CSE 331: Microprocessor Interfacing and Embedded Systems

Assembly Language

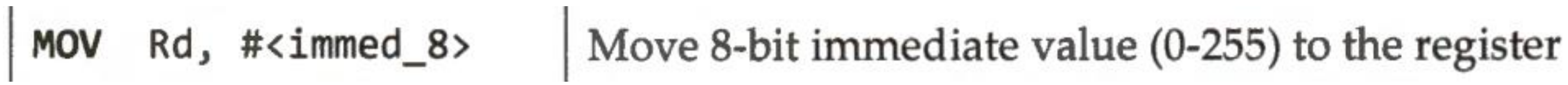
Arithmatic and logic operation

Load and store

https://cpulator.01xz.net/?sys=arm-de1soc

Load and Store [ch 5]

Load Constant into Registers



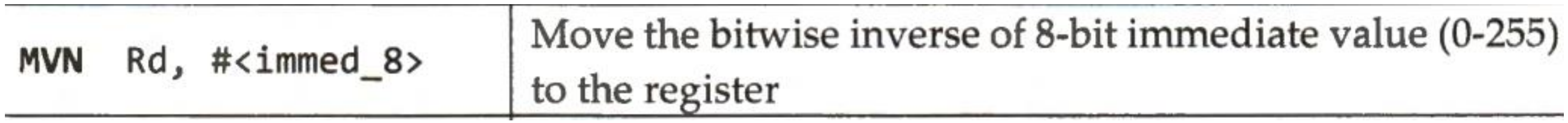
`MOV Rd, #<immed_8>`	Move 8-bit immediate value (0-255) to the register

- Loads the data into the reg Rd.
- Maximum of 8 bits can be loaded in this way

- MOV R0, #42 ; Load decimal 42 into R0
- MOV R1, #0xFF ; Load 255 (hex FF)
- MOV R2, #0b101010 ; Load binary 101010 = decimal 42
- MOV R3, #'A' ; Load ASCII value of 'A' = 65
- MOV R4, #0x41 ; Same as 'A'
- MOV R5, #' ' ; Space character (ASCII 32)

Load and Store

Load Constant into Registers



`MVN Rd, #<immed_8>`	Move the bitwise inverse of 8-bit immediate value (0-255) to the register

- Moves Bitwise inverse or 1's complement of the constant
- Maximum of 8 bits can be loaded in this way

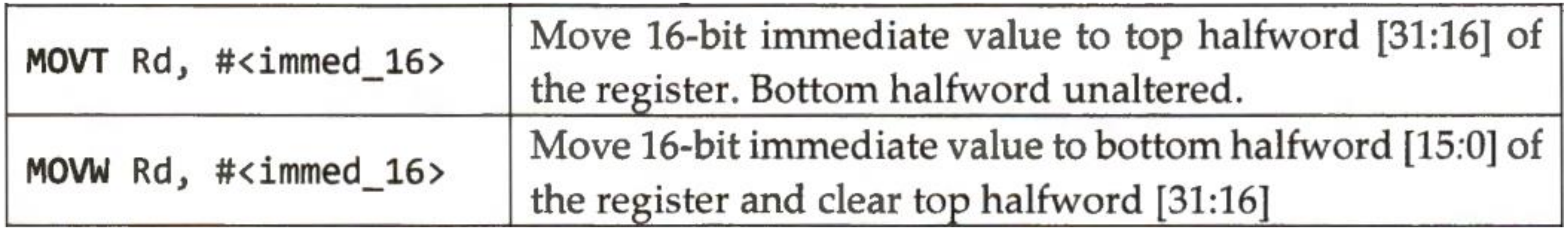
```
MVN R0, #0x00
R0 = 0xFFFFFFFF

MVN R1, #0xFF
R1 = 0xFFFFFF00
```

```
MOV R2, #0b10101010 ; R2 = 0xAA
MVN R3, R2 ; R3 = NOT R2
R3 = 0xFFFFFF55
```

- What if we want to move some negative values in the reg?
- What happens to the flags?
- What is being stored?
- What if we want to store more than 8 bits?

Load and Store

Load Constant into Registers



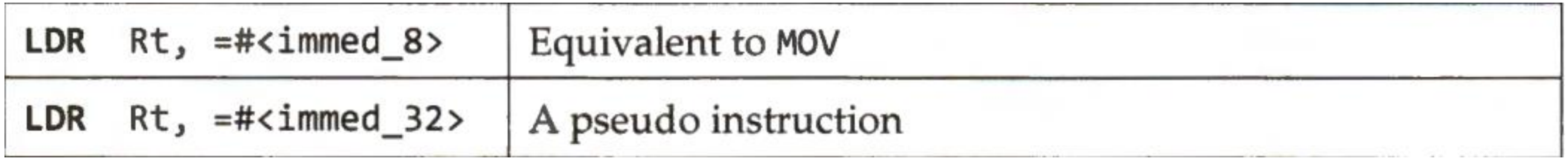
`MOVT Rd, #<immed_16>`	Move 16-bit immediate value to top halfword [31:16] of the register. Bottom halfword unaltered.
`MOVW Rd, #<immed_16>`	Move 16-bit immediate value to bottom halfword [15:0] of the register and clear top halfword [31:16]

- Moves 16 bits of data into the regs
- Can be used to set the values of the bottom and top half separately

```
MOVW R0, #0x5678      ; Load lower 16 bits
MOVT R0, #0x1234      ; Load upper 16 bits
R0 = 0x12345678
```

Load and Store

Load Constant into Registers



`LDR Rt, =#<immed_8>`	Equivalent to MOV
`LDR Rt, =#<immed_32>`	A pseudo instruction

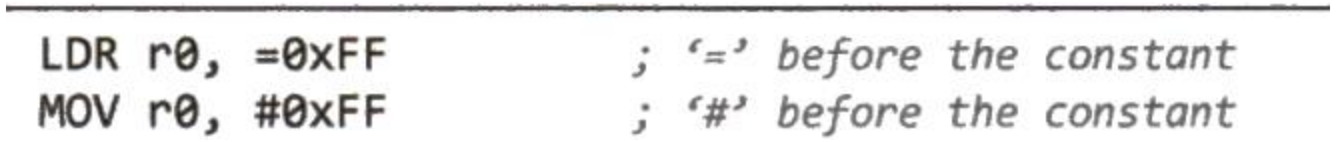
- LDR can both be a pseudo instruction and a real machine instruction that loads a word from memory.
- A pseudo instruction is an instruction that is available to use in an assembly program, but not directly supported by the microprocessor.
- Compilers translate it to one or multiple actual machine instructions when the assembler builds the program into an executable.
- LDR is a pseudo instruction when loading a constant
- LDR is a real machine instruction when accessing data from memory

Load and Store

Load Constant into Registers

- LDR can load a 32 bit constant into the reg
- MOV can only load a 12-bit constant into a register.

- If the constant number is 12-bit , compilers translate the LDR pseudo instruction to MOV or MVN.
- Otherwise, compilers translate it into a regular LDR instruction that uses PC-relative memory address.
- Note the syntax for specifying the constant number in MOV and LDR are different.

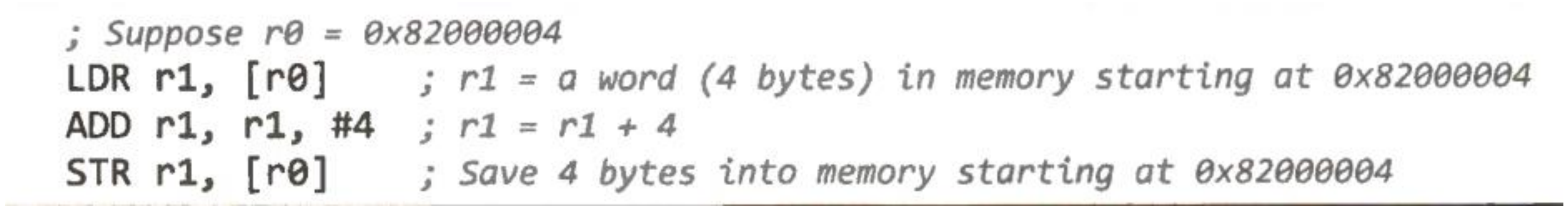


```
LDR r0, =0xFF          ; '=' before the constant
MOV r0, #0xFF          ; '#' before the constant
```

Load and Store

Accessing Data in Memory

- When an assembly program accesses data in memory, the memory address must be in a register.
- A store instruction does the opposite.
- It saves the content of a register to the memory at a given memory address.



```
; Suppose r0 = 0x82000004
LDR r1, [r0]      ; r1 = a word (4 bytes) in memory starting at 0x82000004
ADD r1, r1, #4    ; r1 = r1 + 4
STR r1, [r0]      ; Save 4 bytes into memory starting at 0x82000004
```

