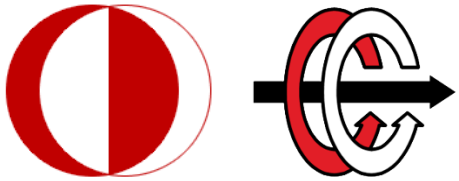
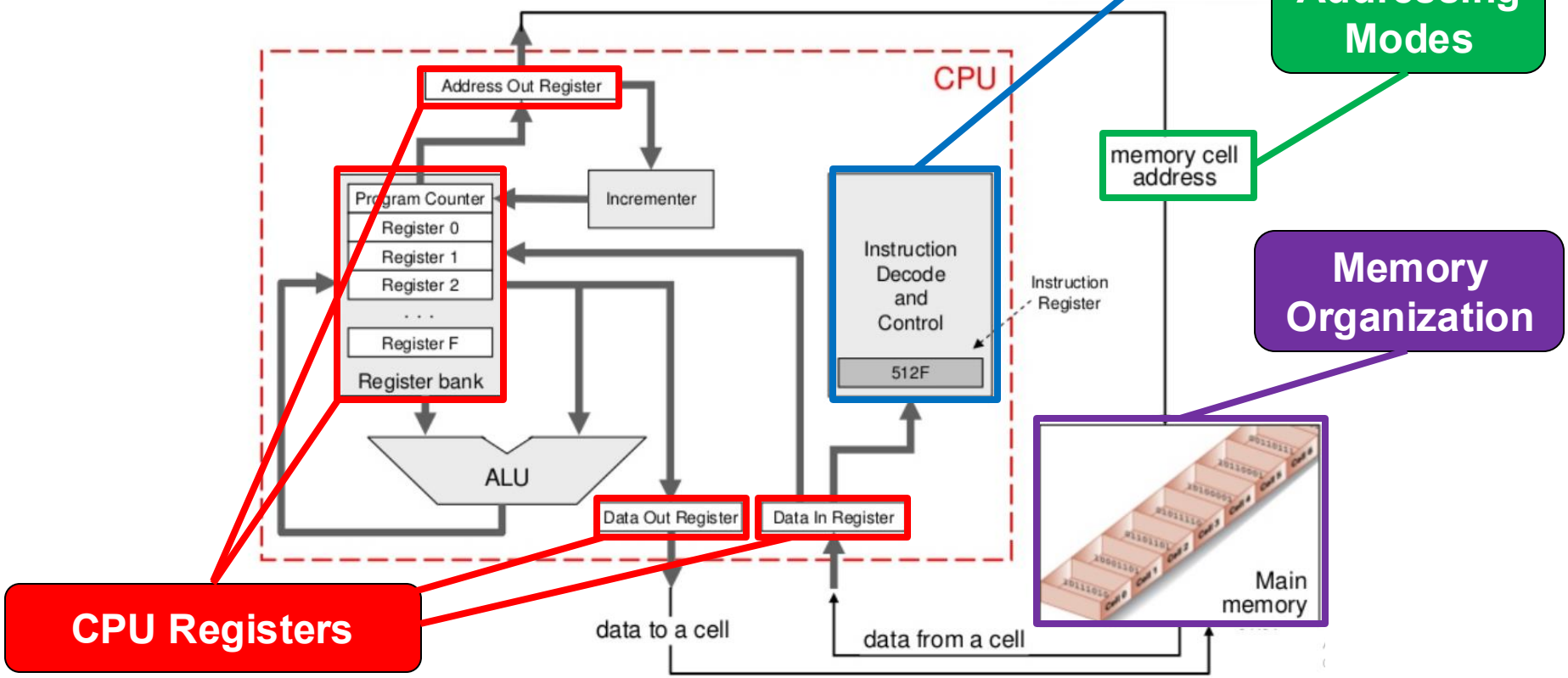


ARM Architecture, Programming Model

Week 2

Programming Model: Overview

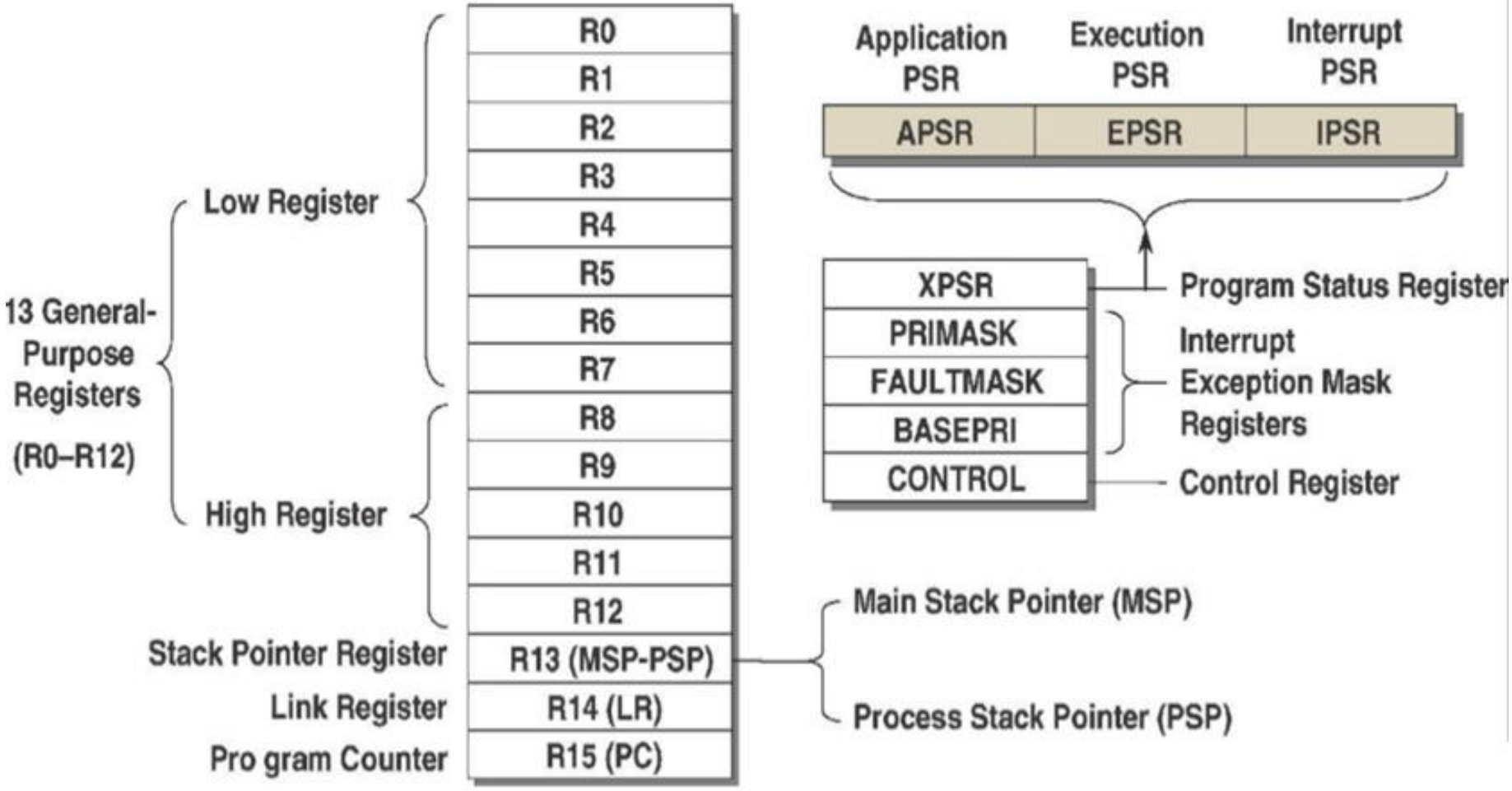
- Required Knowledge for Assembly Programming



Programming model

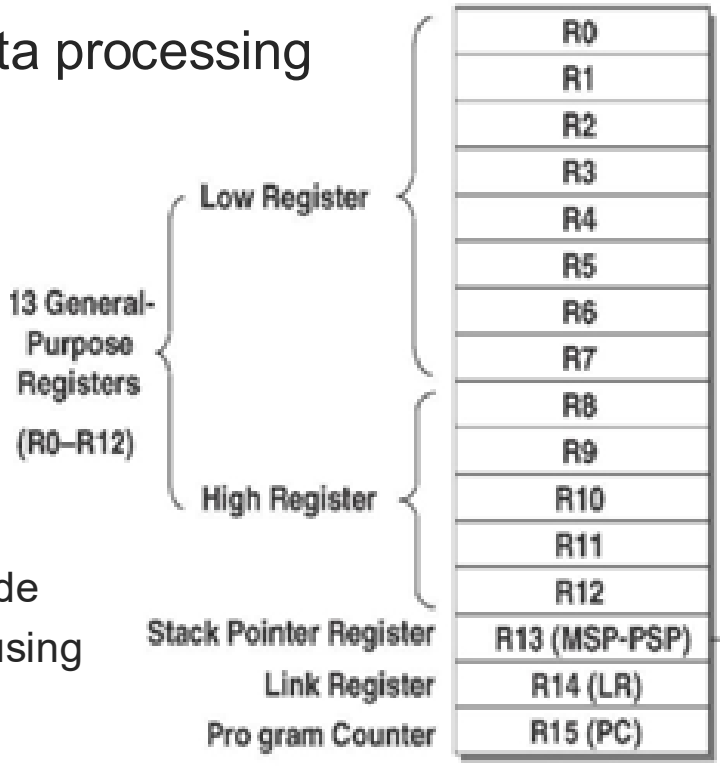
- To be able to program a MCU, assembly language programmer should know in depth the folowing features:
  - CPU registers
  - Instruction set (operations, operands, data sizes and types)
  - Addressing modes
  - Memory organization

