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

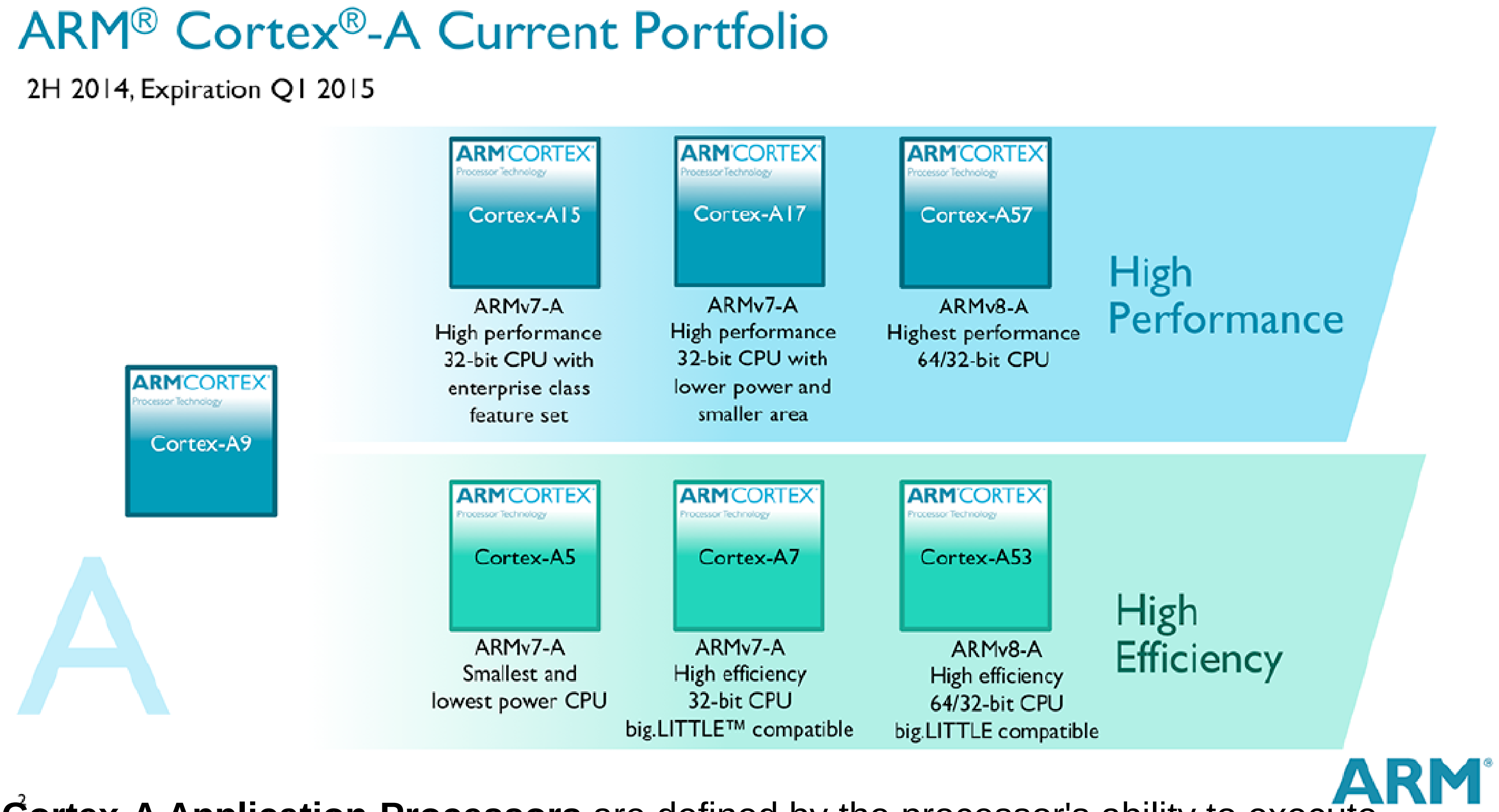
ARM available as microcontrollers, IP cores, etc.

