Arm = Advanced RISC Machines, Ltd.

References:

*Computers as Components, 4th Ed.*, by Marilyn Wolf

*ARM Cortex-M4 User Guide* (link on course web page)

*ARM Architecture Reference Manual* (link on course web page)
Arm instruction set - outline

- Arm versions.
- Arm assembly language.
- Arm programming model.
- Arm memory organization.
- Arm data operations.
- Arm flow of control.
# 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**
  - High security applications

- Earlier classic processors including Arm7, Arm9, Arm11 families

---

**Cortex-A**
- Cortex-A73
- Cortex-A72
- Cortex-A57
- Cortex-A53
- Cortex-A35
- Cortex-A32
- Cortex-A17
- Cortex-A15
- Cortex-A9
- Cortex-A8
- Cortex-A7
- Cortex-A5

**Cortex-R**
- Cortex-R8
- Cortex-R7
- Cortex-R5
- Cortex-R4

**Cortex-M**
- Cortex-M23, M33
- Cortex-M7
- Cortex-M4
- Cortex-M3
- Cortex-M0, M0+

**SecurCore**
- SC000
- SC300

**Classic**
- Arm11
- Arm9
- Arm7
Equipment Adopting Arm Cores

- Tele-parking
- Intelligent toys
- Utility Meters
- IR Fire Detector
- Exercise Machines
- Energy Efficient Appliances
- Intelligent Vending
- MR A

Source: Arm University Program Overview
Arm processors vs. Arm architectures

- **Arm architecture**
  - Describes the details of instruction set, programmer’s model, exception model, and memory map

- **Arm processor**
  - Developed using one of the Arm architectures
  - More implementation details, such as timing information

---

<table>
<thead>
<tr>
<th>Armv4/v4T Architecture</th>
<th>Armv5/ v4E Architecture</th>
<th>Armv6 Architecture</th>
<th>Armv7 Architecture</th>
<th>Armv8 Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Arm7TDMI</td>
<td>e.g. Arm9926EJ-S</td>
<td>Armv6-M e.g. Cortex-M0, M1</td>
<td>Armv7-A e.g. Cortex-A9</td>
<td>Armv8-A e.g. Cortex-A53</td>
</tr>
<tr>
<td>e.g. Armv5/ v4E</td>
<td></td>
<td></td>
<td>Armv7-R e.g. Cortex-R4</td>
<td>Cortex-A57 Armv8-R</td>
</tr>
<tr>
<td>Architecture</td>
<td></td>
<td></td>
<td></td>
<td>Armv8-M, e.g. Cortex-M23, M33</td>
</tr>
<tr>
<td>e.g. Arm1136</td>
<td></td>
<td></td>
<td>Armv7-M e.g. Cortex-M4</td>
<td></td>
</tr>
</tbody>
</table>
Arm Architecture versions

(From Arm.com)
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** (Thumb mode instructions)
  - 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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Arm Architecture</th>
<th>Core Architecture</th>
<th>Thumb®</th>
<th>Thumb®-2</th>
<th>Hardware Multiply</th>
<th>Hardware Divide</th>
<th>Saturated Math</th>
<th>DSP Extensions</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-M0</td>
<td>Armv6-M</td>
<td>Von Neumann</td>
<td>Most</td>
<td>Subset</td>
<td>1 or 32 cycle</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M0+</td>
<td>Armv6-M</td>
<td>Von Neumann</td>
<td>Most</td>
<td>Subset</td>
<td>1 or 32 cycle</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M3</td>
<td>Armv7-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M4</td>
<td>Armv7E-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Optional</td>
</tr>
<tr>
<td>Cortex-M7</td>
<td>Armv7E-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Optional</td>
</tr>
<tr>
<td>Cortex-M23, 33</td>
<td>Armv8-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Optional</td>
</tr>
</tbody>
</table>
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

  destination   source/left    source/right
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
Arm Register Set

Current Visible Registers

Abort Mode
- r0
- r1
- r2
- r3
- r4
- r5
- r6
- r7
- r8
- r9
- r10
- r11
- r12
- r13 (sp)
- r14 (lr)
- r15 (pc)