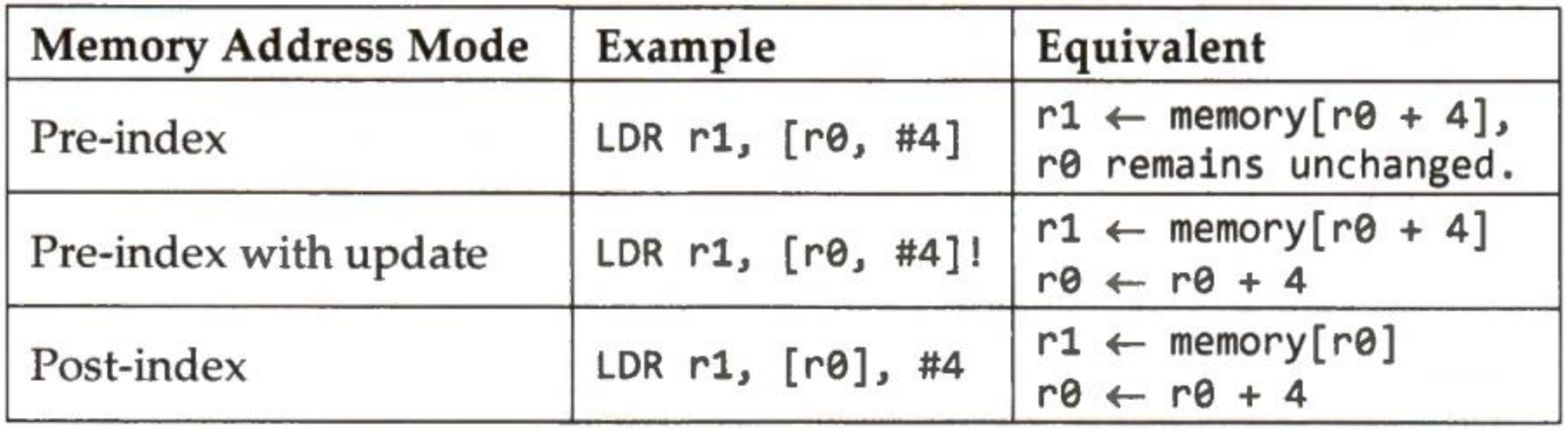
Load and Store

Accessing Data in Memory

```
.global _start
_start:
   LDR R0, =list
   LDR R1, [R0]


.data
list:
   .word 4, 5, 6, -1, -2, 9
```

Load and Store

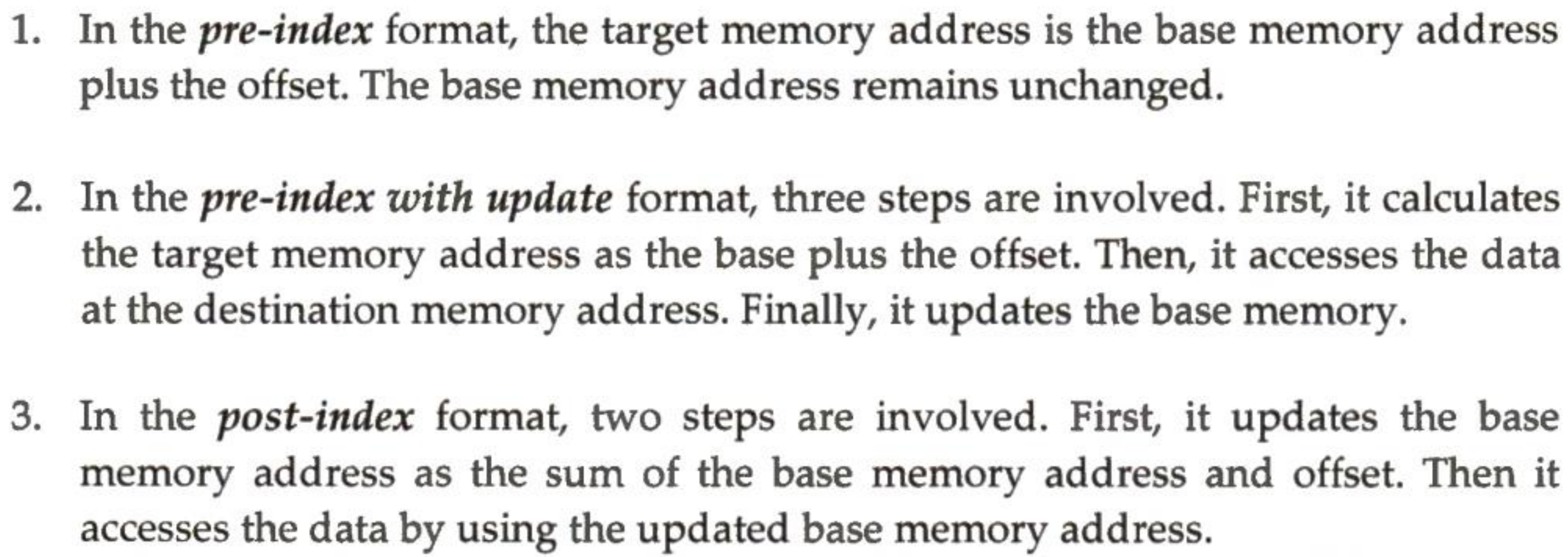
Accessing Data in Memory



Memory Address Mode	Example	Equivalent
Pre-index	`LDR r1, [r0, #4]`	`r1 ← memory[r0 + 4],` `r0 remains 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`

Load and Store

Accessing Data in Memory



1. In the ***pre-index*** format, the target memory address is the base memory address plus the offset. The base memory address remains unchanged.

2. In the ***pre-index with update*** format, three steps are involved. First, it calculates the target memory address as the base plus the offset. Then, it accesses the data at the destination memory address. Finally, it updates the base memory.

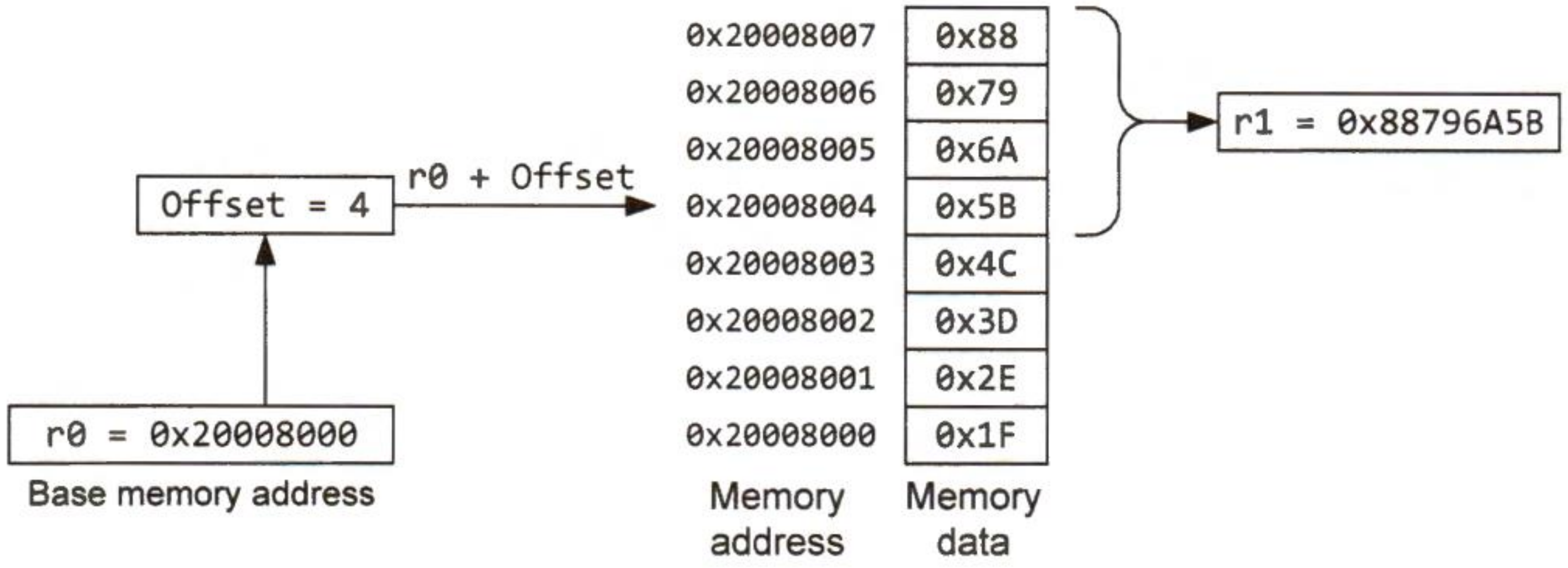
3. In the ***post-index*** format, two steps are involved. First, it updates the base memory address as the sum of the base memory address and offset. Then it accesses the data by using the updated base memory address.

Load and Store

Accessing Data in Memory

```
LDR r1, [r0, #4]  ; Pre-index
```

As shown in Figure 5-3, register `r0` remains unchanged. After loading the word stored at memory address `0x20008004`, the data in register `r1` is `0x88796A5B`.