www.arm.com

Based on Lecture Notes by Marilyn Wolf

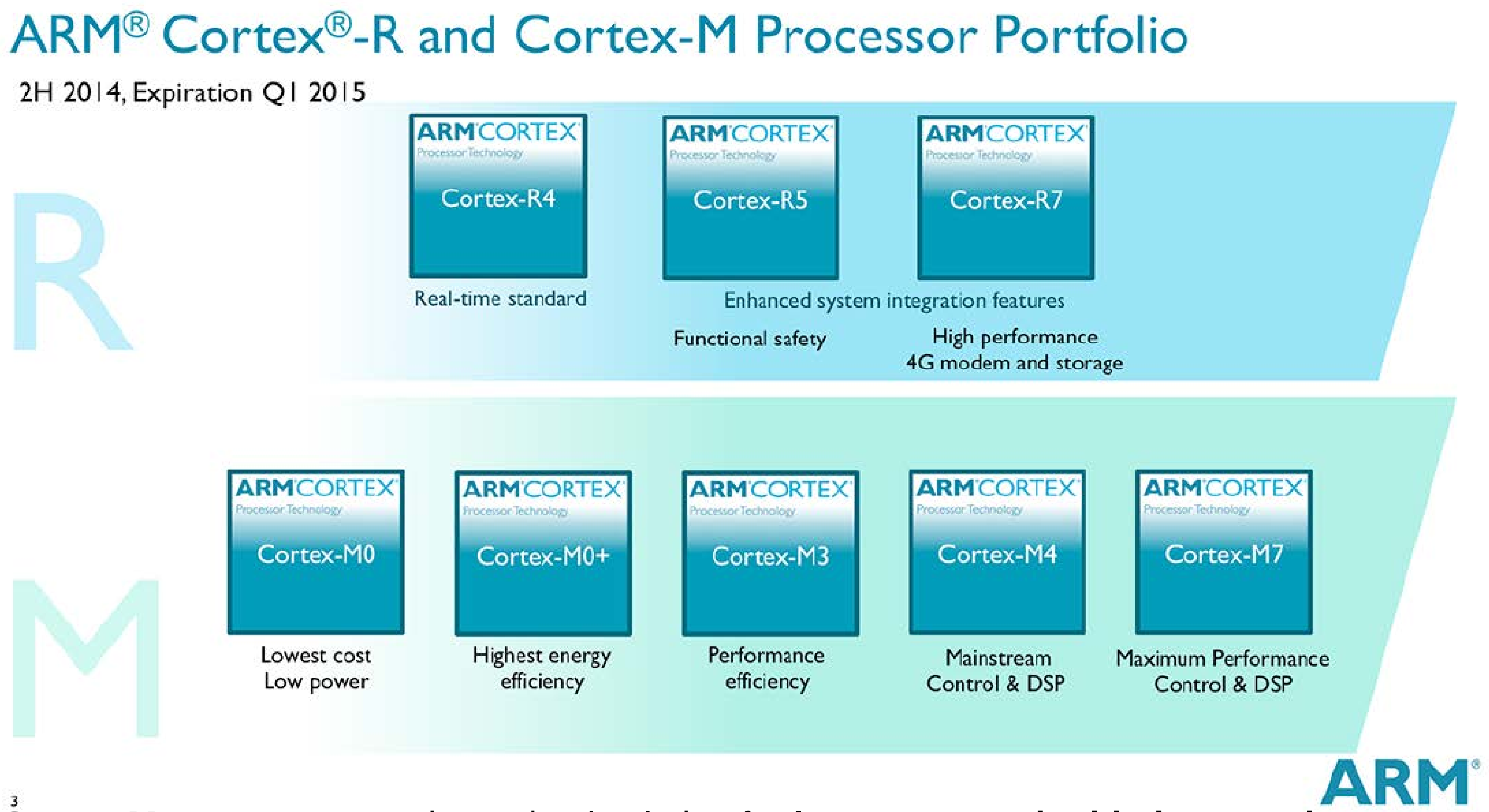
ARM instruction set - outline

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


ARM® Cortex®-A Current Portfolio
2H 2014, Expiration Q1 2015
ARM CORTEX Processor Technology Cortex-A15
ARMv7-A High performance 32-bit CPU with enterprise class feature set
ARM CORTEX Processor Technology Cortex-A17
ARMv7-A High performance 32-bit CPU with lower power and smaller area
ARM CORTEX Processor Technology Cortex-A57
ARMv8-A Highest performance 64/32-bit CPU
High Performance
ARM CORTEX Processor Technology Cortex-A9
ARM CORTEX Processor Technology Cortex-A5
ARMv7-A Smallest and lowest power CPU
ARM CORTEX Processor Technology Cortex-A7
ARMv7-A High efficiency 32-bit CPU big.LITTLE™ compatible
ARM CORTEX Processor Technology Cortex-A53
ARMv8-A High efficiency 64/32-bit CPU big.LITTLE compatible
High Efficiency
A
ARM®

Cortex-A Application Processors are defined by the processor's ability to execute complex operating systems, including Linux, Android, Chrome OS, Tizen, Microsoft Windows (CE/Embedded) and others. This class of processors integrates a Memory Management Unit (MMU) to manage the memory requirements of complex OSs and enable the download and execution of third party software.

•Smartphones *Feature Phones *Tablets / eReaders *Adv. Personal Media Players
•Digital Television *Set-top Boxes/Satellite Receivers *High-End Printers
•Personal Navigation Devices *Server/Enterprise *Wearables *Home Networking


ARM® Cortex®-R and Cortex-M Processor Portfolio

2H 2014, Expiration Q1 2015

R
ARM CORTEX Processor Technology Cortex-R4
Real-time standard
ARM CORTEX Processor Technology Cortex-R5
ARM CORTEX Processor Technology Cortex-R7
Enhanced system integration features
Functional safety
High performance 4G modem and storage
M
ARM CORTEX Processor Technology Cortex-M0
Lowest cost Low power
ARM CORTEX Processor Technology Cortex-M0+
Highest energy efficiency
ARM CORTEX Processor Technology Cortex-M3
Performance efficiency
ARM CORTEX Processor Technology Cortex-M4
Mainstream Control & DSP
ARM CORTEX Processor Technology Cortex-M7
Maximum Performance Control & DSP
ARM
3

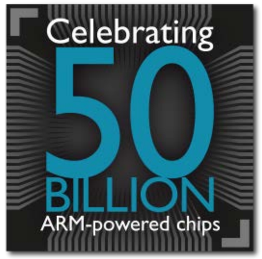
Cortex-M processors are the optimal solution for **low-power embedded computing** applications. The 32-bit Cortex-M processor family is the key to transforming all sorts of embedded systems into smart and connected systems. Often provided as a "black box" with pre-loaded applications, they have limited capability to expand hardware functionality and in most cases no screen.

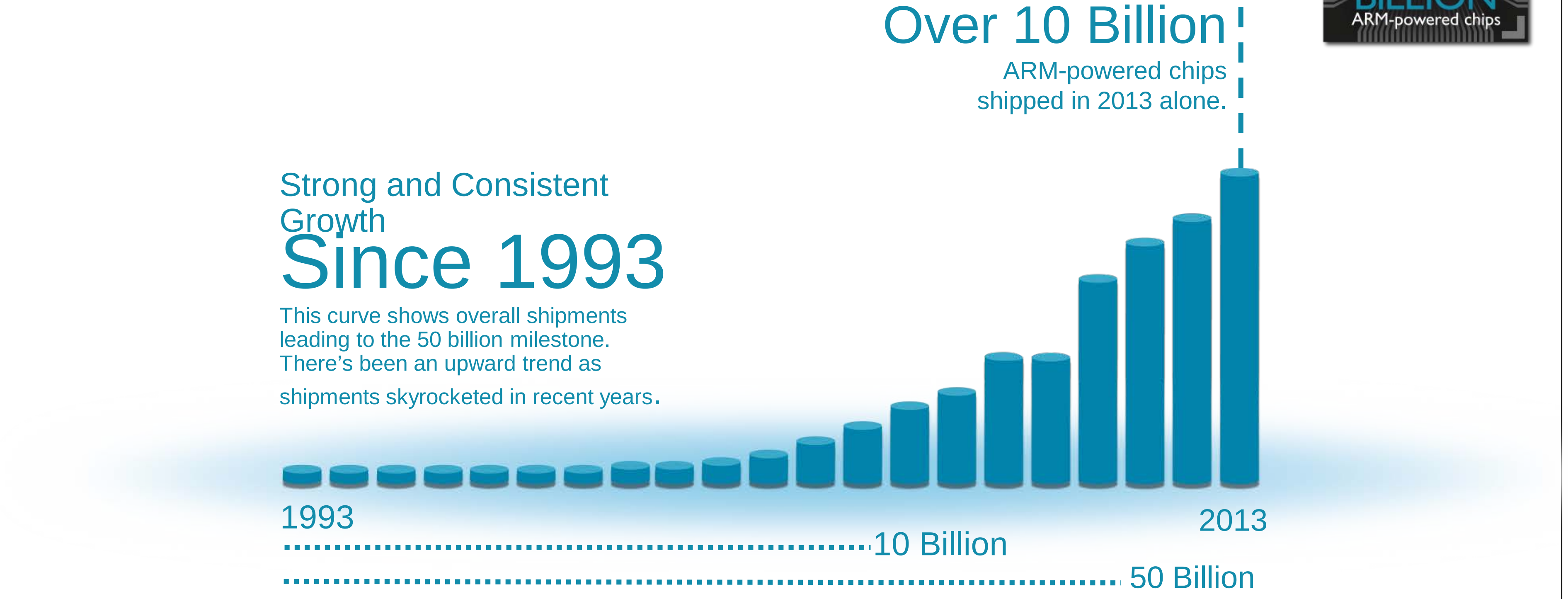
•Merchant MCUs *Automotive Control Systems *Motor Control Systems
•White Goods controllers *Smart Meters *Sensors *Internet of Things