(16 32-bit general-purpose registers)

Banked out Registers
(change during exceptions)

User
- r8
- r9
- r10
- r11
- r12

FIQ
- r13 (sp)
- r14 (lr)

IRQ
- r13 (sp)
- r14 (lr)

SVC
- r13 (sp)
- r14 (lr)

Undef
- r13 (sp)
- r14 (lr)

ELEC 5260/6260/6266 Embedded Systems
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.
CPSR
Current Processor Status Register

Must be in a “privileged” mode to change the CPSR

MRS  rn,CPSR
MSR  CPSR,rn

**2 modes in Cortex: Thread & Handler
Arm status bits

- Every arithmetic, logical, or shifting operation can set CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow)
- Examples:
  - $-1 + 1 = 0$: \(\text{NZCV} = 0110\).
  - $2^{31}-1+1 = -2^{31}$: \(\text{NZCV} = 1001\).

- Setting status bits must be explicitly enabled on each instruction
  - ex. “adds” sets status bits, whereas “add” does not
Arm data instructions

• Basic format:
  \texttt{ADD r0,r1,r2}
  \hspace{1cm}Computes r1+r2, stores in r0.

• Immediate operand: (8-bit constant – can be scaled by $2^k$
  \texttt{ADD r0,r1,#2}
  \hspace{1cm}Computes r1+2, stores in r0.

• Set condition flags based on operation:
  \texttt{ADDS r0,r1,r2}

\textcolor{red}{set status flags}

• Assembler translation:
  \texttt{ADD r1,r2} $\Rightarrow$ \texttt{ADD r1,r1,r2} \hspace{1cm} (but not MUL)
Flexible 2\textsuperscript{nd} operand

- 2\textsuperscript{nd} operand = constant or register
- Constant with optional shift: (#8bit_value)
  - Assembler/Compiler turns constant into one of:
    - 8-bit value, shifted left any #bits (up to 32)
    - 0x00ab00ab, 0xab00ab00, 0xabababab (a,b hex digits)
    - Assembler error if constant cannot be represented as above
- 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 2\textsuperscript{nd} 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)
  \[ [Rd] \leq Op1 + Op2 + C \]
- **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 \([Rd] \leq Op1 \oplus Op2\)
- **LSL, LSR**: logical shift left/right (combine with data op’s)
  
  \[ADD r1, r2, r3, \text{ LSL } #4 : \; [r1] \leq r2 + (r3 \times 16)\]

  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

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

- **CMP**: compare : $\text{Op1} - \text{Op2}$
- **CMN**: negated compare : $\text{Op1} + \text{Op2}$
- **TST**: bit-wise AND : $\text{Op1} \ ^\wedge \text{Op2}$
- **TEQ**: bit-wise XOR : $\text{Op1} \ \text{xor} \ \text{Op2}$
New Thumb2 bit operations

- Bit field insert/clear (to pack/unpack data within a register)
  
  \[
  BFC \ r0,\#5,\#4 \ ; \text{Clear 4 bits of } r0, \text{starting with bit } \#5
  \]
  
  \[
  BFI \ r0,r1,\#5,\#4 \ ; \text{Insert 4 bits of } r1 \text{ into } r0, \text{start at bit } \#5
  \]

- Bit reversal (REV) – reverse order of bits within a register
  
  - Bit \([n]\) moved to bit \([31-n]\), for \(n = 0..31\)
  
  - Example:
    
    \[
    REV \ r0,r1 \ ; \text{reverse order of bits in } r1 \text{ and put in } r0
    \]
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 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 32-bit load pseudo-op*

- Operand cannot be memory address or large constant
- LDR r3,=0x55555555
  - Place 0x55555555 in r3
  - Produces MOV if immediate constant can be found
  - Otherwise put constant in a “literal pool” and use:
    LDR r3,[PC,#immediate-12]
    …..
    DCD 0x55555555 ; in literal pool following code

* Not an actual Arm instruction – translated to Arm ops by the assembler
Arm memory access 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: \texttt{LDR r0,[r1]}
  - with second register: \texttt{LDR r0,[r1,-r2]}
  - with constant: \texttt{LDR r0,[r1,#4]}
  - pre-indexed: \texttt{LDR r0,[r1,#4]!}
  - post-indexed: \texttt{LDR r0,[r1],#8}

Immediate \#offset = 12 bits (2’s complement)
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,LSL #2]` ; address = (r2)+(r3 x 4)

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:**
  \[ \text{LDR } r0, [r1, #16] \]
  - Loading from location \([r1+16]\)

- **Auto-indexing increments base register:**
  \[ \text{LDR } r0, [r1, #16]! \]
  - Loading from location \([r1+16]\), then sets \(r1 = r1 + 16\)

- **Post-indexing fetches, then does offset:**
  \[ \text{LDR } r0, [r1], #16 \]
  - Loads \(r0\) from \([r1]\), then sets \(r1 = r1 + 16\)

- **Recent assembler addition:**
  \[ \text{SWP} \{\text{cond}\} \ rd, rm, [rn] \]
  : swap mem & reg
  \[ M[rn] \rightarrow rd, \ rd \rightarrow M[rn] \]
Arm ADR pseudo-op

- Assembler will try to translate:
  
  LDR Rd, label to LDR Rd, [pc, #offset]

- If address in Code Area, generate address value by performing arithmetic on PC.

- ADR pseudo-op generates instruction required to calculate address (in Code Area ONLY)
  
  ADR r1, LABEL

  (uses MOV, MOVN, ADD, SUB op’s)
Example: C assignments

- **C:** \( x = (a + b) - c; \)
- **Assembler:**
  
  ```assembly
  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**: \( y = a \times (b + c) \);
- **Assembler**:
  
  ```asm
  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**: \( z = (a << 2) \mid (b \& 15); \)
- **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:
  \[ B \text{ label} \]
  Target \(< \pm32M(\text{Arm}), \pm2K(\text{Thumb}), \pm16M(\text{Thumb2})\)

- Conditional branch:
  \[ \text{BNE label} \]
  Target \(< \pm32M(\text{Arm}),-252..+258(T), \pm1M(T2)\)

- Thumb2 additions (compare & branch if zero/nonzero):
  \[ \text{CBZ } r0,\text{label} ; \text{branch if } r0 == 0 \]
  \[ \text{CBNZ } r0,\text{label} ; \text{branch if } r0 != 0 \]
Example: if statement

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

- Assembler:
  
  ```assembly
  ; 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
fbblock 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

\(\textbf{(Arm mode} \text{ only – not available in Thumb/Thumb 2 mode)}\)

\[
\begin{align*}
\text{CMP} & \text{ r0,r1} \\
&; \text{true block} \\
\text{MOVLT} & \text{ r0,#5} \; \text{; generate value for x} \\
\text{ADRLT} & \text{ r4,x} \; \text{; get address for x} \\
\text{STRLT} & \text{ r0,[r4]} \; \text{; store x} \\
\text{ADRLT} & \text{ r4,c} \; \text{; get address for c} \\
\text{LDRLT} & \text{ r0,[r4]} \; \text{; get value of c} \\
\text{ADRLT} & \text{ r4,d} \; \text{; get address for d} \\
\text{LDRLT} & \text{ r1,[r4]} \; \text{; get value of d} \\
\text{ADDLT} & \text{ r0,r0,r1} \; \text{; compute y} \\
\text{ADRLT} & \text{ r4,y} \; \text{; get address for y} \\
\text{STRLT} & \text{ r0,[r4]} \; \text{; store y}
\end{align*}
\]
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

Pseudo-C

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

Thumb2 code