ARM Cortex-M3/4 Registers



ARM Cortex-M3/4 Registers: R0 to R15

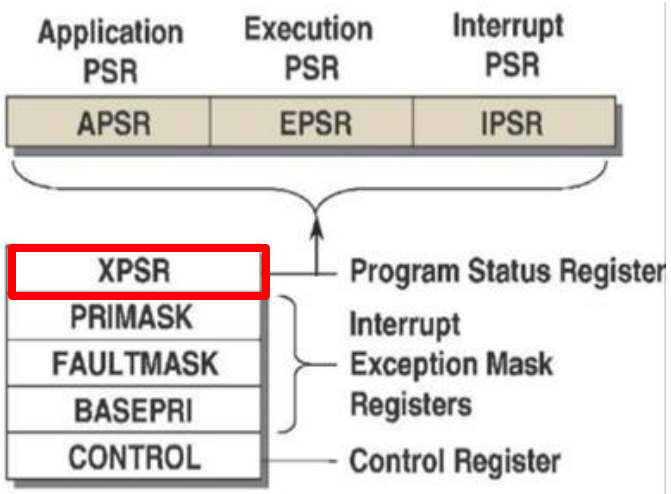
- R0-R12 - General purpose registers for data processing
  - R0-R7 (Low registers) many 16-bit instructions only access these registers
  - R8-R12 (High registers) can be used with 32-bit instructions
- SP - Stack pointer (Banked R13)
  - Can refer to one of two SPs
    - Main Stack Pointer (MSP)
    - Process Stack Pointer (PSP)
  - Uses MSP initially, and whenever in Handler mode
  - In Thread mode, can select either MSP or PSP using CONTROL register
- LR - Link Register (R14)
  - Holds return address when called with Branch & Link instruction (BL)
- PC - Program Counter (R15)



ARM Cortex-M3/4 Registers: PSR

- Program Status Register (PSR) is three views of same register
- Application PSR (APSR)
  - Condition code flag bits Negative, Zero, Carry, Overflow, DSP overflow and saturation, Great-Than or Equal
- Interrupt PSR (IPSR)
  - Holds exception number of currently executing ISR
- Execution PSR (EPSR)
  - ICI/IT, Interruptible-Continuable Instruction, IF-THEN instruction
  - Thumb state, always 1

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE*							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						



ARM Cortex-M3/4 Registers: PSR Details

APSR

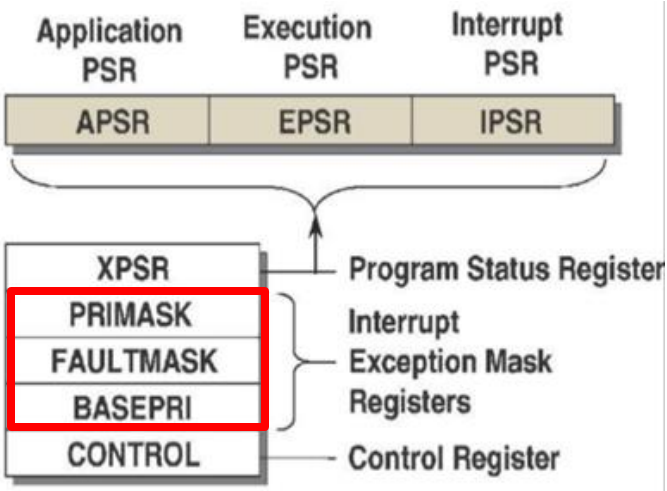
Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	DSP overflow and saturation flag
[26:20]	-	Reserved
[19:16]	GE[3:0]	Greater than or Equal flags.
[15:0]	-	Reserved

IPSR

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. . . .

Interrupt Exception Mask Registers

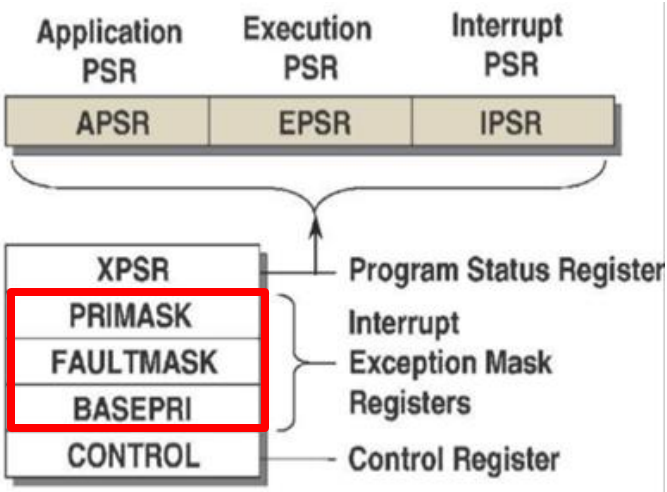
- PRIMASK, FAULTMASK, and BASEPRI
  - Used to mask exceptions based on priority levels
  - Only accessed in the privileged access level
  - By default, all zero, which means the masking (disabling of exception/interrupt) is not active



- PRIMASK Register
  - 1-bit wide interrupt mask register
  - When set, it blocks all exceptions (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception
  - The most common usage for PRIMASK is to disable all interrupts for a time critical process

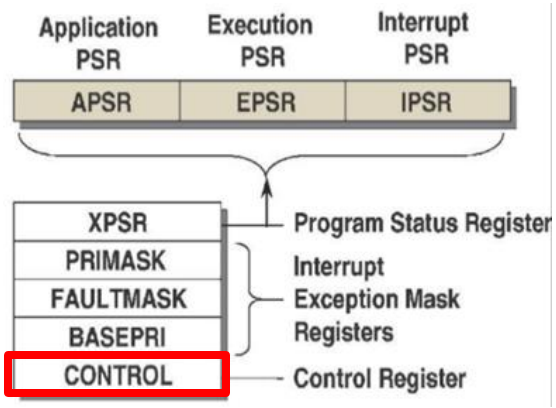
Interrupt Exception Mask Registers

- FAULTMASK Register
  - Very similar to PRIMASK, but it also blocks the HardFault exception
- BASEPRI Register
  - Defines the priority of the executing software
  - It prevents interrupts with lower or equal priority



Control Register

- Controls the stack used and the privilege level for software execution when the processor is in Thread mode
- If implemented, indicates whether the FPU state is active



Bits	Name	Function
[31:3]	-	Reserved.
[2]	FPCA	When floating-point is implemented this bit indicates whether context floating-point is currently active: 0 = no floating-point context active 1 = floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
[1]	SPSEL	Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return: 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = privileged 1 = unprivileged.

Memory Map: Different Types of Memory

- 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
    - Save return addresses for subroutines and interrupt/exception handlers
    - Data to be passed to/from a subroutine/function can be saved
  - 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

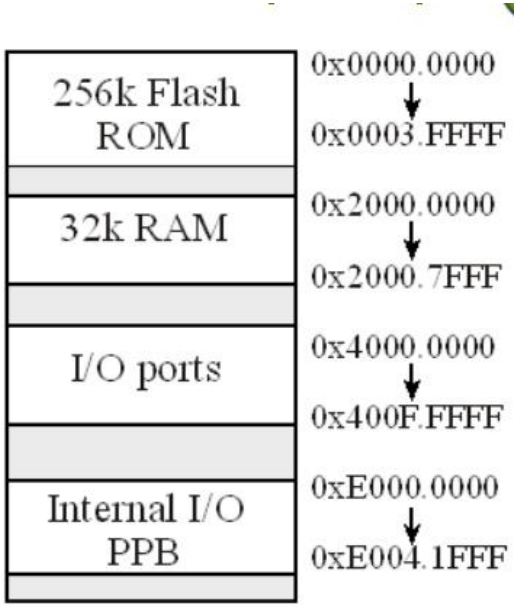
Memory Map: Facts

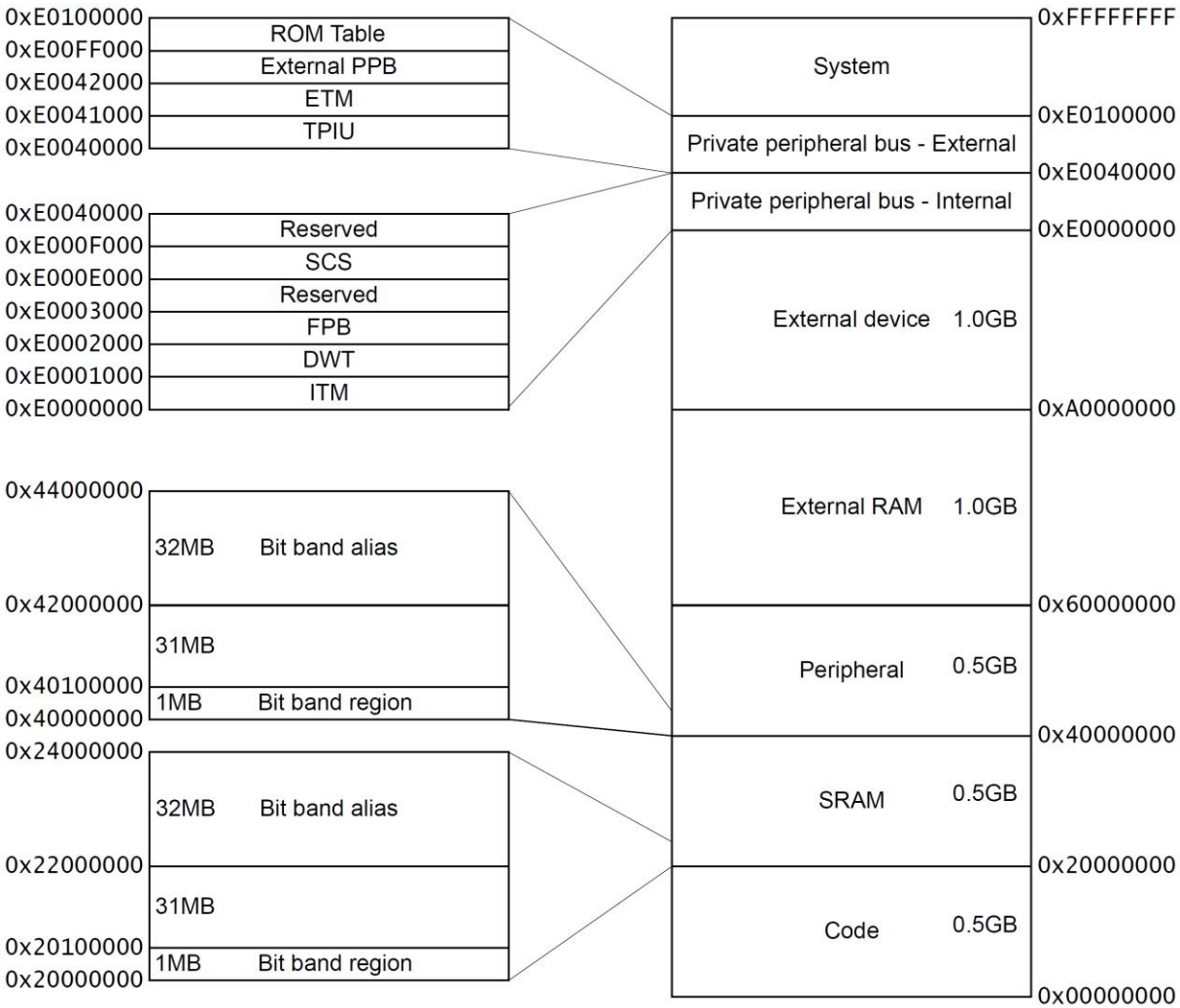
- Cortex-M3 and M4 Processors do not include memories
  - No program memory
  - No SRAM
  - No cache memory
- Memory Access of Cortex-M3 and M4 Processors
  - Generic on-chip bus interface

