

# ARM Processor

ARM = Advanced RISC Machines, Ltd.

ARM licenses IP to other companies

(ARM does not fabricate chips)

2005: ARM had 75% of embedded RISC market, with 2.5 billion processors

ARM available as microcontrollers, IP cores, etc.

[www.arm.com](http://www.arm.com)

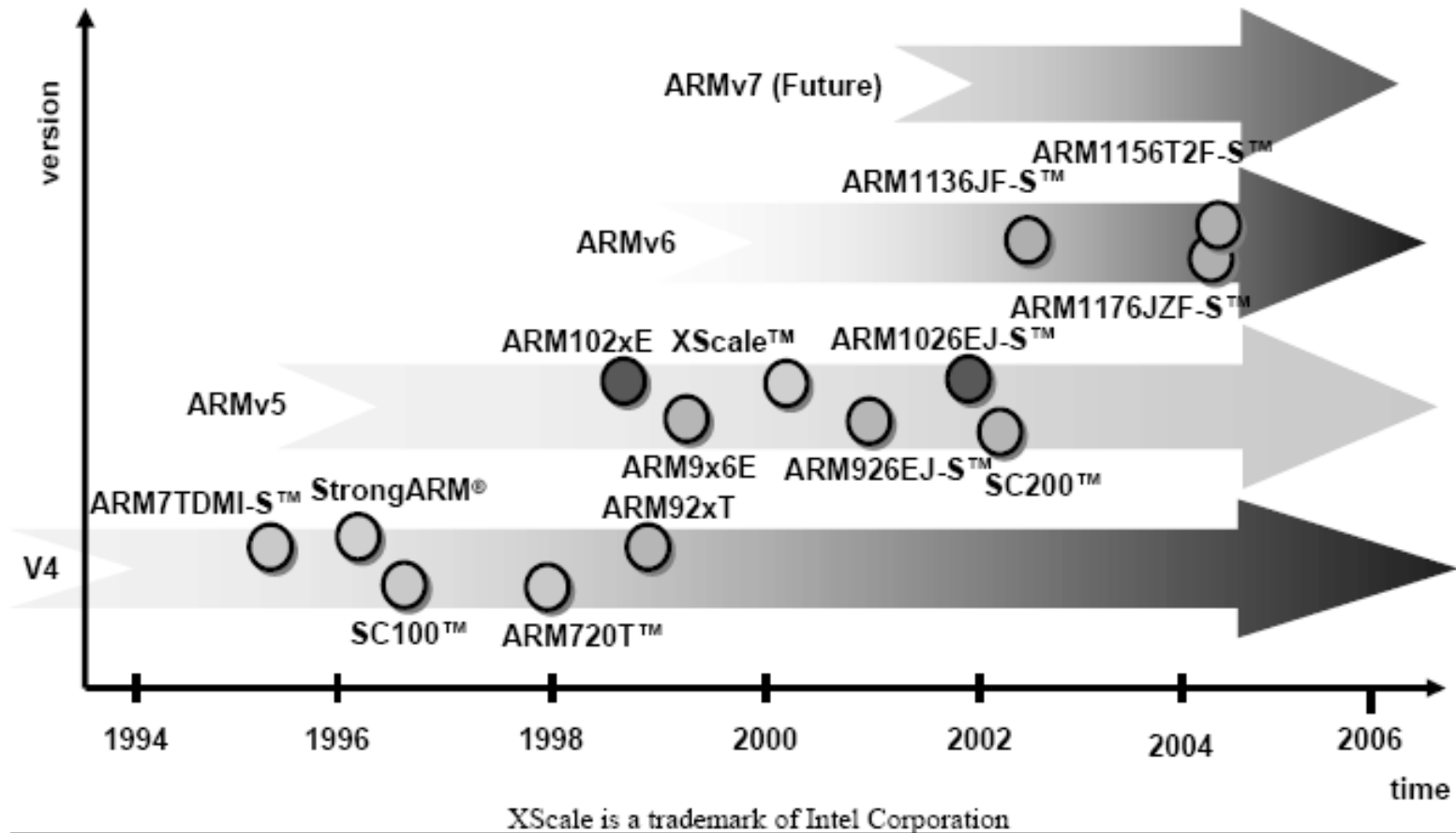
# ARM instruction set - outline

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control.

