RISC CPU Characteristics

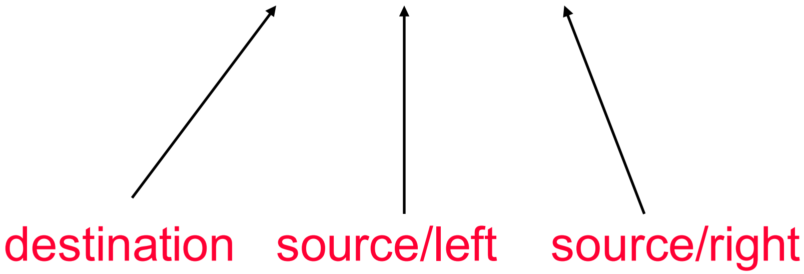
- 32-bit load/store architecture
- Fixed instruction length
- Fewer/simpler instructions than CISC CPU
- Limited addressing modes, operand types
- Simple design easier to speed up, pipeline & scale

Based on Lecture Notes by Marilyn Wolf

ARM assembly language

- Fairly standard RISC assembly language:

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

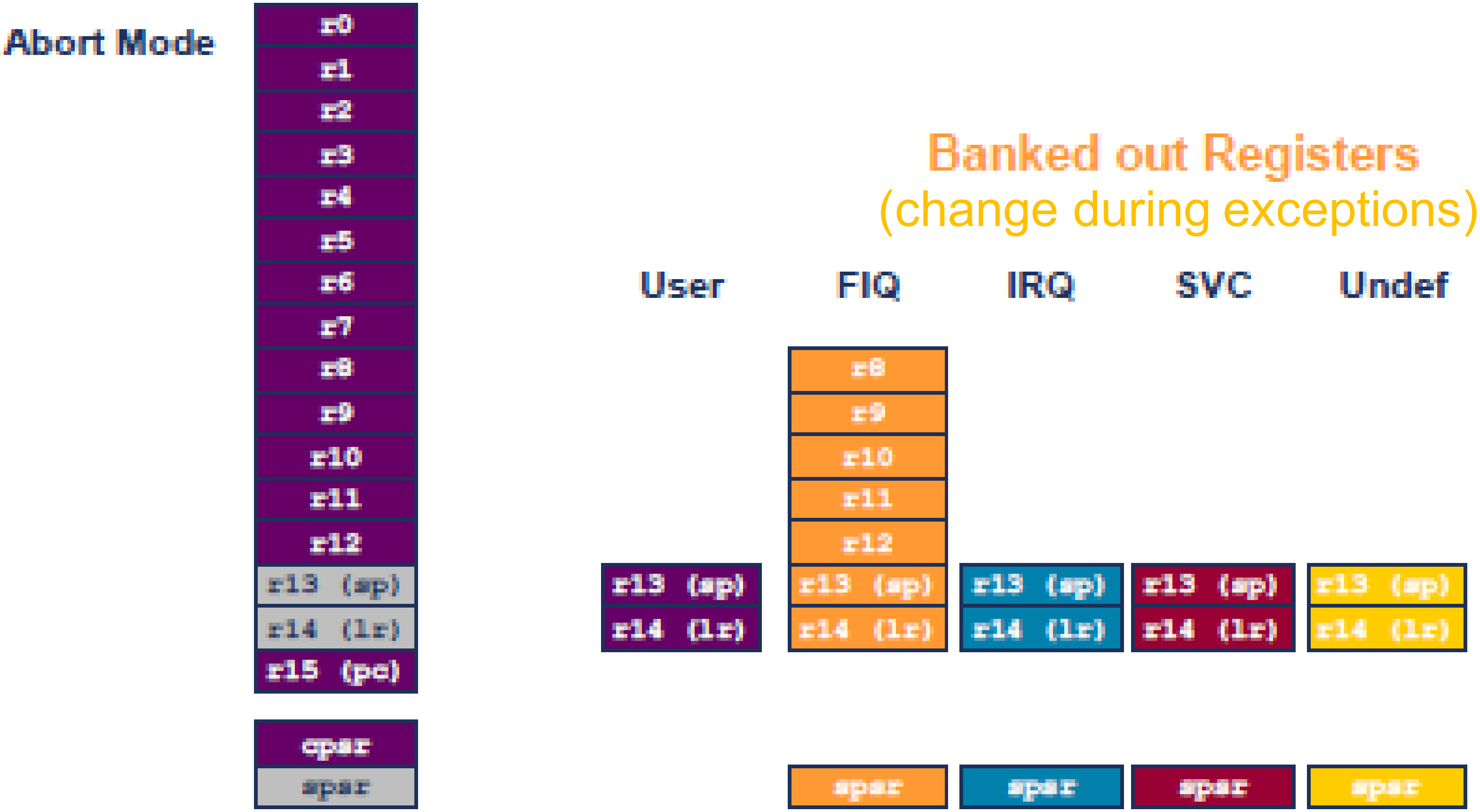


destination source/left source/right

Based on Lecture Notes by Marilyn Wolf

ARM Register Set

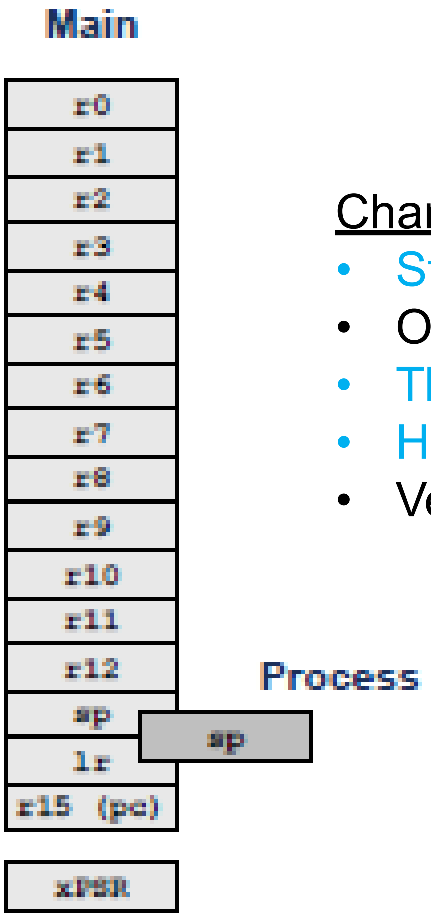
Current Visible Registers (16 32-bit general-purpose registers)