    → Microcontroller vendors can add their own memory system to their design
- Customization for Different Microcontroller Products
  - Different memory configurations
  - Different memory sizes and types
  - Different peripherals

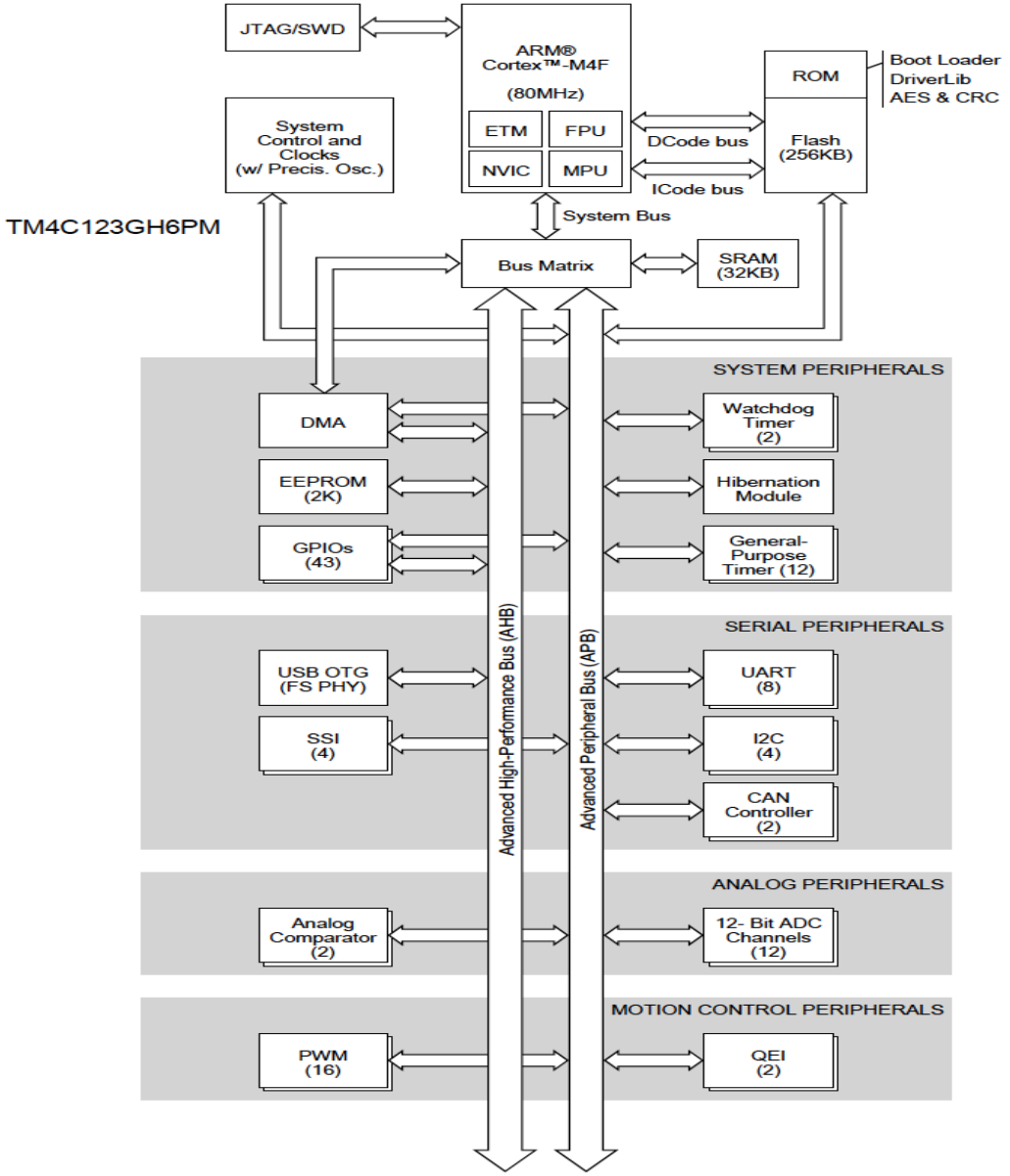
Cortex-M4 Fixed Memory Map

- Example: TM4C123G MCU
  - 256 KB Flash
  - 32 KB RAM
  - 2KB EEPROM



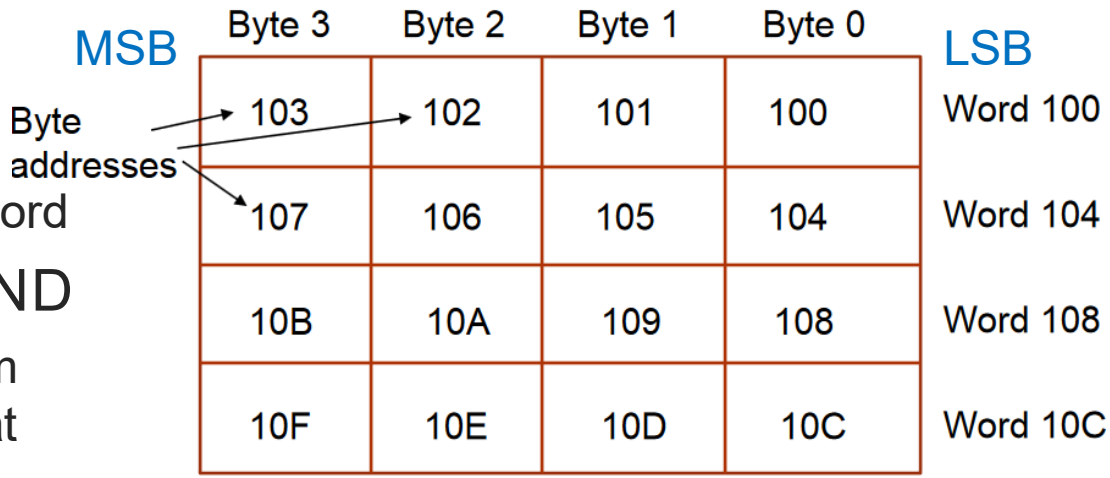


Bus Architecture: TM4C123GH6PM



ARM Cortex-M3/4 Memory Formats Reminder

- Default memory format for ARM CPUs: LITTLE ENDIAN
  - Bytes 0 (LSB) -3 (MSB) hold the first stored word
  - Bytes 4 (LSB) -7 (MSB) hold the second stored word



- Configuration Pin BIGEND
  - Enables hardware system developer to select format
    - Little Endian (Value 0)
    - BE-8: Byte-invariant Big Endian (Value 1)
  - Pin is sampled on reset
  - It is not possible to change endianness when out of reset

ARM Cortex-M3/4 Endianness: Comparison

- Little Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x3	Byte – 0x2	Byte – 0x1	Byte – 0x0
…				
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
…				
…	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

- Big Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x0	Byte – 0x1	Byte – 0x2	Byte – 0x3
…				
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
…				
…	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

Instruction Set: Comparison

- ARM now called AArch32

32-bit	32-bit	32-bit	32-bit	32-bit

- Thumb (captures all ARM 32 bit instructions)

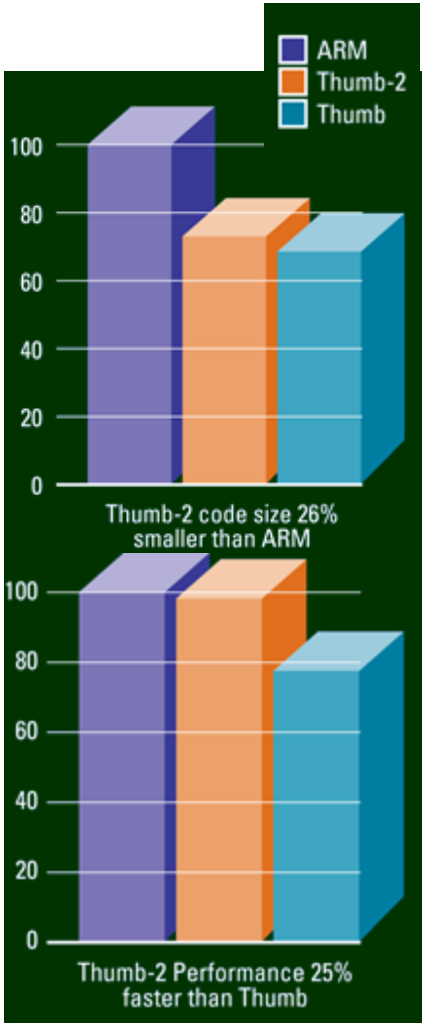
16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit

- Thumb-2

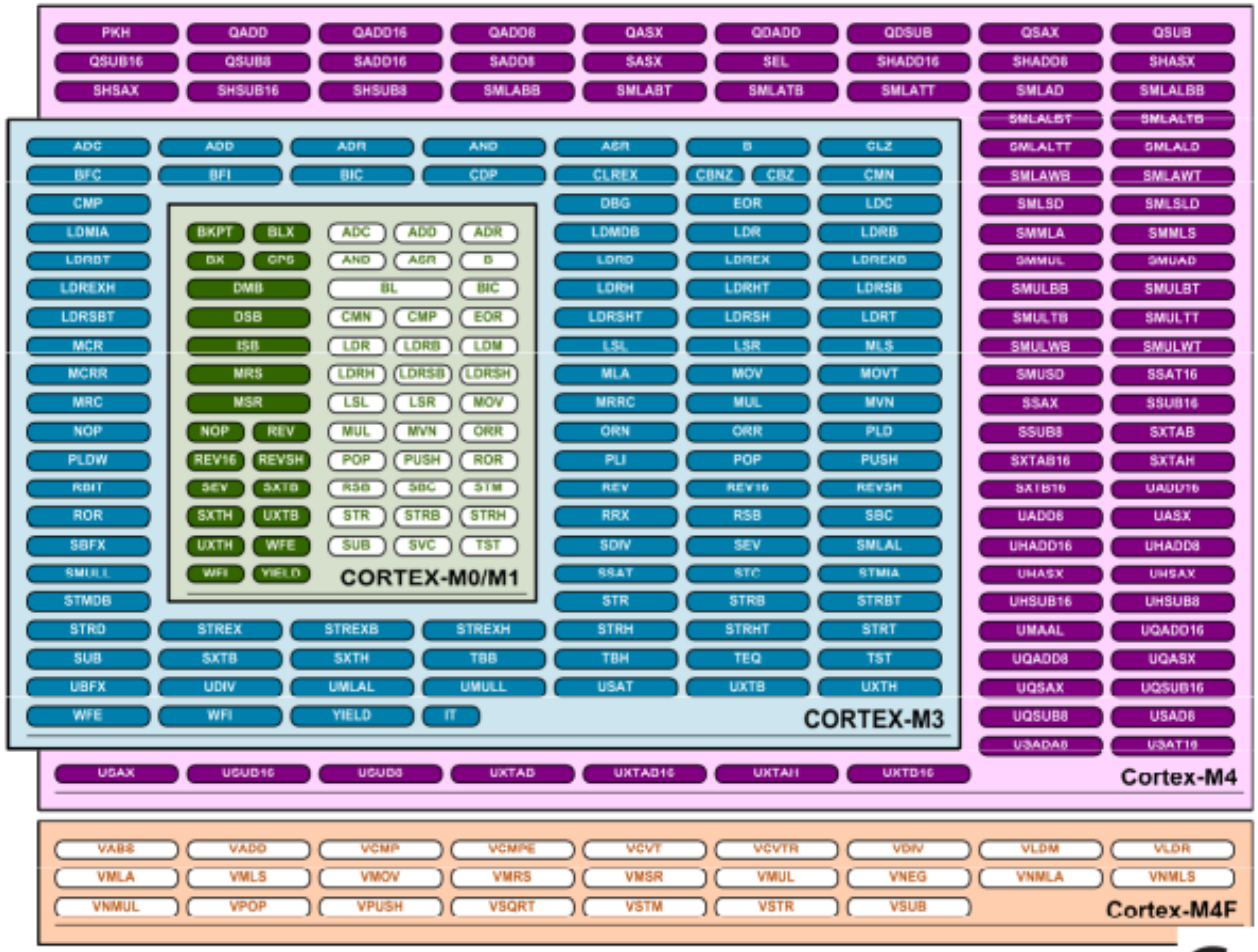
32-bit	32-bit	16-bit	16-bit	16-bit	32-bit	16-bit

- Range of Instructions

Instruction Groups	Cortex-M0, M1	Cortex-M3	Cortex-M4	Cortex-M4 with FPU
16-bit ARMv6-M instructions	●	●	●	●
32-bit Branch with Link instruction	●	●	●	●
32-bit system instructions	●	●	●	●
16-bit ARMv7-M instructions		●	●	●
32-bit ARMv7-M instructions		●	●	●
DSP extensions			●	●
Floating point instructions				●



Instruction Set: Comparison

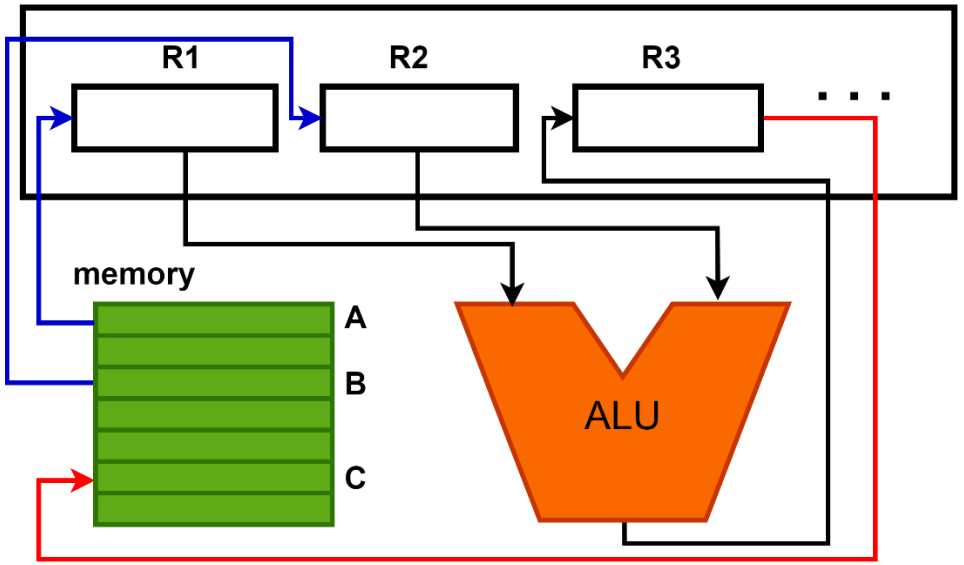


Note: A Cortex® - M4 processor with the floating point unit is frequently referred to as Cortex-M4F

Cortex
Low-Power Leadership from ARM

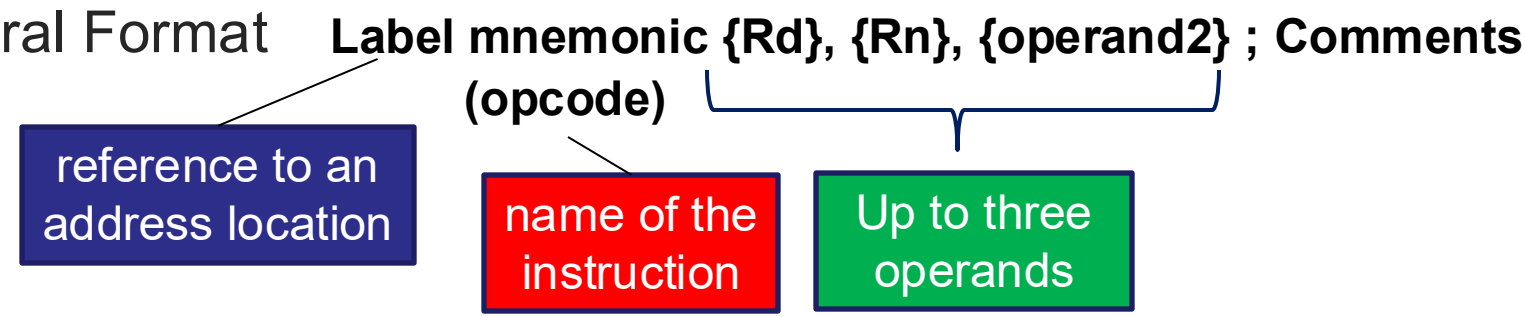
Load/Store Architecture

- Properties
  - ALU operations are simple and are always between registers
  - Use simple dedicated load/store instructions for register-memory transfers (which should be less frequent than ALU operations)
- Example Program: C = A + B
  - Load R1, A
  - Load R2, B
  - Add R3, R1, R2
  - Store R3, C



Instruction Format

- General Format



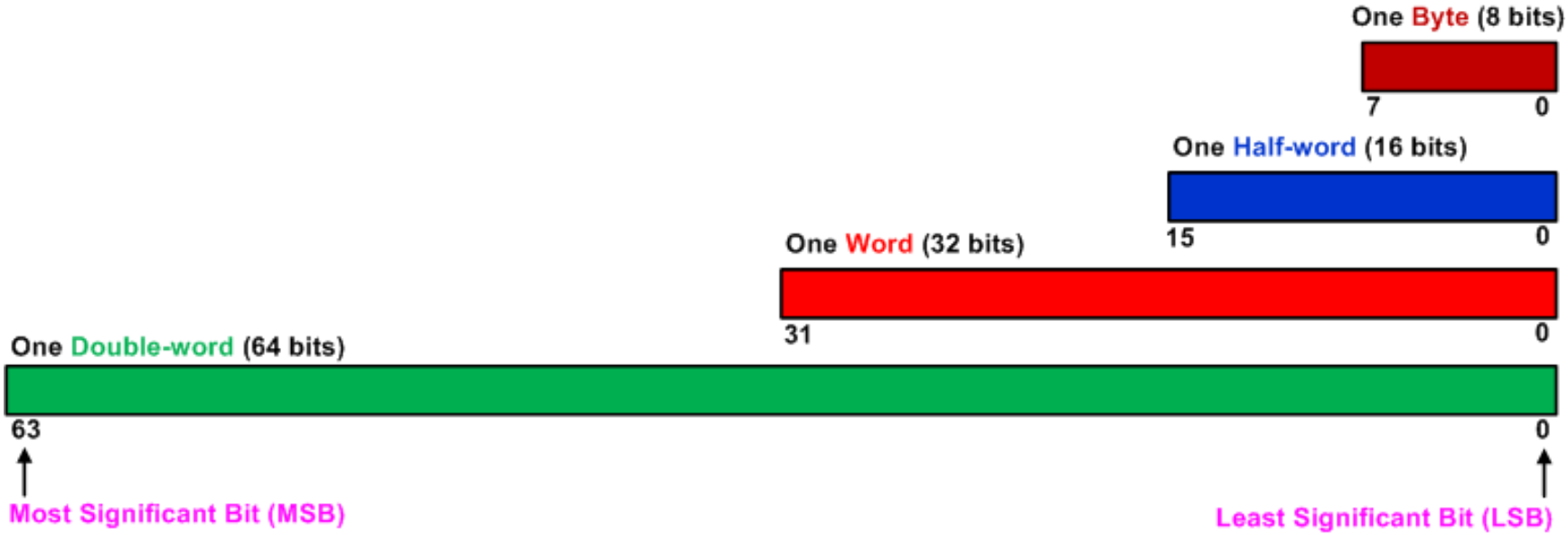
- Examples

```
Loc   ADD   R3,   R2,   R1    ;R2+R1->R3
      ADD   R3,   R2,   #5    ;R2+5->R3
      LDR   R3,   [R2]        ;R3 = value pointed by R2
```

- General Observations
  - If exists, **Rd** is typically the destination register
  - If exists, **Rn** is typically the source register
  - operand2 is the flexible second operand that can be either a register (**Rm**), shifted register or a constant (immed_8r)

Supported Data Types: Overview

- Integer ALU Operations
  - 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)



Supported Data Types: Conversion

- Convert Bytes/Half-words 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 also supports single and double-precision IEEE floating-point data

Addressing Modes: Overview

- Addressing Mode Definition
  - The addressing mode is the format the instruction uses to specify the memory location to read or write data
- Generic Addressing Mode Types
  - Immediate
  - Direct
  - Indirect or indexed
  - Relative
- Unused Addressing Mode Types by ARM
  - 32-bit address can not be included in a 32-bit instruction