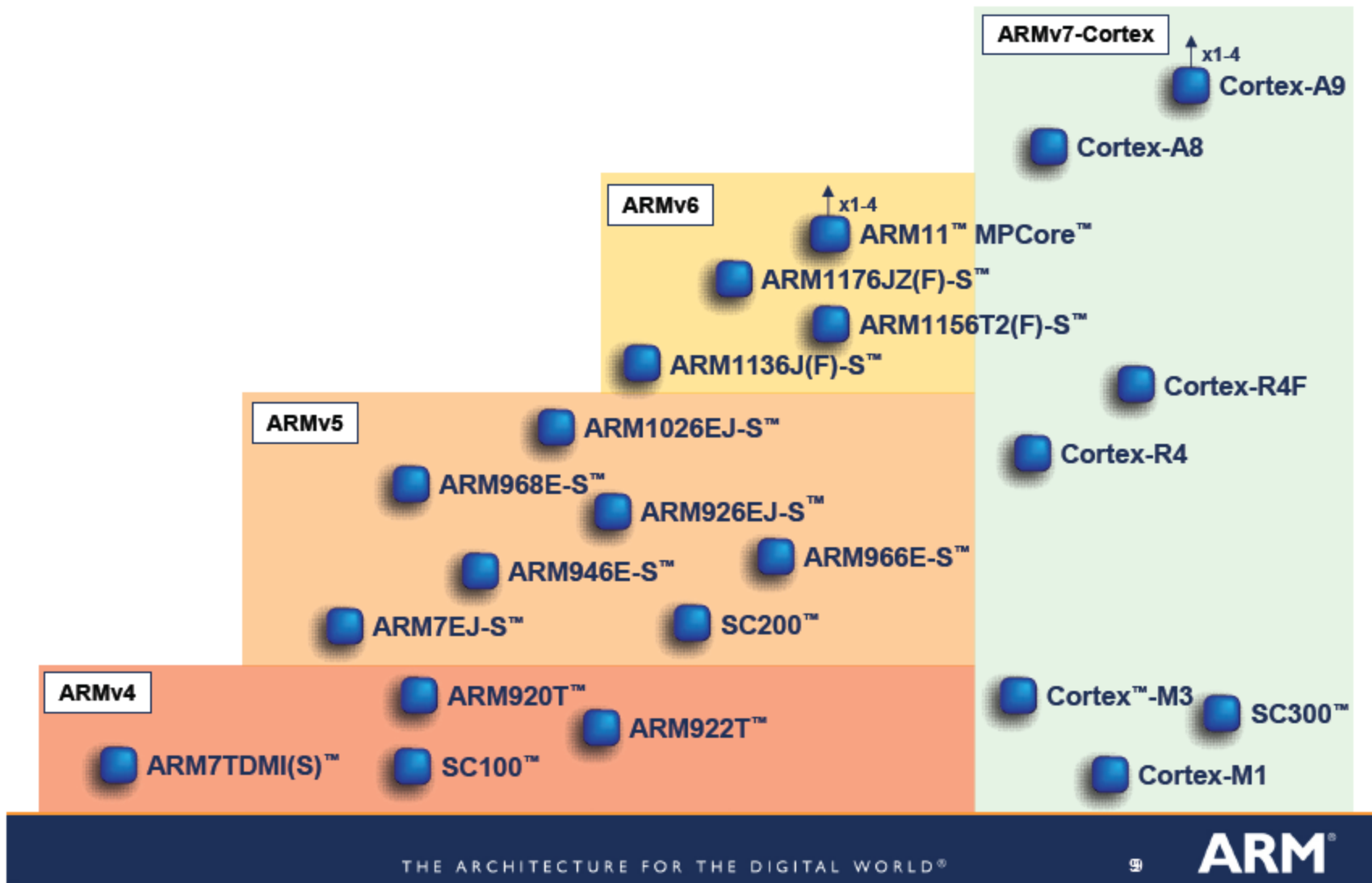
# ARM versions

- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.
  - ARM9 – includes “Thumb” instruction set
  - ARM10 – for multimedia (graphics, video, etc.)
  - ARM11 – high performance + Jazelle (Java)
  - SecurCore – for security app’s (smart cards)
  - Cortex – Optimized for embedded app’s
  - StrongARM – portable communication devices

# Architecture Revisions

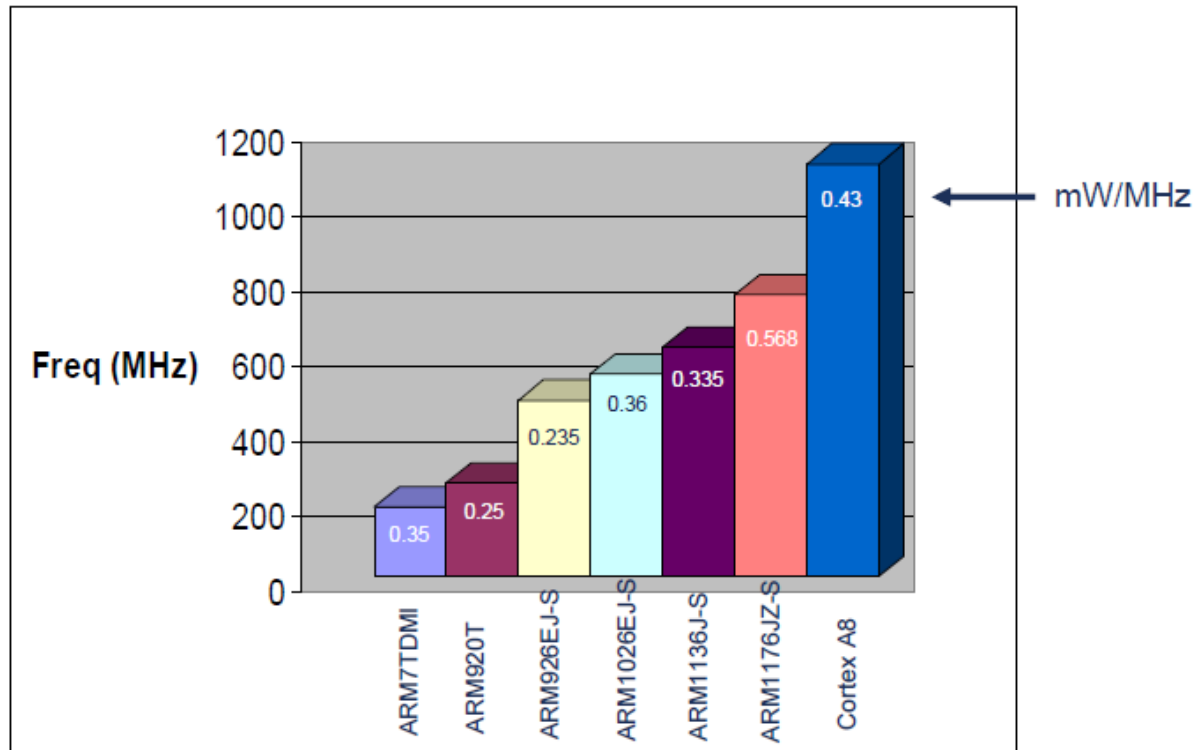


# ARM Architecture versions



Based on Lecture Notes by Marilyn Wolf

# Relative Performance\*



\*Represents attainable speeds in 130, 90 or 65nm processes

# RISC CPU Characteristics

- 32-bit load/store architecture
- Fixed instruction length
- Fewer/simpler instructions
- Limited addressing modes, operand types
- Simpler design easier to speed up, pipeline & scale

# ARM assembly language

- Fairly standard assembly language:

```
                LDR r0,[r8]    ; a comment
label          ADD r4,r0,r1    ; r4=r0+r1
```

The diagram illustrates the components of the `ADD r4,r0,r1` instruction. Three red labels at the bottom are connected by arrows to the instruction fields: `destination` points to `r4`, `source/left` points to `r0`, and `source/right` points to `r1`.