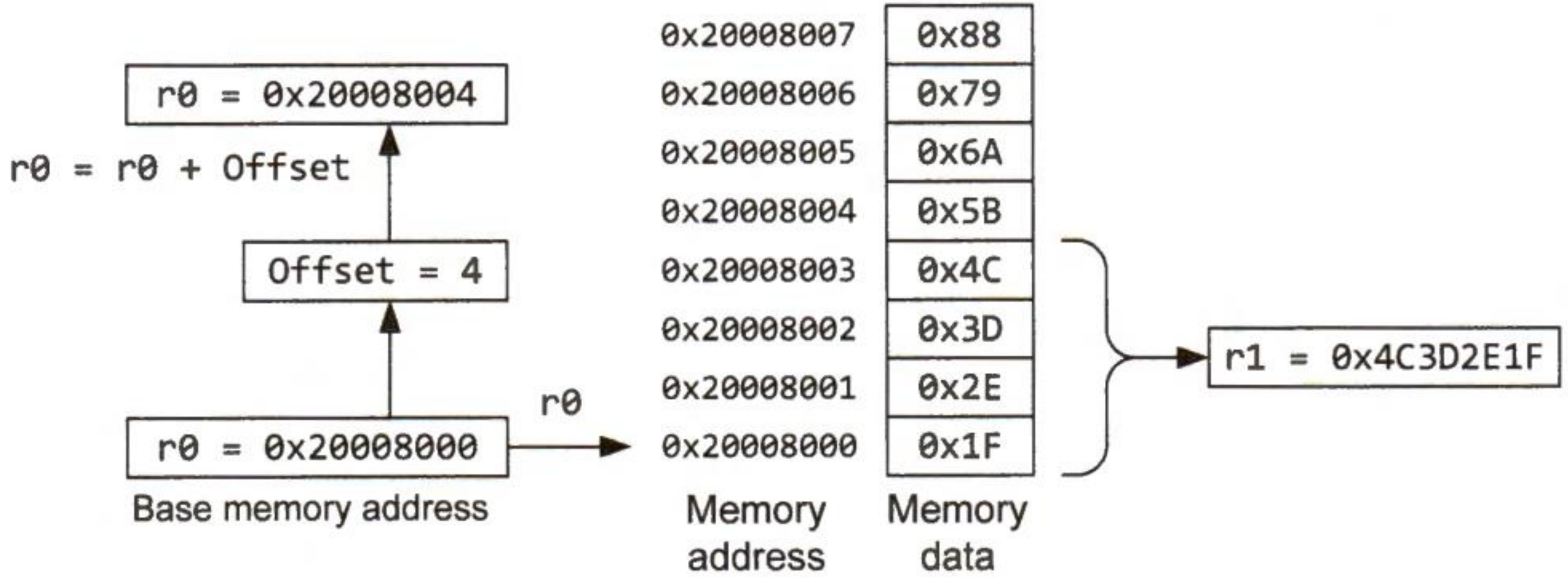


Load and Store

Accessing Data in Memory

```
LDR r1, [r0], #4  ; Post-index
```

As shown in Figure 5-4, the value in register `r0` is incremented by the offset after loading. Register `r1` is fetched from the memory address `0x20008000`.

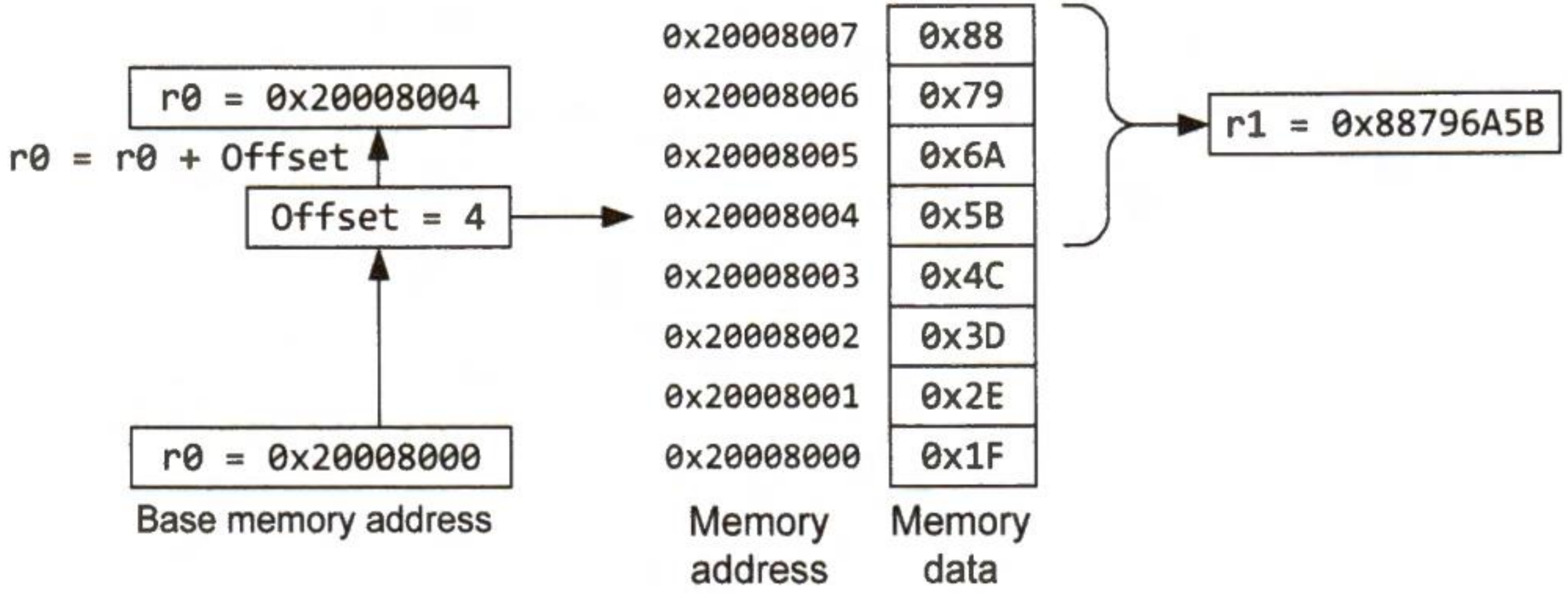


Load and Store

Accessing Data in Memory

```
LDR r1, [r0, #4]!    ; Pre-index with update
```

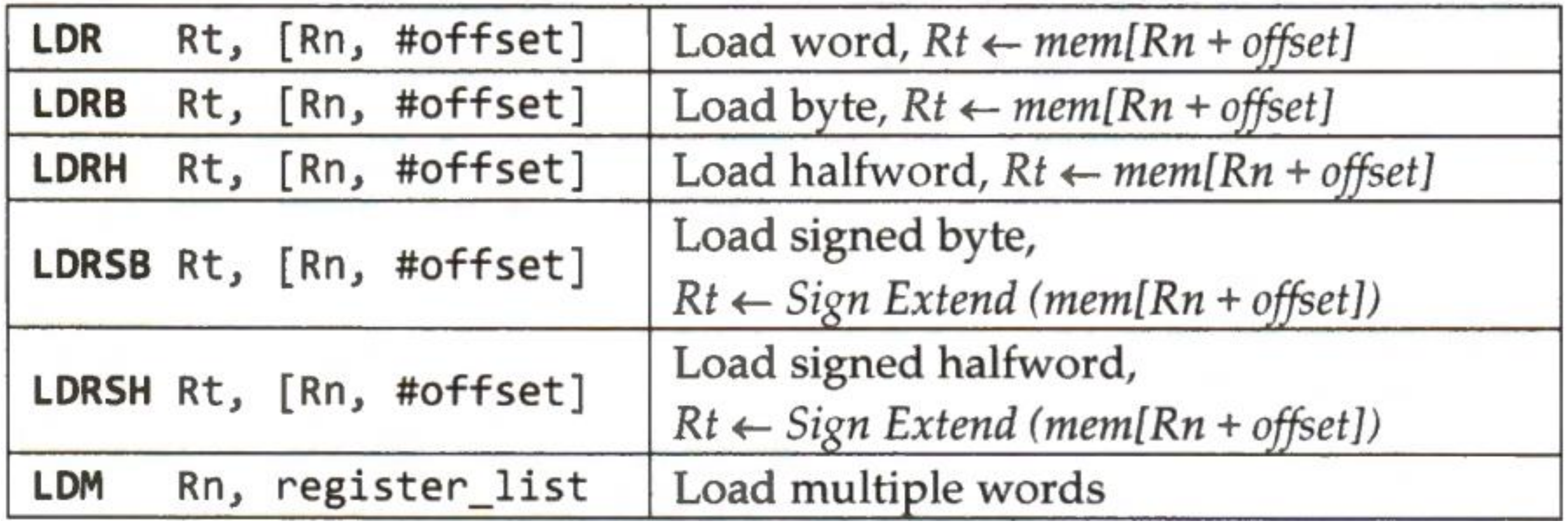
As shown in Figure 5-5, the value in register `r0` is incremented by the offset after loading. Both the post-index and the pre-index with update change the base memory address. However, different with the post-index, the pre-index with update retrieves the word from the memory address `0x20008004`, instead of `0x20008000`.



Load and Store

Accessing Data in Memory

What if we want to load only a byte or a half word?



`LDR    Rt, [Rn, #offset]`	Load word, $Rt \leftarrow mem[Rn + offset]$
`LDRB   Rt, [Rn, #offset]`	Load byte, $Rt \leftarrow mem[Rn + offset]$
`LDRH   Rt, [Rn, #offset]`	Load halfword, $Rt \leftarrow mem[Rn + offset]$
`LDRSB  Rt, [Rn, #offset]`	Load signed byte, $Rt \leftarrow Sign\ Extend\ (mem[Rn + offset])$
`LDRSH  Rt, [Rn, #offset]`	Load signed halfword, $Rt \leftarrow Sign\ Extend\ (mem[Rn + offset])$
`LDM    Rn, register_list`	Load multiple words

- Only the pre-index format is shown.
- Load and store instructions with the other two formats are similar.