Equipment Adopting ARM Cores

Source: ARM University Program Overview

With More Than 50 Billion

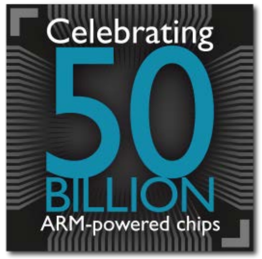


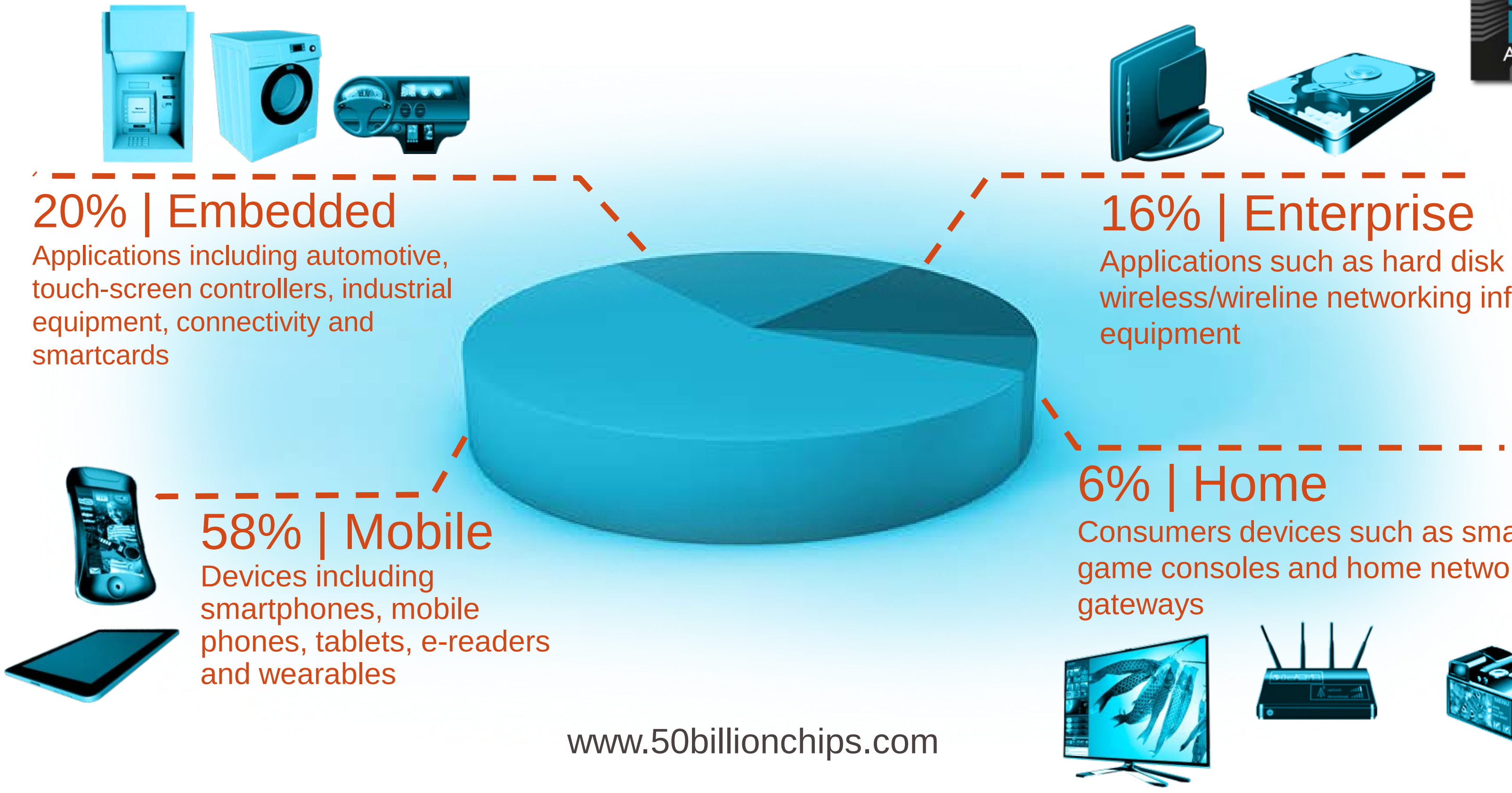


www.50billionchips.com

Source: ARM University Program Overview

Markets we're POWERING





20% | Embedded
Applications including automotive, touch-screen controllers, industrial equipment, connectivity and smartcards

16% | Enterprise
Applications such as hard disk drives, and wireless/wireline networking infrastructure equipment

58% | Mobile
Devices including smartphones, mobile phones, tablets, e-readers and wearables

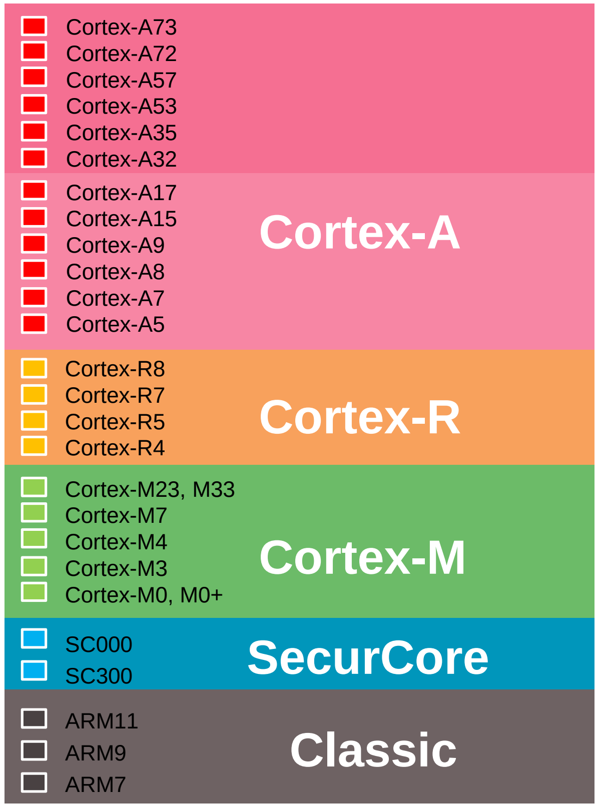
6% | Home
Consumers devices such as smart TVs, game consoles and home networking gateways