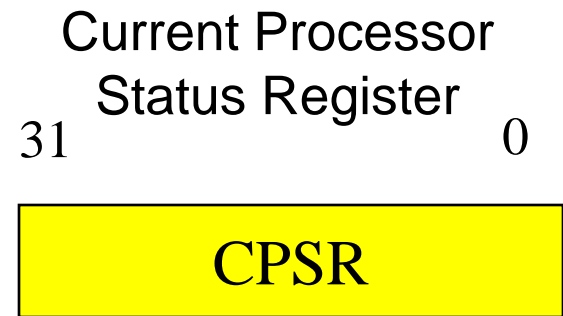


# ARM programming model

16 32-bit general-purpose registers

r0
r1
r2
r3
r4
r5
r6
r7

r8
r9
r10
r11
r12
r13
r14
r15 (PC)



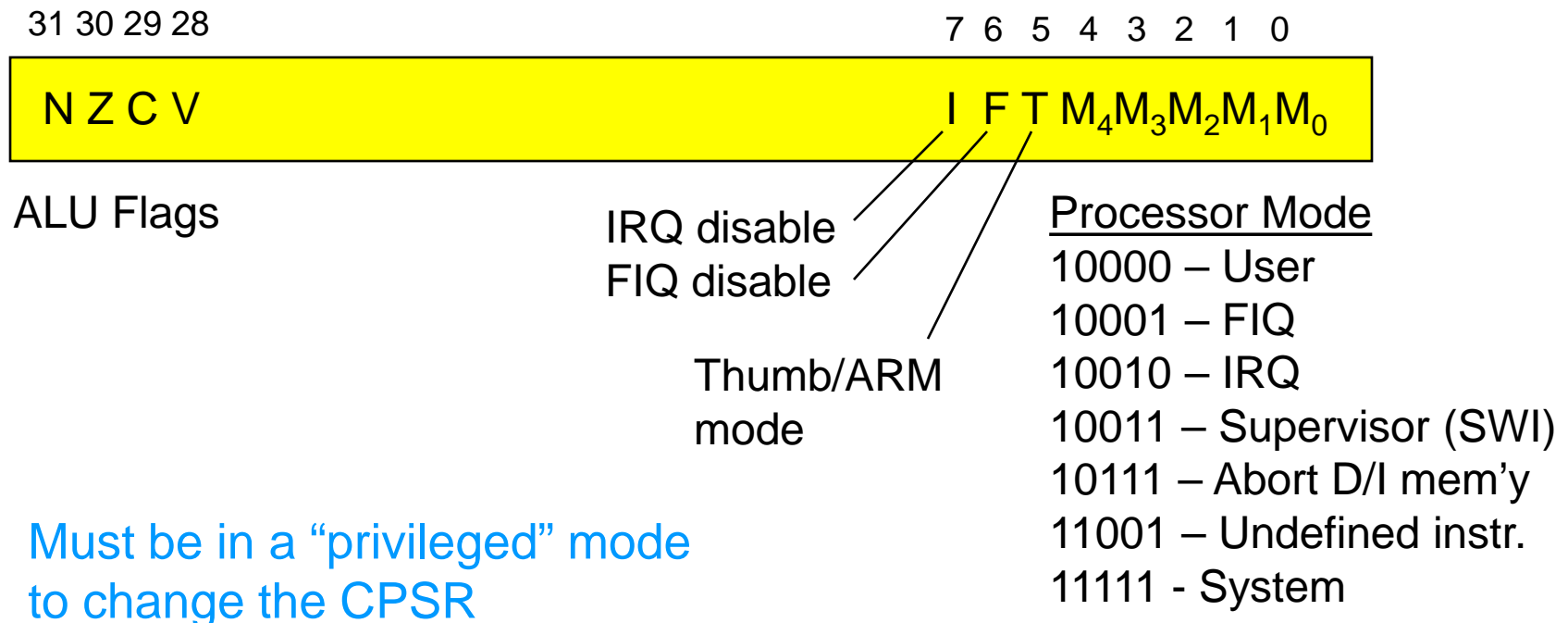
N Z C V

Program Counter

Some registers (r13,r14) change during exceptions

# CPSR

## Current Processor Status Register



**MRS rn,CPSR**

**MSR CPSR,rn**

# Register Set

## Current Visible Registers

Abort Mode

<b>r0</b>
<b>r1</b>
<b>r2</b>
<b>r3</b>
<b>r4</b>
<b>r5</b>
<b>r6</b>
<b>r7</b>
<b>r8</b>
<b>r9</b>
<b>r10</b>
<b>r11</b>
<b>r12</b>
r13 (sp)
r14 (lr)
<b>r15 (pc)</b>
<b>cpsr</b>
spsr

## Banked out Registers

User

FIQ

IRQ

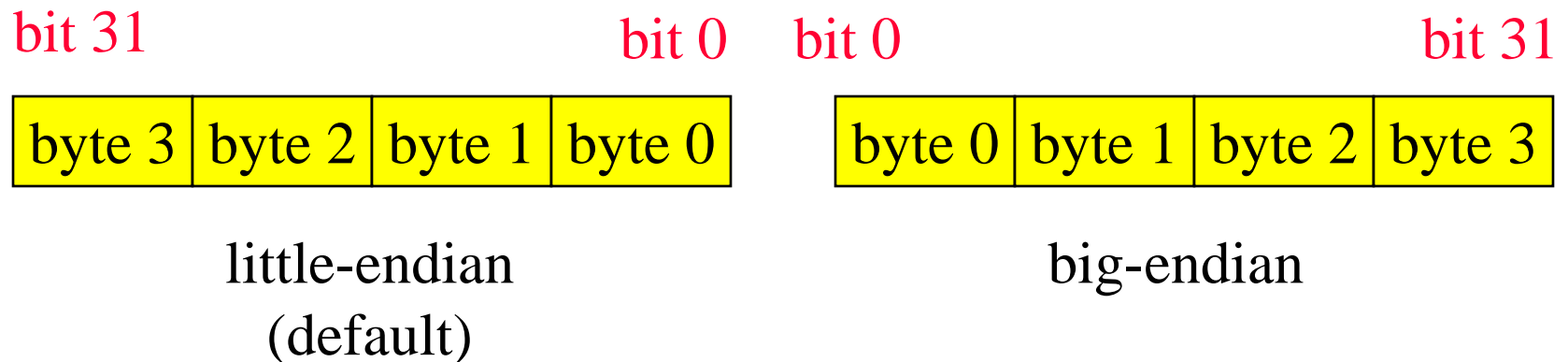
SVC

Undef

	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

# Endianness

- Relationship between bit and byte/word ordering defines “endianness”:



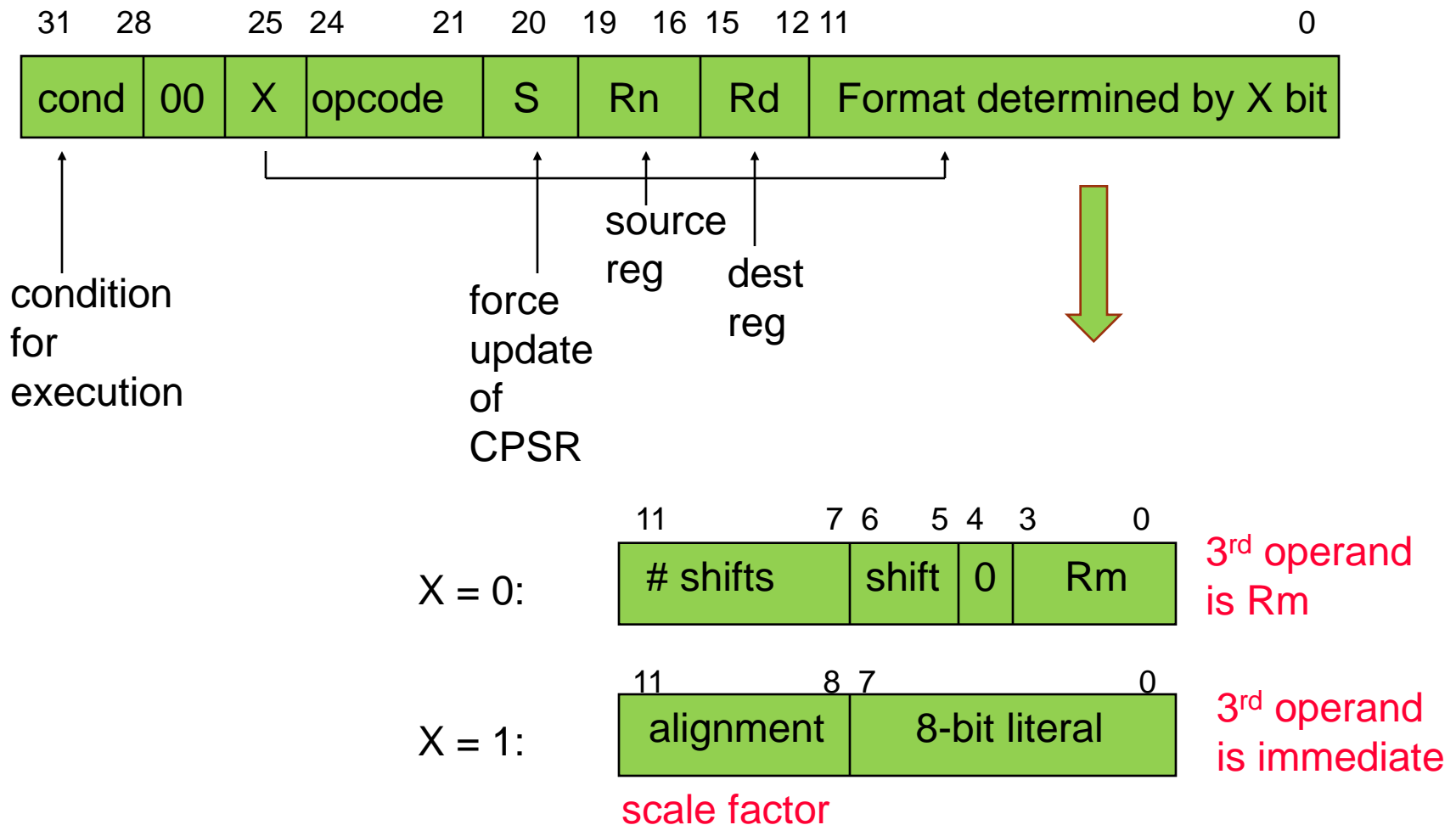
# ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
  - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

# ARM status bits

- Every arithmetic, logical, or shifting operation can set CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow)
- Examples:
  - $-1 + 1 = 0$ :      NZCV = 0110.
  - $2^{31}-1+1 = -2^{31}$ :      NZCV = 1001.
- Setting status bits must be explicitly indicated on each instruction
  - ex. “adds” sets status bits, whereas “add” does not

# ARM Instruction Code Format



# ARM data instructions

- Basic format:

ADD r0 , r1 , r2

- Computes  $r1 + r2$ , stores in r0.

- Immediate operand: (8-bit constant – can be scaled by  $2^k$ )

ADD r0 , r1 , #2

- Computes  $r1 + 2$ , stores in r0.

- Set condition flags based on operation:

ADDS r0 , r1 , r2

↑  
set status flags

- Recently-added assembler translation:

ADD r1,r2 = ADD r1,r1,r2      (but not MUL)



# ARM arithmetic instructions

- ADD, ADC : add (w. carry)  
 $[Rd] \leq Op1 + Op2 + C$
- SUB, SBC : subtract (w. carry)  
 $[Rd] \leq Op1 - Op2 + (C - 1)$
- RSB, RSC : reverse subtract (w. carry)  
 $[Rd] \leq Op2 - Op1 + (C - 1)$
- MUL: multiply (32-bit product – no immediate for Op2)  
 $[Rd] \leq Op1 \times Op2$
- MLA : multiply and accumulate (32-bit result)  
 $MLA\ Rd, Rm, Rs, Rn : [Rd] \leq (Rm \times Rs) + Rn$

# ARM logical instructions

- AND, ORR, EOR: bit-wise logical op's
- BIC : bit clear  $[Rd] \leftarrow Op1 \wedge \overline{Op2}$
- LSL, LSR : logical shift left/right (combine with data op's)

ADD r1,r2,r3, LSL #4 :  $[r1] \leftarrow r2 + (r3 \times 16)$

Vacated bits filled with 0's

- ASL, ASR : arithmetic shift left/right (maintain sign)
- ROR : rotate right
- RRX : rotate right extended with C from CPSR

33-bit shift:



# ARM comparison instructions

- CMP : compare :  $Op1 - Op2$
- CMN : negated compare :  $Op1 + Op2$
- TST : bit-wise AND :  $Op1 \wedge Op2$
- TEQ : bit-wise XOR :  $Op1 \text{ xor } Op2$
- These instructions only set the NZCV bits of CPSR – no other result is saved. (“Set Status” is implied)

# ARM move instructions

- MOV, MVN : move (negated)

MOV r0, r1 ; sets r0 to r1

MOVN r0, r1 ; sets r0 to  $\overline{r1}$

MOV r0, #55 ; sets r0 to 55

- Can use shift modifier to scale a value:

MOV r0, r1, LSL #6 ; [r0]  $\leq$  r1 x 64

- Recently-added pseudo-Op:

LSL rd, rn, shift = MOV rd, rn, LSL shift

# ARM load/store instructions

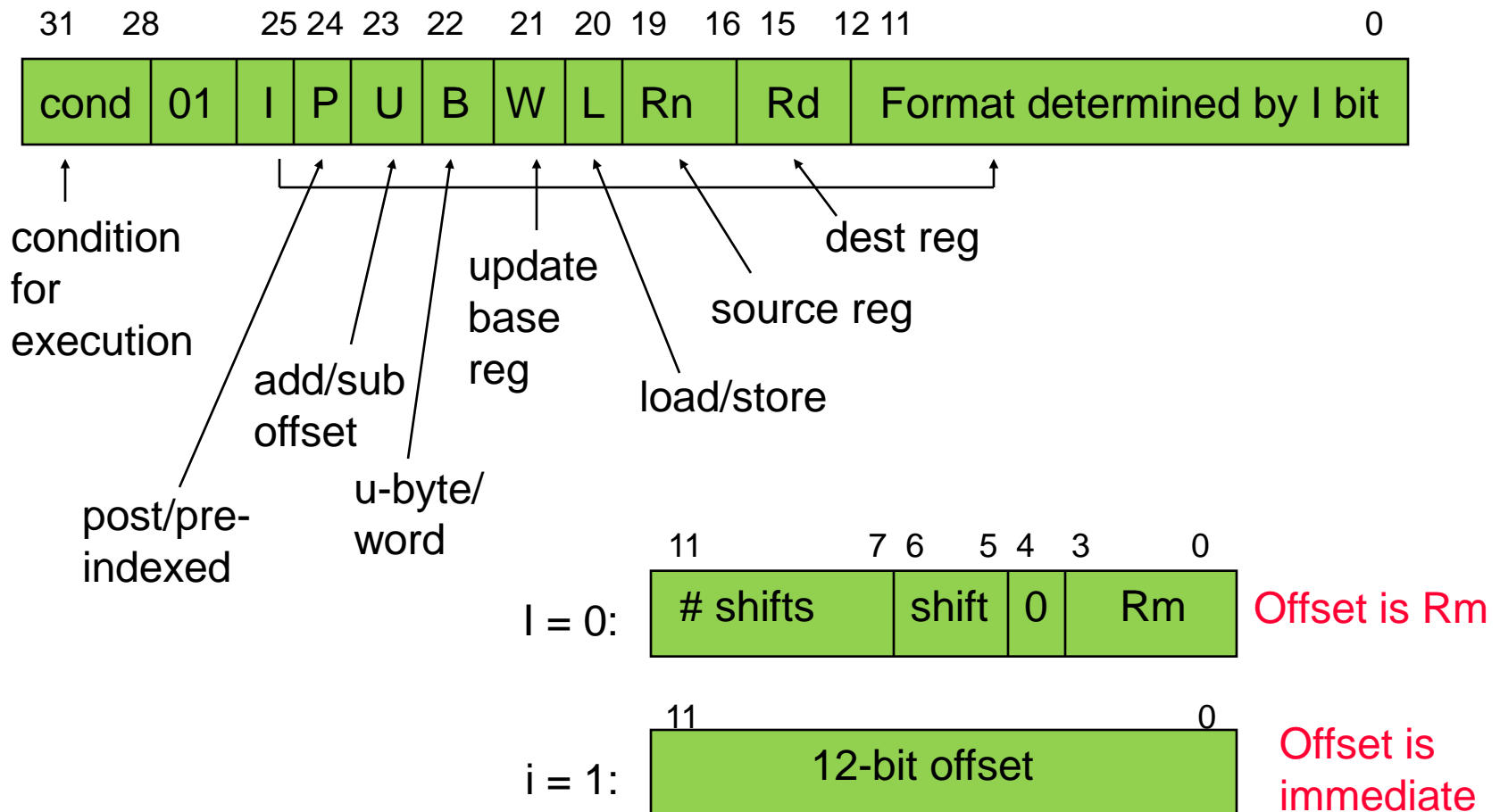
- Load operand from memory into target register
  - LDR – load 32 bits
  - LDRH – load halfword (16 bit unsigned #) & zero-extend to 32 bits
  - LDRSH – load signed halfword & sign-extend to 32 bits
  - LDRB – load byte (8 bit unsigned #) & zero-extend to 32 bits
  - LDRSB – load signed byte & sign-extend to 32 bits
- Store operand from register to memory
  - STR – store 32-bit word
  - STRH – store 16-bit halfword (right-most 16 bits of register)
  - STRB : store 8-bit byte (right-most 8 bits of register)

# ARM load/store addressing


- Addressing modes: **base address + offset**
  - register indirect : `LDR r0, [r1]`
  - with second register : `LDR r0, [r1, -r2]`
  - with constant : `LDR r0, [r1, #4]`
  - pre-indexed: `LDR r0, [r1, #4]!`
  - post-indexed: `LDR r0, [r1], #8`

**Immediate #operand = 12 bits (2's complement)**

# ARM Load/Store Code Format



# ARM load/store examples

- `ldr r1,[r2]` ; address = (r2)
- `ldr r1,[r2,#5]` ; address = (r2)+5
- `ldr r1,[r2,#-5]` ; address = (r2)-5
- `ldr r1,[r2,r3]` ; address = (r2)+(r3)
- `ldr r1,[r2,-r3]` ; address = (r2)-(r3)
- `ldr r1,[r2,r3,SHL #2]` ; address=(r2)+(r3 x 4)  


Scaled index

Base register r2 is not altered in these instructions



# ARM load/store examples

(base register updated by auto-indexing)

- `ldr r1,[r2,#4]!`      ; use address =  $(r2)+4$   
                          ;  $r2 \leq (r2)+4$       (pre-index)
- `ldr r1,[r2,r3]!`      ; use address =  $(r2)+(r3)$   
                          ;  $r2 \leq (r2)+(r3)$       (pre-index)
- `ldr r1,[r2],#4`      ; use address =  $(r2)$   
                          ;  $r2 \leq (r2)+4$       (post-index)
- `ldr r1,[r2],[r3]`      ; use address =  $(r2)$   
                          ;  $r2 \leq (r2)+(r3)$       (post-index)

# Additional addressing modes

- Base-plus-offset addressing:

LDR r0, [r1, #16]

- Loads from location  $[r1+16]$

- Auto-indexing increments base register:

LDR r0, [r1, #16]!

- Loads from location  $[r1+16]$ , then sets  $r1 = r1 + 16$

- Post-indexing fetches, then does offset:

LDR r0, [r1], #16

- Loads r0 from  $[r1]$ , then sets  $r1 = r1 + 16$

- Recent assembler addition:

**SWP**{cond} rd, rm, [rn] : swap mem & reg

$M[rn] \rightarrow rd, rd \rightarrow M[rn]$

# ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction  
(with only 32-bit instruction).
- Assembler will try to translate:  
LDR Rd,label = LDR Rd,[pc,#offset]
- Generate address value by performing arithmetic on PC.  
(if address in code section)
- ADR pseudo-op generates instruction required to calculate address (in code section ONLY)  
ADR r1,LABEL  
(uses MOV,MOVN,ADD,SUB op's)

# ARM 32-bit load pseudo-op

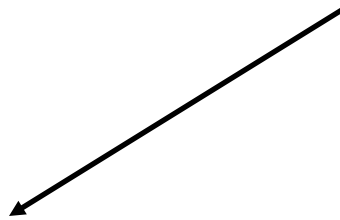
- LDR r3,=0x55555555
  - Produces MOV if immediate constant can be found
  - Otherwise put constant in a “literal pool”

LDR r3,[PC,#immediate-12]

.....

DCD 0x55555555

;in literal pool following code



# Example: C assignments

- C: `x = (a + b) - c;`

- Assembler:

```
ADR r4,a           ; get address for a
LDR r0,[r4]         ; get value of a
ADR r4,b           ; get address for b, reusing r4
LDR r1,[r4]         ; get value of b
ADD r3,r0,r1        ; compute a+b
ADR r4,c           ; get address for c
LDR r2,[r4]         ; get value of c
SUB r3,r3,r2        ; complete computation of x
ADR r4,x           ; get address for x
STR r3,[r4]         ; store value of x
```

# Example: C assignment

- C:  $y = a * (b + c);$

- Assembler:

```
LDR r4,=b      ; get address for b
LDR r0,[r4]     ; get value of b
LDR r4,=c      ; get address for c
LDR r1,[r4]     ; get value of c
ADD r2,r0,r1    ; compute partial result
LDR r4,=a      ; get address for a
LDR r0,[r4]     ; get value of a
MUL r2,r2,r0    ; compute final value for y
LDR r4,=y      ; get address for y
STR r2,[r4]     ; store y
```

# Example: C assignment

- C: `z = (a << 2) | (b & 15);`

- Assembler:

```
LDR r4,=a           ; get address for a
LDR r0,[r4]          ; get value of a
MOV r0,r0,LSL 2      ; perform shift
LDR r4,=b           ; get address for b
LDR r1,[r4]          ; get value of b
AND r1,r1,#15        ; perform AND
ORR r1,r0,r1         ; perform OR
LDR r4,=z           ; get address for z
STR r1,[r4]          ; store value for z
```

# ARM flow control operations

- All operations can be performed conditionally, testing CPSR:
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:  
B #100
- Can be performed conditionally:  
BNE #100



# Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
```

```
LDR r4,=a           ; get address for a
```

```
LDR r0,[r4]         ; get value of a
```

```
LDR r4,=b           ; get address for b
```

```
LDR r1,[r4]         ; get value for b
```

```
CMP r0,r1           ; compare a < b
```

```
BLE fblock          ; if a >= b, branch to false block
```

# If statement, cont'd.