Banked out Registers
(change during exceptions)

Based on Lecture Notes by Marilyn Wolf

ARM Cortex register set



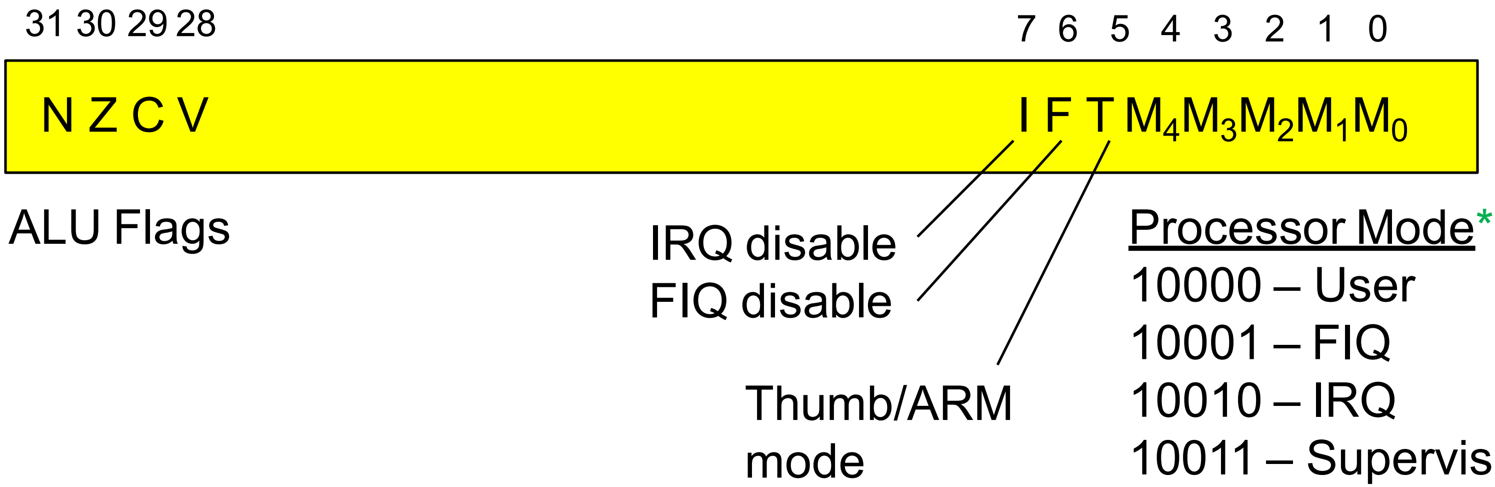
Changes from standard ARM architecture:

- Stack-based exception model
- Only two processor modes
- Thread Mode for User tasks*
- Handler Mode for OS tasks and exceptions*
- Vector table contains addresses

*Only SP changes between modes

Based on Lecture Notes by Marilyn Wolf

CPSR
Current Processor Status Register



<u>Processor Mode</u>**

10000 – User
10001 – FIQ
10010 – IRQ
10011 – Supervisor (SWI)
10111 – Abort D/I mem'y
11001 – Undefined instr.
11111 - System