Load and Store

Accessing Data in Memory

Difference between **LDRB** and **LDRSB**

LDRB R0, [R1]

- Suppose memory at `[R1]` = `0xFF` (255 in decimal)
- R0 = `0x000000FF` → **zero-extended**

LDRSB R0, [R1]

- Suppose memory at [R1] = 0xFF → interpreted as -1 in 8-bit signed
- R0 = 0xFFFFFFFF → sign-extended

Load and Store

Accessing Data in Memory

How does LDM work

LDM R1, {R0, R2, R3}

- Loads 3 words (32 bits each) from memory starting at address in R1
- Stores them into R0, R2, and R3 respectively
- Does **not** change R1

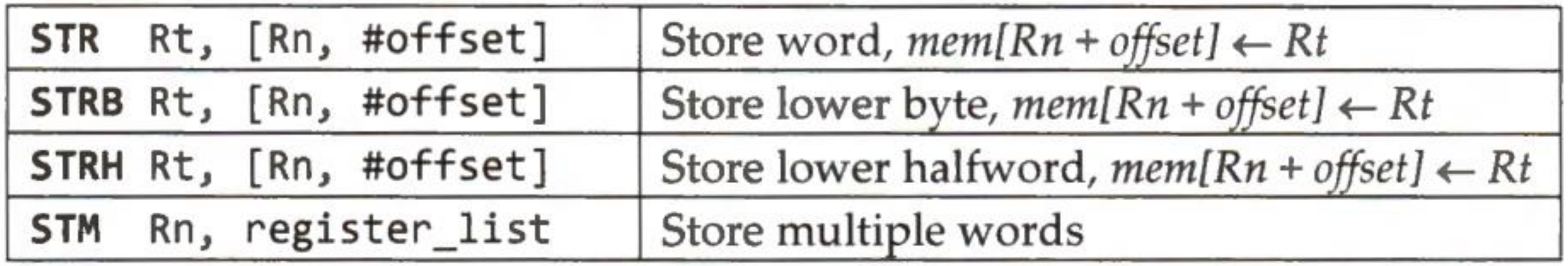
LDM R1!, {R4, R5}

- How does it work?
- Loads two 32-bit values from [R1] and [R1+4] into R4 and R5
- Then updates R1 = R1 + 8 (2 registers × 4 bytes)

Load and Store

<u>Storing Data in Memory</u>

Store value of a register in memory



`STR  Rt, [Rn, #offset]`	Store word, $mem[Rn + offset] \leftarrow Rt$
`STRB Rt, [Rn, #offset]`	Store lower byte, $mem[Rn + offset] \leftarrow Rt$
`STRH Rt, [Rn, #offset]`	Store lower halfword, $mem[Rn + offset] \leftarrow Rt$
`STM  Rn, register_list`	Store multiple words

Load and Store

Storing Data in Memory

```
LDR R4, =buffer
LDR R0, =0x12345678
STR R0, [R4]        ; Stores 4 bytes starting at address buffer[0]

Result:
78 56 34 12   ; Little-endian

LDR R0, =0x12345678
STRB R0, [R4, #4] ; Stores only the lowest byte (0x78) at buffer[4]
Result:
78 at [buffer+4] location

MOV R0, #0x12345678
STRH R0, [R4, #6] ; Stores 0x5678 at buffer[6]
Result:
78 56
```

Load and Store

Storing Data in Memory

STM example

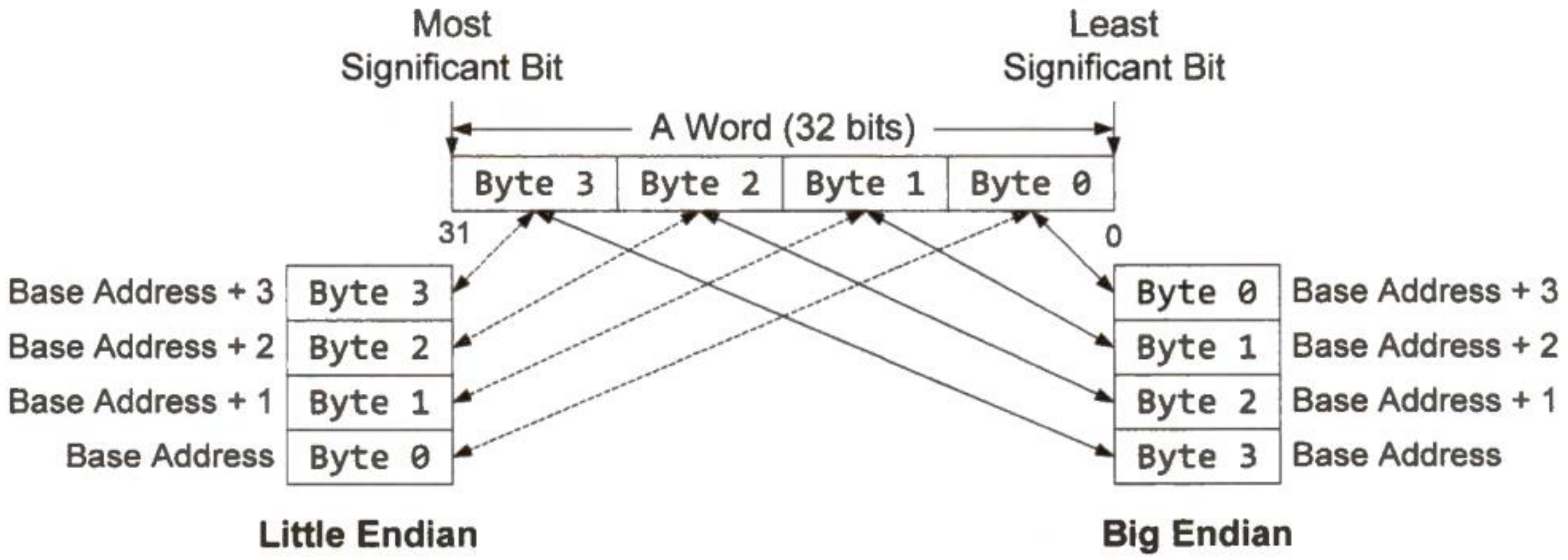
STM R1, {R2, R3}
- Stores R2 to `[R1]`
- Stores R3 to `[R1 + 4]`
- Does **not** change R1

STM R1!, {R4, R5}
- Stores:
- R4 → `[R1]`
- R5 → `[R1 + 4]`
- Then updates `R1 = R1 + 8` (2 registers × 4 bytes)

Load and Store

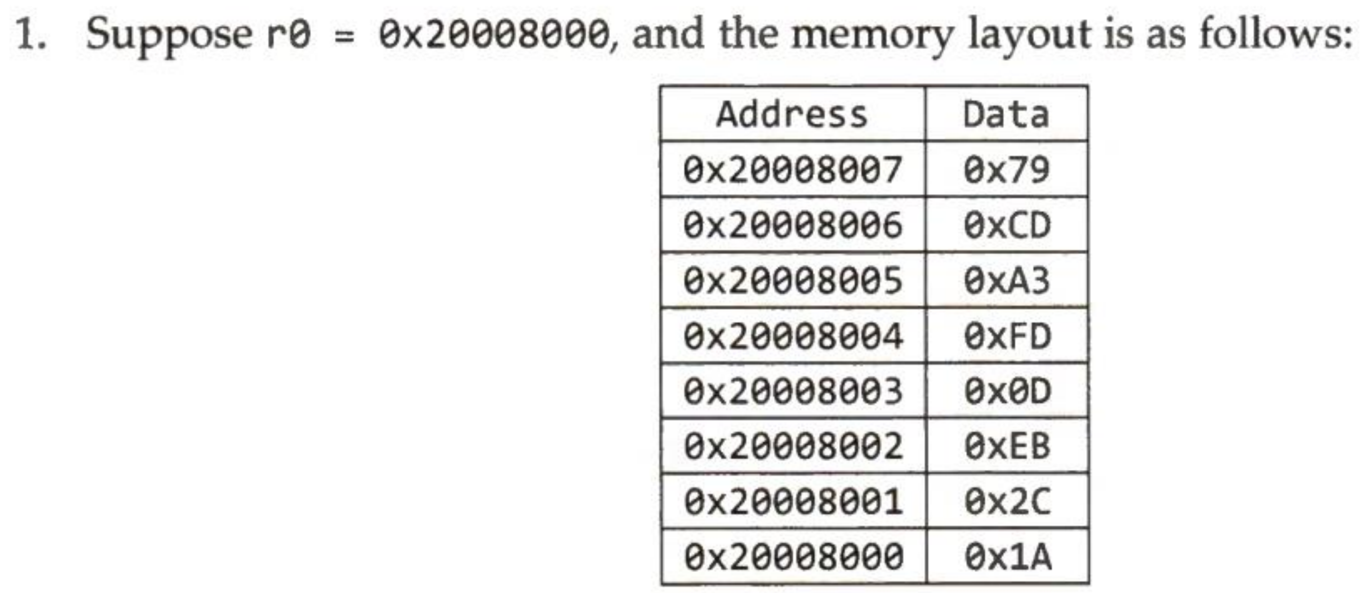
Storing Data in Memory

- Little endian means the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.)
- Big endian means the high-order byte of the number is stored at the lowest address, and the low-order byte at the highest address. (The big end comes first.)



