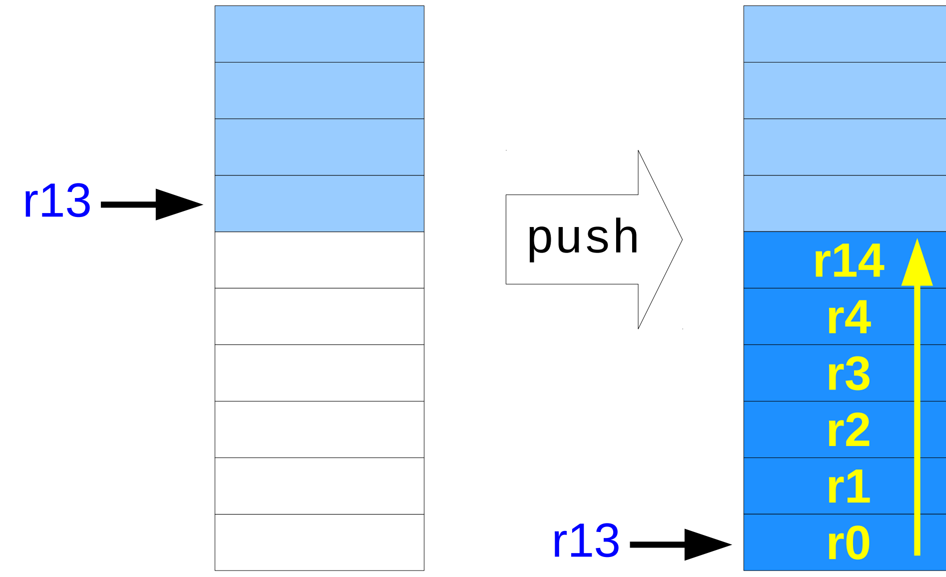
**STMFD r13!, {r0-r4,r14}**

- **store** multiple data
- **push** operation
  on a Full Down stack

- stack top is full (filled)
- stack is growing
  downward

- stack pointer
- top of a stack
- ! : updated after
  push operation

- multiple data to be