Must be in a "privileged" mode to change the CPSR

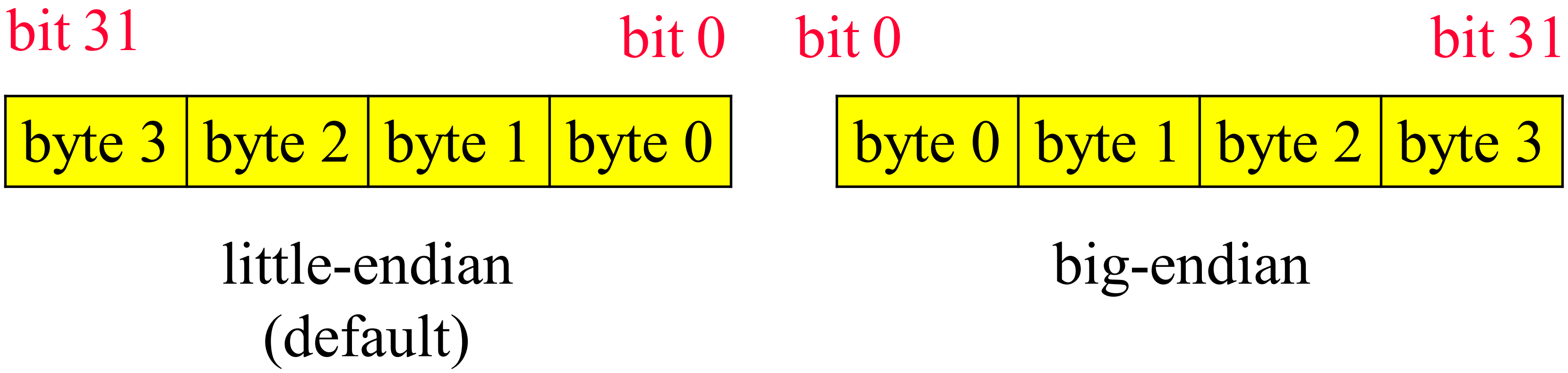
```
MRS  rn,CPSR
MSR  CPSR,rn
```

**2 modes in Cortex: Thread & Handler

Based on Lecture Notes by Marilyn Wolf

Endianness

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



bit 31 | byte 3 | byte 2 | byte 1 | byte 0 | bit 0

little-endian (default)

bit 0 | byte 0 | byte 1 | byte 2 | byte 3 | bit 31

big-endian

Based on Lecture Notes by Marilyn Wolf

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.
- Configure at power-up in either little- or bit-endian mode.

ARM status bits

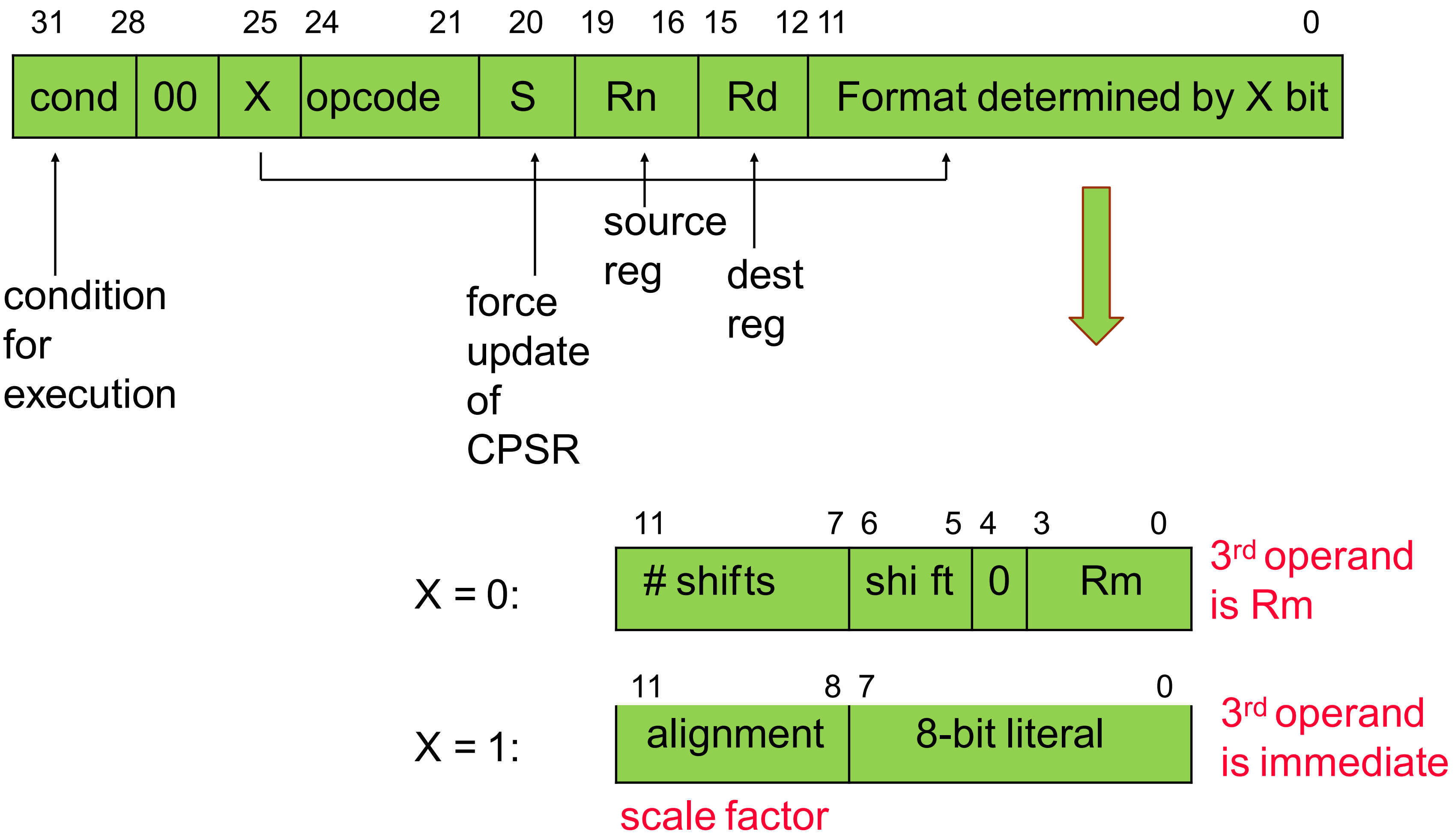
- Every arithmetic, logical, or shifting operation can set CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow)
- Examples:

