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
ARM instruction set architecture

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

(From arm.com)
History

- ARM9 – includes “Thumb” instruction set
- ARM10 – for multimedia (graphics, video, etc.)
- ARM11 – high performance + Jazelle (Java)
- SecurCore – for security app’s (smart cards)
- Cortex-M – Optimized for microcontrollers
- Cortex-A - High performance (multimedia systems)
- Cortex-R – Optimized for real-time app’s
Instruction Sets

- Thumb-2
  (16-bit and 32-bit instructions)
- Cortex-M4
- Cortex-M3
- Cortex-M0
- Thumb
  (16-bit Instructions)
- ARM
  (32-bit Instructions)
ARM Cortex-M instruction sets

Cortex-M0/M1

Cortex-M3

Cortex-M4

Cortex-M4 FPU
Programmer’s model of a CPU

- What information is specified in an “instruction” to accomplish a task?
  - **Operations**: add, subtract, move, jump
  - **Operands**: data manipulated by operations
    - # of operands per instruction (1-2-3)
  - **Data sizes & types**
    - # bits (1, 8, 16, 32, …)
    - signed/unsigned integer, floating-point, character …
  - **Locations of operands**
    - **Memory** – specify location by a memory “address”
    - **CPU Registers** – specify register name/number
    - **Immediate** – data embedded in the instruction code
    - Input/output device “ports”/interfaces
RISC vs. CISC architectures

- **CISC = “Complex Instruction Set Computer”**
  - Rich set of instructions and options to minimize #operations required to perform a given task
  - Example: Intel x86 instruction set architecture

- **RISC = “Reduced Instruction Set Computer”**
  - Fixed instruction length
  - Fewer/simpler instructions than CISC CPU 32-bit load/store architecture
  - Limited addressing modes, operand types
  - Simple design easier to speed up, pipeline & scale
  - Example: ARM architecture

Program execution time =
\[(\# \text{ instructions}) \times (\# \text{ clock cycles/instruction}) \times (\text{clock period})\]
ARM instruction format

Add instruction:  
ADD  R1, R2, R3 ;2nd source operand = register
ADD  R1, R2, #5 ;2nd source operand = constant

1. operation:   binary addition (compute  \( R1 = R2 + 5 \))
2. destination: register \( R1 \)  (replaces original contents of \( R1 \))
3. left-hand operand:   register \( R2 \)
4. right-hand operand:
    Option 1:  register \( R3 \)
    Option 2:  constant 5 (# indicates constant)

operand size: 32 bits  (all arithmetic/logical instructions)
operand type: signed or unsigned integer
ARM assembly language

- Fairly standard assembly language format:

```
LDR r0, [r8] ; a comment

label ADD r4, r0, r1 ; r4 = r0 + r1
```

- `Label` (optional) refers to the location of this instruction
- `Memory address/pointer`
- `Destination`    `Source/left`    `Source/right`

label (optional) refers to the location of this instruction
Processor core registers

- All registers are 32 bits wide
- 13 general purpose registers
  - Registers r0 – r7 (Low registers)
  - Registers r8 – r12 (High registers)
  - Use to hold data, addresses, etc.
- 3 registers with special meaning/usage
  - Stack Pointer (SP) – r13
  - Link Register (LR) – r14
  - Program Counter (PC) – r15
- xPSR – Program Status Register
  - Composite of three PSRs
  - Includes ALU flags (N,Z,C,V)
Program status register (PSR)

- Program Status Register $xPSR$ is a composite of 3 PSRs:
  - **APSR** - Application Program Status Register – ALU condition flags
    - N (negative), Z (zero), C (carry/borrow), V (2’s complement overflow)
    - Flags set by ALU operations; tested by conditional jumps/execution
  - **IPSR** - Interrupt Program Status Register
    - Interrupt/Exception No.
  - **EPSR** - Execution Program Status Register
    - T bit = 1 if CPU in “Thumb mode” *(always for Cortex-M4)*, 0 in “ARM mode”
    - IT field – If/Then block information
    - ICI field – Interruptible-Continuuable Instruction information
- $xPSR$ stored on the stack on exception entry
Data types supported in ARM

