ARM Cortex-M4 Programming Model

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.

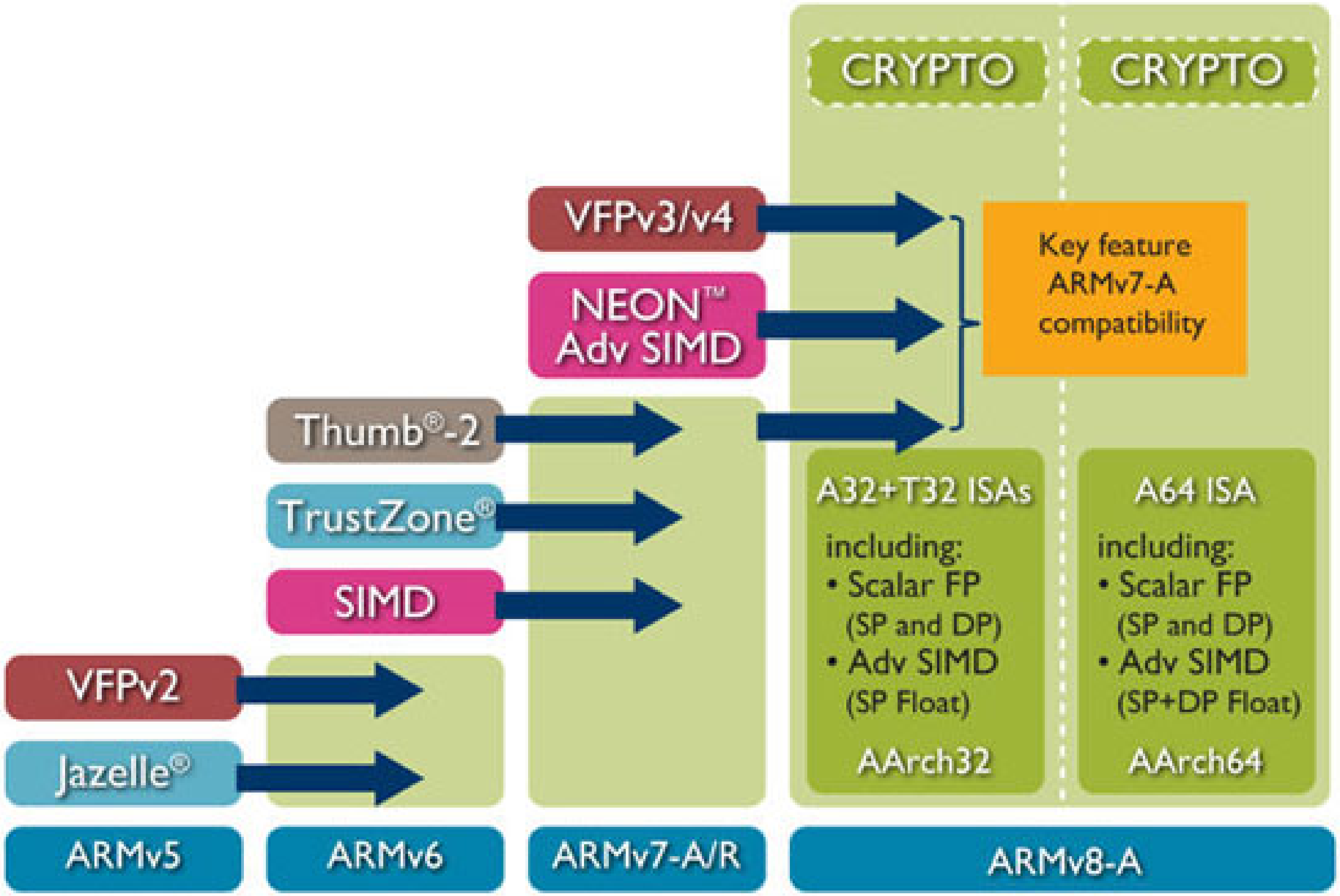
www.arm.com

ARM instruction set architecture

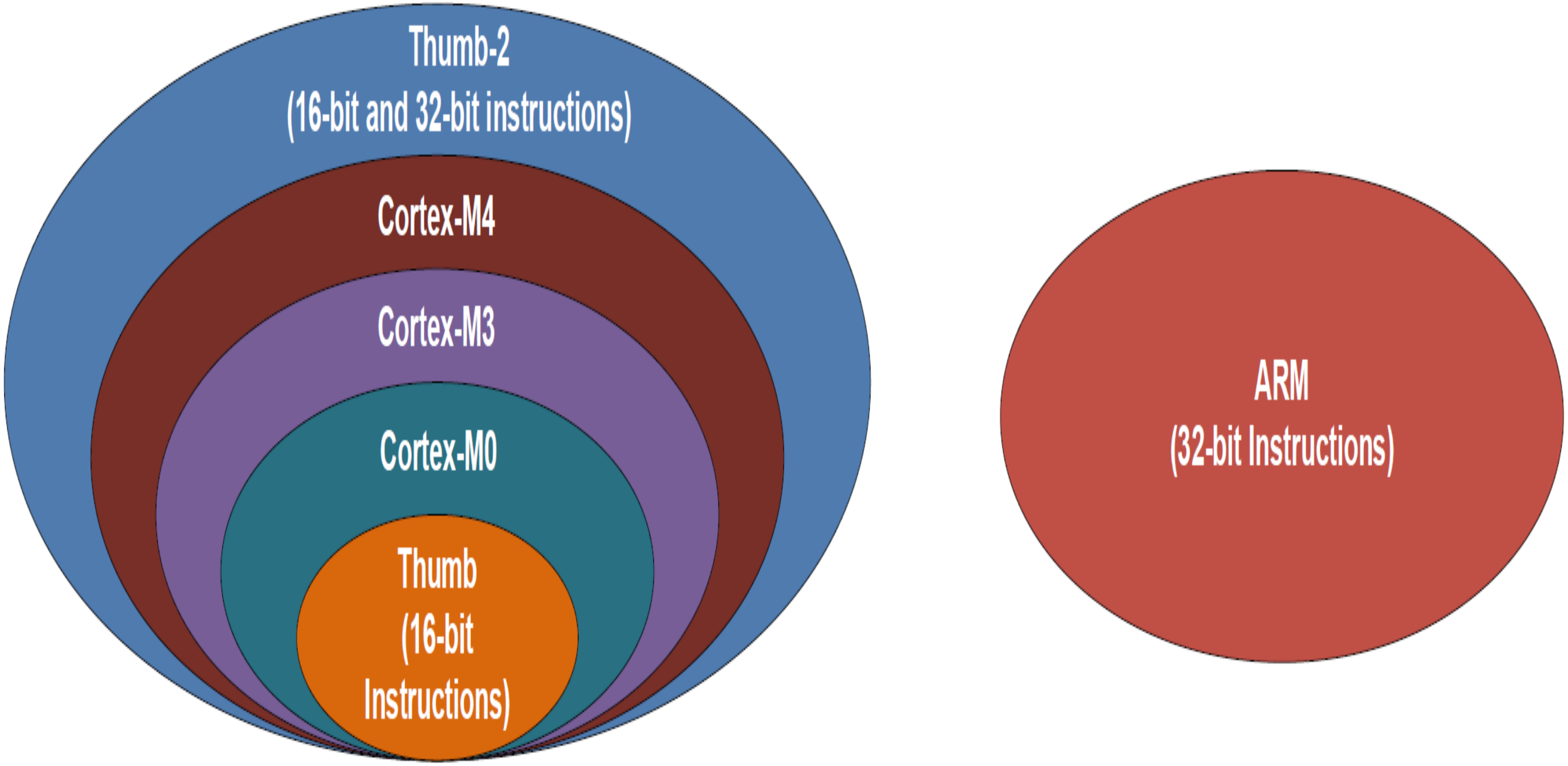
- ARM versions.
- ARM programming model.
- ARM memory organization.
- ARM assembly language.
- ARM data operations.
- ARM flow of control.

ARM Architecture versions

(From arm.com)



Instruction Sets



Arm Processor Families

Cortex-A series (advanced application)

- High-performance processors for open OSs
- App's: smartphones, digital TV, server solutions, and home gateways.

Cortex-R series (real-time)

- Exceptional performance for real-time applications
- App's: automotive braking systems and powertrains.