  -1 + 1 = 0: NZCV = 0110.

  2^{31}-1+1 = -2^{31}: NZCV = 1001.

- Setting status bits must be explicitly enabled on each instruction
  - ex. "adds" sets status bits, whereas "add" does not

ARM Instruction Code Format



31 28		25	24 21	20	19 16	15 12	11 0
cond	00	X	opcode	S	Rn	Rd	Format determined by X bit

condition for execution

force update of CPSR

source reg

dest reg

X = 0:

11 7	6 5	4	3 0
# shifts	shift	0	Rm

3rd operand is Rm

X = 1:

11 8	7 0
alignment	8-bit literal

scale factor

3rd operand is immediate

Based on Lecture Notes by Marilyn Wolf

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

- Assembler translation:

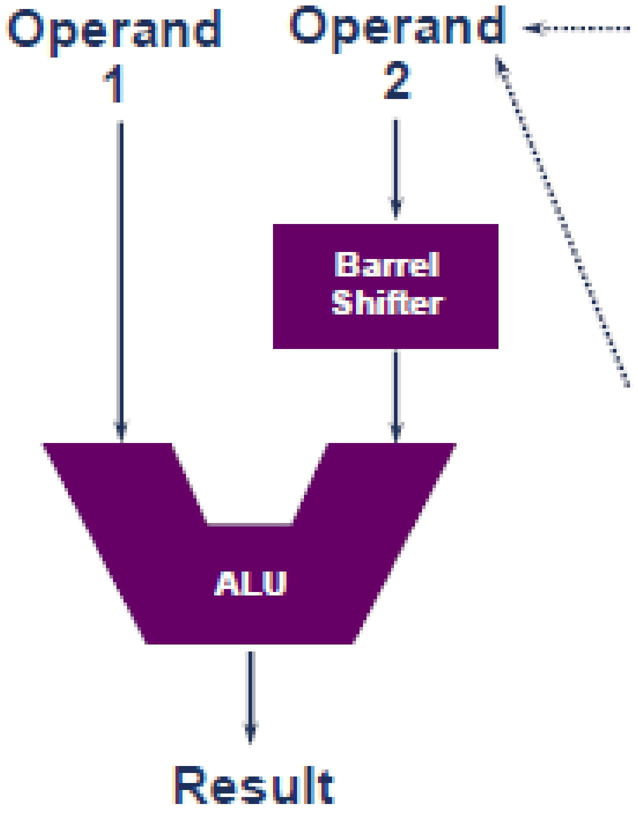
ADD r1,r2 => ADD r1,r1,r2 (but not MUL)

Based on Lecture Notes by Marilyn Wolf

Flexible 2nd operand

- 2nd operand = constant or register
- Constant with optional shift: (#8bit_value)
  - 8-bit value, shifted left any #bits (up to 32)
  - 0x00ab00ab, 0xab00ab00, 0xabababab (a,b hex digits)
- Register with optional shift: Rm,shift_type,#nbits
  - shift_type = ASR, LSL, LSR, ROR, with nbits < 32
  - shift_type RRX (rotate through X) by 1 bit

Barrel shifter for 2nd operand



Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

ARM arithmetic instructions

- ADD, ADC : add (w.carry)

  [Rd] <= Op1 + Op2 + C
- SUB, SBC : subtract (w.carry)

  [Rd] <= Op1 – Op2 + (C – 1)
- RSB, RSC : reverse subtract (w.carry)

  [Rd] <= OP2 – Op1 + (C – 1)
- MUL: multiply (32-bit product – no immediate for Op2)

  [Rd] <= Op1 x Op2
- MLA : multiply and accumulate (32-bit result)