  pushed on the stack



# LDM : Store Multiple Data

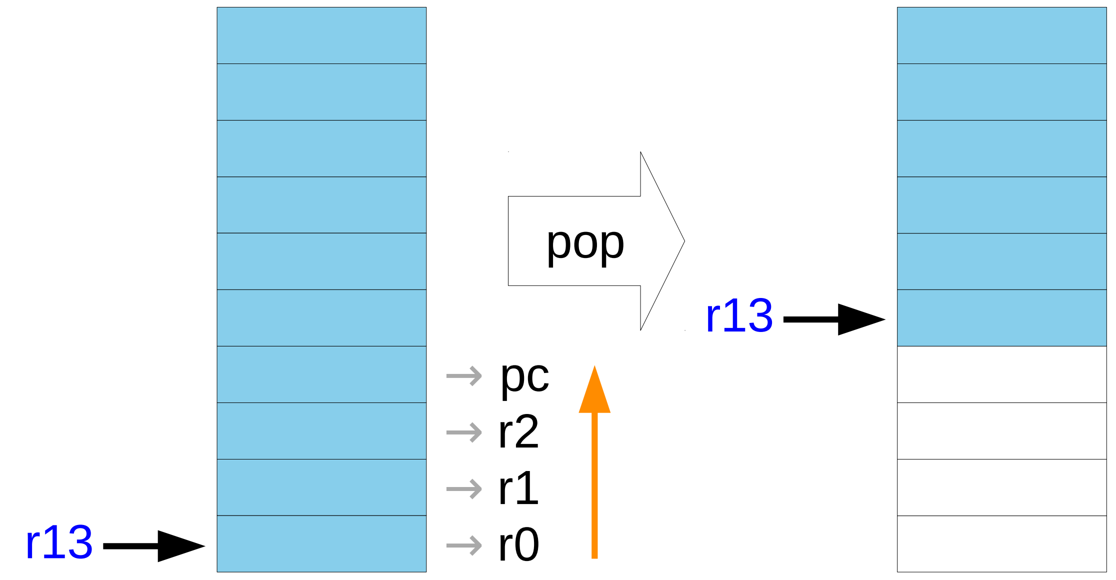
**LDMFD** **r13!,** **{r0-r2,pc}**

- **load** multiple data
- **pop** operation
  on a Full Down stack

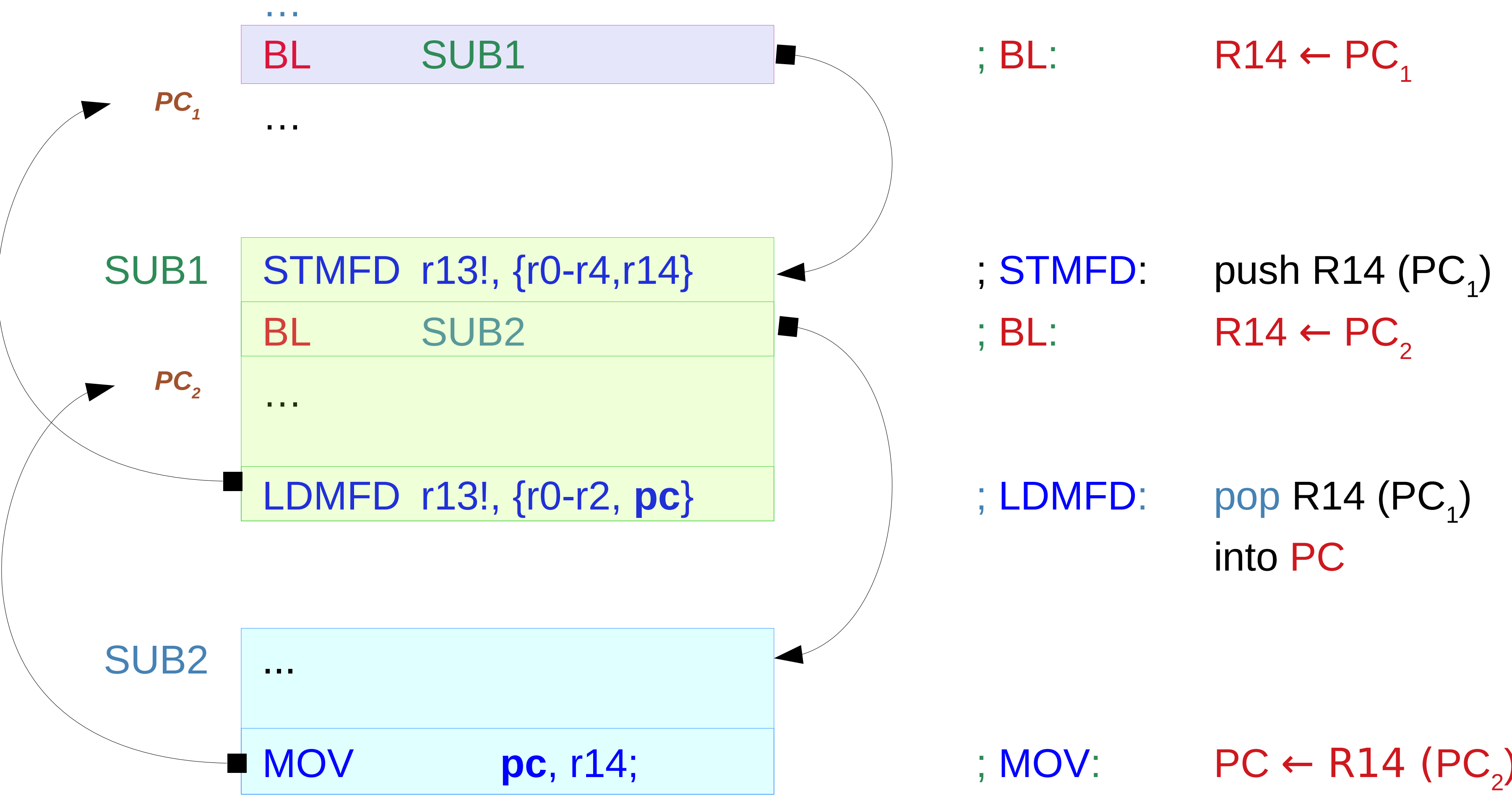
- stack top is full (filled)
- stack is shrinking
  upward