Cortex-M series (microcontroller)

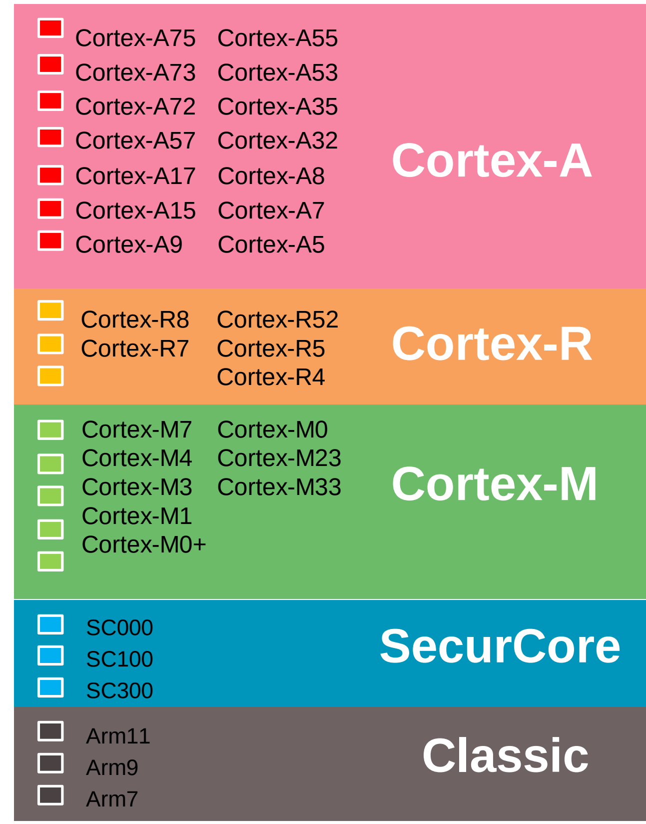
- Cost-sensitive solutions for deterministic microcontroller applications
- App's: microcontrollers, smart sensors, automotive body electronics, and airbags.