Load and Store

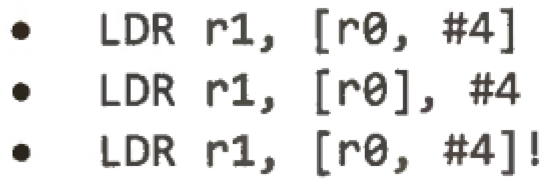
Excercise



1. Suppose `r0 = 0x20008000`, and the memory layout is as follows:

Address	Data
0x20008007	0x79
0x20008006	0xCD
0x20008005	0xA3
0x20008004	0xFD
0x20008003	0x0D
0x20008002	0xEB
0x20008001	0x2C
0x20008000	0x1A

- What are the values of R1 and R0 if followings are run separately?



```
•   LDR r1, [r0, #4]
•   LDR r1, [r0], #4
•   LDR r1, [r0, #4]!
```

Load the given data into the memory and analyze where they are getting stored:

solution:

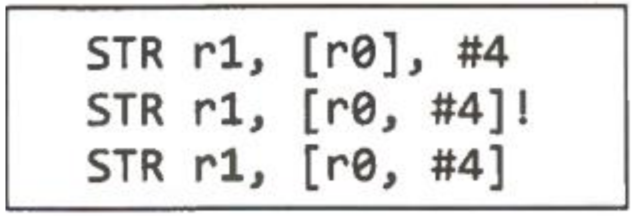
```
.global _start
_start:
   LDR r0, =list

.data
list:
.byte   0x1A,0x2C,0xEB,0x0D,0x
FD,0xA3, 0xCD,0x79
```

Load and Store

Excercise

Suppose r0 = 0x20000000 and r1 = 0x12345678. All bytes in memory are initialized to 0x00. Suppose the following assembly program has run successfully. Draw a table to show the memory value.

Write a program that loads these values into r0 and r1.
Then execute these following code to see the memory maps.



```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, #4]
```

```
LDR R0, =#0x20000000
LDR R1, =#0x12345678

STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, #4]
```

Load and Store

Excercise

What is the value in register r1 in the following instructions if r0 = 0x20008000? Assume the system is little endian.

```
(1) LDRSB r1, [r0]
(2) LDRSH r1, [r0]
(3) LDRB  r1, [r0]
(4) LDRH  r1, [r0]
```

Memory address	Data
0x20008002	0xA1
0x20008001	0xB2
0x20008000	0xC3
0x20007FFF	0xD4
0x20007FFE	0xE5

Arithmetic and Logic

Program Status Register

- Cortex-M processors have five status flags: negative (N), zero (Z), overflow (V), carry © and saturation (Q).
- The negative flag (N) is set if the result of ALU is negative (i.e., bit[31] is 1) and is cleared otherwise.
- The zero flag (Z) is set if the ALU result is zero, and is cleared otherwise.
- The carry flag (C) is set if a carry occurs in unsigned addition, and is cleared otherwise. For unsigned subtraction, it is set if no borrow has occurred, and is cleared otherwise.
- The overflow flag (V) is set if an overflow takes place when performing a signed addition or subtraction, and is cleared otherwise.
- The saturation flag (Q) is set if an SSAT or USAT instruction causes saturation, and is cleared otherwise.

Arithmetic and Logic

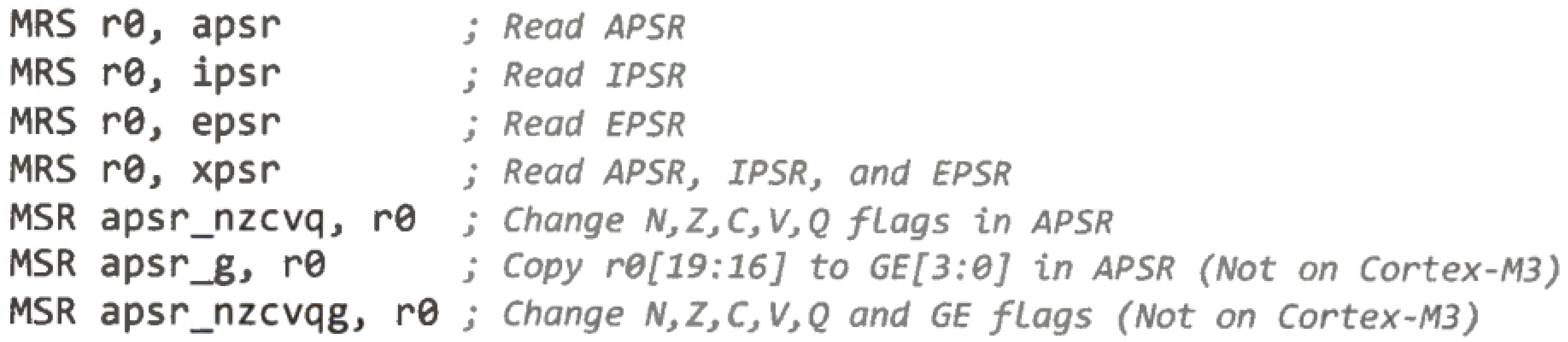
Program Status Register

- These flags are stored in the program status register (PSR)
- The PSR reg is a combination of three special registers.
- Application program status register (APSR)
- Interrupt program status register (IPSR)
- Execution program status register (EPSR)
- APSR, IPSR, and EPSR have no overlap in bit fields
- The processor combines them into one register PSR, or called xPSR,

Arithmetic and Logic

<u>Program Status Register</u>

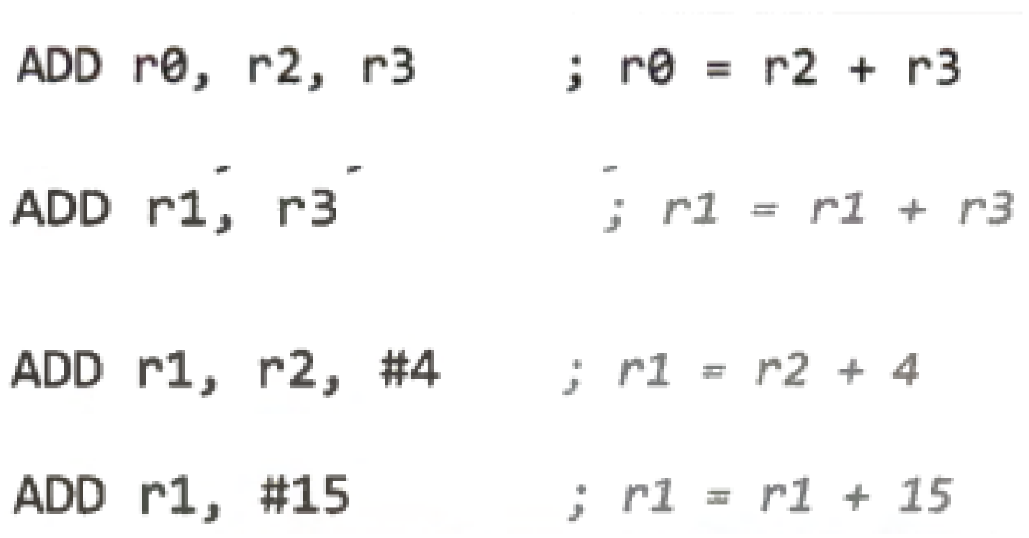
- These special registers can only be accessed by using two special instructions
- MSR (move from a general register to a special register).
- MRS (move from a special register to a general register)



```
MRS r0, apsr          ; Read APSR
MRS r0, ipsr          ; Read IPSR
MRS r0, epsr          ; Read EPSR
MRS r0, xpsr          ; Read APSR, IPSR, and EPSR
MSR apsr_nzcvq, r0    ; Change N,Z,C,V,Q flags in APSR
MSR apsr_g, r0        ; Copy r0[19:16] to GE[3:0] in APSR (Not on Cortex-M3)
MSR apsr_nzcvqg, r0   ; Change N,Z,C,V,Q and GE flags (Not on Cortex-M3)
```

Arithmetic and Logic

Addition

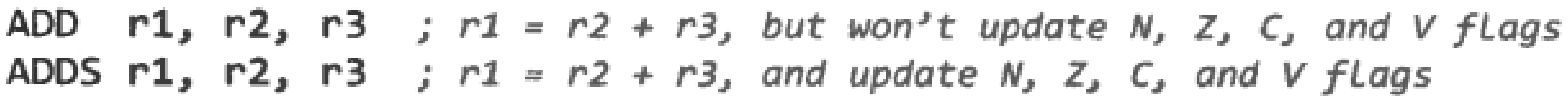


```
ADD r0, r2, r3      ; r0 = r2 + r3

ADD r1, r3           ; r1 = r1 + r3

ADD r1, r2, #4      ; r1 = r2 + 4

ADD r1, #15         ; r1 = r1 + 15
```

Arithmetic and Logic

Addition with updating PSR register

- If the S suffix is appended to an instruction mnemonic, the processor modifies the status flags ADD vs ADDS based on the computation result.



```
ADD  r1, r2, r3  ; r1 = r2 + r3, but won't update N, Z, C, and V flags
ADDS r1, r2, r3  ; r1 = r2 + r3, and update N, Z, C, and V flags
```

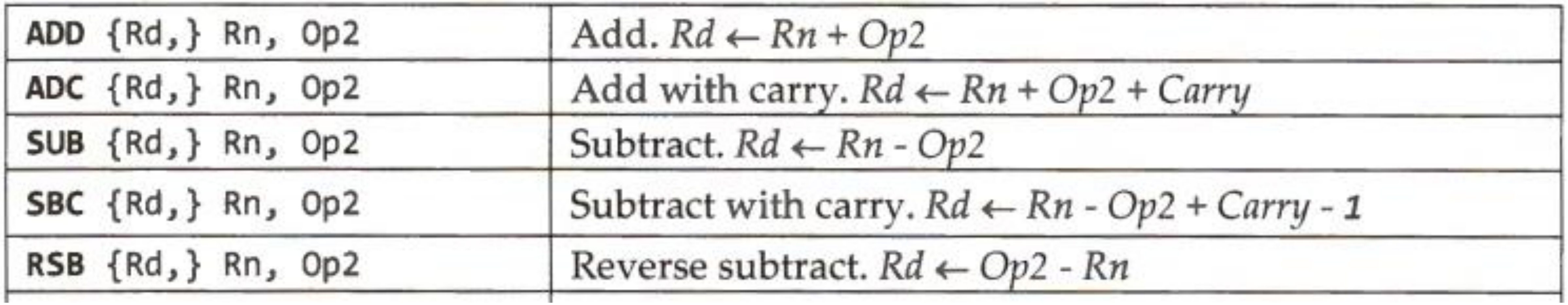
MOV R0, #1
MOV R1, #-1
ADD R2, R0, R1 ; R2 = 1 - 1 = 0
Flags not affected