- stack pointer
- top of a stack
- ! : updated after
  push operation

- multiple data to be
  pushed on the stack



# Subroutine return instructions



# Supervisor calls

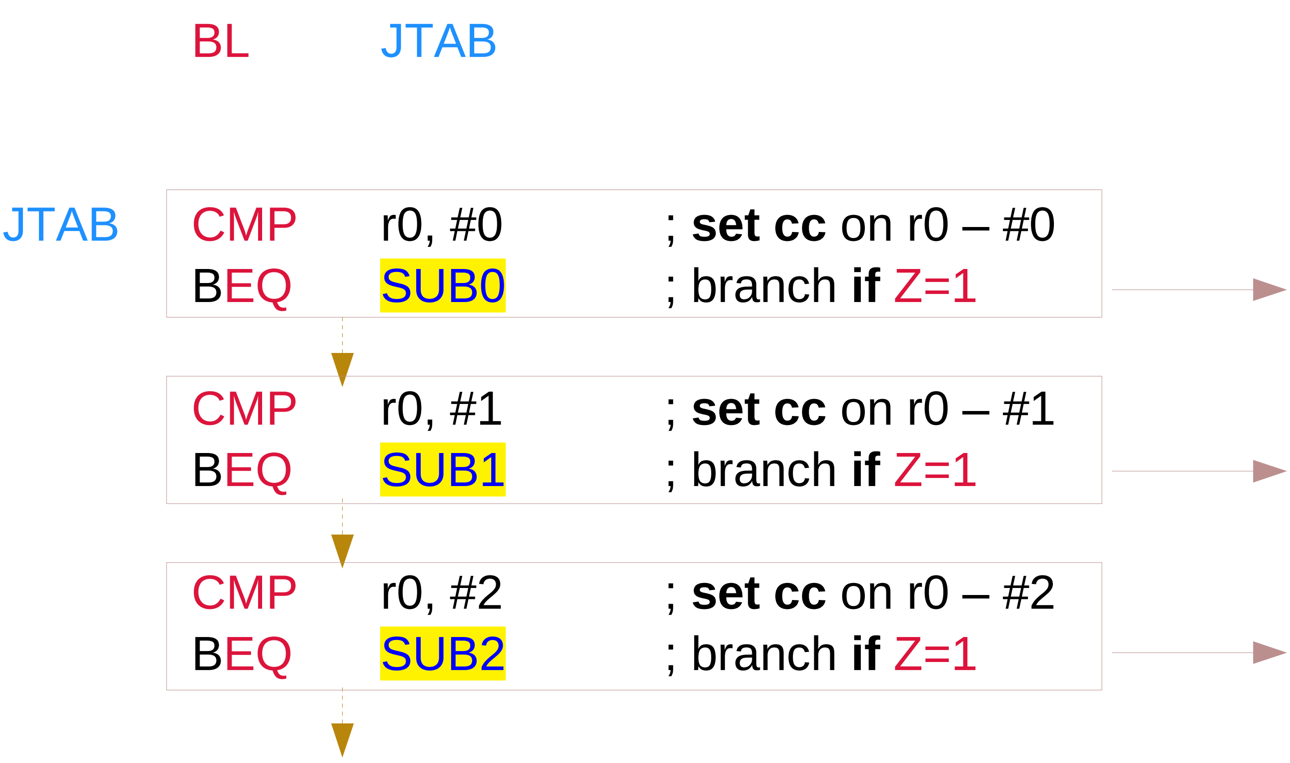
```
SWI     SWI_WriteC
SWI     SWI_Exit
```

SWI_WriteC: IO subroutine ()

SWI_Exit: returns control from a user program to the monitor program

to call privileged routine to access the system software
use **SWI** (SoftWare Interrupt – Supervisor Call)