  MLA Rd,Rm,Rs,Rn : [Rd] <= (Rm x Rs) + Rn

Based on Lecture Notes by Marilyn Wolf

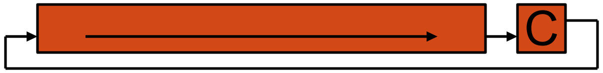
ARM logical instructions

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

  ADD r1,r2,r3, LSL #4 : [r1] <= r2 + (r3x16)

  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:



New Thumb2 bit operations

- Bit field insert/clear (to pack/unpack data within a register)

```
BFC r0,#5,#4   ;Clear 4 bits of r0, starting with bit #5
BFI   r0,r1,#5,#4  ;Insert 4 bits of r1 into r0, start at bit #5
```

- Bit reversal (REV) – reverse order of bits within a register
  - Bit [n] moved to bit [31-n], for n = 0..31
  - Example:

```
REV  r0,r1 ;reverse order of bits in r1 and put in r0
```

Based on Lecture Notes by Marilyn Wolf

ARM comparison instructions

These instructions only set the NZCV bits of CPSR – no other result is saved. (“Set Status” is implied)

- CMP : compare : Op1 – Op2
- CMN : negated compare : Op1 + Op2
- TST : bit-wise AND : Op1 ^ Op2
- TEQ : bit-wise XOR : Op1 xor Op2

Based on Lecture Notes by Marilyn Wolf

ARM move instructions

- MOV, MVN : move (negated), constant = 8 or 16 bits

```
MOV   r0, r1     ; sets r0 to r1
MOVN  r0, r1     ; sets r0 to r̅1̅
MOV   r0, #55    ; sets r0 to 55
MOV   r0,#0x5678   ;Thumb2 r0[15:0]
MOVT  r0,#0x1234   ;Thumb2 r0[31:16]
```

- Use shift modifier to scale a value:

```
MOV   r0,r1,LSL #6 ; [r0] <= r1 x 64
```

- Special pseudo-op:

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

Based on Lecture Notes by Marilyn Wolf

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-most16 bits of register)
  - STRB : store 8-bit byte (right-most 8 bits of register)