SecurCore series

- High-security applications such as smartcards and e-government

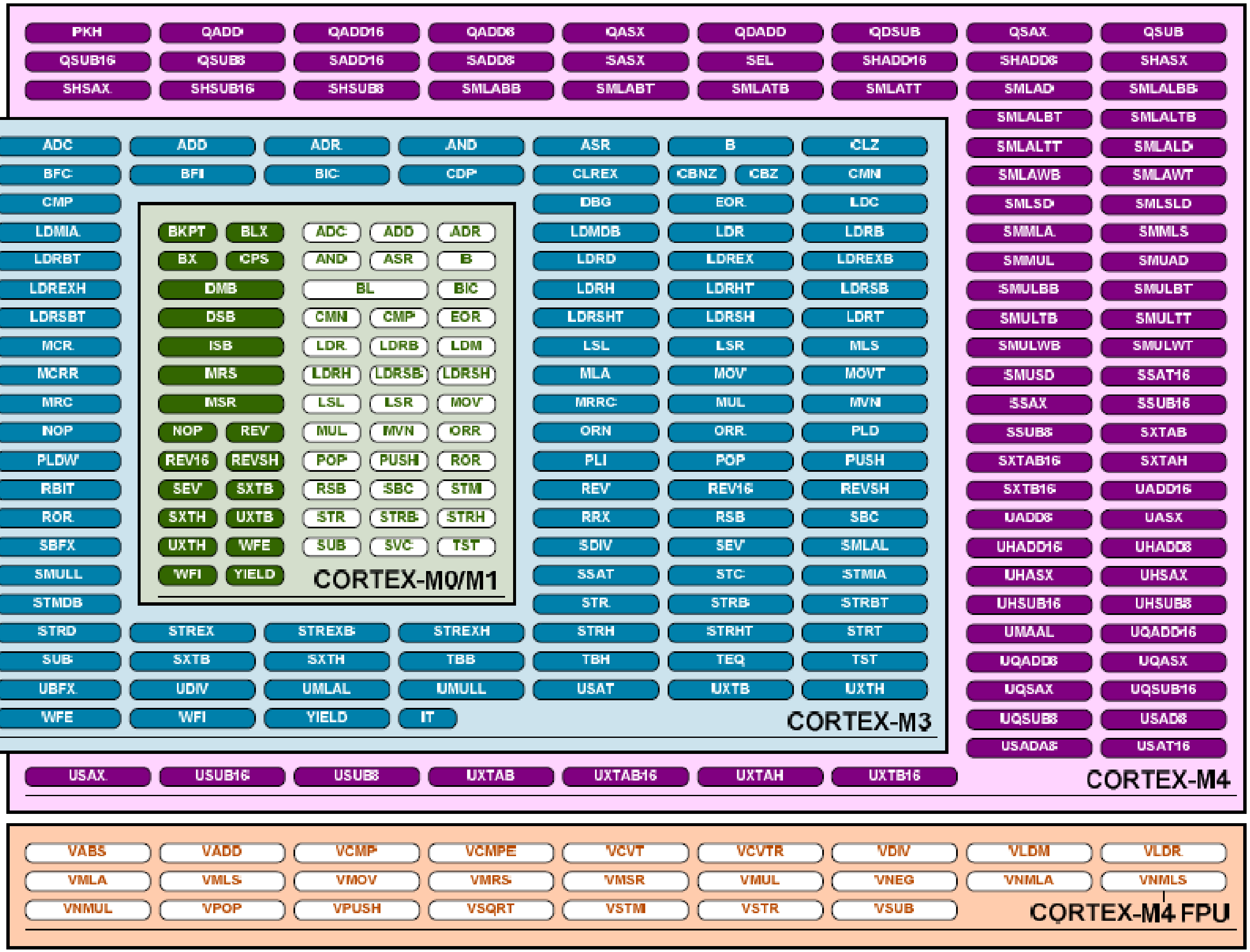
Classic processors

- Include Arm7, Arm9, and Arm11 families



As of Nov 2017

ARM Cortex-M instruction sets



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 =

(# instructions) x (# clock cycles/instruction) x (clock period)

ARM instruction format

Add instruction: ADD R1, R2, R3 ;2^{nd} source operand = register

ADD R1, R2, #5 ;2^{nd} source operand = constant

1 2 3 4

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:

memory address/pointer

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

destination source/left source/right

label (optional) refers to the location of this instruction

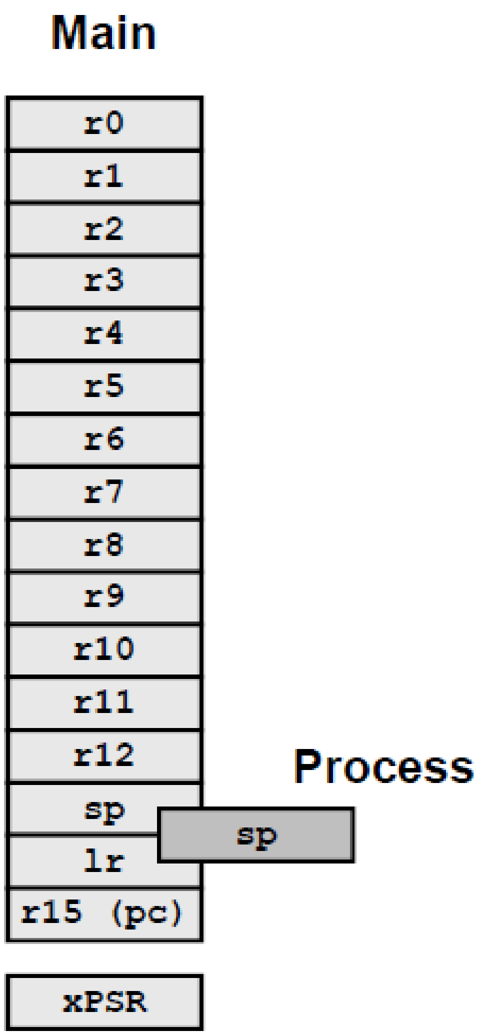
Processor core registers

- All registers are 32 bits wide

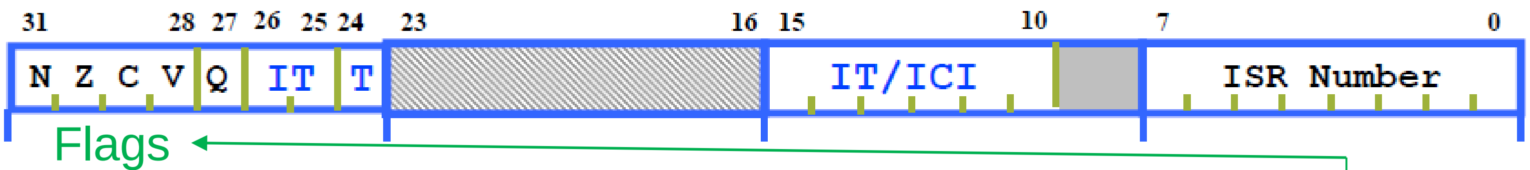
- 13 general purpose registers
  - Registers r0 – r7 (Low registers)
  - Registers r8 – r12 (High registers)
  - Use to hold data, addresses, etc.