www.50billionchips.com

Source: ARM University Program Overview

ARM processor families

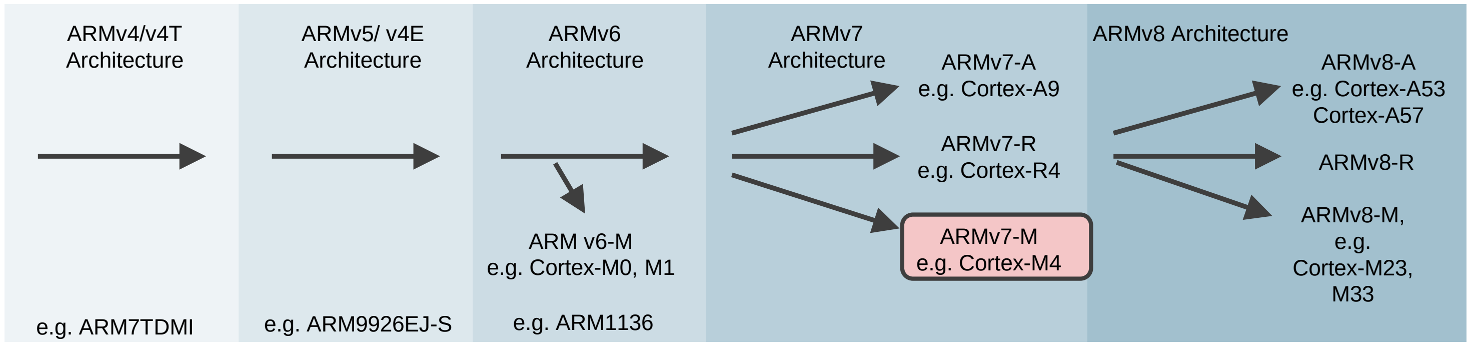
- **Cortex-A series (Application)**
  - High performance processors capable of full Operating System (OS) support
  - Applications include smartphones, digital TV, smart books
- **Cortex-R series (Real-time)**
  - High performance and reliability for real-time applications;
  - Applications include automotive braking system, powertrains
- **Cortex-M series (Microcontroller)**
  - Cost-sensitive solutions for deterministic microcontroller applications
  - Applications include microcontrollers, smart sensors
- SecurCore series for high security applications
- Earlier classic processors including ARM7, ARM9, ARM11 families



Cortex Low-Power Leadership from ARM

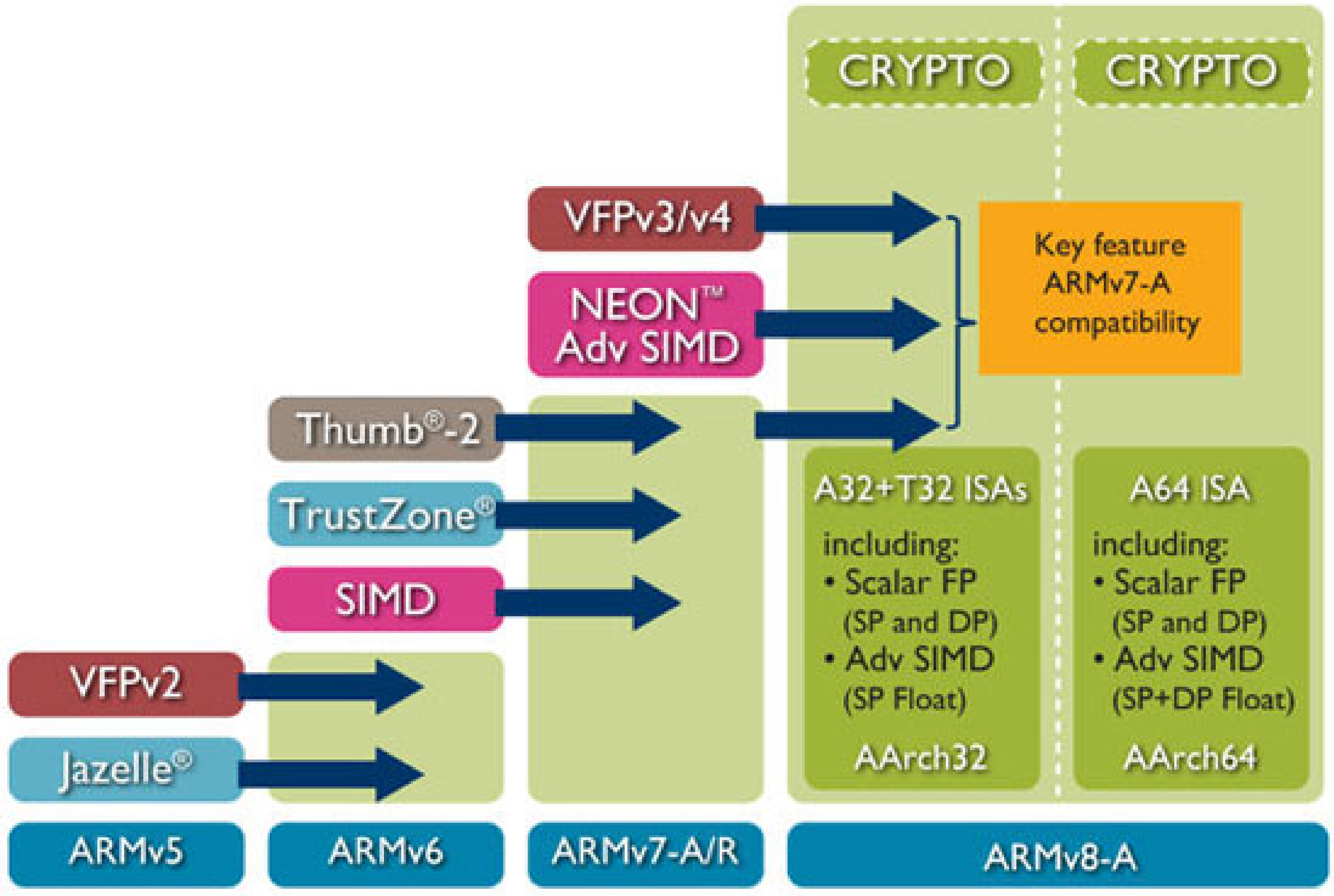
ARM processors vs. ARM architectures

- **ARM architecture**
  - Describes the details of instruction set, programmer's model, exception model, and memory map
  - Documented in the Architecture Reference Manual
- **ARM processor**
  - Developed using one of the ARM architectures
  - More implementation details, such as timing information
  - Documented in processor's Technical Reference Manual



ARM Architecture versions

(From arm.com)


CRYPTO
CRYPTO
VFPv3/v4
NEON™ Adv SIMD
Key feature ARMv7-A compatibility
Thumb®-2
TrustZone®
SIMD
VFPv2
Jazelle®
A32+T32 ISAs including: • Scalar FP (SP and DP) • Adv SIMD (SP Float)
AArch32
A64 ISA including: • Scalar FP (SP and DP) • Adv SIMD (SP+DP Float)
AArch64
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

ARM Cortex-M series

- Cortex-M series: Cortex-M0, M0+, M3, M4, M7, M22, M23
  - Low cost, low power, bit and byte operations, fast interrupt response
- Energy-efficiency
  - Lower energy cost, longer battery life
- Smaller code
  - Lower silicon costs
- Ease of use
  - Faster software development and reuse
- Embedded applications
  - Smart metering, human interface devices, automotive and industrial control systems, white goods, consumer products and medical instrumentation