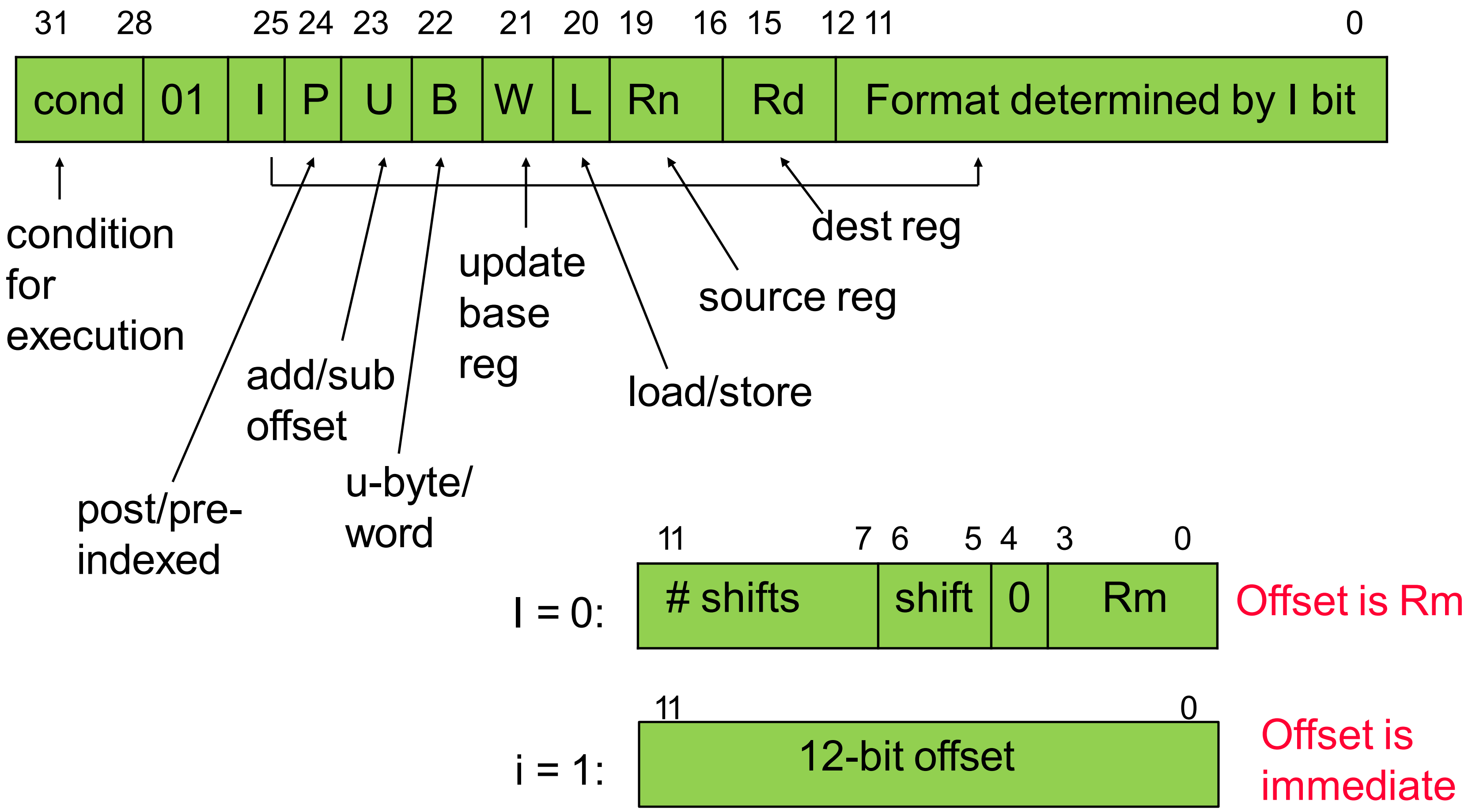
ARM load/store addressing

- 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 #offset = 12 bits (2’s complement)

Based on Lecture Notes by Marilyn Wolf

ARM Load/Store Code Format



31 28		25	24	23	22	21	20	19 16	15 12	11 0
cond	01	I	P	U	B	W	L	Rn	Rd	Format determined by I bit

condition for execution

post/pre-indexed

add/sub offset

u-byte/word

update base reg

load/store

source reg

dest reg

I = 0:

11 7	6 5	4	3 0
# shifts	shift	0	Rm

Offset is Rm

i = 1:

11 0
12-bit offset

Offset is immediate

Based on Lecture Notes by Marilyn Wolf

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)+$\underbrace{(r3 \times 4)}_{\text{Scaled index}}$

Base register r2 is not altered in these instructions

Based on Lecture Notes by Marilyn Wolf

ARM load/store examples

(base register updated by auto-indexing)

- ldr r1,[r2,#4]! ; use address = (r2)+4

  ; r2<=(r2)+4 (pre-index)
- ldr r1,[r2,r3]! ; use address = (r2)+(r3)

  ; r2<=(r2)+(r3) (pre-index)
- ldr r1,[r2],#4 ; use address = (r2)

  ; r2<=(r2)+4 (post-index)
- ldr r1,[r2],[r3] ; use address = (r2)

  ; r2<=(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] -> rd, rd -> M[rn]

Based on Lecture Notes by Marilyn Wolf

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

Based on Lecture Notes by Marilyn Wolf

Example: C assignments

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

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

Example: C assignment

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

Based on Lecture Notes by Marilyn Wolf

Example: C assignment

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

Based on Lecture Notes by Marilyn Wolf

ARM flow control operations

- All operations can be performed conditionally, testing CPSR (only branches in Thumb/Thumb2):
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:

```
      B label
 Target < ±32M(ARM),±2K(Thumb),±16M(Thumb2)
```

- Conditional branch:

```
   BNE label
 Target < ±32M(ARM),-252..+258(T),±1M(T2)
```

- Thumb2 additions (compare & branch if zero/nonzero):

```
 CBZ  r0,label  ;branch if r0 == 0
 CBNZ r0,label  ;branch if r0 != 0
```

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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 ...
```

Based on Lecture Notes by Marilyn Wolf

Example: Conditional instruction implementation

(ARM mode only – not available in Thumb/Thumb 2 mode)