- Integer ALU operations are performed **only on 32-bit data**
  - Signed or unsigned integers
- Data sizes in memory:
  - Byte (8-bit), Half-word (16-bit), Word (32-bit), Double Word (64-bit)
- Bytes/half-words are converted to 32-bits when moved into a register
  - Signed numbers – extend sign bit to upper bits of a 32-bit register
  - Unsigned numbers – fill upper bits of a 32-bit register with 0’s
  - **Examples:**
    - 255 (unsigned byte) 0xFF => 0x000000FF (fill upper 24 bits with 0)
    - -1 (signed byte) 0xFF => 0xFFFFFFFF (fill upper 24 bits with sign bit 1)
    - +1 (signed byte) 0x01 => 0x00000001 (fill upper 24 bits with sign bit 0)
    - -32768 (signed half-word) 0x8000 => 0xFFFF8000 (sign bit = 1)
    - 32768 (unsigned half-word) 0x8000 => 0x00008000
    - +32767 (signed half-word) 0x7FFF => 0x00007FFF (sign bit = 0)
- Cortex-M4F supports single and double-precision IEEE floating-point data
  (Floating-point ALU is **optional** in Cortex-M4 implementations)
C/C++ language data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bits)</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>[-2^7 .. +2^7–1] = [-128 .. +127]</td>
</tr>
<tr>
<td>signed char</td>
<td></td>
<td>Compiler-specific (not specified in C standard)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARM compiler default is signed</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>[0 .. 2^8–1] = [0..255]</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>[-2^{15} .. +2^{15}–1]</td>
</tr>
<tr>
<td>signed short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned short</td>
<td></td>
<td>[0 .. 2^{16}–1]</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>[-2^{31} .. +2^{31}–1] (natural size of host CPU)</td>
</tr>
<tr>
<td>signed int</td>
<td></td>
<td>int specified as signed as signed in the C standard</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>[0 .. 2^{32}–1]</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>[-2^{31} .. +2^{31}–1]</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>[-2^{63} .. +2^{63}–1]</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>IEEE single-precision floating-point format</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>IEEE double-precision floating-point format</td>
</tr>
</tbody>
</table>
Directive: Data Allocation

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Memory Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCB</td>
<td>Define Constant Byte</td>
<td>Reserve 8-bit values</td>
</tr>
<tr>
<td>DCW</td>
<td>Define Constant Half-word</td>
<td>Reserve 16-bit values</td>
</tr>
<tr>
<td>DCD</td>
<td>Define Constant Word</td>
<td>Reserve 32-bit values</td>
</tr>
<tr>
<td>DCQ</td>
<td>Define Constant</td>
<td>Reserve 64-bit values</td>
</tr>
<tr>
<td>SPACE</td>
<td>Defined Zeroed Bytes</td>
<td>Reserve a number of zeroed bytes</td>
</tr>
<tr>
<td>FILL</td>
<td>Defined Initialized Bytes</td>
<td>Reserve and fill each byte with a value</td>
</tr>
</tbody>
</table>

**DCx**: reserve space and initialize value(s) for ROM

*(initial values ignored for RAM)*

**SPACE**: reserve space without assigning initial values

*(especially useful for RAM)*
Directive: Data Allocation

AREA myData, DATA, READWRITE

hello DCB "Hello World!",0 ; Allocate a string that is null-terminated

dollar DCB 2,10,0,200 ; Allocate integers ranging from -128 to 255

scores DCD 2,3.5,-0.8,4.0 ; Allocate 4 words containing decimal values

miles DCW 100,200,50,0 ; Allocate integers between -32768 and 65535

p SPACE 255 ; Allocate 255 bytes of zeroed memory space

f FILL 20,0xFF,1 ; Allocate 20 bytes and set each byte to 0xFF

binary DCB 2_01010101 ; Allocate a byte in binary

octal DCB 8_73 ; Allocate a byte in octal

char DCB ‘A’ ; Allocate a byte initialized to ASCII of ‘A’
Memory usage

- **Code memory** (normally read-only memory)
  - Program instructions
  - Constant data