```asm
    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, r5 ; do if r0 <= r1
```
Example: C switch statement

- **C:**
  ```c
  switch (test) { case 0: ... break; case 1: ... }
  ```

- **Assembler:**
  ```assembly
  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 casel
  ... 
  ```
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
casel instructions

(TBH similar, but with 16-bit offsets/DCI)
Finite impulse response (FIR) filter

\[ f = \sum_{1 \leq i \leq n} c_i x_i \]

\(X_i\)'s are data samples
\(C_i\)'s are constants
Example: FIR filter

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

- Assembler
  ; loop initiation code
  ```assembly
  MOV r0,#0          ; use r0 for I
  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,r0,LSL #2] ; get c[i]
  LDR r6,[r5,r0,LSL #2] ; get x[i]
  MUL r4,r4,r6 ; compute c[i]*x[i]
  ADD r2,r2,r4 ; add into running sum f
  ADD r0,r0,#1 ; add 1 to i
  CMP r0,r1 ; exit?
  BLT loop ; if i < N, continue

; Finalize result
  LDR r3,=f ; point to f
  STR r2,[r3] ; f = result
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
    ldr    r2,=f        ;point to f
    str    r0,[r2]      ;f = result
here   b     here

AREA MyData, DATA
    carray dcd  1,2,3
    xarray dcd  10,20,30
    f     space   4
END

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

• Branch and link instruction:
  \textbf{BL} \texttt{foo} \texttt{;} \textit{Copies current PC to r14.}

• To return from subroutine:
  \textbf{BX} \texttt{r14} \texttt{;} \textit{branch to address in r14}
  or:
  \texttt{MOV r15,r14} \texttt{--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:

```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
```

(Omit if using Cortex-M startup code)
Nested subroutine calls (2)

```c
; 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 return address
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 stack and return
```
Nested subroutine calls (3)

; void f2 (int r) {
; int g;
; g = r+5;
}

Func2  ldr  r4,[r13] ; 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

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

* Alternate register designation
  a1-a4 : argument/result/scratch
  v1-v8: variables

ELEC 5260/6260/6266 Embedded Systems
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\}
uP startup: `startup_stm32l476.s`

- **Stack definition:**
  ```
  Stack_Size EQU 0x400;
  AREA STACK, NOINIT, READWRITE, ALIGN=3
  Stack_Mem SPACE Stack_Size
  __initial_sp
  ```

- **Vector table:**
  ```
  AREA RESET, DATA, READONLY
  __Vectors DCD __initial_sp ; Top of Stack
  DCD Reset_Handler ; Reset Handler
  DCD NMI_Handler ; NMI Handler
  ```

- **Reset handler:**
  ```
  Reset_Handler PROC
  EXPORT Reset_Handler [WEAK]
  IMPORT SystemInit
  IMPORT __main
  LDR R0, =SystemInit
  BLX R0
  LDR R0, =__main
  BX R0
  ```
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 1 to 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.
Arm Instruction Code Format

cond 00 X opcode S | Rn Rd | Format determined by X bit

<table>
<thead>
<tr>
<th>31 28 25 24 21 20 19 16 15 12 11 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond 00 X opcode S Rn Rd</td>
</tr>
</tbody>
</table>

- condition for execution
- force update of CPSR
- source reg
dest reg

X = 0:
- # shifts
- shift
- 0
- Rm

X = 1:
- alignment
- 8-bit literal

X = 0:
- 3rd operand is Rm

X = 1:
- 3rd operand is immediate

3rd operand

scale factor
### Arm Load/Store Code Format

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>condition</td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>I</td>
</tr>
<tr>
<td>24</td>
<td>P</td>
</tr>
<tr>
<td>23</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>W</td>
</tr>
<tr>
<td>20</td>
<td>L</td>
</tr>
<tr>
<td>19</td>
<td>Rn</td>
</tr>
<tr>
<td>16</td>
<td>Rd</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

- **cond**: Condition for execution
- **I**: Indicates whether the operation is a load or store
- **P**: Pre-indexed
- **U**: Update base register
- **B**: Base register
- **W**: Word or u-byte operation
- **L**: Indexed or post-indexed
- **Rn**: Source register
- **Rd**: Destination register

#### Offset Calculation

- **I = 0:**
  - # shifts: 11 bits
  - shift: 6 bits
  - 0: 1 bit
  - Rm: 4 bits
  - Offset is Rm

- **i = 1:**
  - 12-bit offset
  - Offset is immediate