    → ARM does not use the direct addressing mode

Addressing Modes: Immediate

- Properties
  - Data itself is contained in the instruction
  - Once the instruction is fetched no additional memory access cycles are needed to get the data

```
Opcode    Rd, #constant
```

- Example

```
MOV    R0, #100      : R0=100
ADD    R0,#0x64      ; R0=R0+100
```

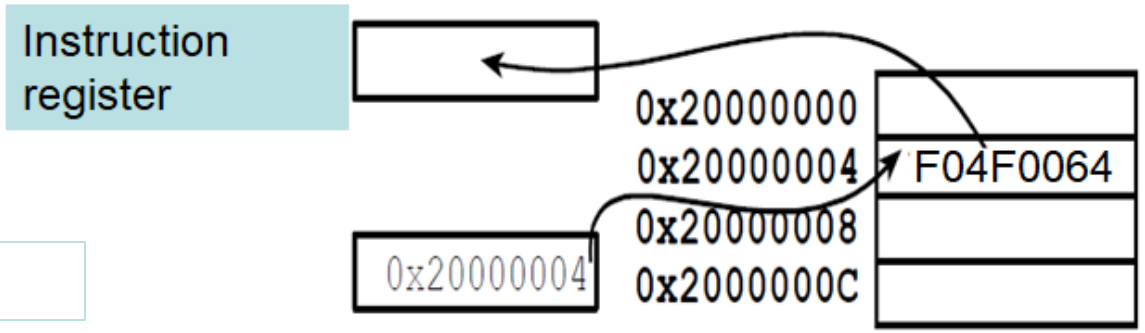
→ This addressing mode is only used to get, load or read data

Addressing Modes: Immediate

- Illustration
  - Assume next instruction is in memory address 0x20000004
  - Instruction is fetched from memory and written in IR (instruction register)
  - Instruction is executed: decimal value 100 is written in register R0

Move

```
LOC     MOV     R0, #100          ; R0=100
; the location of the instruction in the memory is 0x20000004,
; LOC=0x20000004
; machine code of the instruction is 0xF04F0064
```



R0	100

Overview

Adressing Modes: Register

- Properties
  - Involves the use of registers to hold the data to be manipulated
  - Memory is not accessed when this addressing mode is executed
    → This mode is relatively fast

```
Opcode    Rd, Rn
```

- Example

```
MOV R3,R2: Copy the contents of R2 into R3
```

Adressing Modes: Indexed

- Properties
  - The data is in memory and a register contains a pointer to the data
  - Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data

Opcode R1,[R2, optional offset, optional shift], optional offset

→ Can include an offset from the index address

→ Can include updating index register with offset (pre- or post- access)

- Several Forms of Indexed Addressing

Load / Store

LDR{type}	**Rd,[Rn]**	**;load memory at [Rn] to Rd**
STR{type}	**Rt,[Rn]**	**;store Rt to memory at [Rn]**
LDR{type}	**Rd,[Rn, #n]**	**;load memory at [Rn+n] to Rd, ;Rn unchanged**

B, SB, H, SH, - (see later)

Overview

Adressing Modes: Indexed

- Different Versions for Load Instructions

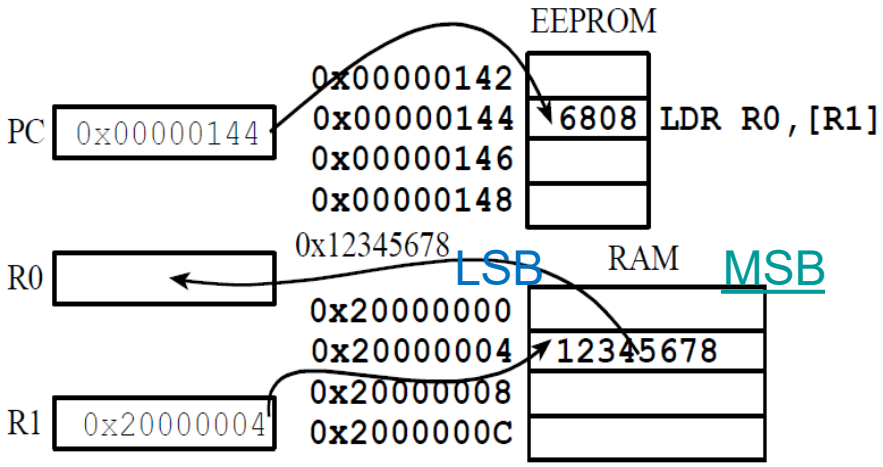
Name	Alternative Name	ARM Examples
Indexed, base	Register indirect	**LDR R0, [R1]**
Pre-indexed, base with displacement	Register indirect with offset	**LDR R0, [R1, #4]**
Pre-indexed, autoindexing	Register indirect, pre-incrementing	**LDR R0, [R1, #4]!**
Post-indexing, autoindexed	Register indirect, post-increment	**LDR R0, [R1], #4**

Adressing Modes: Indexed, Base

- Illustration for Load Instruction
  - Assume R1 points to the memory address 0x20000004
  - We want to load the value at that address into R0

LDR R0,[R1] ; R0 = value pointed to by R1
; location of the instruction in memory is 0x00000144
; machine code of the instruction is 0x6808
; R1 = 0x20000004

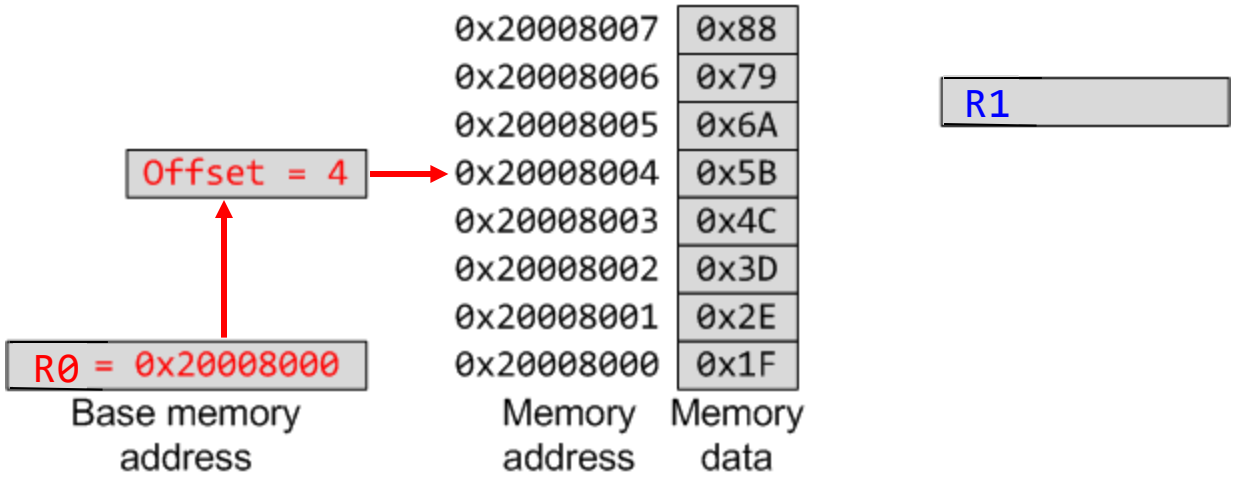
→ After the execution,
R0 = 0x78563412



Adressing Modes: Pre-Index

- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address with offset 4 in register R1

```
LDR R1, [R0, #4]
```



Adressing Modes: Pre-Index

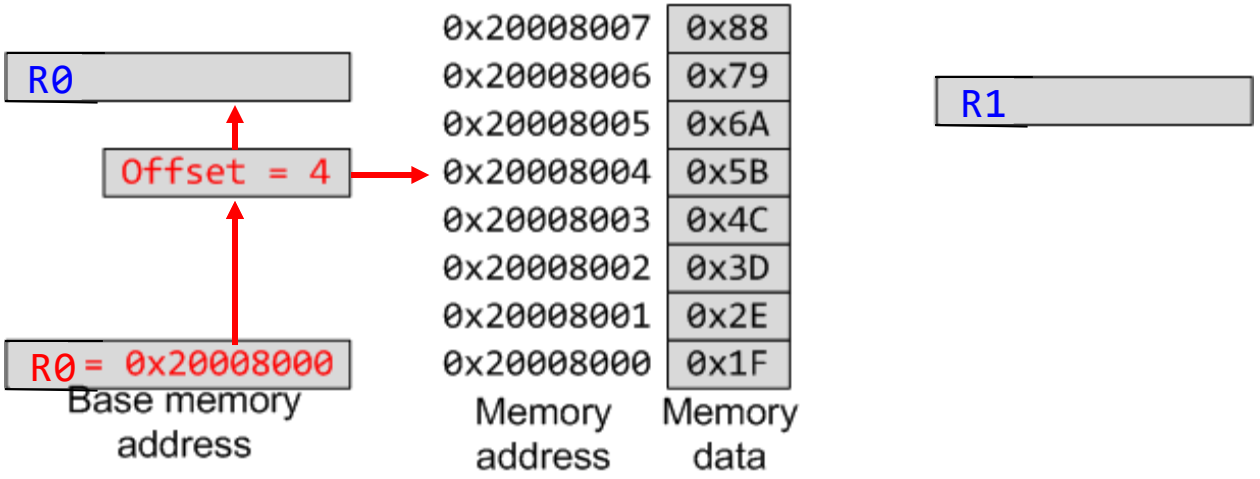
- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address with offset 4 in register R1

```
LDR R1, [R0, #4]
```

Adressing Modes: Pre-Index with Update

- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address with offset 4 in R1
  - We update (increment) the memory address stored in R0