MOV R0, #1
MOV R1, #-1; Use 2's complement
ADDS R2, R0, R1   ; R2 = 1 + (-1) = 0
Flags:

- **Z = 1** (result is zero)
- **N = 0**
- **C, V** = unaffected here

Arithmetic and Logic

Addition and Subtraction Operations



`ADD {Rd,} Rn, Op2`	Add. $Rd \leftarrow Rn + Op2$
`ADC {Rd,} Rn, Op2`	Add with carry. $Rd \leftarrow Rn + Op2 + Carry$
`SUB {Rd,} Rn, Op2`	Subtract. $Rd \leftarrow Rn - Op2$
`SBC {Rd,} Rn, Op2`	Subtract with carry. $Rd \leftarrow Rn - Op2 + Carry - 1$
`RSB {Rd,} Rn, Op2`	Reverse subtract. $Rd \leftarrow Op2 - Rn$

Arithmetic and Logic

Addition and Subtraction Operations

ADDC or **ADC** = **Add with Carry**:

ADC Rd, Rn, Operand2 ; Rd = Rn + Operand2 + Carry

- It **includes the current carry flag (C)** in the result.
- Used when doing **multi-word (e.g. 64-bit) addition** in a 32-bit system.

```
MOV R0, #0xFFFFFFFF       ; R0 = max 32-bit unsigned int
MOV R1, #1
ADD R2, R0, R1 ;          ;R2 = R0 + R1 = 0x00000000 (carry occurs, but ignored)

MOVS R0, #0xFFFFFFFF      ;To Set carry via overflow
ADDS R0, R0, #1           ; R0 = 0, C = 1

MOV R3, #0
ADC R4, R3, #5 ;           R4 = R3 + 5 + carry = 6
```

Arithmetic and Logic

Addition and Subtraction Operations

Remember during subtraction:

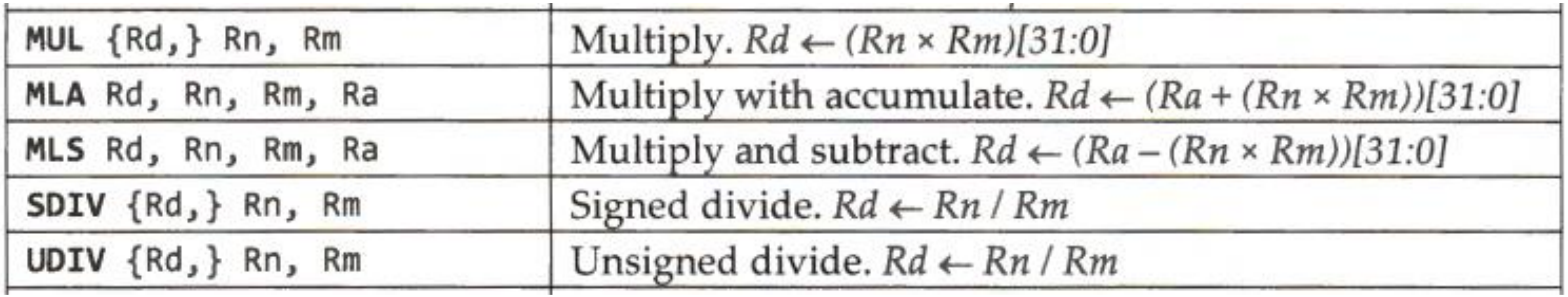
- **C = 1** means **no borrow**
- **C = 0** means **borrow occurred**

```
MOV R0, #10
MOV R1, #3
SUB R2, R0, R1        ; R2 = 10 - 3 = 7

; Assume carry is 1 (no previous borrow)
MOVS R0, #5
SUBS R0, R0, #2        ; R0 = 3, C = 1

MOV R1, #1
SBC R2, R0, R1         ; R2 = 3 - 1 - (1 - C) = 3 - 1 - 0 = 2
```

```
MOV R0, #3
MOV R1, #10
RSB R2, R0, R1 ; R2 = 10 - 3 = 7
```

Arithmetic and Logic

Multiplication and Division Operations



`MUL {Rd,} Rn, Rm`	Multiply. $Rd \leftarrow (Rn \times Rm)[31:0]$
`MLA Rd, Rn, Rm, Ra`	Multiply with accumulate. $Rd \leftarrow (Ra + (Rn \times Rm))[31:0]$
`MLS Rd, Rn, Rm, Ra`	Multiply and subtract. $Rd \leftarrow (Ra - (Rn \times Rm))[31:0]$
`SDIV {Rd,} Rn, Rm`	Signed divide. $Rd \leftarrow Rn / Rm$
`UDIV {Rd,} Rn, Rm`	Unsigned divide. $Rd \leftarrow Rn / Rm$

Arithmetic and Logic

Multiplication and Division Operations

MOV R1, #5 MOV R2, #3
MUL R0, R1, R2 ; R0 = R1 *
R2 = 5 * 3 = 15

- `MUL` computes **32-bit × 32-bit → 32-bit result**
- If result exceeds 32 bits, **higher bits are lost (truncated)**

Arithmetic and Logic

Multiplication and Division Operations

```
MOV R1, #2
MOV R2, #3
MOV R3, #5
MLA R0, R1, R2, R3   ; R0 = (2 * 3) + 5 = 11
```

- Only the **lower 32 bits** of the product are used.

Arithmetic and Logic

Multiplication and Division Operations

```
MOV R1, #2      ; Rm = 2
MOV R2, #3      ; Rs = 3
MOV R3, #10     ; Rn = 10
MLS R0, R1, R2, R3   ; R0 = R3 - (R1 × R2)
```

R0 = 10 - (2 × 3) = 10 - 6 = 4

- Only the **lower 32 bits** of the product are used.
- `MLS` **does NOT** update condition flags — there is **no `MLSS`** variant.
- If you need flags (e.g., N/Z), you must use:

- `MUL R4, Rm, Rs`
- `SUBS Rd, Rn, R4   ; Updates flags`

Arithmetic and Logic

Multiplication and Division Operations

- **SDIV** stands for **Signed Division**.
- It performs **signed integer division** of one register by another and stores the result in a destination register.

```
MOV R1, #10      ; Rn = 10
MOV R2, #-2      ; Rm = -2
SDIV R0, R1, R2   ; R0 = R1 / R2 = 10 / -2 = -5
```

- **Signed** division: operands and result are interpreted as **signed 32-bit integers**
- **Remainder is discarded** (only the quotient is returned)
- Division by **zero** causes **UNPREDICTABLE behavior** — check before dividing!

- SDIV is not supported by the processor we are using in the simulator

Arithmetic and Logic

Multiplication and Division Operations

- **`UDIV`** stands for **Unsigned Division**.
- It performs **unsigned integer division** of one 32-bit register by another and stores the **quotient** in a destination register.

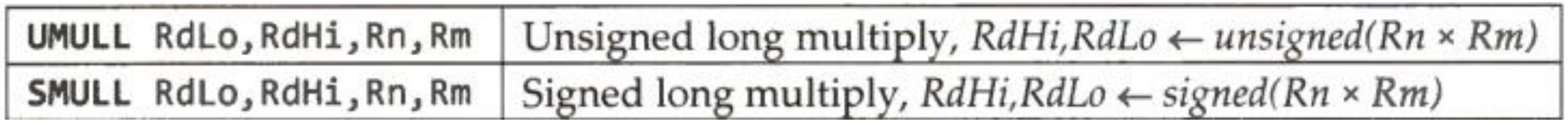
```
MOV R1, #10      ; Rn = 10
MOV R2, #3       ; Rm = 3
UDIV R0, R1, R2   ; R0 = R1 / R2 = 10 / 3 = 3 (quotient only)
```

- Operands are treated as **unsigned 32-bit integers**
- **Only the quotient** is returned — the **remainder is discarded**
- **No flags** (N, Z, C, V) are updated

•UDIV is not supported by the processor we are using in the simulator

Arithmetic and Logic

Long Multiplication Instructions



`UMULL RdLo,RdHi,Rn,Rm`	Unsigned long multiply, *RdHi,RdLo* $\leftarrow$ *unsigned(Rn* × *Rm)*
`SMULL RdLo,RdHi,Rn,Rm`	Signed long multiply, *RdHi,RdLo* $\leftarrow$ *signed(Rn* × *Rm)*

- Two registers are used to store a 64-bit result
- The high register (RdHi) holds the most significant 32 bits
- The low register (Rd Lo) holds the least significant 32 bits.