- **Data memory** (normally read/write memory – RAM)
  - Variable data/operands

- **Stack** (located in data memory)
  - Special Last-In/First-Out (LIFO) data structure
    - Save information temporarily and retrieve it later
    - Return addresses for subroutines and interrupt/exception handlers
    - Data to be passed to/from a subroutine/function
  - Stack Pointer register (r13/sp) points to last item placed on the stack

- **Peripheral addresses**
  - Used to access registers in “peripheral functions” (timers, ADCs, communication modules, etc.) **outside** the CPU
Cortex-M4 processor memory map

**Cortex** peripheral function registers (NVIC, tick timer, etc.)

**STM32F407 microcontroller:**

Peripheral function registers

SRAM1 (128Kbyte):
- [0x2000_0000 .. 0x2001_FFFF]

SRAM2 (64Kbyte):
- [0x1000_0000 .. 0x1000_FFFF]

Flash memory (1MByte):
- [0x0800_0000 .. 0x0800F_FFFF]

We will use Flash for code, SRAM1 for data.

All ARM addresses are 32 bits
Endianness

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

<table>
<thead>
<tr>
<th>little-endian (default)</th>
<th>big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit 31</td>
<td>bit 0</td>
</tr>
<tr>
<td>bit 0</td>
<td>bit 0</td>
</tr>
<tr>
<td>bit 0</td>
<td>bit 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byte 3</th>
<th>byte 2</th>
<th>byte 1</th>
<th>byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x78</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12</td>
<td>0x34</td>
<td>0x56</td>
<td>0x78</td>
</tr>
</tbody>
</table>

**Example:**
32-bit data = 0x12345678

<table>
<thead>
<tr>
<th>Address: 100</th>
<th>0x78</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0x56</td>
</tr>
<tr>
<td>102</td>
<td>0x34</td>
</tr>
<tr>
<td>103</td>
<td>0x12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address: 100</th>
<th>0x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0x34</td>
</tr>
<tr>
<td>102</td>
<td>0x56</td>
</tr>
<tr>
<td>103</td>
<td>0x78</td>
</tr>
</tbody>
</table>
Physical memory organization

- Physical memory may be organized as N bytes per addressable word
  - ARM memories normally 4-bytes wide
  - “Align” 32-bit data to a Word boundary (address that is a multiple of 4)
    - All bytes of a word must be accessible with one memory read/write

<table>
<thead>
<tr>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>102</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>107</td>
<td>106</td>
<td>105</td>
<td>104</td>
</tr>
<tr>
<td>10B</td>
<td>10A</td>
<td>109</td>
<td>108</td>
</tr>
<tr>
<td>10F</td>
<td>10E</td>
<td>10D</td>
<td>10C</td>
</tr>
</tbody>
</table>

Word 100
Word 104
Word 108
Word 10C

ARM instructions can read/write 8/16/32-bit data values
AREA string_copy, CODE, READONLY
EXPORT __main
ALIGN ENTRY
__main PROC

strcpy LDR r1, =srcStr ; Retrieve address of the source string
LDR r0, =dstStr ; Retrieve address of the destination string
loop LDRB r2, [r1], #1 ; Load a byte & increase src address pointer
STRB r2, [r0], #1 ; Store a byte & increase dst address pointer
CMP r2, #0 ; Check for the null terminator
BNE loop ; Copy the next byte if string is not ended
stop B stop ; Dead loop. Embedded program never exits.

ENDP

AREA myData, DATA, READWRITE
ALIGN

srcStr DCB "The source string.",0 ; Strings are null terminated
dstStr DCB "The destination string.",0 ; dststr has more space than srcstr

END
The **AREA** directive indicates to the assembler the start of a new data or code section.

- Areas are the basic independent and indivisible unit processed by the linker.
- Each area is identified by a name and areas within the same source file cannot share the same name.
- An assembly program must have at least one code area.
- By default, a code area can only be read and a data area may be read from and written to.