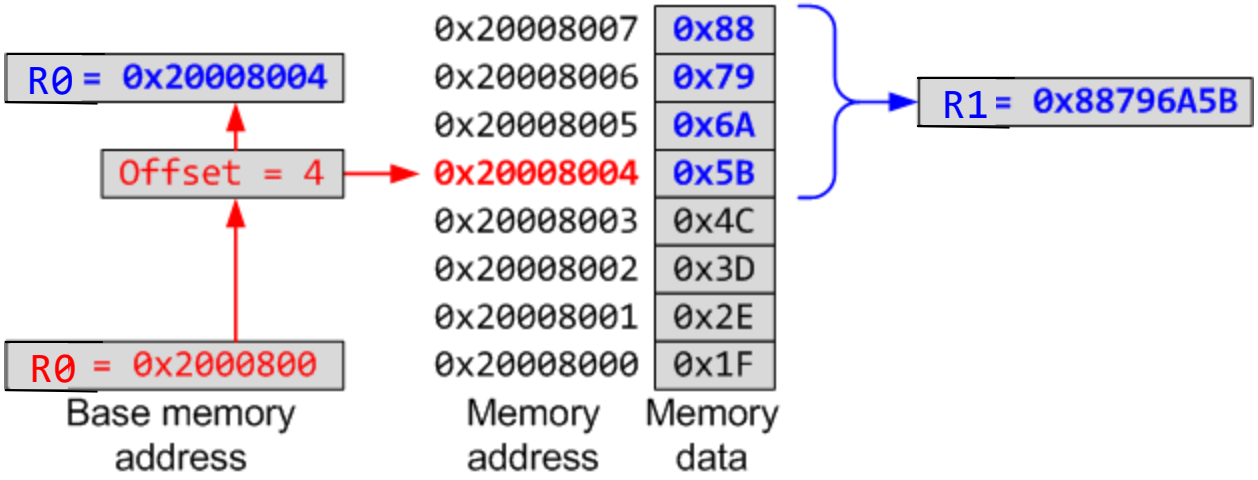
```
LDR R1, [R0, #4]!
```



Adressing Modes: Pre-Index with Update

- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address with offset 4 in R1
  - We update (increment) the memory address stored in R0

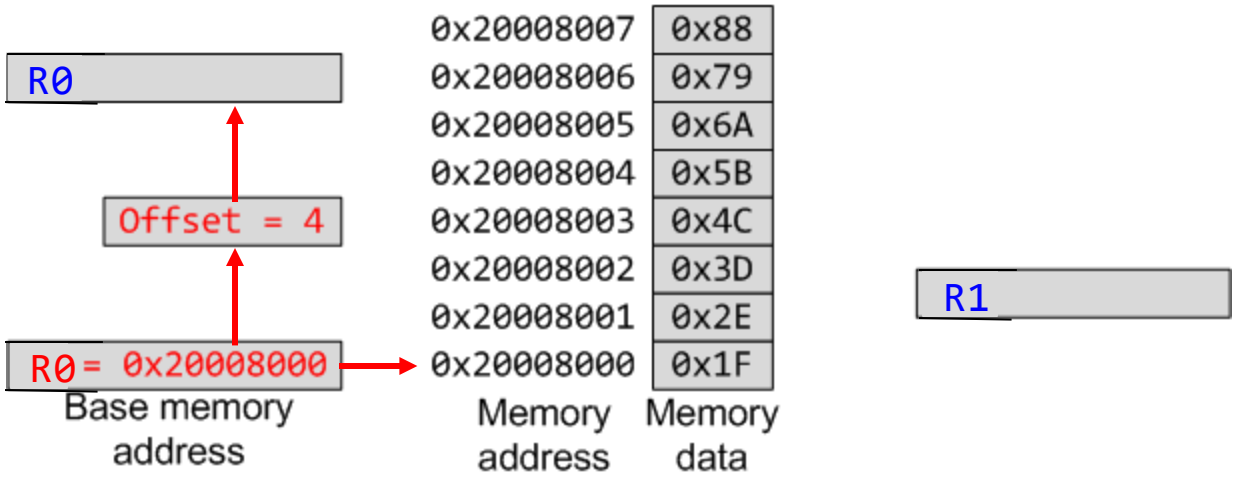
```
LDR R1, [R0, #4]!
```



Adressing Modes: Post-Index

- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address in R1
  - We update (increment) the memory address stored in R0 with offset 4

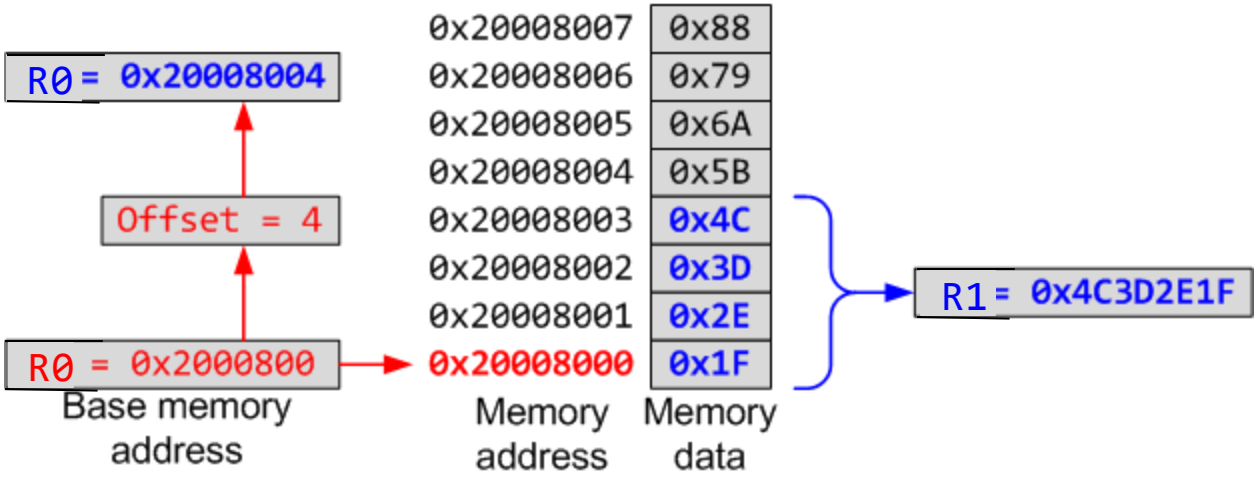
```
LDR R1, [R0], #4
```



Adressing Modes: Post-Index

- Illustration for Load Instruction
  - Assume R0 points to the memory address 0x2000800
  - We load the value at that address in R1
  - We update (increment) the memory address stored in R0 with offset 4

```
LDR R1, [R0], #4
```



Adressing Modes: Indexed

- Summary

Index Format	Example	Equivalent
Pre-index	LDR R1, [R0, #4]	R1 ← memory[R0 + 4], R0 is unchanged
Pre-index with update	LDR R1, [R0, #4]!	R1 ← memory[R0 + 4] R0 ← R0 + 4
Post-index	LDR R1, [R0], #4	R1 ← memory[R0] R0 ← R0 + 4

Store Instructions: Overview

- STR Rt, [Rs]:
  - Save data in register Rt into memory
  - The memory address is specified in a base register Rs.
  - Pre and post-indexed addressing modes can be used like LDR
- Reminder

STR{type} R1,[R0, optional offset, optional shift], optional offset

Adressing Modes: Indexed, Base

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x20008000
  - We want to store the value in R1 at that address

```
STR R1, [R0]; R0 = 0x20008000,  R1=0x76543210
```

R0 before store

0x20008000

R0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Indexed, Base

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x20008000
  - We want to store the value in R1 at that address

```
STR R1, [R0]; R0 = 0x20008000,  R1=0x76543210
```

R0 before store

0x20008000

R0 after store

0x20008000

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

Adressing Modes: Pre-Index

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address with offset 4

```
STR R1, [R0, #4]; R0 = 0x20008000,  R1=0x76543210
```

R0 before store

0x20008000

R0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Pre-Index

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address with offset 4

```
STR R1, [R0, #4]; R0 = 0x20008000,  R1=0x76543210
```

R0 before store

0x20008000

R0 after store

0x20008000

Memory Address	Memory Data
0x20008007	**0x76**
0x20008006	**0x54**
0x20008005	**0x32**
0x20008004	**0x10**
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Pre-Index with Update

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address with offset 4
  - We update (increment) the memory address stored in R0

```
STR R1, [R0, #4]!
; R0=0x20008000, R1=0x76543210
```

R0 before store

0x20008000

R0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Pre-Index with Update

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address with offset 4
  - We update (increment) the memory address stored in R0

```
STR R1, [R0, #4]!
; R0=0x20008000, R1=0x76543210
```

R0 before store

0x20008000

R0 after store

0x20008004

Memory Address	Memory Data
0x20008007	**0x76**
0x20008006	**0x54**
0x20008005	**0x32**
0x20008004	**0x10**
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Post-Index

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address
  - We update (increment) the memory address stored in R0 by 4

```
STR R1, [R0], #4
; R0=0x20008000, R1=0x76543210
```

R0 before store

0x20008000

R0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Adressing Modes: Post-Index

- Illustration for Store Instruction
  - Assume R0 points to the memory address 0x2000800
  - We store the value in R1 at that address
  - We update (increment) the memory address stored in R0 by 4

```
STR R1, [R0], #4
; R0=0x20008000, R1=0x76543210
```

R0 before store

0x20008000

R0 after store

0x20008004

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

Adressing Modes: Indexed

- Summary

Index Format	Example	Equivalent
Pre-index	STR R1, [R0, #4]	R1 → memory[R0 + 4], R0 is unchanged
Pre-index with update	STR R1, [R0, #4]!	R1 → memory[R0 + 4] R0 ← R0 + 4
Post-index	STR R1, [R0], #4	R1 → memory[R0] R0 ← R0 + 4

PC-Relative Addressing Mode with LDR

- Explanation
  - Indexed addressing using the PC (program counter) as the pointer
  - Instruction representation: PC value plus or minus a numeric offset
  - Assembler calculates the required offset from the label and the address of the current instruction
  - If the offset is too big, the assembler produces an error
  - Usage for branching, calling functions, etc.

Example of Literal Read	Description
`LDRB Rt,[PC, #offset]`	Load unsigned byte into Rt using PC offset
`LDRSB Rt,[PC, #offset]`	Load and signed extend a byte data into Rt using PC offset
`LDRH Rt,[PC, #offset]`	Load unsigned half-word into Rt using PC offset
`LDRSH Rt,[PC, #offset]`	Load and signed extend a half-word data into Rt using PC offset
`LDR Rt, [PC, #offset]`	Load a word data into Rt using PC offset
`LDRD Rt,Rt2,[PC, #offset]`	Load a double-word into Rt and Rt2 using PC offset

R15

Pseudo Instructions

Requires 2 instructions

- Example

Address	Source Program	Debug Disassembly
	MOV**32** r0,#0x12345678	MOVW r0,#0x5678 MOVT r0,#0x1234
0x0000050C	LDR r3,=0x12345678	LDR r3,[PC,#offset]
0x0000050E		
.....		
0x0000051C	0x78	; loaded value (32 bit)
0x0000051D	0x56	; in literal pool following the code
0x0000051E	0x34	
0x0000051F	0x12	

Current Address

Next Address

Destination Address

****

offset = Destination Address – (PC value pointing next instruction) = (0x51C – 0x50E) = 0xE

Pseudo Instructions

Requires 2 instructions

- Example

Address	Source Program	Debug Disassembly
	MOV**32** r0,#0x12345678	MOVW r0,#0x5678 MOVT r0,#0x1234
0x0000050C	LDR r3,=0x12345678	LDR r3,[PC,#offset]
0x0000050E		
.....		
0x0000051C	0x78	; loaded value (32 bit)
0x0000051D	0x56	; in literal pool following the code
0x0000051E	0x34	
0x0000051F	0x12	

Current Address

Next Address

Destination Address

****

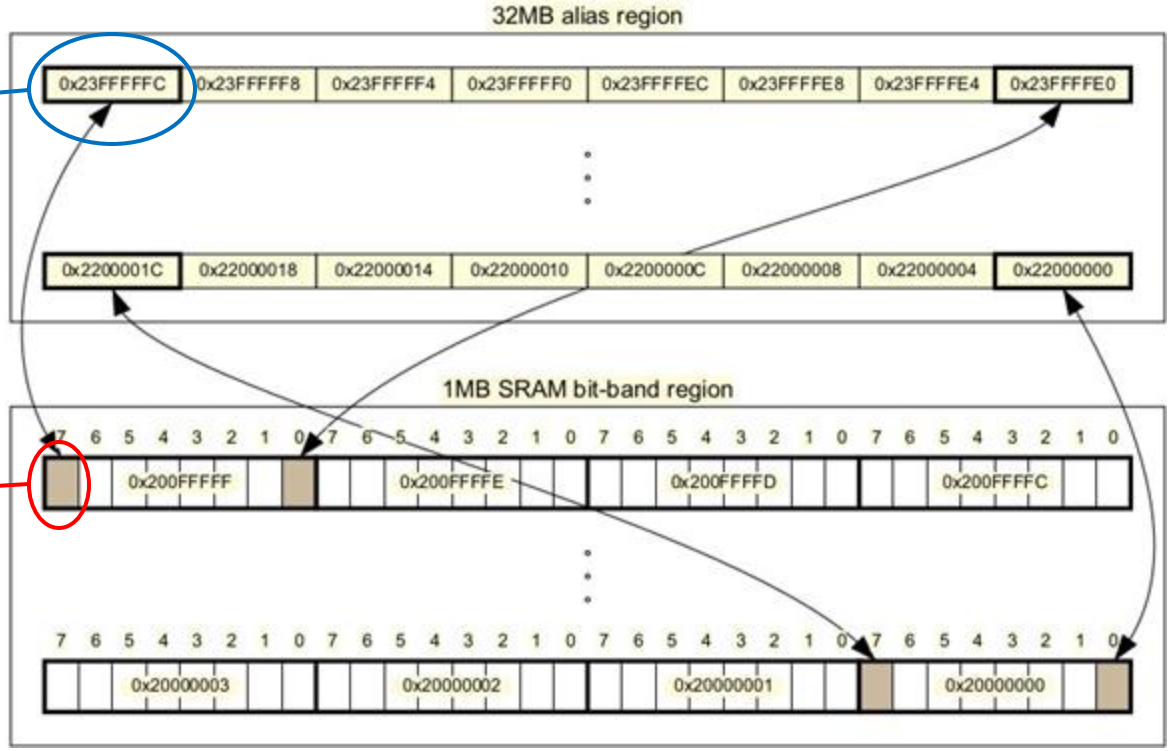
offset = Destination Address – (PC value pointing next instruction) = (0x51C – 0x50E) = 0xE

Bit-Banding: Overview

- A region of memory is used as Bit-band region
- Each bit in the Bit-band region is mapped to an entire word in a second memory region called Bit-band Alias Region

Word in the Bit-band Alias Region

Corresponding bit in the Bit-band Region



Bit-Banding: Explanation

- Detailed Operation
  - Writing 1/0 to bit[0] in the Bit-band Alias word, writes 1/0 to the Bit-band bit
  - Bits [31-1] of the Bit-band Alias word have no effect on the Bit-band bit
- Advantage
  - When writing programs for embedded systems, flags (single bit) are needed
  - Changing the value of a flag in memory requires
    - Reading the word of the flag to some register
    - Modifying the bit of the flag
    - Writing the word of the flag to memory

    } Read-Modify-Write
  - Bit-banding
    - Write to a word in the Bit-band Alias region performs a write to the corresponding bit in the Bit-band region.
    - Reading a word in the Bit-band Alias region will return the value of the corresponding bit in the Bit-band region.

    → Efficient operation on flags and avoidance of race conditions

Bit-Banding: Example

- Set bit 0 in word data in address 0x2000000 without Bit-band

```
LDR  R0,=0x20000000      ; immediate write address in R0
LDR   R1,[R0]            ; read word from address
ORR  R1, #0x1            ; modify bit in register R1
STR   R1,[R0]            ; write back
```

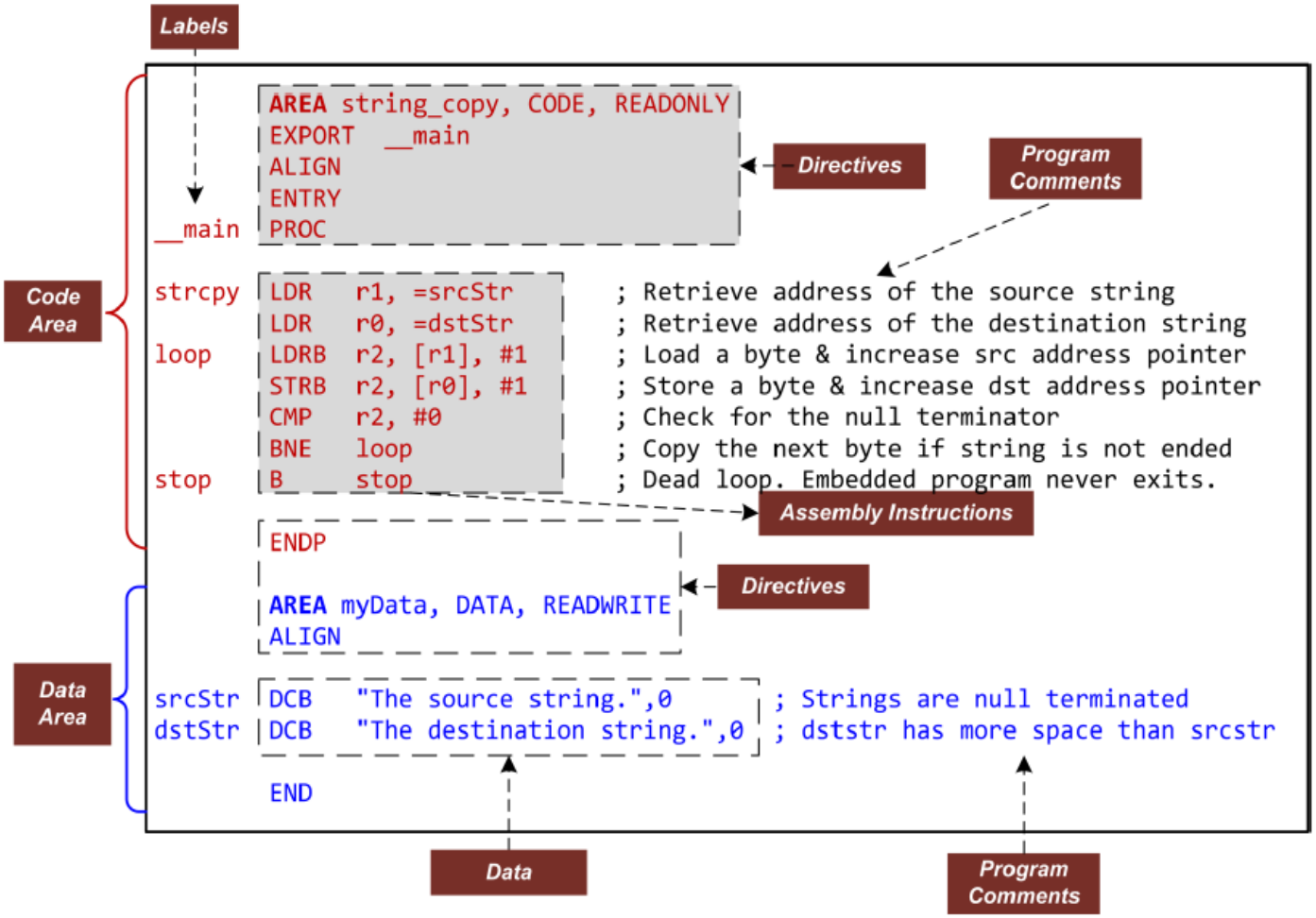
3 instructions

- Set bit 0 in word data in address 0x2000000 with Bit-band

```
LDR  R0,=0x22000000      ; set-up alias address
MOV R1,#1                ; load #1
STR  R1,[R0]             ; write
```

- Exercise
  - Set bit 6 in word data in address 0x2000008 with and without Bit-band

Anatomy of an Assembly Program



Assembly Directives

- Description
  - Directives are used to provide key information for assembly
  - Important: Directives are NOT instructions
  - Examples

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Directive: AREA

- Indicate 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
- 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 (READONLY)
- A data area may be read from and written to (READWRITE).

```
            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
```

Directive: ENTRY

- The ENTRY directive marks the first instruction to be executed within an application program
- There must be exactly one ENTRY directive in an application, no matter how many source files the application has

```
            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
```

Directive: END

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

```
            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
```

Directive: PROC and ENDP

- 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 function within another function

```
            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
```

Directive: EXPORT and IMPORT

- 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 symbol must be defined in another file
- The IMPORT is similar to the “extern” keyword in C

```
            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
```

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

Directive: Data Allocation Examples

```
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’
```