```
; 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
```

Based on Lecture Notes by Marilyn Wolf

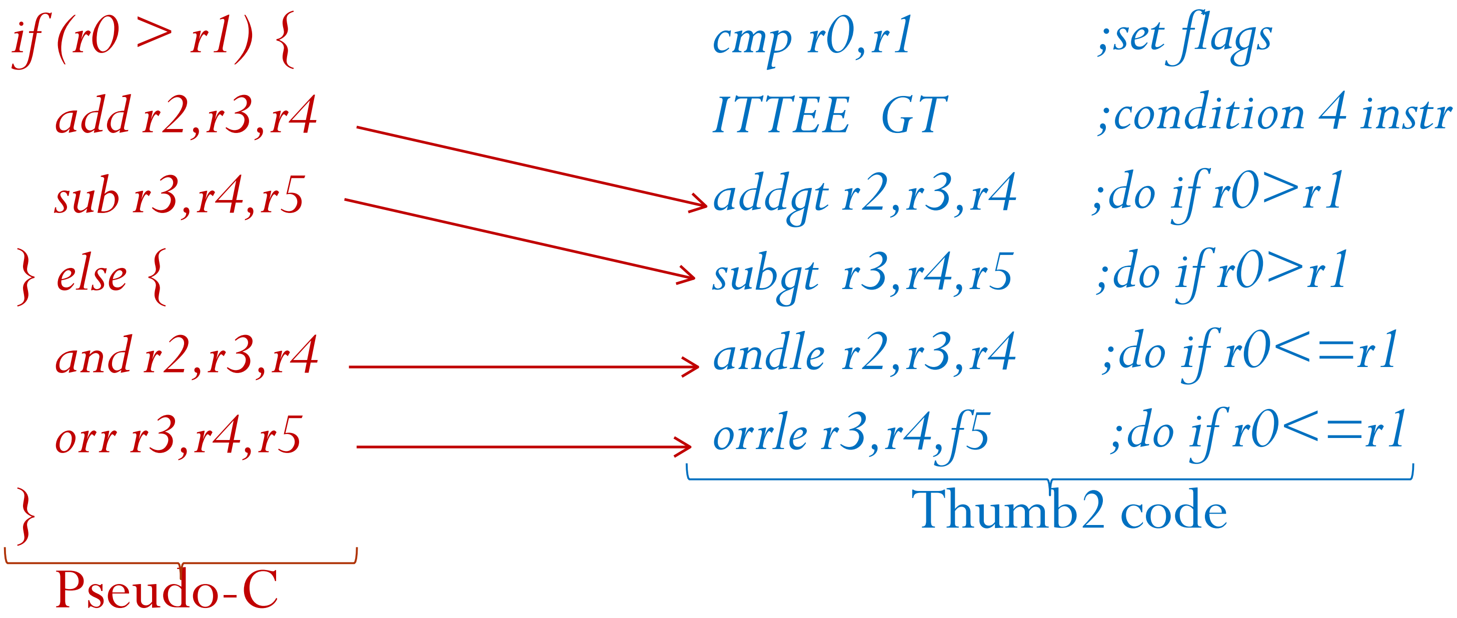
Conditional instruction implementation, cont'd.

```
; 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
```

Based on Lecture Notes by Marilyn Wolf

Thumb2 conditional execution

- (IF-THEN) instruction, IT, supports conditional execution in Thumb2 of up to 4 instructions in a "block"
  - Designate instructions to be executed for THEN and ELSE
  - Format: ITxyz condition, where x,y,z are T/E/blank



```
if (r0 > r1) {
    add r2,r3,r4
    sub r3,r4,r5
} else {
    and r2,r3,r4
    orr r3,r4,r5
}
```

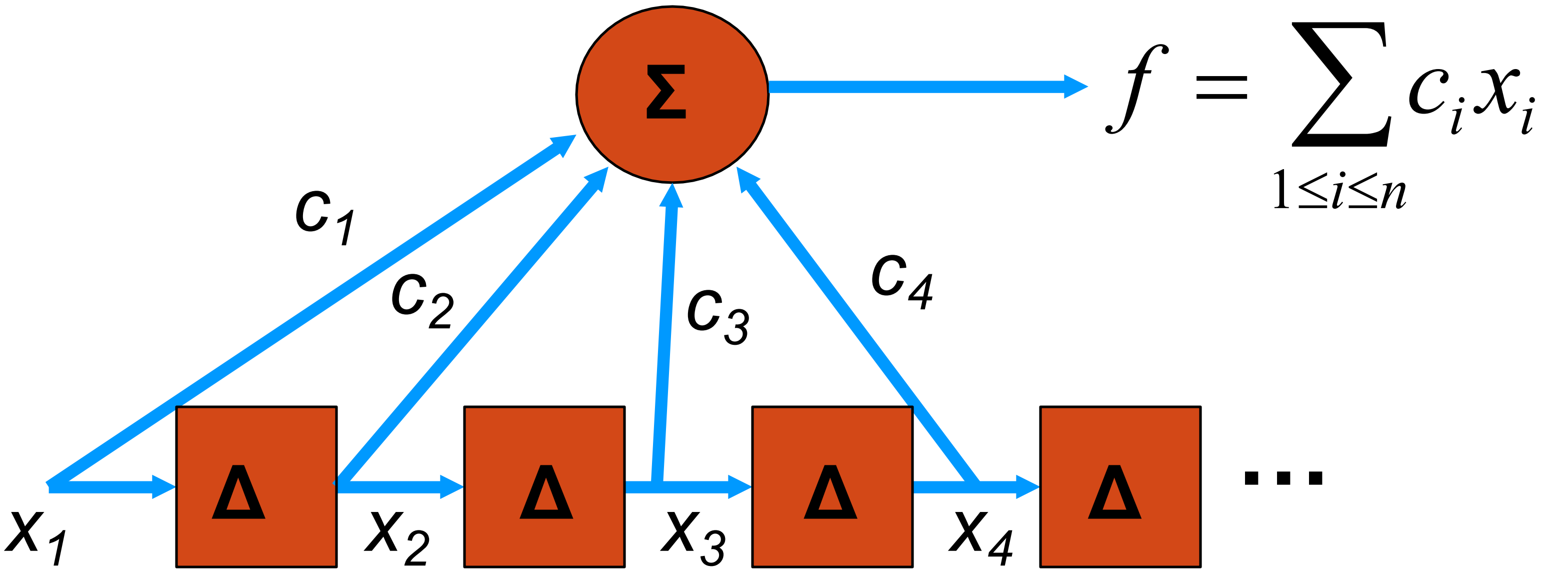
Pseudo-C