ARM Cortex-M processor profile

- M0: Optimized for size and power (13 µW/MHz dynamic power)
- M0+: Lower power (11 µW/MHz dynamic power), shorter pipeline
- M3: Full Thumb and Thumb-2 instruction sets, single-cycle multiply instruction, hardware divide, saturated math, (32 µW/MHz)
- **M4: Adds DSP instructions, optional floating point unit**
- M7: designed for embedded applications requiring high performance
- M23, M33: include ARM TrustZone® technology for solutions that require optimized, efficient security

ARM Cortex-M series family

Processor	ARM Architecture	Core Architecture	Thumb®	Thumb®-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	ARMv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	ARMv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M3	ARMv7-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	No	No
Cortex-M4	ARMv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional
Cortex-M7	ARMv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional
Cortex-M23, 33	ARMv8-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional

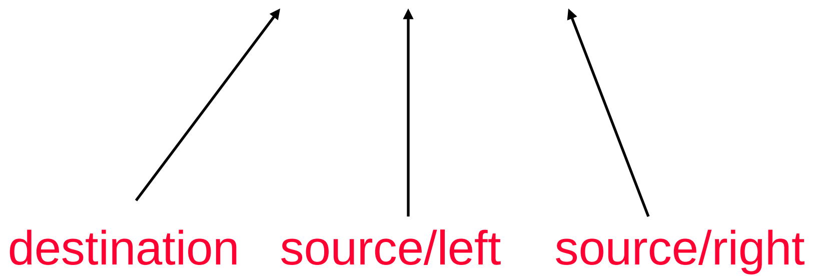
RISC CPU Characteristics

- 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

ARM assembly language

- Fairly standard RISC assembly language:

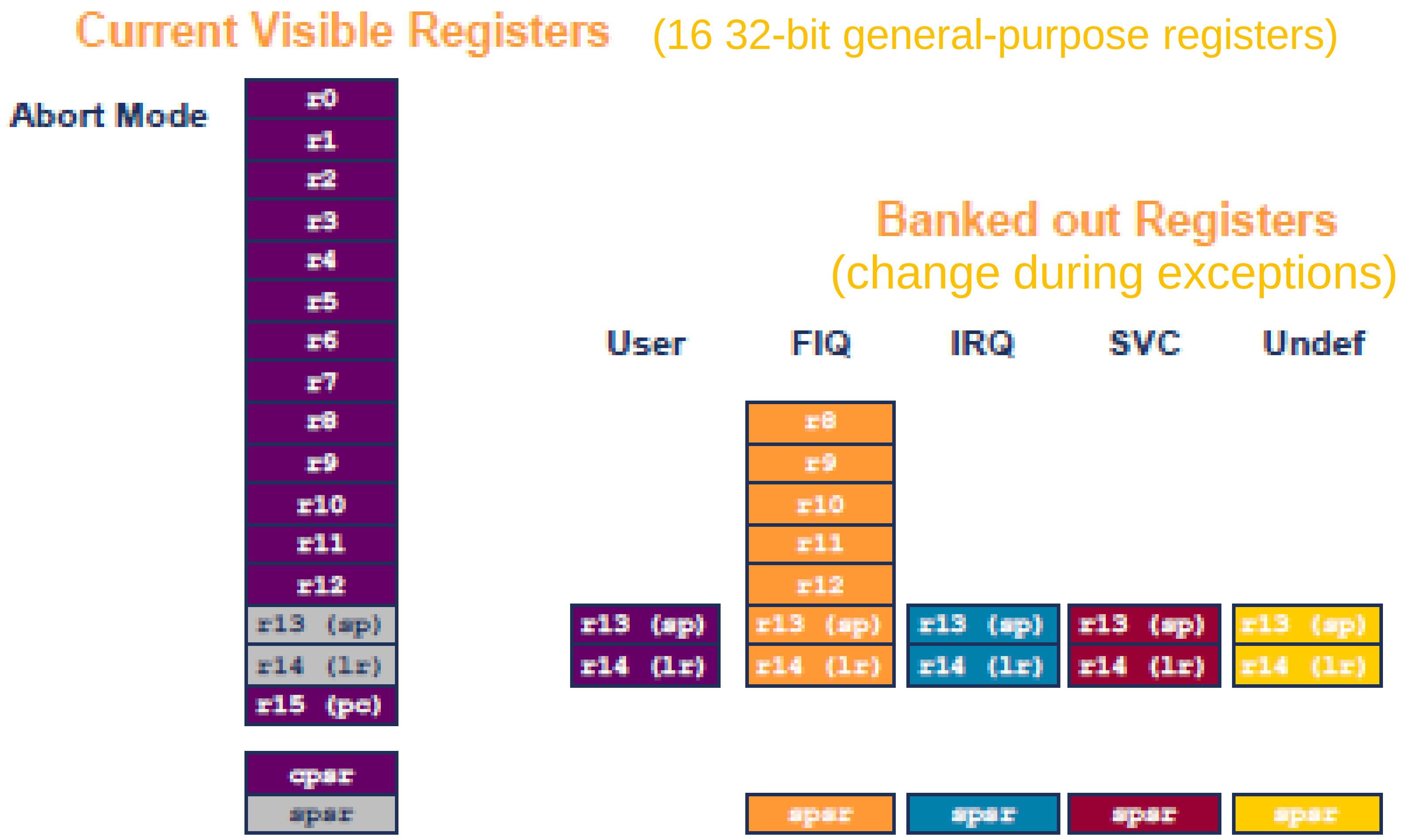
```
         LDR r0,[r8]   ; a comment
label    ADD r4,r0,r1  ;r4=r0+r1
```



ARM Register Set

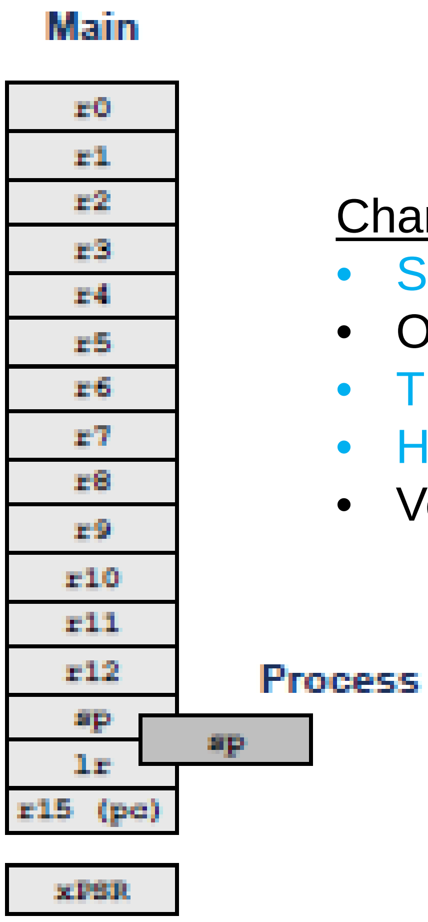
(16 32-bit general-purpose registers)