Directive: EQU and RN

- 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

```
; 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

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

Directive: ALIGN

- The default ALIGN directive aligns the current location within the code to a word (4-byte) boundary
- ALIGN 2 can be used to align on a halfword (2-byte) boundary

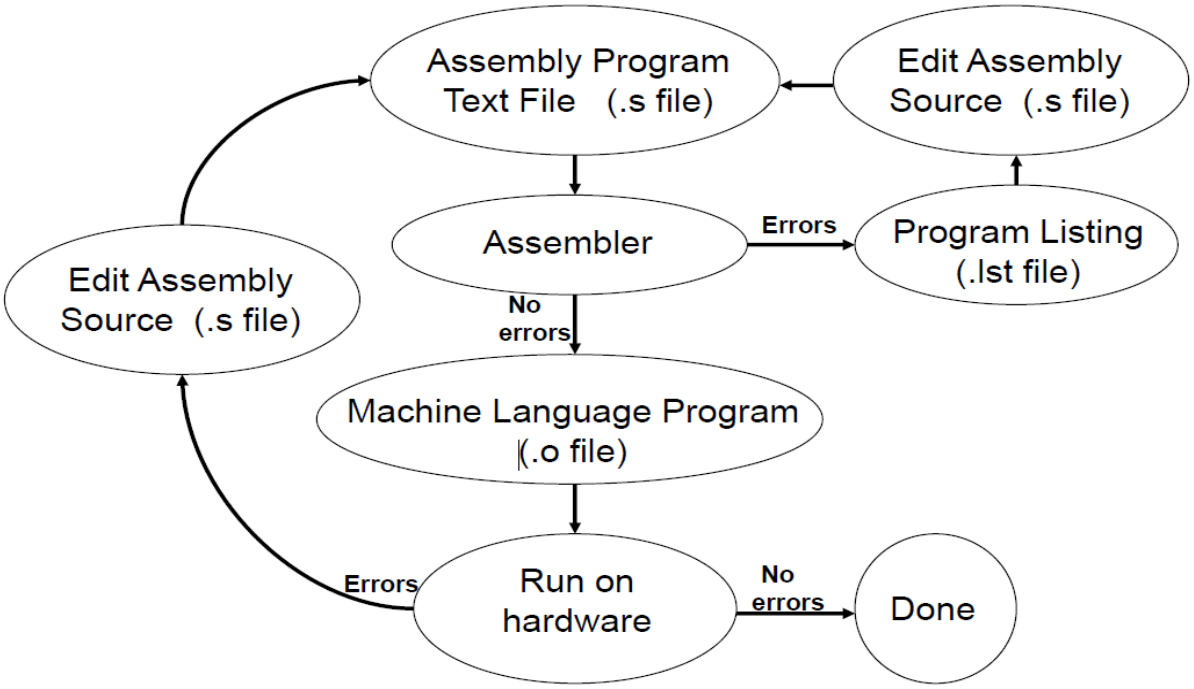
```
  AREA example, CODE, ALIGN = 3   ; Memory address begins at a multiple of 8
  ADD r0, r1, r2                  ; Instructions start at a multiple of 8

  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 (3) of a word (4)
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
```

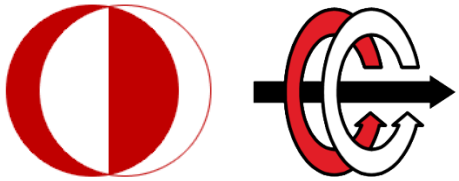
Directive: INCLUDE or GET

- 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

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

Assembly Process



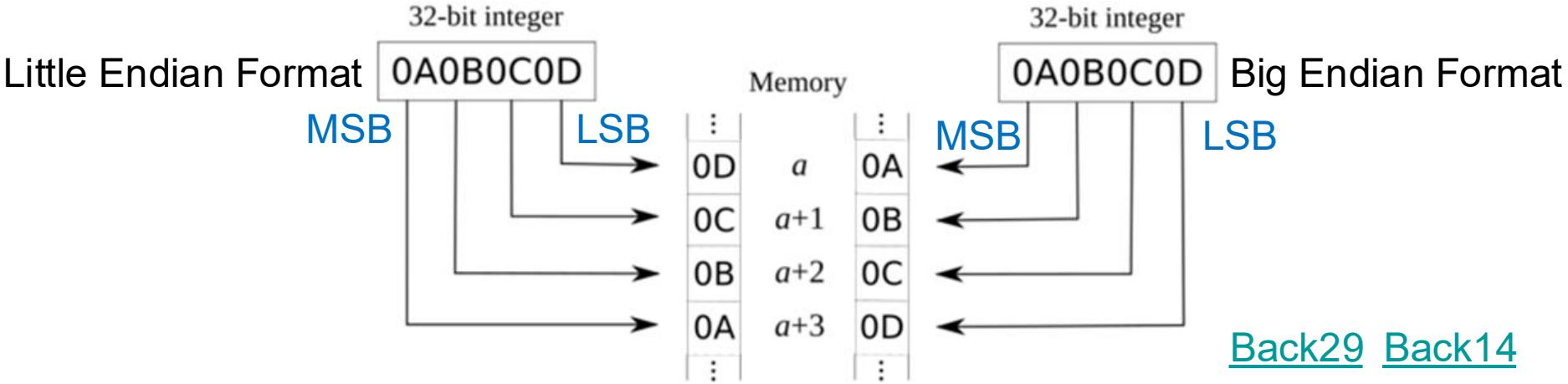


ARM Architecture, Programming Model

Week 2

Memory: Data Encoding

- Little Endian Format
  - Least significant byte (LSB) is stored in the lowest address of the memory
  - Most significant byte (MSB) is stored in the highest address of the memory
  - Note: ARM is Little Endian by default
- Big Endian Format
  - Least significant byte (LSB) is stored in the highest address of the memory
  - Most significant byte (MSB) is stored in the lowest address of the memory



Back29 Back14

Memory Access Instructions for Various Data Sizes

Instruction	Dest	Source	Operations
MOV	R4,	R0	; Copy value from R0 to R4
MOVS	R4,	R0	; Copy value from R0 to R4 with APSR (flags) update
MRS	R7,	PRIMASK	; Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	; Copy value of R2 into CONTROL (special register)
MOV	R3,	#0x34	; Set R3 value to 0x34
MOVS	R3,	#0x34	; Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	; Set R6 to a 16-bit constant 0x1234
MOVT	R6,	#0x8765	; Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	; Move negative value of R7 into R3

Back

Instructions for Transferring Data within the Processor

	LDR{type}	STR{type}
Data Type	**Load (Read from Memory)**	**Store (Write to Memory)**
8-bit unsigned	LDRB	STRB
8-bit signed	LDRSB	STRB
16-bit unsigned	LDRH	STRH
16-bit signed	LDRSH	STRH
32-bit	LDR	STR
Multiple 32-bit	LDM	STM
Double-word (64-bit)	LDRD	STRD
Stack operations (32-bit)	POP	PUSH

Back

Memory Access Instructions with Immediate Offset

Example of Pre-index Accesses Note: the #offset field is optional	Description
`LDRB Rd, [Rn, #offset]`	Read byte from memory location Rn + offset
`LDRSB Rd, [Rn, #offset]`	Read and signed extend byte from memory location Rn + offset
`LDRH Rd, [Rn, #offset]`	Read half-word from memory location Rn + offset
`LDRSH Rd, [Rn, #offset]`	Read and signed extended half-word from memory location Rn + offset
`LDR Rd, [Rn, #offset]`	Read word from memory location Rn + offset
`LDRD Rd1,Rd2, [Rn, #offset]`	Read double-word from memory location Rn + offset
`STRB Rd, [Rn, #offset]`	Store byte to memory location Rn + offset
`STRH Rd, [Rn, #offset]`	Store half-word to memory location Rn + offset
`STR Rd, [Rn, #offset]`	Store word to memory location Rn + offset
`STRD Rd1,Rd2, [Rn, #offset]`	Store double-word to memory location Rn + offset

Memory Access Instructions with Immediate Offset and Write Back

Example of Pre-index with Write Back Note: the #offset field is optional	Description
`LDRB Rd, [Rn, #offset]!`	Read byte with write back
`LDRSB Rd, [Rn, #offset]!`	Read and signed extend byte with write back
`LDRH Rd, [Rn, #offset]!`	Read half-word with write back
`LDRSH Rd, [Rn, #offset]!`	Read and signed extended half-word with write back
`LDR Rd, [Rn, #offset]!`	Read word with write back
`LDRD Rd1,Rd2, [Rn, #offset]!`	Read double-word with write back
`STRB Rd, [Rn, #offset]!`	Store byte to memory with write back
`STRH Rd, [Rn, #offset]!`	Store half-word to memory with write back
`STR Rd, [Rn, #offset]!`	Store word to memory with write back
`STRD Rd1,Rd2, [Rn, #offset]!`	Store double-word to memory with write back

Memory Access Instructions with Post-Indexing

Example of Post Index Accesses	Description
`LDRB Rd,[Rn], #offset`	Read byte from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRSB Rd,[Rn], #offset`	Read and signed extended byte from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRH Rd,[Rn], #offset`	Read half-word from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRSH Rd,[Rn], #offset`	Read and signed extended half-word from memory [Rn] to Rd, then update Rn to Rn+offset
`LDR Rd,[Rn], #offset`	Read word from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRD Rd1,Rd2,[Rn], #offset`	Read double-word from memory[Rn] to Rd1, Rd2, then update Rn to Rn+offset
`STRB Rd,[Rn], #offset`	Store byte to memory[Rn] then update Rn to Rn+offset
`STRH Rd,[Rn], #offset`	Store half-word to memory[Rn] then update Rn to Rn+offset
`STR Rd,[Rn], #offset`	Store word to memory[Rn] then update Rn to Rn+offset
`STRD Rd1,Rd2,[Rn], #offset`	Store double-word to memory[Rn] then update Rn to Rn+offset

Memory Access Instructions with Register Offset

Example of Register Offset Accesses	Description
`LDRB Rd, [Rn, Rm{, LSL #n}]`	Read byte from memory location Rn + (Rm << n)
`LDRSB Rd, [Rn, Rm{, LSL #n}]`	Read and signed extend byte from memory location Rn + (Rm << n)
`LDRH Rd, [Rn, Rm{, LSL #n}]`	Read half-word from memory location Rn + (Rm << n)
`LDRSH Rd, [Rn, Rm{, LSL #n}]`	Read and signed extended half-word from memory location Rn + (Rm << n)
`LDR Rd, [Rn, Rm{, LSL #n}]`	Read word from memory location Rn + (Rm << n)
`STRB Rd, [Rn, Rm{, LSL #n}]`	Store byte to memory location Rn + (Rm << n)
`STRH Rd, [Rn, Rm{, LSL #n}]`	Store half-word to memory location Rn + (Rm << n)
`STR Rd, [Rn, Rm{, LSL #n}]`	Store word to memory location Rn + (Rm << n)