Arithmetic and Logic

Long Multiplication Instructions

```
MOV R0, #0xFFFFFFFF ; Rm = 4294967295
MOV R1, #2 ; Rs = 2
UMULL R2, R3, R0, R1 ;  [R3:R2] = R0 * R1
```

- `R0 * R1 = 0xFFFFFFFF * 2`
- ` = 8589934590 (decimal)`
- ` = 0x1 FFFFFFFE (9 digit)`

- `R2 holds 0xFFFFFFFE (lower 32 bits)`
- `R3 holds 0x00000001 (upper 32 bits)`

- `[R3:R2] = 0x00000001 FFFFFFFE = 8589934590`

Arithmetic and Logic

Long Multiplication Instructions

```
MOV R0, #0x80000000    ; Rm = -2147483648 (most negative signed 32-bit)
MOV R1, #2             ; Rs = 2
SMULL R2, R3, R0, R1   ; [R3:R2] = R0 * R1 = -2147483648 * 2
```

- -2147483648 × 2 = -4294967296
- To represent -4294967296 in 64-bit two’s complement:
- Positive 4294967296 = 0x0000000100000000
- So, -4294967296 = **two’s complement** of that:
- 0xFFFFFFFF 00000000

- Lower 32 bits (R2) = 0x00000000
- Upper 32 bits (R3) = 0xFFFFFFFF

Arithmetic and Logic

Looking into the flags

ADDS:

Flag	Meaning
N	Set if result is negative (bit 31 = 1)
Z	Set if result = 0
C	Set if **carry out** from bit 31 (unsigned overflow)
V	Set if **signed overflow** occurs

```
MOV R0, #0xFFFFFFFF
ADD R1, R0, #1 ; 0xFFFFFFFF + 1 = 0x00000000 → C = 1, Z = 1, V = 0, N = 0

MOV R0, #0x7FFFFFFF ; Rn = 2147483647 (max positive signed 32-bit)
ADD R1, R0, #1 ; R1 = R0 + 1 = 0x80000000 → C=0 , Z=0 , V=1, N=1
```

Arithmetic and Logic

Looking into the flags

SUBS:

SUBS Rd, Rn, Operand2

Flag	Set When…
N	The result is **negative** (bit 31 = 1)
Z	The result is **zero**
C	**No borrow** occurs → Rn ≥ Operand2 (unsigned comparison)
V	**Signed overflow** occurs → Rn and Operand2 have **different signs**, and the result has an **unexpected sign**

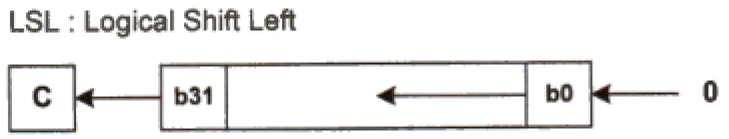
```
MOV R0, #5
SUBS R1, R0, #5 ; R1 = 5 - 5 = 0
Flags:
N = 0, C = 1(No borrow), Z =1, V=0
```

```
MOV R0, #0x80000000 ; R0 = -2147483648 (min signed int)
SUBS R1, R0, #1 ; -2147483648 - 1 = overflow!
Flags:
C = 1, V =1
```

Arithmetic and Logic

Shifts and Rotates

- **LSL** stands for **Logical Shift Left**.
- It shifts the bits of a register **to the left**, and fills the empty bits on the right with **zeros**.
- It's essentially the same as **multiplying by 2^n**, for unsigned numbers.



Syntax
LSL Rd, Rm,#shift_amount

Arithmetic and Logic

Shifts and Rotates

Example:

```
MOV R0, #5 ; R0 = 0b00000000_00000000_00000000_00000101
LSL R1, R0, #1 ; R1 = R0 << 1 = 0b00000000_00000000_00000000_00001010
```

```
R0 = 0000 0000 0000 0000 0000 0000 0000 0101 ; = 5
LSL R1, R0, #2
R1 = 0000 0000 0000 0000 0000 0000 0001 0100  ; = 20
```

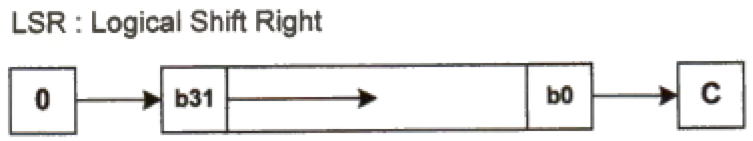
($5 \times 2^2 = 20$)

Instruction	Result
LSL R1, #1	×2
LSL R1, #3	×8
LSL R1, #0	No change

Arithmetic and Logic

Shifts and Rotates

- **LSR** stands for **Logical Shift Right**.
- It shifts the bits of a register **to the right**, and fills the empty bits on the **left** with **zeros**.
- It's equivalent to **dividing by 2^n** for **unsigned** numbers.



Syntax

LSR Rd, Rm, #shift_amount

Arithmetic and Logic

Shifts and Rotates

Example:

```
MOV R0, #0b1000       ; R0 = 00000000 00000000 00000000 00001000 = 8
LSR R1, R0, #1        ; R1 = R0 >> 1 = 00000000 00000000 00000000 00000100 = 4
```

```
R0 = 0b00000000_00000000_00000000_10100000  ; 160
LSR R1, R0, #3
R1 = 0b00000000_00000000_00000000_00010100  ; 20
```

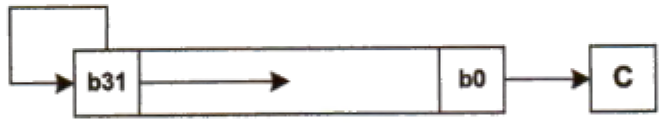
$160 \div 2^3 = 20$

Arithmetic and Logic

Shifts and Rotates

- **ASR** stands for **Arithmetic Shift Right**.
- It shifts bits to the **right**, just like LSR, but preserves the **sign** of the number by filling the **leftmost bits with the sign bit (bit 31)**.

ASR: Arithmetic Shift Right



Syntax
ASR Rd, Rm, #shift_amount

Arithmetic and Logic

Shifts and Rotates

Example:

```
MOV R0, #0b1000       ; R0 = 00000000 00000000 00000000 00001000 = 8
LSR R1, R0, #1        ; R1 = R0 >> 1 = 00000000 00000000 00000000 00000100 = 4
```

Example: Negative Number

```
MOV R0, #-40      ; R0 = 0xFFFFFFD8 = 11111111_11111111_11111111_11011000
ASR R1, R0, #2    ; Should divide by 4, keeping negative sign
;R1 = 0xFFFFFFF6 = 11111111_11111111_11111111_1111 0110 = -10
```

Example: Positive Number

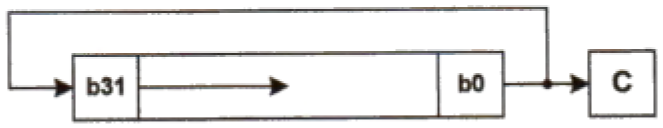
```
MOV R0, #40       ; R0 = 0b00000000_00000000_00000000_00101000
ASR R1, R0, #2    ; R1 = 00000000_00000000_00000000_00001010 = 40 / 4 = 10
```

Arithmetic and Logic

<u>Shifts and Rotates</u>

- **ROR** stands for **Rotate Right**.
- It rotates the bits of a register to the **right**, but
- Unlike LSR, the bits that fall off on the right are **wrapped around and reinserted at the left**.

ROR: Rotate Right



Syntax
ROR Rd, Rm, #shift_amount

Arithmetic and Logic

Shifts and Rotates

Example:

```
MOV R0, #0x80000001       ; R0: 10000000 00000000 00000000 00000001
ROR R1, R0, #1            ;  R1 = 11000000 00000000 00000000 00000000
```

Example:

```
MOV R0, #0x00000081      ; Binary: 00000000...10000001
ROR R1, R0, #4           ; Rotate right by 4
```

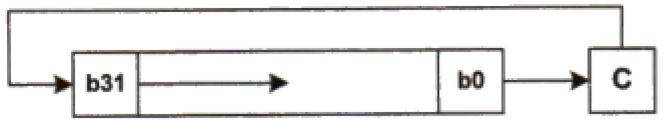
R1 = **0001**0000 00000000 00000000 00001000

Arithmetic and Logic

Shifts and Rotates

- **RRX** stands for **Rotate Right with Extend**.
- It rotates the bits of a register **right by 1 bit**, and instead of just rotating the lowest bit to the top (like ROR), it **shifts all bits right by one** and fills the **MSB (bit 31)** with the **value of the carry flag (C)**.

RRX: Rotate Right Extended



Syntax
MOVS Rd, Rm, RRX

- RRX **must** be used with the S suffix (like MOVS), because it **reads and updates the carry flag.**
- Takes the current value of Rm
- Shifts it **right by 1 bit**
- Inserts the **carry flag (C)** into **bit 31**
- The **bit shifted out of bit 0** becomes the **new carry**

Arithmetic and Logic

Shifts and Rotates

Example:

```
MOV R0, #0x80000001      ; 10000000...00000001
MOVS R1, R0, RRX         ; Carry flag (C) assumed = 1
```

Before:

- `R0 = 10000000 00000000 00000000 00000001`
- Carry = 1

After RRX:

- Shift right by 1 → all bits move one place to the right
- Bit 0 (1) moves to carry
- Carry (1) moves into bit 31

R1 = 11000000 00000000 00000000 00000000 ; = 0xC0000000
C = 1

Arithmetic and Logic

Barrel Shifter

- The **Barrel Shifter** is a hardware unit in ARM that allows you to perform **shifts or rotates on the second operand** of data-processing instructions **in a single instruction cycle**.
- You can shift, rotate, or extend a register while using it in an instruction like **`ADD`, `MOV`, `ORR`, etc.**

You can apply these shifts to **Operand2**:

Shift Type	Description
LSL #n	Logical Shift Left by n
LSR #n	Logical Shift Right by n
ASR #n	Arithmetic Shift Right by n
ROR #n	Rotate Right by n
RRX	Rotate Right with Extend (through carry)

Arithmetic and Logic

Barrel Shifter

Example 1: MOV with Barrel Shifter

```
MOV R0, #1 ; R0 = 0x00000001
MOV R1, R0, LSL #3 ;  R1 = R0 << 3 = R0* 2^3 = 0x00000008
```

Example 2: ADD with Barrel Shifter

```
MOV R2, #2
MOV R3, #5
ADD R4, R2, R3, LSL #2 ; R4 = R2 + (R3 << 2)
```

Why Is This Useful?