(change during exceptions)



Based on Lecture Notes by Marilyn Wolf

ARM Cortex register set

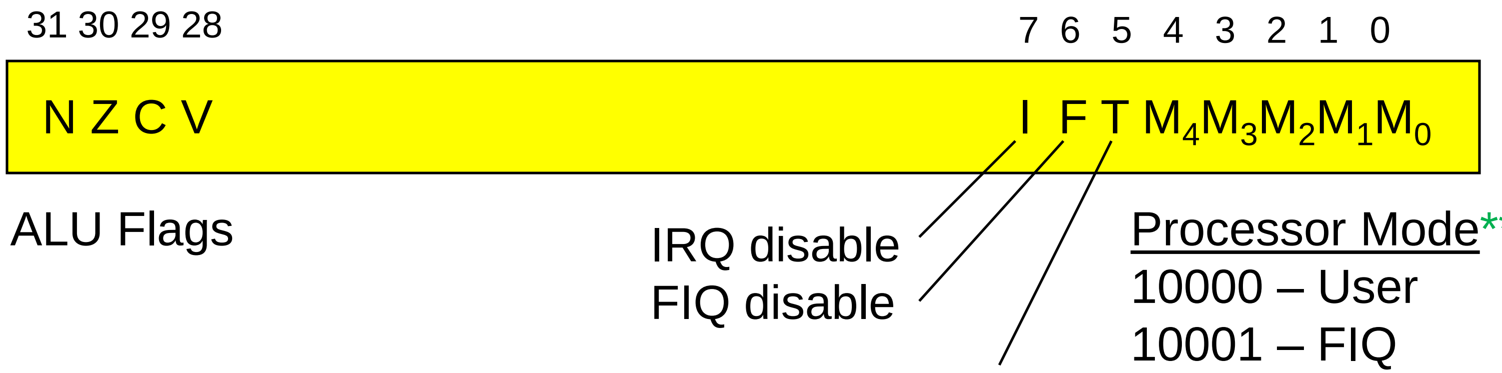


Changes from standard ARM architecture:

- 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

CPSR
Current Processor Status Register



ALU Flags

Processor Mode**

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

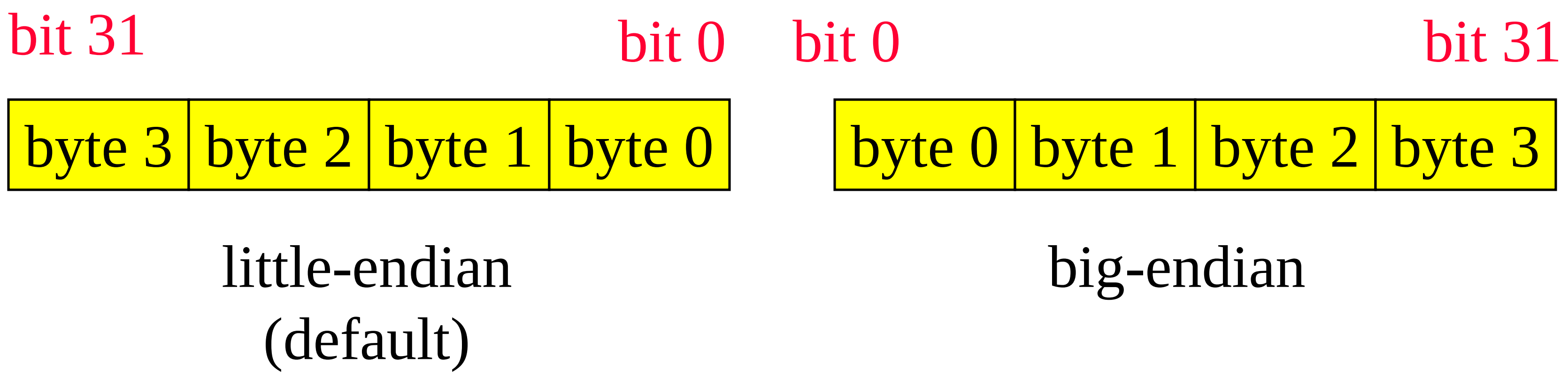
Must be in a "privileged" mode
to change the CPSR

MRS rn,CPSR
MSR CPSR,rn

**2 modes in Cortex:
Thread & Handler

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

ARM status bits

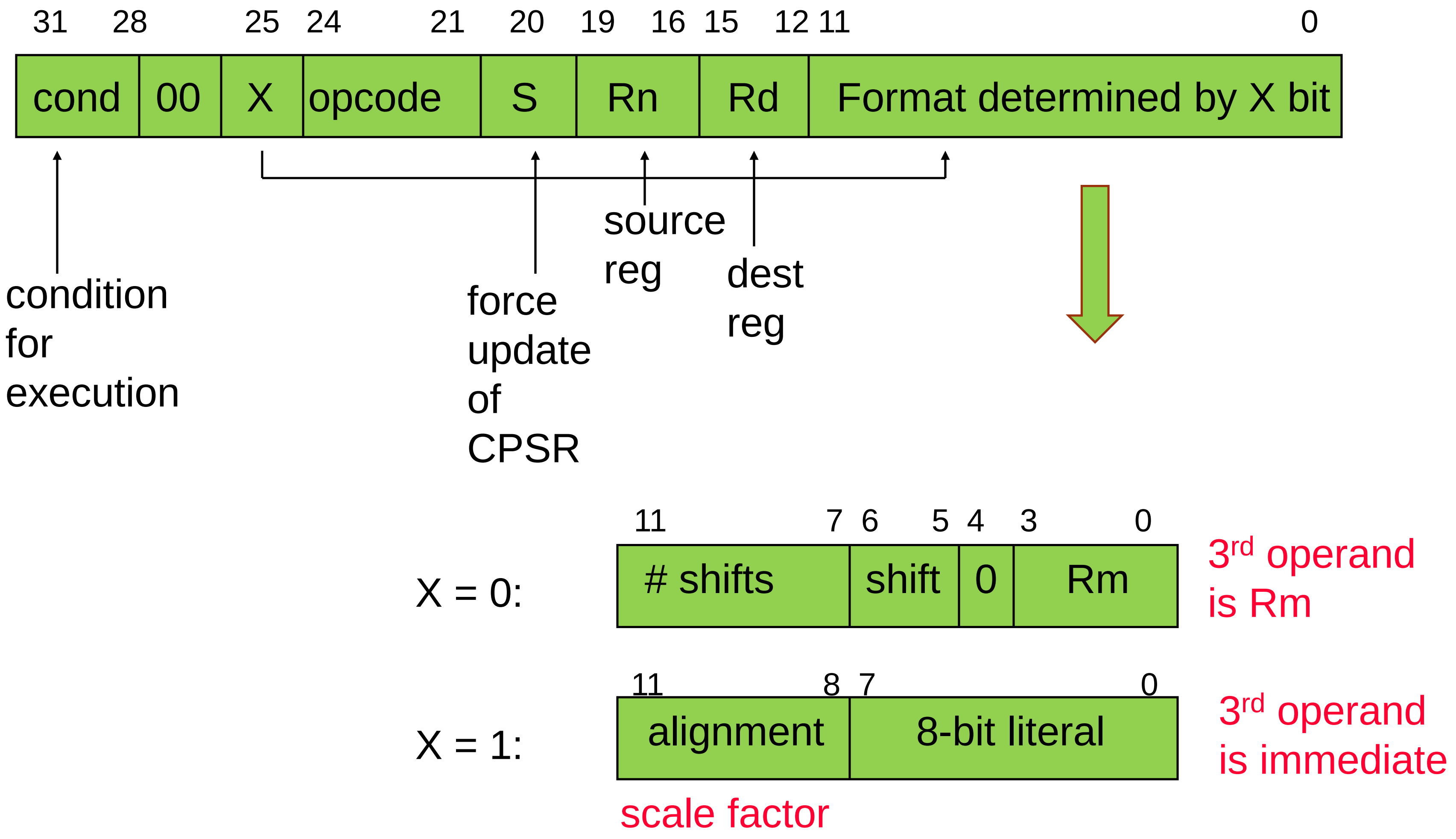
- Every arithmetic, logical, or shifting operation <u>can</u> set CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow)
- Examples:

  -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



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
```

$\uparrow$ set status flags

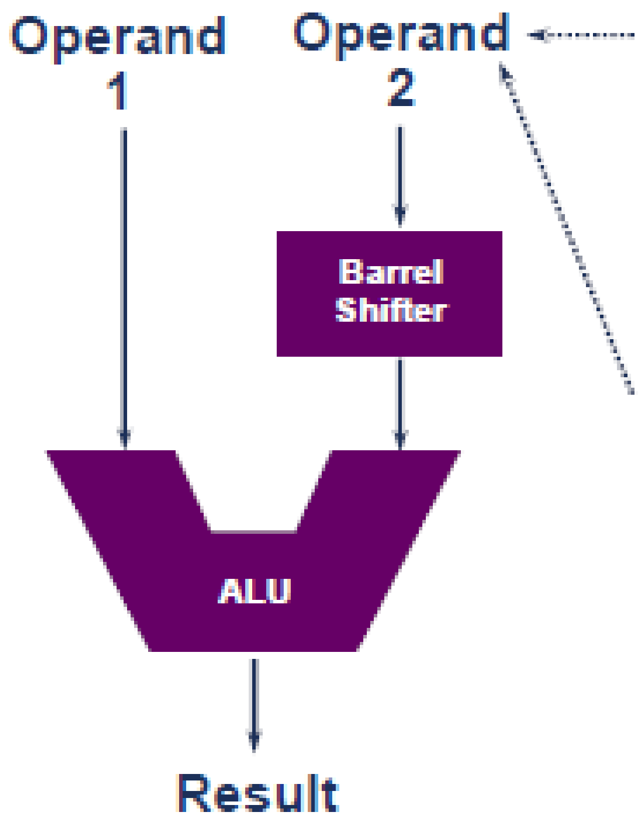
- Assembler translation:

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

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

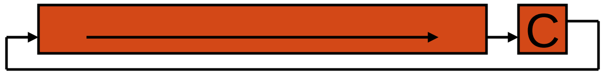
ARM logical instructions

- AND, ORR, EOR: bit-wise logical op's
- BIC : bit clear $\quad$ [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
```

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

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` $\overline{\texttt{r1}}$

  `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`

ARM load/store instructions

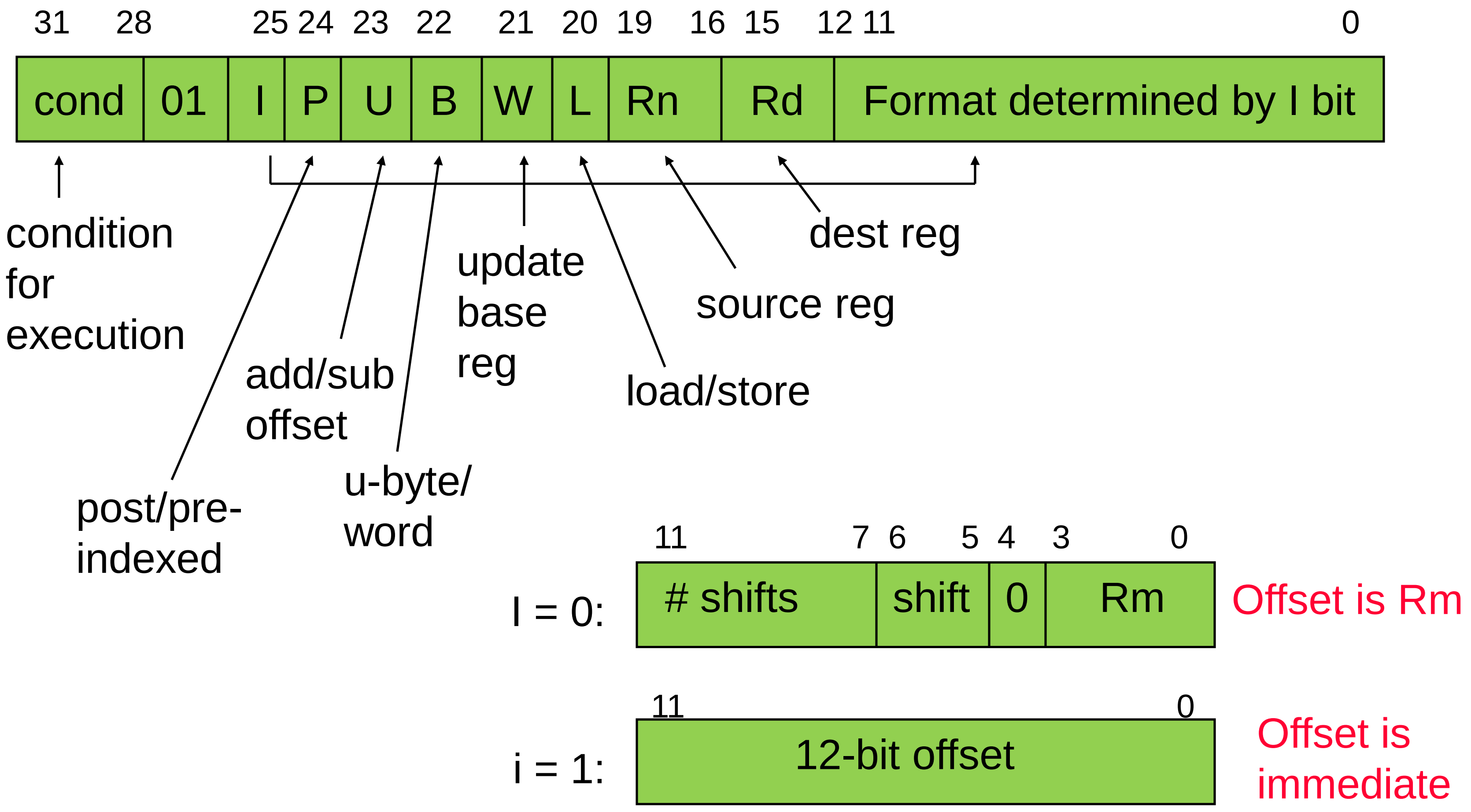
- Load operand from memory into target register
  - LDR – load 32 bits
  - LDRH – load halfword (16 bit unsigned #) & zero-extend to 32 bits
  - LDRSH – load signed halfword & sign-extend to 32 bits
  - LDRB – load byte (8 bit unsigned #) & zero-extend to 32 bits
  - LDRSB – load signed byte & sign-extend 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)