```
; true block
MOV r0,#5           ; generate value for x
LDR r4,=x           ; get address for x
STR r0,[r4]         ; store x
LDR r4,=c           ; get address for c
LDR r0,[r4]         ; get value of c
LDR r4,=d           ; get address for d
LDR r1,[r4]         ; get value of d
ADD r0,r0,r1        ; compute y
LDR r4,=y           ; get address for y
STR r0,[r4]         ; store y
B after            ; branch around false block
```

# If statement, cont'd.

`; false block`

`fblock LDR r4,=c ; get address for c`

`LDR r0,[r4] ; get value of c`

`lDR r4,=d ; get address for d`

`LDR r1,[r4] ; get value for d`

`SUB r0,r0,r1 ; compute a-b`

`LDR r4,=x ; get address for x`

`STR r0,[r4] ; store value of x`

`after ...`

# Example: Conditional instruction implementation

```
; true block
MOVLt r0,#5      ; generate value for x
ADRLT r4,x       ; get address for x
STRLT r0,[r4]    ; store x
ADRLT r4,c       ; get address for c
LDRLT r0,[r4]    ; get value of c
ADRLT r4,d       ; get address for d
LDRLT r1,[r4]    ; get value of d
ADDLT r0,r0,r1   ; compute y
ADRLT r4,y       ; get address for y
STRLT r0,[r4]    ; store y
```

# Conditional instruction implementation, cont'd.

`; false block`

`ADRGE r4,c ; get address for c`

`LDRGE r0,[r4] ; get value of c`

`ADRGE r4,d ; get address for d`

`LDRGE r1,[r4] ; get value for d`

`SUBGE r0,r0,r1 ; compute a-b`

`ADRGE r4,x ; get address for x`

`STRGE r0,[r4] ; store value of x`

# Example: switch statement

- C:

```
switch (test) { case 0: ... break; case 1: ... }
```

- Assembler:

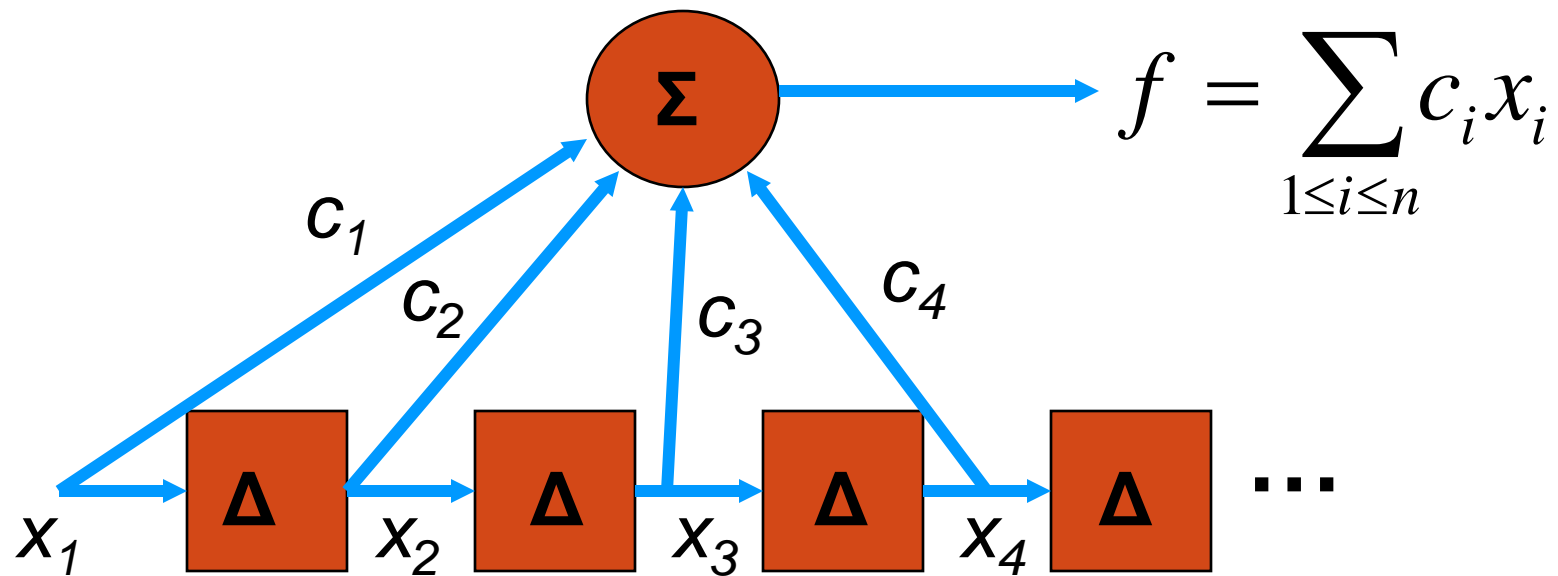
```
LDR r2,=test           ; get address for test
LDR r0,[r2]             ; load value for test
ADR r1,switchtab        ; load switch table address
LDR r15,[r1,r0,LSL #2] ; index switch table
```

```
switchtab DCD case0
```

```
          DCD case1
```

```
          ...
```

# Finite impulse response (FIR) filter



$x_i$ 's are data samples  
 $c_i$ 's are constants

# Example: FIR filter

- C:

```
for (i=0, f=0; i<N; i++)  
    f = f + c[i]*x[i];
```

- Assembler

*; loop initiation code*

```
MOV r0,#0           ; use r0 for I  
MOV r8,#0           ; use separate index for arrays  
LDR r2,=N           ; get address for N  
LDR r1,[r2]         ; get value of N  
MOV r2,#0           ; use r2 for f  
LDR r3,=c           ; load r3 with base of c  
LDR r5,=x           ; load r5 with base of x
```



# FIR filter, cont'.d

; loop body

loop