- 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-Continuable 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
  - <u>Examples:</u>
    - 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 <u>optional</u> in Cortex-M4 implementations)

C/C++ language data types

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

Directive: Data Allocation

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

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

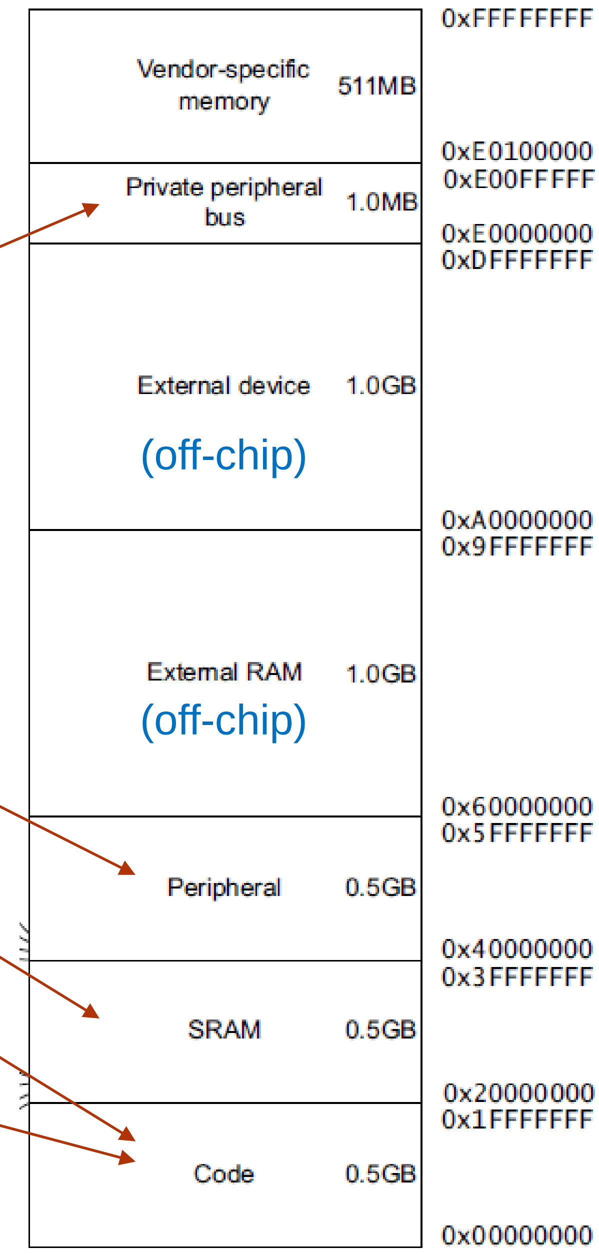
```
 AREA   myData, DATA, READWRITE
hello   DCB   "Hello World!",0  ; Allocate a string that is null-terminated
dollar  DCB   2,10,0,200        ; Allocate integers ranging from -128 to 255
scores  DCD   2,3,-8,4           ; Allocate 4 words containing decimal values
miles   DCW   100,200,50,0      ; 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]

All ARM addresses are 32 bits

We will use Flash for code, SRAM1 for data.

Endianness

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

little-endian (default)

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

big-endian (option)

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

Example:
32-bit data =
0x12345678

Address:	
100	0x78
101	0x56
102	0x34
103	0x12

103	102	101	100
12	34	56	78

Address:	
100	0x12
101	0x34
102	0x56
103	0x78

100	101	102	103
12	34	56	78

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

Byte addresses

Byte 3	Byte 2	Byte 1	Byte 0	
103	102	101	100	Word 100
107	106	105	104	Word 104
10B	10A	109	108	Word 108
10F	10E	10D	10C	Word 10C

ARM instructions can read/write 8/16/32-bit data values

First Assembly

Code Area

```
        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
```

Data Area

```
        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
```

Directive: AREA

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- 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.

Directive: END

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- The END directive indicates the end of a source file.
- Each assembly program must end with this directive.

Directive: ENTRY

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- 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

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- 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

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        IMPORT  sinx                 ; Function sinx defined in another file
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        BL      sinx                 ; Call the sinx function
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- 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.

Directive: EQU

```
; 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
```

- 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.

Example:

```
MOV  R0, #MyConstant    ; Constant 1234 placed in R0
```

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
```

ALIGN generally used as in this example, to align a variable to its data type.

Directive: INCLUDE or GET

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

- 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.