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

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<=(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]

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 (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:

```
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 (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
```

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

(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
```

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
```

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) {                  cmp r0,r1           ;set flags
   add r2,r3,r4                 ITTEE  GT           ;condition 4 instr
   sub r3,r4,r5    ------->     addgt r2,r3,r4      ;do if r0>r1
} else {           ------->     subgt r3,r4,r5      ;do if r0>r1
   and r2,r3,r4    ------->     andle r2,r3,r4      ;do if r0<=r1
   orr r3,r4,r5    ------->     orrle r3,r4,f5      ;do if r0<=r1
}
```

Thumb2 code

Pseudo-C

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 pc,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
          DCD case1
          ...
```

Example: switch statement with new "Table Branch" instruction

Branch address = PC + 2*offset from table of offsets
Offset = byte (TBB) or half-word (TBH)

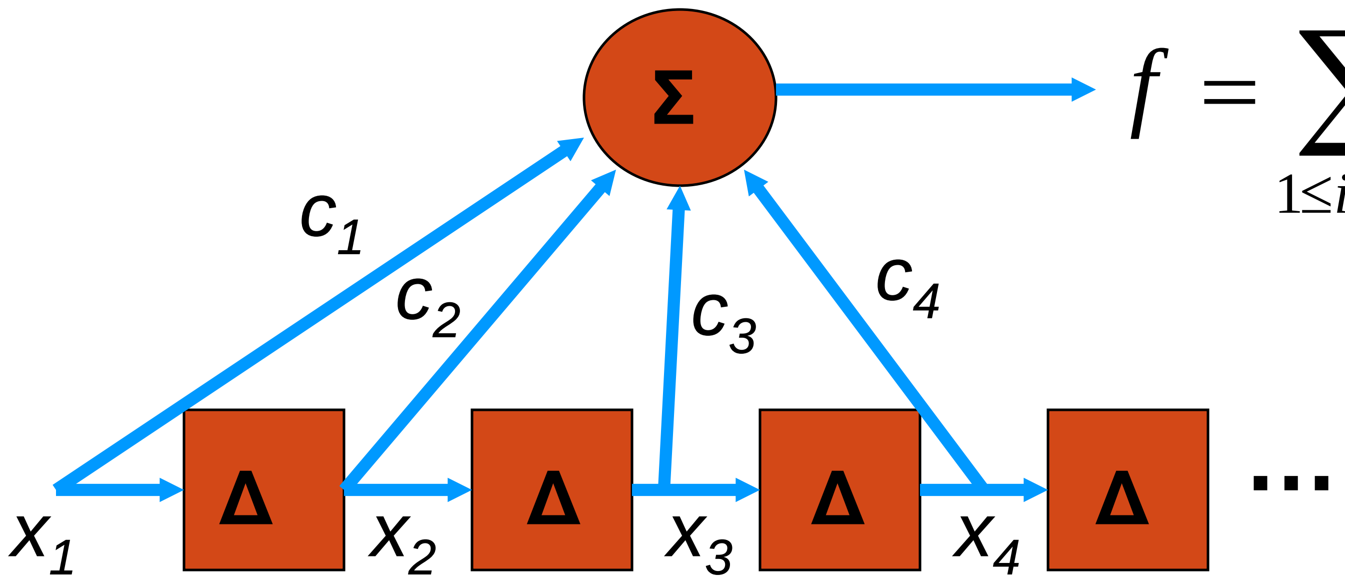
- C:

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

- Assembler:

```
          LDR r2,=test   ; get address for test
          LDR r0,[r2]    ; load value for test
          TBB [pc,r0]    ; add offset byte to PC
switchtab DCB (case0 – switchtab) >> 1  ;byte offset
          DCB (case1 – switchtab) >> 1  ;byte offset
case0  instructions
case1  instructions
(TBH similar, but with 16-bit offsets/DCI)
```

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

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:

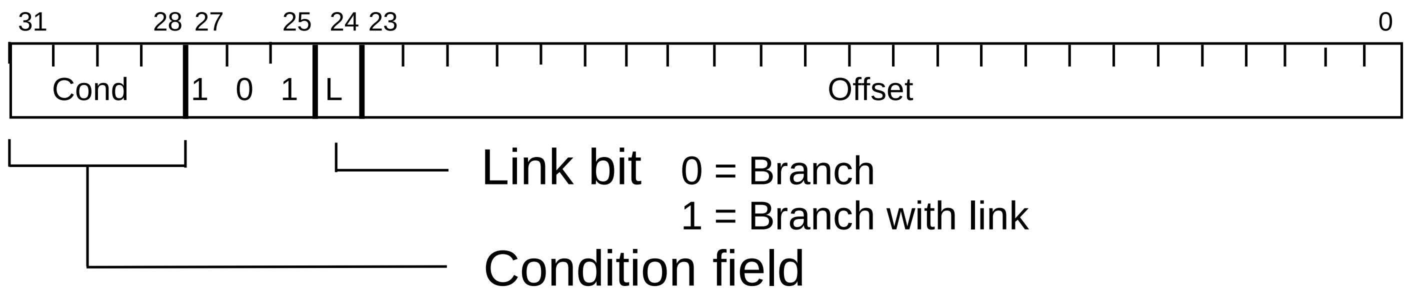
```
   BL foo          ;Copies current PC to r14.
```

- To return from subroutine:

```
   BX  r14         ; branch to address in r14
 or:
   MOV r15,r14    --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

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)

```
;void f1(int a){
;         f2(a);}
Func1   LDR r0,[r13]            ; load arg a 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)

```
; void f2 (int r){
;       int g;
;       g = r+5; }
Func2   ldr  r4,[r14]       ;get argument r from stack
        add r5,r4,#5        ;r5 = argument g
        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	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

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

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

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)

Summary

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