---

<table>
<thead>
<tr>
<th>Array</th>
<th>AREA myData, DATA, READWRITE ; Define a data section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
<tr>
<td>__main</td>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
</tr>
<tr>
<td>__main</td>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td>__main</td>
<td>ENTRY ; Mark the entrance to the entire program</td>
</tr>
<tr>
<td>__main</td>
<td>PROC ; PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td>__main</td>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td>__main</td>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td>__main</td>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>
Directive: END

The END directive indicates the end of a source file.

Each assembly program must end with this directive.
### Directive: ENTRY

<table>
<thead>
<tr>
<th>Array</th>
<th>AREA myData, DATA, READWRITE ; Define a data section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
<tr>
<td></td>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
</tr>
<tr>
<td>__main</td>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td>ENTRY</td>
<td>__main PROC ; Mark the entrance to the entire program</td>
</tr>
<tr>
<td></td>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td></td>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td>END</td>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>

- The ENTRY directive marks the first instruction to be executed within an application.
- **There must be one and only one entry directive in an application**, no matter how many source files the application has.
## Directive: PROC and ENDP

<table>
<thead>
<tr>
<th>Array</th>
<th>AREA myData, DATA, READWRITE ; Define a data section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
<tr>
<td></td>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
</tr>
<tr>
<td></td>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td></td>
<td>ENTRY ; Mark the entrance to the entire program</td>
</tr>
<tr>
<td>__main</td>
<td>PROC ; PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td></td>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td></td>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td></td>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>

- PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).
- A single source file can contain multiple subroutines, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a subroutine within another subroutine.
## Directive: EXPORT and IMPORT

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array</strong></td>
<td>AREA myData, DATA, READWRITE ; Define a data section</td>
</tr>
<tr>
<td></td>
<td>DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
<tr>
<td><strong>__main</strong></td>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
</tr>
<tr>
<td></td>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td></td>
<td>IMPORT sinx ; Function sinx defined in another file</td>
</tr>
<tr>
<td></td>
<td>ENTRY ; Mark the entrance to the entire program</td>
</tr>
<tr>
<td></td>
<td>__main PROC ; PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td></td>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td></td>
<td>BL sinx ; Call the sinx function</td>
</tr>
<tr>
<td></td>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td></td>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>

- The EXPORT declares a symbol and makes this symbol visible to the linker.
- The IMPORT gives the assembler a symbol that is not defined locally in the current assembly file.
- The IMPORT is similar to the “extern” keyword in C.
The EQU directive associates a symbolic name to a numeric constant. Similar to the use of `#define` in a C program, the EQU can be used to define a constant in an assembly code.

- The RN directive gives a symbolic name to a specific register.

```assembly
; Interrupt Number Definition (IRQn)
BusFault_IRQn   EQU -11 ; Cortex-M3 Bus Fault Interrupt
SVCall_IRQn     EQU -5 ; Cortex-M3 SV Call Interrupt
PendSV_IRQn     EQU -2 ; Cortex-M3 Pend SV Interrupt
SysTick_IRQn    EQU -1 ; Cortex-M3 System Tick Interrupt
MyConstant      EQU 1234 ; Constant 1234 to use later

Dividend        RN   6 ; Defines dividend for register 6
Divisor         RN   5 ; Defines divisor for register 5
```
Directive: ALIGN

AREA example, CODE, ALIGN = 3 ; Memory address begins at a multiple of 8
ADD r0, r1, r2 ; Instructions start at a multiple of 8

AREA myData, DATA, ALIGN = 2 ; Address starts at a multiple of four
a DCB 0xFF ; The first byte of a 4-byte word
ALIGN 4, 3 ; Align to the last byte of a word
b DCB 0x33 ; Set the fourth byte of a 4-byte word
c DCB 0x44 ; Add a byte to make next data misaligned
ALIGN ; Force the next data to be aligned
d DCD 12345 ; Skip three bytes and store the word
Directive: INCLUDE or GET

- The INCLUDE or GET directive is to include an assembly source file within another source file.
- It is useful to include constant symbols defined by using EQU and stored in a separate source file.

```assembly
INCLUDE constants.s ; Load Constant Definitions
AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
...
ENDP
END
```