

ESET 349 - Microcontroller Architecture

Loads, Stores, and Addressing

Dr. Muhammad Faeyz Karim

INTRODUCTION

- Dynamic analysis of instruction usage gives a good indication of the types of operations that are performed most frequently.
- For typical programs, half the instructions deal with data movement between registers and memory.
- Therefore loading and storing data efficiently is critical to processor performance

Memory

- Memory can be view as contiguous storage elements that hold data. It is like a long string of mailboxes, where your letter (data) is stored in a box with a specific number on it (address).
- Width of each storage element is a byte.
- Size of memory is usually measured in megabytes (MB , 2^{20}), gigabytes (GB, 2^{30}) or even terabytes (TB, 2^{40})
- ARM7 has 32 bits address bus which means it can interfaced to 2^{32} or 4GB of memory space

Types of Memory

- ROM : Read Only Memory, memory contents are retained even without power supply.
- EEPROM : Electrically Erasable Programmable ROM
- SRAM : Static Random Access Memory, contents are lost when power is off.
- DRAM : Dynamic Random Access Memory, contents must be constantly refreshed otherwise they will be lost even when power is on.

Instruction Class

- ARM instructions can be broadly separated into 3 basic classes:
- **1. Data Movement**
 - Memory load/store
 - Register Transfers
- **2. Data Operation**
 - Arithmetic
 - Logical
 - Register movement
 - Comparison and test
- **3. Flow Control**
 - Branch
 - Conditional execution

ARM Instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Access memory with load and store instructions only
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the 'S' suffix to the instruction

Load / Store Instructions

- The ARM is a Load / Store Architecture:
 - – Does not support memory to memory data processing operations
 - – Must move data values into registers before using them
- This might sound inefficient, but in practice isn't:
 - – Load data values from memory into registers
 - – Process data in registers using a number of data processing instructions which are not slowed down by memory access
 - – Store results from registers out to memory
- The ARM has three sets of instructions which interact with main memory. These are:
 - – Single register data transfer (LDR / STR)
 - – Block data transfer (LDM/STM)
 - – Single Data Swap (SWP)

Load and Store Instructions

- Only two basic instructions are used for data transfer between memory and processor registers.
- LDR: **L**oad**D** words from memory into a **R**egister
- STR: **S**Tore words from a **R**egister into memory
- Basic syntax:
 - LDR/STR{<cond>}{type} <Rd>, <addressing_mode>
 - where <Rd> = destination (for LDR) & source (for STR)
 - cond = condition flag
 - type = byte, halfword, word(default), signed & unsigned

Common Load/Store Instructions

Loads	Stores	Size and Type
LDR	STR	Word (32 bits)
LDRB	STRB	Byte (8 bits)
LDRH	STRH	Halfword (16 bits)
LDRSB		Signed Byte
LDRSH		Signed Halfword
LDM	STM	Multiple Words

Addressing Modes

ARM uses a fixed-length instruction, with the lowest 12 bits available to specify immediate address

- not sufficient to cover the full 2^{32} address space
- hence do not support direct addressing

ARM only provides indirect addressing modes

1. Register indirect addressing
2. PC-relative addressing

Register Indirect Addressing

- An address is available in a register

- Example:

LDR r0, [r1]

- Here, the r1 content is known as the 'base address', and r1 is called the **base address register**.

- Can be further extended to:

- a) Pre-indexed addressing
- b) Pre-indexed with write-back addressing (uses "!")
- c) Post-indexed addressing (implicit write-back)

Load and Store Word or Byte: Base Register

- The memory location to be accessed is held in a base register
 - STR r0, [r1] ; Store contents of r0 to location pointed ; to by contents of r1.
 - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.

Source
Register
for STR

r0
0 x 5

Base
Register

r1
0 x 200

Memory

0 x 200	0 x 5
0 x 201	0 x 0
0 x 202	0 x 0
0 x 203	0 x 0

r2

Destination
Register
for LDR

Load Half Word Example

LDRH r11, [r0] ;load a halfword into r11

r0 content
before/after load

0x00008000

r11 before load

0x12345678

r11 after load

0x0000FFEE

Memory

Address	Data
---------	------

0x8000	0xEE
--------	------

0x8001	0xFF
--------	------

0x8002	0x90
--------	------

0x8003	0xA7
--------	------

Signed Byte Load Example

LDRSB r11, [r0] ;load a signed byte into r11

r0 content
before/after load

0x00008000

r11 before load

0x12345678

r11 after load

0xFFFFFFFF

Memory

Address	Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7

More Register-Indirect Examples

LDR r0, [r1]

; load r0 with the content of the memory location

; pointed to in r1

STR r2, [r1]

; store content of r2 to memory location with address

; pointed to in r1

LDRB r0, [r1] ; load byte size data

STRH r0, [r1] ; store halfword size data

LDRSB r0, [r1] ; load signed byte

Pre-Indexed

Pre-indexed addressing adds an offset to the base address before executing the load/store.

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]{!}

Example: LDR r0, [r1, #8]

This instruction loads r0 with the content of memory location at (base address + 8).

Optional “!” specifies to write the effective address back into Rn after execution of instruction. Otherwise Rn retains original value.

Useful for addressing an element in a data structure. For example, access a particular register of a peripheral through the peripheral base address and its offset.

Pre-Indexed Load Example

The load instruction **LDR r9, [r12, r8, LSL #2]**

- Will load a word from the memory address given by the base register of r12 and an offset value created from register r8, after shifting left by 2 bits.
- Effective address : $ea<r12+r8*4>$
- If base register r12 contain value 0x4000, offset register r8 contains 0x20, the effective address will be 0x4080

More Pre-Indexed LDR Examples

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]{!}

;r6 gets data from ea(r0+(r1/64))

```
LDR    r6, [r0, r1, ASR #6]
```

;r0 gets data from ea(r1-8)

```
LDR    r0, [r1, #-8]
```

;r0 gets data from ea(r1-(r2*8))

```
LDR    r0, [r1, -r2, LSL #3]
```

More Complex Pre-Indexed LDR Examples

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]{!}

;r5 gets 2 bytes of data from ea(r9) and is sign
;extended to fill the 32 bit r5 register
LDRSH R5, [R9]

;r3 gets 1 byte of data from ea(r8+3) and is sign
;extended to fill the 32 bit r3 register
LDRSB R3, [R8, #3]

;r4 gets 1 byte of data from ea(r10+193) and is sign
;extended to fill the 32 bit r4 register
LDRSB R4, [R10, #0xC1]

;r4 gets 1 byte of data from ea(r10+193) and is sign
;extended to fill the 32 bit r4 register and r10 is
;updated to contain r10 <- r10+193
LDRSB R4, [R10, #0xC1]!

Pre-Indexed with Write-Back

Pre-indexed addressing with write-back automatically updates the base address before executing the load/store.

Example: `LDR r0, [r1, #4]!`

This instruction adds 4 to the base register r1, loads r0 with the content of memory location (now is at base address + 4), and increments r1 by 4.

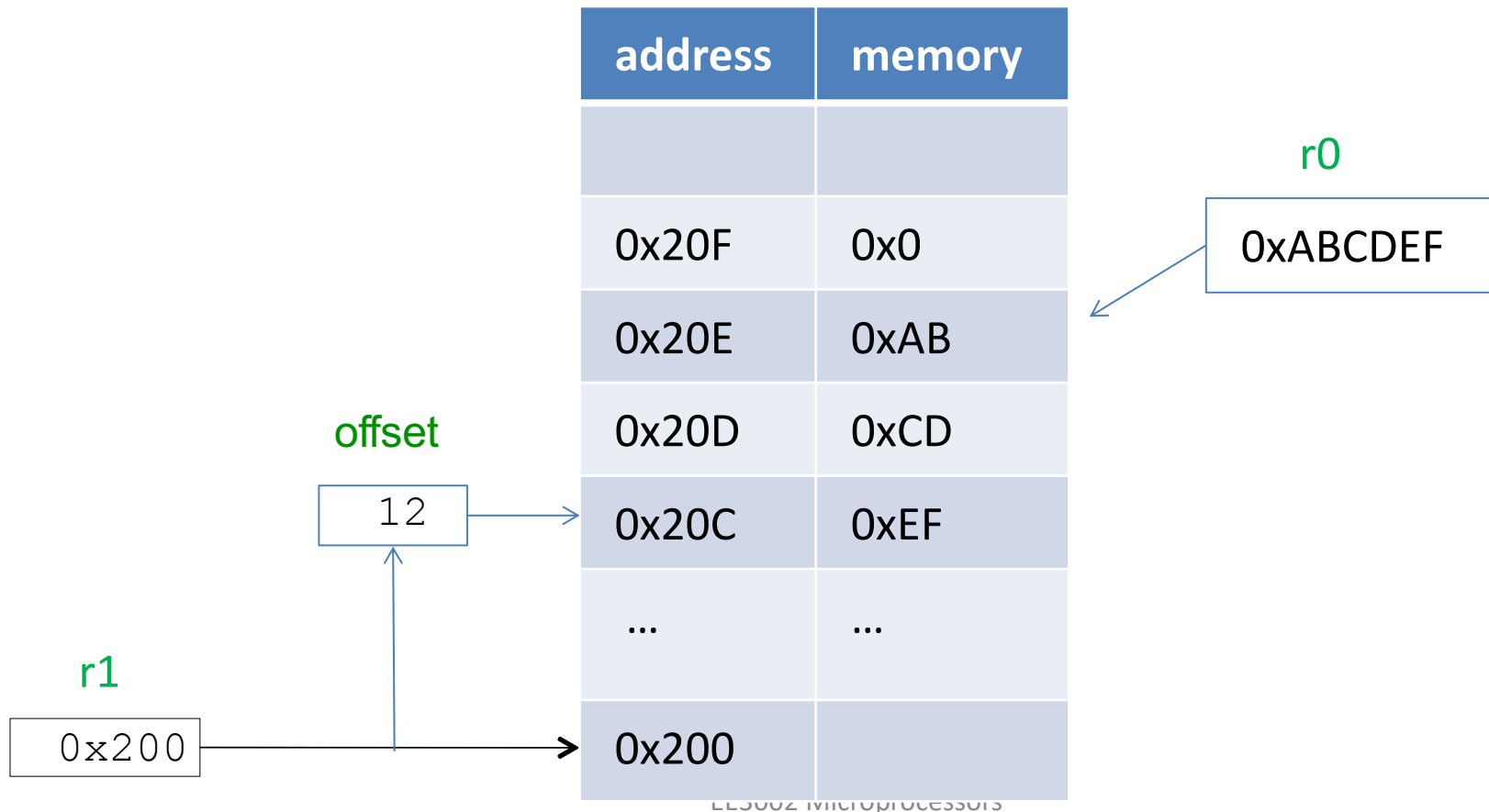
Increment to the base address is done before the execution of the load instruction.

Useful for automatic stepping through a lookup table with a starting address placed in the base address register.

Pre-Indexed STR Example

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]{!}

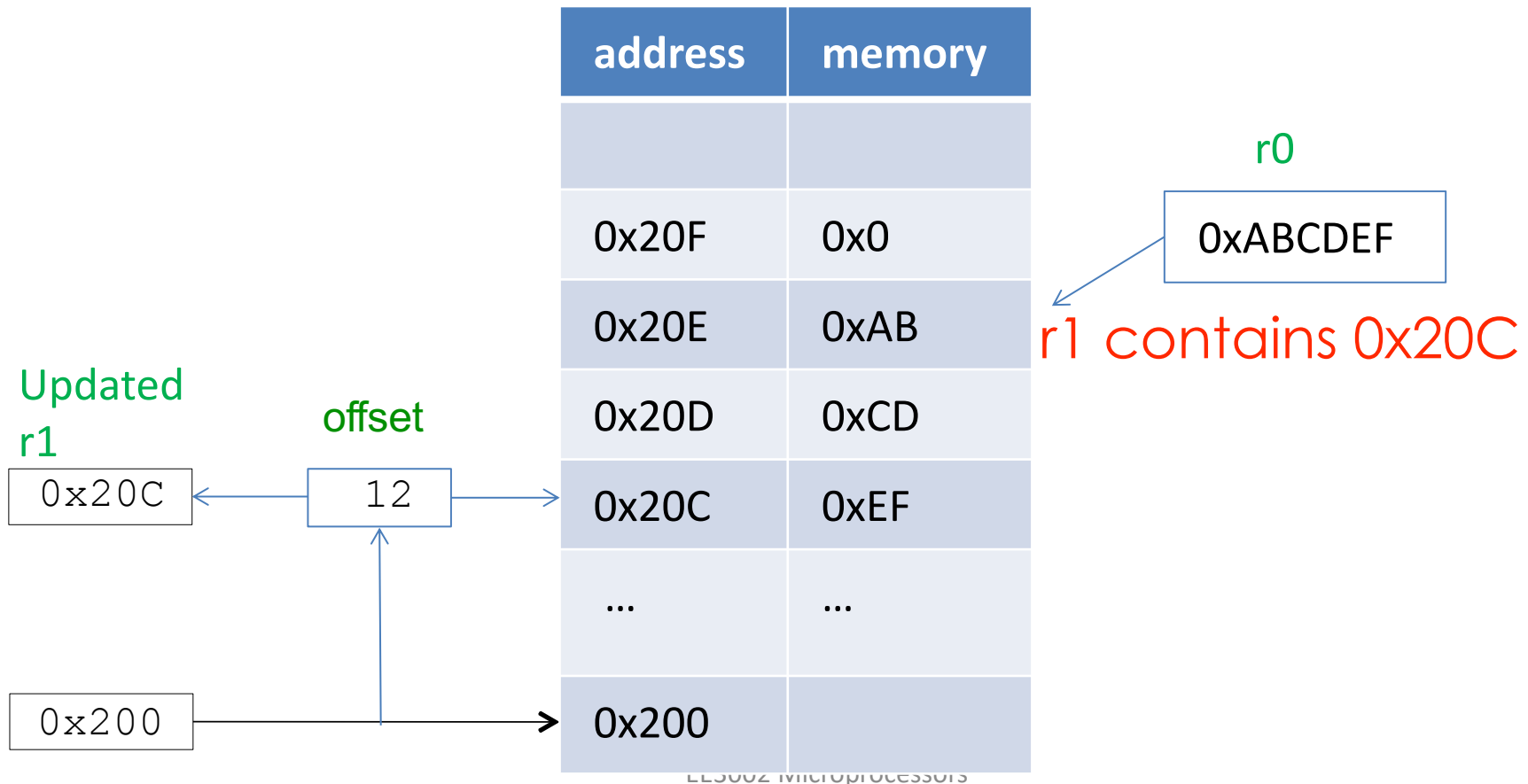
STR r0, [r1, #12]; writes 0xABCDEF to address 0x20C



Pre-Indexed Example with Writeback

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]{!}

STR r0, [r1, #12]!; writes 0xABCDEF to address ;0x20C and updates r1



Post-Indexed

- Post-indexed addressing automatically updates the base address after executing the load/store.

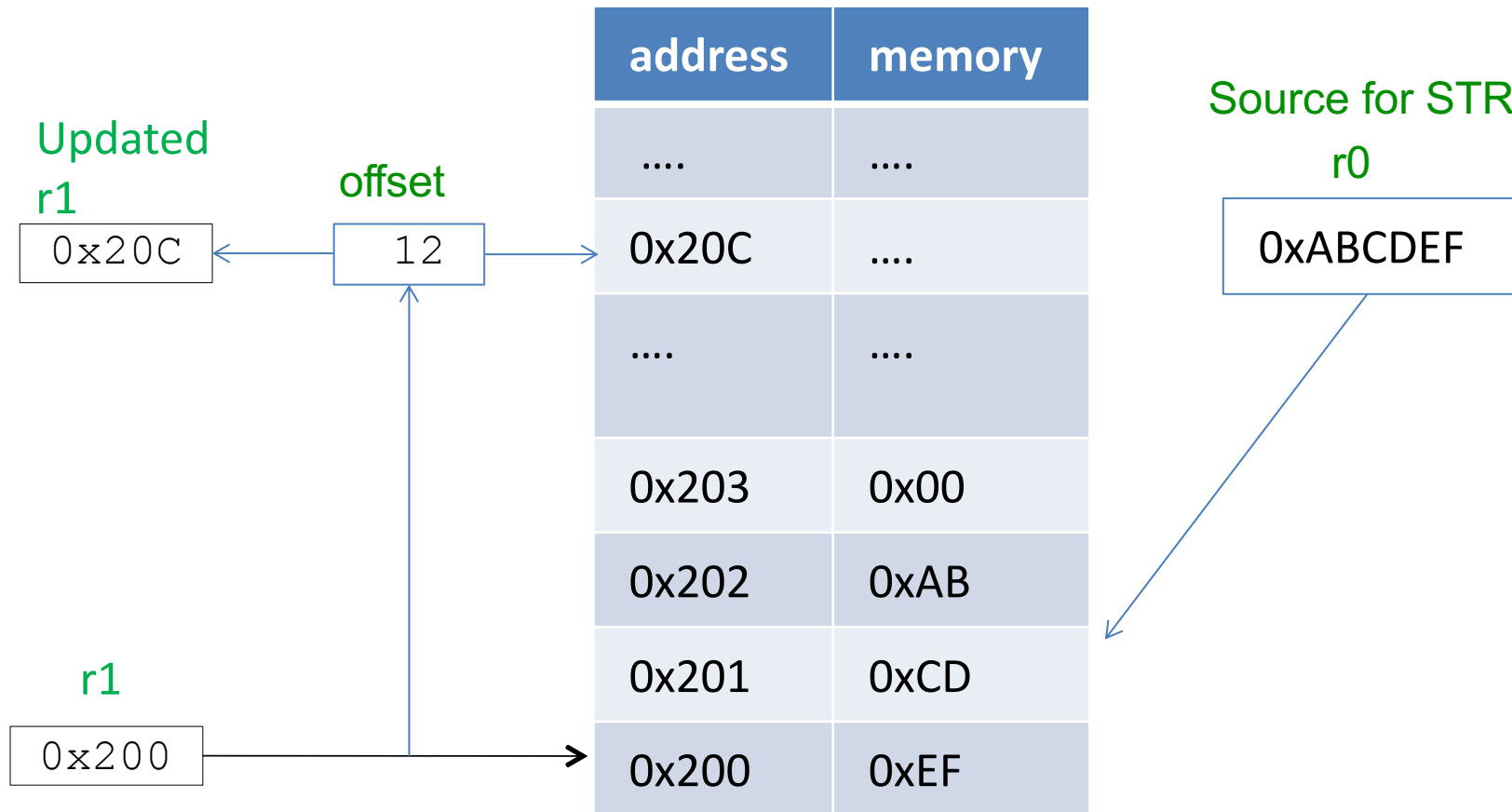
Example: STR r0, [r1], #4;

- This instruction stores the content of r0 into the memory location pointed to in base address register r1, executes the store operation, and increases the base address value by 4. Increment is done after the execution.
- Note that '!' is not needed for post-indexed addressing since the update is implicit.
- Useful for storing a list of data into a table with a starting address pointed to by the base address value in r1.

Post-Indexed Example

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]

STR r0, [r1], #12; writes 0xABCDEF to address ;0x200 and updates r1



Another Example Using Post Indexed

;Memory write to address 0x8000

STR r3, [r8], #4

**r3 content
before/after STR**

0xFEEDBABE

r8 before STR

0x00008000

r8 after STR

0x00008004

Memory after STR

Address Data

0x8003

0xFE

0x8002

0xED

0x8001

0xBA

0x8000

0xBE

More Post-Indexed Examples

LDR | STR{<cond>} <Rd>, [<Rn>, <offset>]

;32 bit data in r7 written to ea(r0) and
;r0 is updated to contain $r0 \leftarrow r0 + 24$ after write
STR r7, [r0], #24

;r2 gets 32 bits of data from address beginning at
;ea(r0) and r0 is updated to contain $r0 \leftarrow r0 + (r4/16)$
LDR r2, [r0], r4, ASR #4

;r3 gets 16 bits of data from address beginning at
;ea(r9) and r9 is updated to contain $r9 \leftarrow r9 + 2$
LDRH r3, [r9], #2

;16 bit data written from r2 to address beginning at
;ea(r5) and r5 is updated to contain $r5 \leftarrow r5 + 8$
STRH r2, [r5], #8

More Complex Post-Indexed Examples

- To auto-increment the base register to location 0x1F4 use

STR r0, [r1], #-12 ;r1=0x200 initially

- If r2 contains 3, auto-increment base register to 0x20C by multiplying r2 by 4

STR r0, [r1], r2, LSL #2

Assembler Operators

- Primitive operations can be performed on data during assembly process
- A:MOD:B A modulo B
- A:ROL:B Rotate A left by B bits
- A:ROR:B Rotate A right by B bits
- A:SHL:B Shift A left by B bits
- A:SHR:B Shift A right by B bits
- A+B Add A to B
- A-B Subtract B from A
- A:AND:B Bitwise AND of A and B
- A:EOR:B Bitwise Exclusive OR of A and B
- A:OR:B Bitwise OR of A and B

Examples

- `ORR r1, r1, #1:SHL:3`
;set bit 3 of r1
;1 shifted left three times is binary 1000
- `DCD (0x8321:SHL:4):OR:2`
;is equivalent to `DCD 0x83212`
- `MOV r0, #((1:SHL:14):OR:(1:SHL:12))`
;is equivalent to
;MOV r0, #0x5000

Example Workings

DCD (0x8321:SHL:4):OR:2

Hexadecimal

Binary

8	3	2	1
1000	0011	0010	0001

Shift left by 4 bits

				0000
Or				0000000000000000000000010
1000	0011	0010	0001	0010
8	3	2	1	2

Example Workings

MOV r0, #((1:SHL:14) :OR: (1:SHL:12))

(1:SHL:14) => shifting 1 14 bits to the left

In binary notation

=> 100 0000 0000 0000

(1:SHL:12) => 001 0000 0000 0000 OR

 101 0000 0000 0000

Hexadecimal 5 0 0 0

Endian

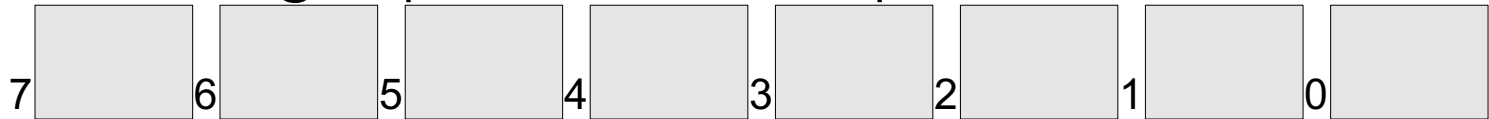
- Term arises from paper D. Cohen [1981]
- ARM supports both conventions
- Only arises with systems that have smaller-sized memory storage than wordsize
- Should storage be from “left-to-right” or “right-to-left”?
- Intel x86 uses “right-to-left”, Motorola 68X (now FreeScale) uses “left-to-right”
- ARM allows for either
- Core has input pin BIGEND, when asserted, results in Big Endian

Effect Of Endianess

- The ARM can be set up to access its data in either little or big-endian format.
- Little Endian: - Least significant byte of a word is stored **bits 0-7** of an addressed word.
- Big Endian: - Least significant byte of a word is stored **bits 24-31** of an addressed word.
- This has no real relevance unless data is stored in words and then accessed in smaller sized quantities (halfwords or bytes).
- Which byte / halfword accessed will depend on endianness of the system involved.

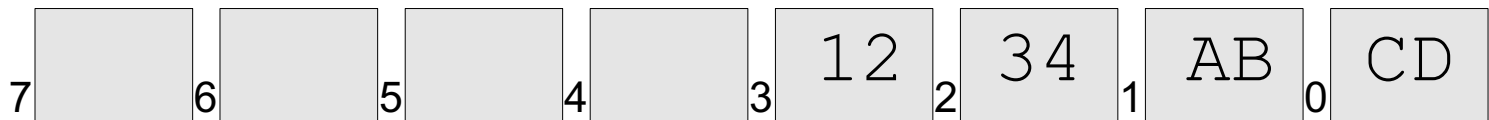
Big And Little Endian

Take this example (0x1234ABCD) and **store it into a memory array**. Our memory array is just a big block of byte-wide storage spaces, with sequential addresses:

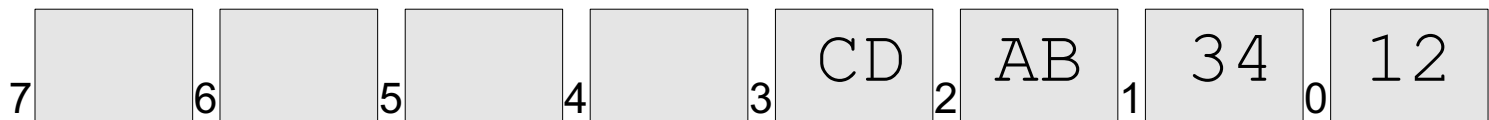


Starting from address 0, the number can go into memory in 2 ways,

Little-endian:



and big-endian

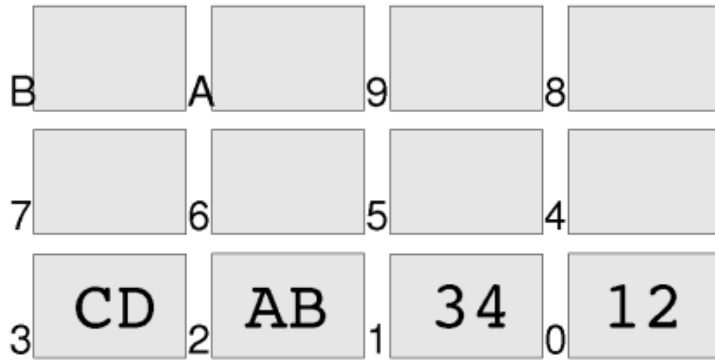


Big And Little Endian

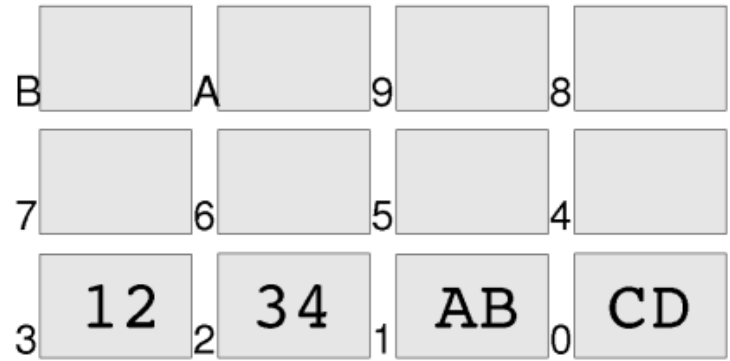
Sometimes **the way we draw our memory array** is different (i.e. for 32bit machines, we draw memory in a different way).

We can apply our example (0x1234ABCD) to both:

32-bit memory, big-endian



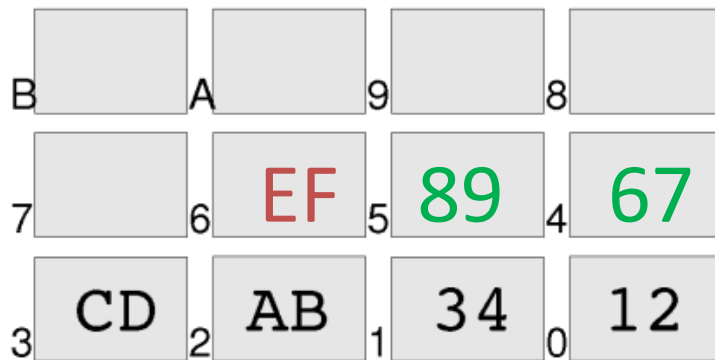
32-bit memory, little-endian



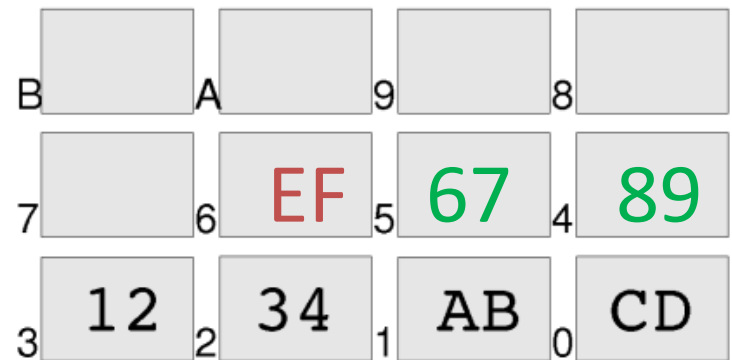
Big And Little Endian

To make things more interesting, let's store a halfword (0x6789) and a byte (0xEF) right after the 32bit value:

32-bit memory, big-endian



32-bit memory, little-endian



Summary

- Load/Store Architecture
- Load/Store Instructions
- Pre-index and Post-index addressing
- Endianess