- ARM allows **shifting directly in the instruction**, unlike x86
- It reduces instruction count and improves performance
- You get combined operations like "shift and add" or "rotate and or" without extra lines

Arithmetic and Logic

Bitwise Logic Operations

- The **AND** instruction performs a **bitwise logical AND** between two operands.
- It compares **each bit** of two registers, and the result is 1 only if **both bits are 1** — otherwise, it is 0

Syntax

AND Rd, Rn, Operand2

```
MOV R0, #0b10101010    ; = 0xAA
MOV R1, #0b11001100    ; = 0xCC
AND R2, R0, R1         ; R2 = R0 & R1

 R0 = 10101010
 R1 = 11001100
      ---------------
  R2 = 10001000
```

Arithmetic and Logic

Bitwise Logic Operations

- The **AND** instruction performs a **bitwise logical AND** between two operands.
- It compares **each bit** of two registers, and the result is 1 only if **both bits are 1** — otherwise, it is 0

Syntax

```
AND Rd, Rn, Operand2

MOV R0, #0b10101010     ; = 0xAA
MOV R1, #0b11001100     ; = 0xCC
AND R2, R0, R1          ; R2 = R0 & R1

 R0 = 10101010
 R1 = 11001100
     ---------------
  R2 = 10001000
```

Example 2: Masking with AND

```
MOV R0, #0x12345678     ; Some value
MOV R1, #0x0000FF00     ; Mask: keep only middle byte
AND R2, R0, R1

R0 = 0x12345678
R1 = 0x0000FF00
R2 = 0x00005600
```

Arithmetic and Logic

Bitwise Logic Operations

- **ORR** stands for **bitwise OR** in ARM assembly.
- It performs a **logical OR** operation between each bit of two operands.
- A bit in the result is 1 if **either** (or both) corresponding bits are 1.

Syntax
ORR Rd, Rn, Operand2

```
MOV R0, #0b10101010    ; = 0xAA
MOV R1, #0b11001100    ; = 0xCC
ORR R2, R0, R1         ; R2 = R0 & R1

 R0 = 10101010
 R1 = 11001100
    ----------------
 R2 = 11101110
```

Example 2: Setting Bits with ORR

```
MOV R0, #0x12340000
MOV R1, #0x00005678
ORR R2, R0, R1

R0 = 0x12340000
R1 = 0x00005678
R2 = 0x12345678
```

Barrel Shifter Example

```
ORR R3, R1, R2, LSL #4 ;
R3 = R1 OR (R2 << 4)
```

Arithmetic and Logic

Bitwise Logic Operations

- **EOR** stands for **Exclusive OR** in ARM assembly.
- It performs a bitwise **XOR** between two operands:
- The result bit is **1** if the **bits are different**, The result bit is **0** if the **bits are the same**.

Syntax

EOR Rd, Rn, Operand2

```
MOV R0, #0b10101010    ; = 0xAA
MOV R1, #0b11001100    ; = 0xCC
EOR R2, R0, R1         ; R2 = R0 & R1

 R0 = 10101010
 R1 = 11001100
     ---------------
 R2 = 01100110
```

Example 2: Toggling Bits (Flipping)

```
MOV R0, #0b11110000    ; R0 = 0xF0
MOV R1, #0b00001111    ; R1 = 0x0F
EOR R2, R0, R1         ; Toggle all bits

R2 = 0b11111111 = 0xFF
```

Arithmetic and Logic

Bitwise Logic Operations

- **`BIC`** stands for **Bit Clear** in ARM assembly.

- **`Rd = Rn AND (NOT Operand2)`**
- So instead of just ANDing the bits like AND, it **inverts `Operand2` first**, then **ANDs** it with Rn.

Syntax
BIC Rd, Rn, Operand2

```
MOV R0, #0b11111111     ; 0xFF
MOV R1, #0b00001111     ; 0x0F
BIC R2, R0, R1          ; R2 = R0 & ~(R1)

        R0 = 11111111
        R1 = 00001111
NOT R1 = 11110000
R2 = R0 AND (NOT R1) = 11110000
```

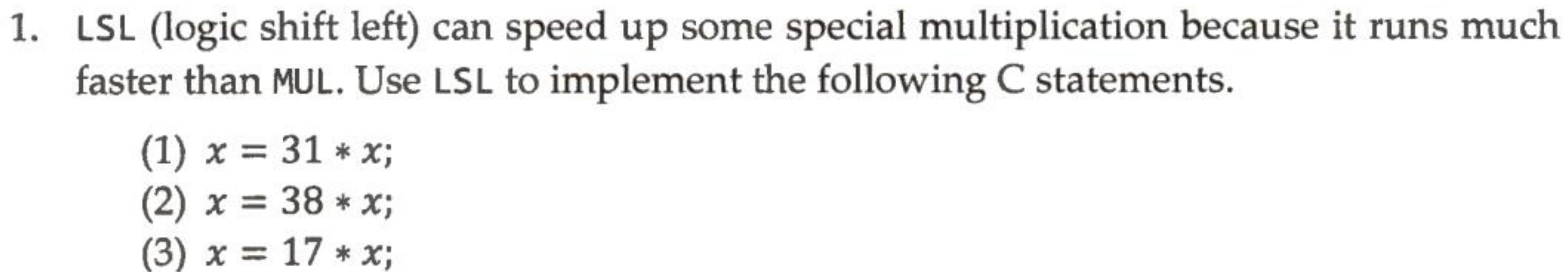
Example 2: Clearing a Bit

```
MOV R0, #0b10101010     ; R0 = 0xAA
MOV R1, #0b10000000     ; R1 = 0x80
BIC R2, R0, R1          ; Clear bit 7 of R0

R0 = 10101010
R1 = 10000000
~R1 = 01111111
R2 = R0 & ~R1 = 00101010 = 0x2A
```

Arithmetic and Logic

Swap the upper halfword and lower half word of a register

```
.global _start
_start:

LDR r0, =0xAAAABBBB ; Load example value into r0

MOV r1, r0 ; Copy r0 to r1

LSR r2, r1, #16 ; r2 = upper halfword (r1 >> 16)

MOV r1, r1, LSL #16 ; r1 = lower halfword shifted to
upper

ORR r0, r1, r2 ; combine upper and lower swapped
```



1. LSL (logic shift left) can speed up some special multiplication because it runs much faster than MUL. Use LSL to implement the following C statements.

   (1) $x = 31 * x;$
   (2) $x = 38 * x;$
   (3) $x = 17 * x;$

✅ **Strategy:**

Break the constant multiplier into a sum of powers of 2:

Constant	Binary Representation	Decomposition
31	11111	16+8+4+2+1
38	100110	32+4+2
17	10001	16+1

Assume:

- Input `x` is in `r0`
- Result is stored back in `r0`
- Use temporary registers `r1`, `r2`, etc.

Computing x = 31 * x;

```
MOV r1, r0, LSL #4 ;  r1 = x << 4 = 16x
ADD r1, r1, r0, LSL #3 ; r1 += 8x → 24x
ADD r1, r1, r0, LSL #2 ; r1 += 4x → 28x
ADD r1, r1, r0, LSL #1 ; r1 += 2x → 30x
ADD r0, r1, r0 ; r0 = 30x + x = 31x
```

2. Suppose `r0 = 0x0F0F0F0F`, and `r1 = 0xFEDCBA98`, find the result of the following operations.

```
(1) EOR r3, r1, r0
(2) ORR r3, r1, r0
(3) AND r3, r1, r0
(4) BIC r3, r1, r0
(5) BFI r3, r1, #4, #8
(6) MVN r3, r1
(7) MVN r3, r0
(8) MVN r3, r0
    ADD r3, r1, r3
```

4. Translate the following C statement into an assembly program, assuming 16-bit signed integers x, y and z (*i.e.*, signed short) are stored in 32-bit register `r0`, `r1`, and `r2`, respectively.

$$x = x * y + z - x;$$

5. Translate the following C statement into an assembly program, assuming 16-bit unsigned integers x and y (*i.e.*, unsigned short) are stored in register `r0`, and `r1`, respectively.

$$x = x \,\%\, y;$$

6. Write an assembly program that calculates the value of the following given polynomial, assuming signed integers x and y are stored in register `r0` and `r1`, respectively.

$$y = 3x^3 - 7x^2 + 10x - 11.$$

14. Suppose `r0` = `0xFFFFFFFF`, `r1` = `0x00000001`, and `r2` = `0x00000000`. Initially the `N`, `Z`, `C`, and `V` flags are zero. Find the value of the `N`, `Z`, `C`, and `V` flags of the following instructions. (Assume each instruction runs individually, *i.e.*, these instructions are not part of a program.)

```
(1) ADD  r3, r0, r2
(2) SUBS r3, r0, r0
(3) ADDS r3, r0, r2
(4) LSL  r3, r0, #1
(5) LSRS r3, r1, #1
(6) ANDS r3, r0, r2
```