# Jump tables – method 1

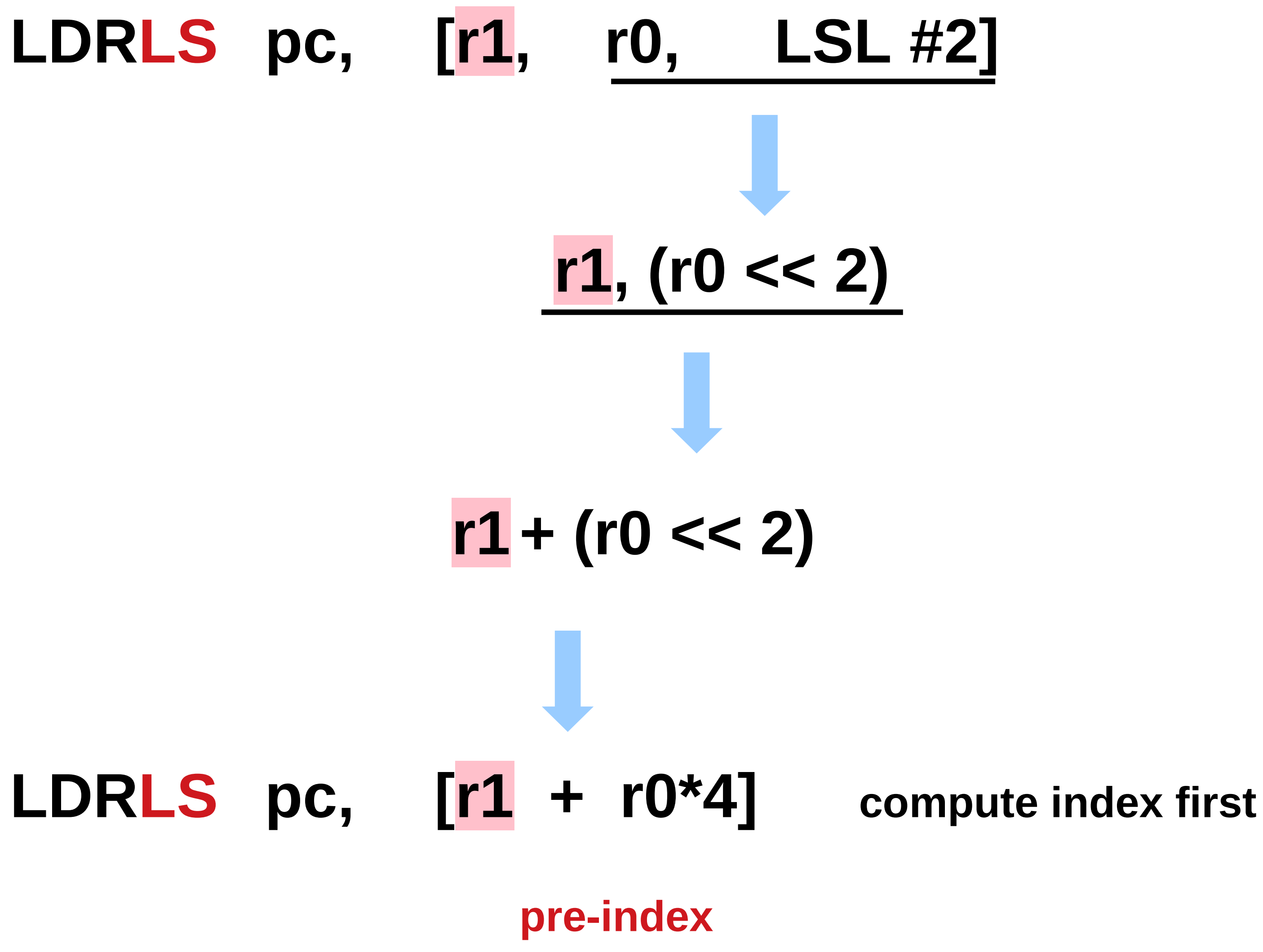


# Jump tables – method 2

```
        BL    JTAB
        …
JTAB    ADR     r1, STAB                ; r1 ← STAB address
        CMP     r0, #SMAX
        LDRLS   pc, [r1, r0, LSL #2]    ; LDR pc, SUB0 or SUB1 or …
        B       ERROR

STAB    DCD     SUB0                    ; STAB+0 byte address     0*4
        DCD     SUB1                    ; STAB+4 byte address     1*4
        DCD     SUB2                    ; STAB+8 byte address     2*4
                                        ; r1 + (r0 << 2)          r0*4
                                        ; r1 = STAB
                                        ; r0 = 0, 1, 2,  …   < #SMAX
        …
```

# Pre-index Addressing



# Cost of branch operation

```
CMP      r0, #SMAX
LDRLS    pc, [r1, r0, LSL #2]     ; load when normal range (frequent)
B        ERROR
; too many conditional executions          (cost high)
```

```
CMP      r0, #SMAX
BNI      ERROR                    ; branch when out of range (infrequent)
LDR      pc, [r1,r0,LSL#2]
; rare conditional branch operations       (cost effective)
```

### References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf