Loads, Stores, and Addressing

EE3002/ IM2003 Microprocessor Part 1

Chapter 5 of textbook

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})
- ARM7TDMI 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**oa**D** words from memory into a **R**egister
- STR: **ST**ore 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 r1content is known as the ‘base address’, and r1 is called the **base address register**.

- Can be further extended to:

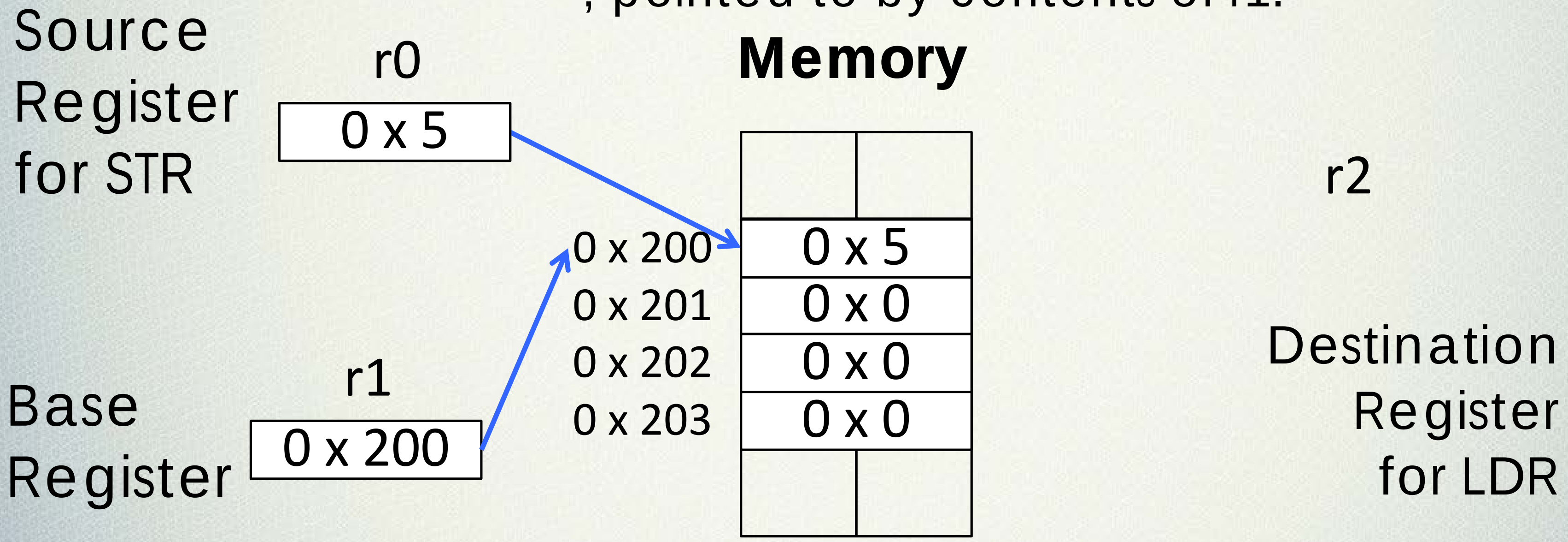
a) Pre-indexed addressing

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.



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

0xFFFFFFEE

Memory

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

More Register-Indirect Examples

```
 LDR      r0, [r1]
; load r0with the content of the memory location
; pointed to in r1

 STR      r2, [r1]
; store content of r2to 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 r0with 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-index 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 Examples

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

;r3 data written to ea(r0+(r5*8)) STR

r3, [r0, r5, LSL #3]

;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*4))

LDR r0, [r1, -r2, LSL #2]

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 r1by 4.

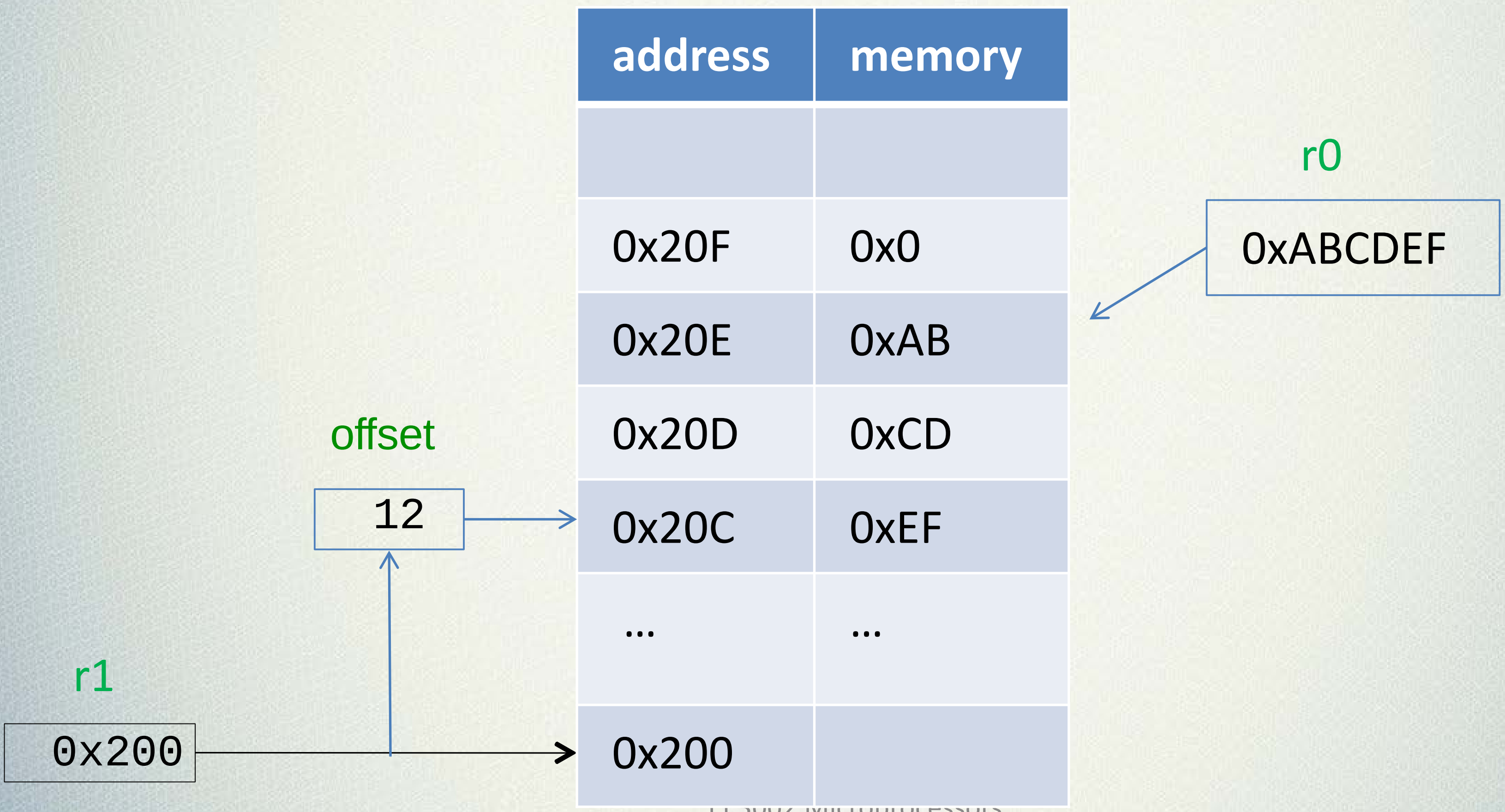
Increment to the base address is done <u>before</u> the execution of the load instruction.

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

Pre-Indexed Example

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

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



address	memory
0x20F	0x0
0x20E	0xAB
0x20D	0xCD
0x20C	0xEF
…	…
0x200	

STR Pre-indexed Addressing

- To store to location 0x1f4instead use

– STR r0, [r1,#-12]

- To auto-increment base pointer to 0x20cuse

– STR r0, [r1, #12]!

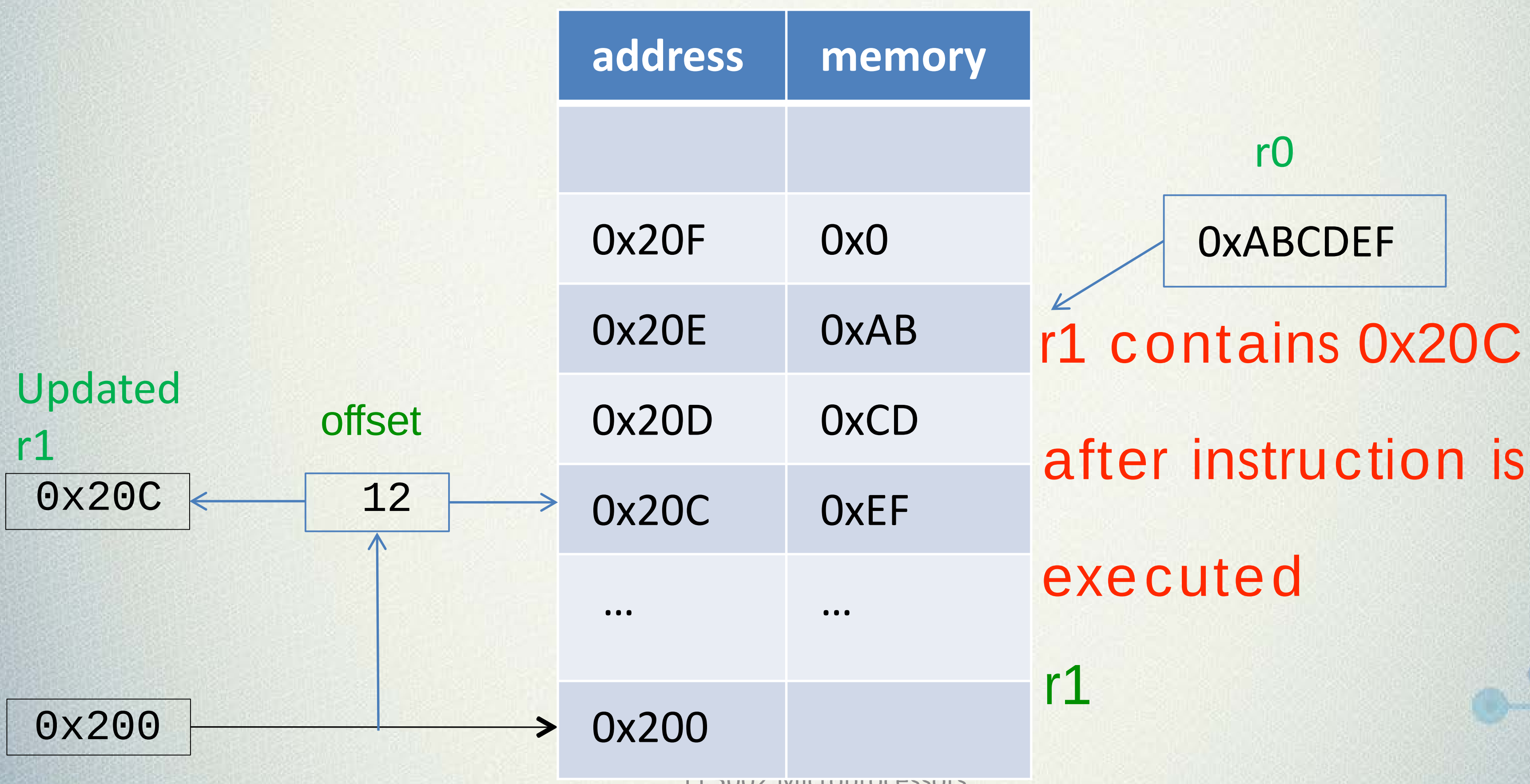
- If r2contains 3, access 0x20cby multiplying this by 4

– STR r0, [r1, r2, LSL #2]

Pre-Indexed Example with Writeback

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

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



address	memory
0x20F	0x0
0x20E	0xAB
0x20D	0xCD
0x20C	0xEF
…	…
0x200	

r1 contains 0x20C after instruction is executed

Pre-Indexed 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]!
```

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 <u>after</u> 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 Examples

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

;32 bit data in r7 written to ea(r0) and
;r0 is updated to contain r0 <- 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 <- r0+(r4/16) LDR r2, [r0], r4, ASR #4

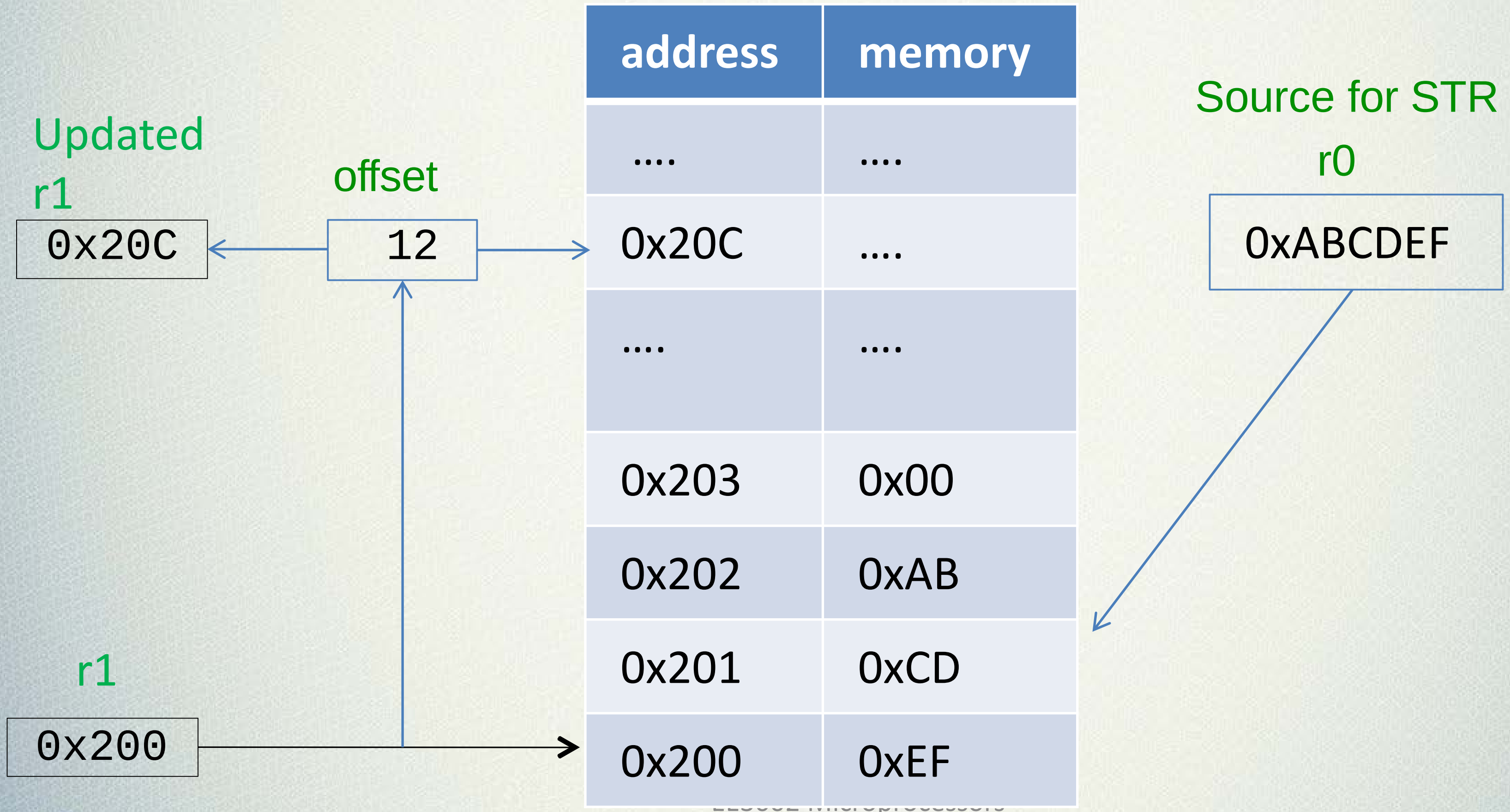
;r3 gets 16 bits of data from address beginning at
;ea(r9)and r9 is updated to contain r9 <- r9+2 LDRH r3, [r9], #2

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

Post-Indexed Example

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

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



address	memory
….	….
0x20C	….
….	….
0x203	0x00
0x202	0xAB
0x201	0xCD
0x200	0xEF

STR: Post-indexed Addressing

```
STR     r0, [r1], #12          ;writes 0x5 to address 0x200
```

- To auto-increment the base register to location 0x1f4instead use:
  – STR r0, [r1], #-12 ;r1=0x200
- If r2contains 3, auto-increment base register to 0x20cby multiplying this by 4:
  – STR r0, [r1], r2, LSL #2

Store Example Using Post Indexed

```
STR      r3, [r8], #4 ;Memory write to address 0x8000
```

r3 content before/after store

0xFEEDBABE

r8 before store

0x00008000

r8 after store

0x00008004

Memory after Store

Address	Data
0x8000	0xBE
0x8001	0xBA
0x8002	0xED
0x8003	0xFE

More Examples

```
LDR     r5, [r3]                ;load r5 with data from
                                ;ea <r3>
STRB    r0, [r9]                ;store data in r0 to
                                ;ea<r9>
STR     r3, [r0, r5, LSL #3]              ;store data in r3
                              ;ea<r0+r5*8>
LDR     r1, [r0, #4]!      ;load r1 from ea<r0+4>,
                                ;r0<-r0+4
STRB          r7, [r6, #-1]! ;store byte to ea<r6-1>,
                                ;r6<-r6-1
LDR     r3, [r9], #4            ;load r3 from ea<r9>,
                                ;r9<-r9+4
STR     r2, [r5], #8            ;store word to ea<r5>,
                                ;r5<-r5+8
```

Example - Memory String Copy

```
SRAM_BASE  EQU        0x40000000      ;address to store string
           AREA       StrCopy, CODE
           ENTRY                      ;mark first instruction
Main       adr     r1, srcstr         ;pointer to source string
           ldr     r0, =SRAM_BASE     ;pointer to destination string
strcopy
           ldrb      r2,  [r1],       ;load byte, update address
           #1  strb       r2,         ;store byte, update address
           [r0],     #1   cmp         ;check for zero terminator
           r2, #0                     ;loop if terminator not reached
stop       bne                        ;halt processing, loop forever
srcstr    DCB strcopy“This is my (source) string”,0
          END b             ;end of file marker
           stop
```

Endian

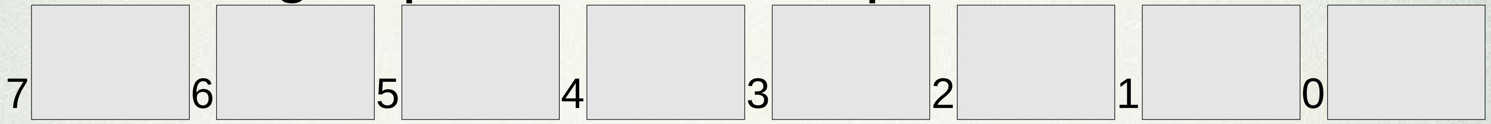
- Term arises from paper D. Cohen [1981]
- ARM supports both conventions
- Only arises with systems that have smaller-sized memory storage than wordsize
- Should storage be from “left-to-right” or “right-to- left”?
- Intel x86 uses “right-to-left”, Motorola 68X (now FreeScale) uses “left-to-right”
- 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 endianess of the system involved.

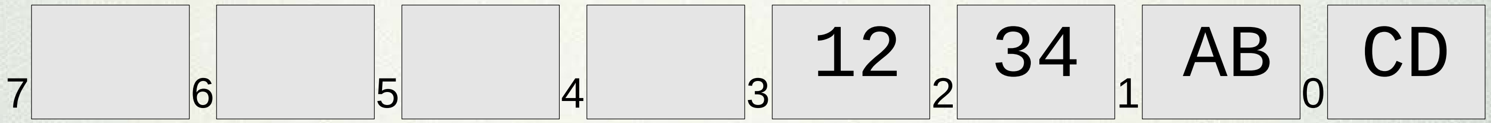
Big And Little Endian

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

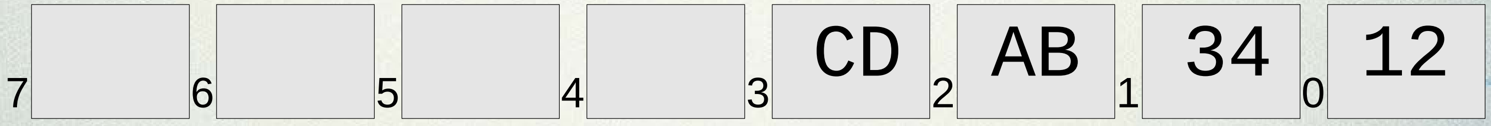


Starting from address 0, the number can go into memory in 2 ways,

littleendian:



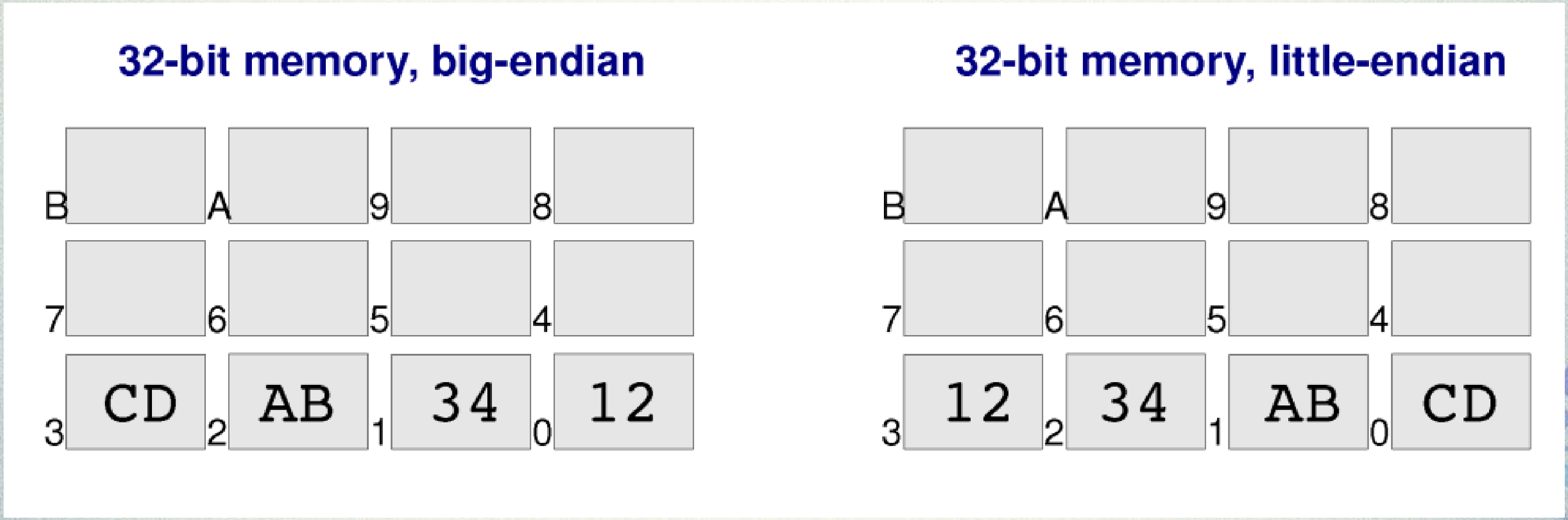
and bigendian



Big And Little Endian

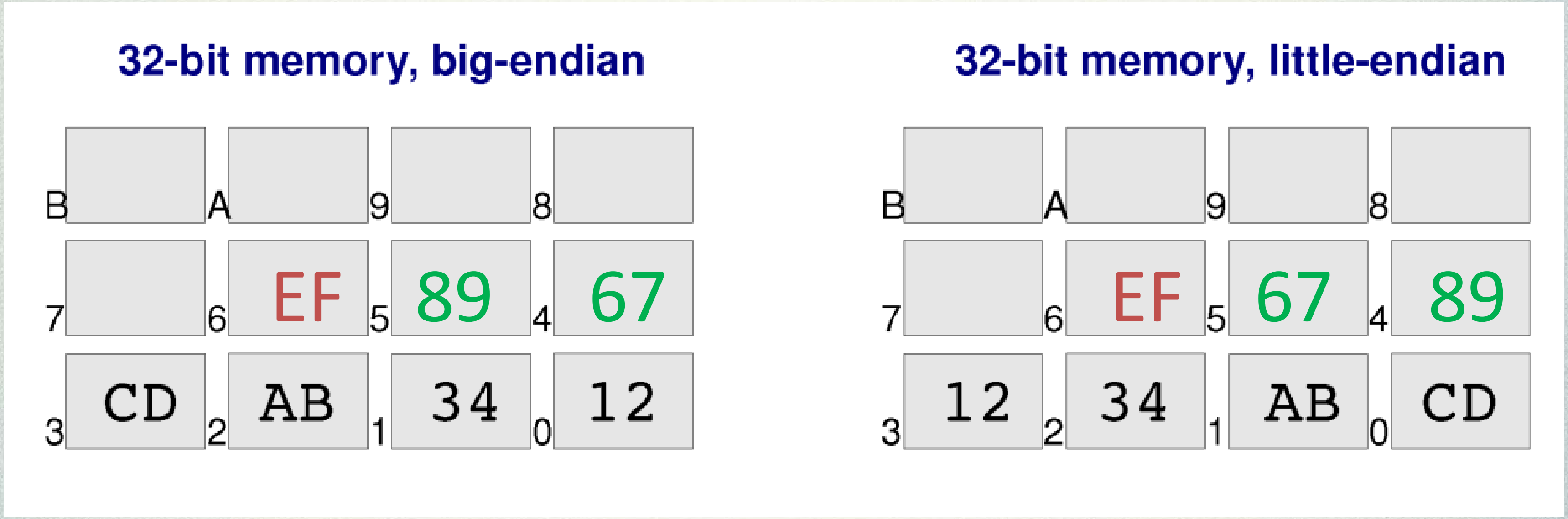
Sometimes the way we draw our memory array is different (i.e. for 32bit machines, we draw memory in a different way).

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



Big And Little Endian

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



Changing Endianness

```
•  ; On entry    : R0 holds the word to be swapped
•  ;On exit      : R0 holds the swapped word
•  ;:R1, R2 and R3 are destroyed
•  Byteswap    ; first three instruction does initialisation
•        MOV R2,  #0xff  ;R2 = 0xff
•        ORR R2,  R2, #0xff0000  ;R2 = 0x00ff00ff
•        MOV R3, R2, LSL #8 ;R3 = 0xff00ff00
•   ; repeat the following code for each word to swap
•                ;R0 = W X Y Z
•         AND R1, R2, R0, ROR #24 ;R1 =  0  Y 0 W
•         AND R0, R3, R0, ROR #08 ;R0 =  Z  0 X  0
•         ORR  R0, R0, R1 ;R0 =  Z  Y X W
```

Summary

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