```
cmp r0,r1            ;set flags
ITTEE  GT            ;condition 4 instr
addgt r2,r3,r4       ;do if r0>r1
subgt  r3,r4,r5      ;do if r0>r1
andle r2,r3,r4       ;do if r0<=r1
orrle r3,r4,f5       ;do if r0<=r1
```

Thumb2 code

Based on Lecture Notes by Marilyn Wolf

Finite impulse response (FIR) filter



$$f = \sum_{1 \le i \le n} c_i x_i$$

X_i's are data samples

C_i's are constants

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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

Based on Lecture Notes by Marilyn Wolf

ARM subroutine linkage

- Branch and link instruction:

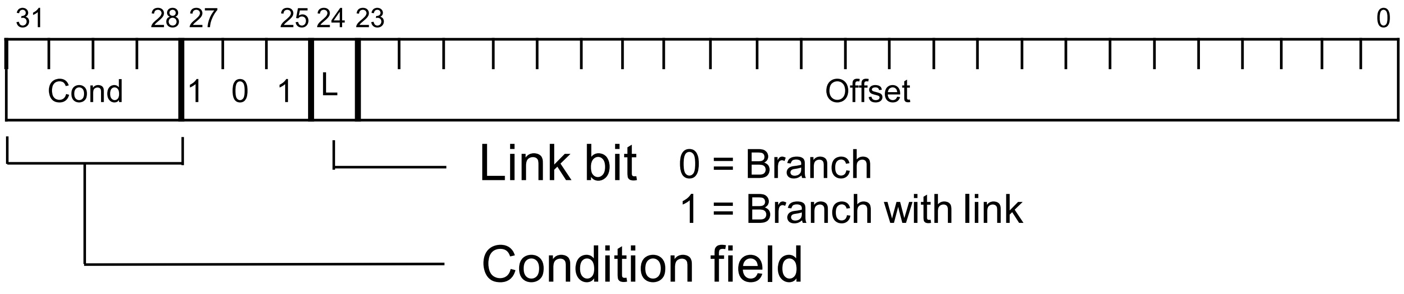
```
  BL foo          ; copies current PC to r14.
```

- To return from subroutine:

```
  BX   r14        ; branch to address in r14
or:
  MOV r15,r14      --Not recommended for Cortex
```

- 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 CPU shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range(ARM
  - Thumb: ± 16 Mbyte (unconditional),± 1 Mbyte (conditional)
  - How to perform longer branches?
  - Bcond is only conditional instruction allowed outside of IT block

Based on Lecture Notes by Marilyn Wolf

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);
 }
```

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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
```

Based on Lecture Notes by Marilyn Wolf

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	Ip (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
  stmda r13,{r8-r12,r14}   ;r13 not updated at end
  ```

  Lowest numbered register at lowest memory address

Based on Lecture Notes by Marilyn Wolf

ARM assembler additions

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

Mutual exclusion support

- Test and set a "lock/semaphore" for shared data access
  - Lock=0 indicates shared resource is unlocked (free to use)
  - Lock=1 indicates the shared resource is "locked" (in use)
- LDREX Rt,[Rn{,#offset}]
  - read lock value into Rt from memory to request exclusive access to a resource
  - Cortex notes that LDREX has been performed, and waits for STRTX
- STREX Rd,Rt,[Rn{,#offset}]
  - Write Rt value to memory and return status to Rd
  - Rd=0 if successful write, Rd=1 if unsuccessful write
  - Cortex notes that LDREX has been performed, and waits for STRTX
  - "fail" if LDREX by another thread before STREX performed by first thread
- CLREX
  - Force next STREX to return status of 1to Rd (cancels LDREX)

Mutual exclusion example

- Location "Lock"is 0 if a resource is free, 1 if not free

```
        ldr      r0,=Lock      ;point to lock
        mov      r1,#1         ;prepare to lock the resource
try     ldrex    r2,[r0]       ;read Lock value
        cmp      r2,#0         ;is resource unlocked/free?
        itt      eq            ;next 2 ops if resource free
        strexeq  r2,r1,[r0]    ;store 1 in Lock
        cmpeq    r2,#0         ;was store successful?
        bne      try           ;repeat loop if lock unsuccessful
```

LDREXB/LDREXH - STREXB/STREXH for byte/halfword Lock

Based on Lecture Notes by Marilyn Wolf

Common assembler directives

- Allocate storage and store initial values (CODE area)

Label	DCD	value1,value2…	allocate word
Label	DCW	value1,value2…	allocate half-word
Label	DCB	value1,value2…	allocate byte

- Allocate storage without initial values (DATA area)

Label	SPACE	n	reserve n bytes (uninitialized)

Based on Lecture Notes by Marilyn Wolf

Summary

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