```
LDR r4,[r3,r8]    ; get c[i]
LDR r6,[r5,r8]    ; get x[i]
MUL r4,r4,r6      ; compute c[i]*x[i]
ADD r2,r2,r4      ; add into running sum f
ADD r8,r8,#4      ; add word offset to array index
ADD r0,r0,#1      ; add 1 to i
CMP r0,r1         ; exit?
BLT loop          ; if i < N, continue
```

# FIR filter with MLA & auto-index

```
AREA TestProg, CODE, READONLY
```

```
ENTRY
```

```
        mov     r0,#0           ;accumulator
        mov     r1,#3           ;number of iterations
        ldr     r2,=carray      ;pointer to constants
        ldr     r3,=xarray      ;pointer to variables
loop     ldr     r4,[r2],#4      ;get c[i] and move pointer
        ldr     r5,[r3],#4      ;get x[i] and move pointer
        mla     r0,r4,r5,r0     ;sum = sum + c[i]*x[i]
        subs    r1,r1,#1        ;decrement iteration count
        bne     loop           ;repeat until count=0
here     b       here
carray dcd     1,2,3
xarray dcd     10,20,30
```

```
END
```

Also, need “time delay” to prepare x array for next sample

# ARM subroutine linkage

- Branch and link instruction:

**BL** `foo` ; Copies current PC to r14.

- To return from subroutine:

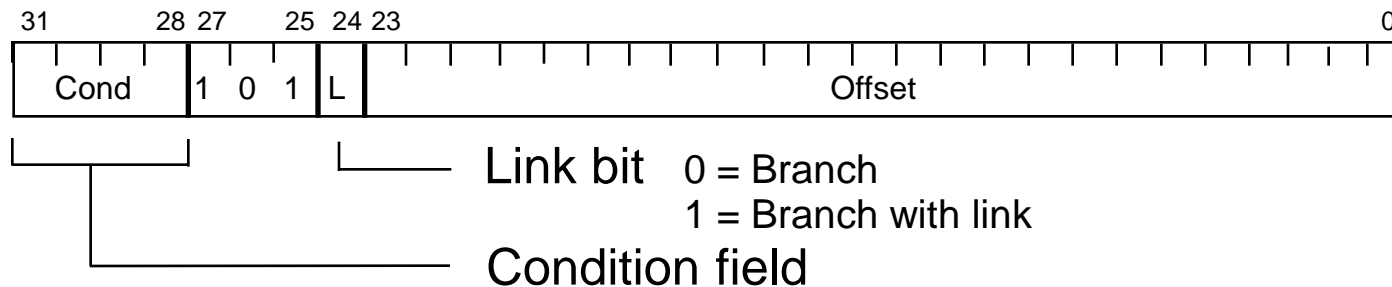
**BX** `r14` ; branch to address in r14

or:

`MOV r15, r14`

- May need subroutine to be “reentrant”
  - interrupt it, with interrupting routine calling the subroutine (2 instances of the subroutine)
  - support by creating a “stack” (not supported directly)

# Branch instructions (B, BL)



- The processor core shifts the offset field left by 2 positions, **sign-extends** it and adds it to the PC
  - $\pm 32$  Mbyte range
  - **How to perform longer branches?**

# Nested subroutine calls

- Nested function calls in C:

```
void f1(int a){
    f2(a);}
void f2 (int r){
    int g;
    g = r+5; }
main () {
    f1(xyz);
}
```

# Nested subroutine calls (1)

- Nesting/recursion requires a “coding convention” to save/pass parameters:

```
        AREA Code1, CODE
Main    LDR r13, =StackEnd    ;r13 points to last element on stack
        MOV    r1, #5        ;pass value 5 to func1
        STR    r1, [r13, #-4]! ; push argument onto stack
        BL     func1         ; call func1()
here    B       here
```

# Nested subroutine calls (2)

; Function func1()

Func1    LDR r0,[r13]                    ; load arg into r0 from stack

          ; call func2()

          STR r14,[r13,#-4]!            ; store func1's return adrs

          STR r0,[r13,#-4]!            ; store arg to f2 on stack

          BL func2                      ; branch and link to f2

          ; return from func1()

          ADD r13,#4                    ; "pop" func2's arg off stack

          LDR r15, [r13],#4            ; restore register and return

# Nested subroutine calls (3)

; Function func2()

Func2    BX        r14        ;preferred return instruction

; Stack area

          AREA    Data1,DATA

Stack     SPACE 20        ;allocate stack space

StackEnd

          END



# Register usage conventions

Reg	Usage*	Reg	Usage*
r0	a1	r8	v5
r1	a2	r9	v6
r2	a3	r10	v7
r3	a4	r11	v8
r4	v1	r12	lp (intra-procedure scratch reg.)
r5	v2	r13	sp (stack pointer)
r6	v3	r14	lr (link register)
r7	v4	r15	pc (program counter)

\* Alternate register designation

a1-a4 : argument/result/scratch

v1-v8: variables

Based on Lecture Notes by Marilyn Wolf

# Saving/restoring multiple registers

- LDM/STM – load/store multiple registers
  - LDMIA – increment address after xfer
  - LDMIB – increment address before xfer
  - LDMDA – decrement address after xfer
  - LDMDB – decrement address before xfer
  - LDM/STM default to LDMIA/STMIA

Examples:

`ldmia r13!, {r8-r12,r14}` ;r13 updated at end

`stmdb r13, {r8-r12,r14}` ;r13 not updated at end

# ARM assembler new additions

- PUSH {reglist} = STMDB sp!, {reglist}
- POP {reglist} = LDMIA sp!, {reglist}

# Summary

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
  - Some multi-register operations take longer.
- All instructions can be executed conditionally.