L3 (CHAPTER 6)

Programming in Assembly
Part 2: Data Manipulation
Exercises

Load Instructions

▸ **LDR rt, [rs]**

- ▸ fetch data from memory into register rt.
- ▸ The memory address is specified in register rs.
- ▸ For Example:

```
; Assume r0 = 0x08200004
; Load a word:
LDR r1, [r0]                    ; r1 = Memory.word[0x08200004]
```

Store Instructions

- **STR rt, [rs]:**

  - save data in register rt into memory
  - The memory address is specified in a base register rs.
  - For Example:

    ```
    ; Assume r0 = 0x08200004
    ; Store a word
    STR r1, [r0]                 ; Memory.word[0x08200004] = r1
    ```
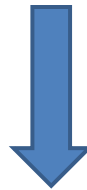
Question: Load-Modify-Store

C statement

$$X = X + 1;$$

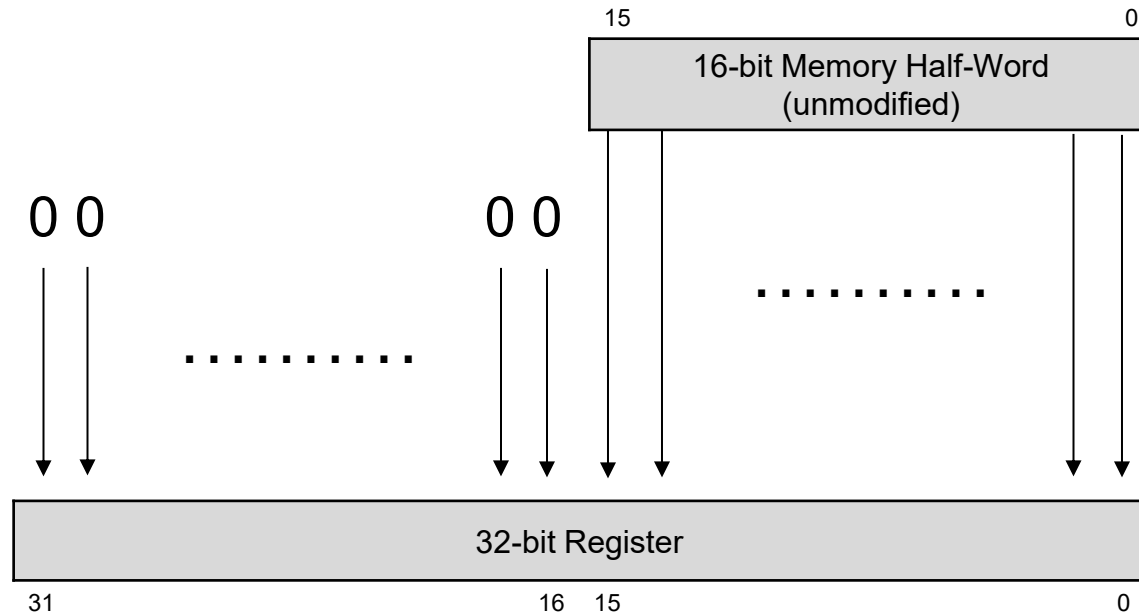; Assume the memory address of x is stored in r1.
Write the assembler program for the C statement.

Loading Constants: LDR

- LDR r_d,=*constant*
  - A special "pseudo-operation" that will work for any constant up to 32 bits wide.
  - You simply write what appears to be a regular ARM instruction (except that an equal sign is substituted for the pound sign) and let the assembler sort out the most efficient way to achieve your objective:
  - Converted to MOV or MVN if possible
  - Else converts to LDR r_d,[pc,#imm]
- *Examples:*
  - *LDR R1,=10 ;assembler replaces this by MOV R1,#10.*
  - *LDR R1,=−15 ;assembler replaces this by MVN R1,#14.*
  - *LDR R1,=−127435 ;assembler replaces this by a memory reference instruction that loads the constant −127435 from a separate memory location.*

LDRH (Load Halfword)

15   0

16-bit Memory Half-Word (unmodified)

0 0   0 0

· · · · · · · · · ·   · · · · · · · · · ·

32-bit Register

31   16   15   0

When loading 8- or 16-bit data into a 32-bit register, the operand itself is always right justified within the register and its most significant bits filled according to whether the value is signed or unsigned. Unsigned operands less than 32 bits wide must fill the extra bit positions with zeroes (called "zero-filling").

LDRSH (Load Signed Halfword)

15   0

16-bit Memory Half-Word (unmodified)

· · · · · · · · · ·

· · · · · · · · · ·

32-bit Register

31   16   15   0
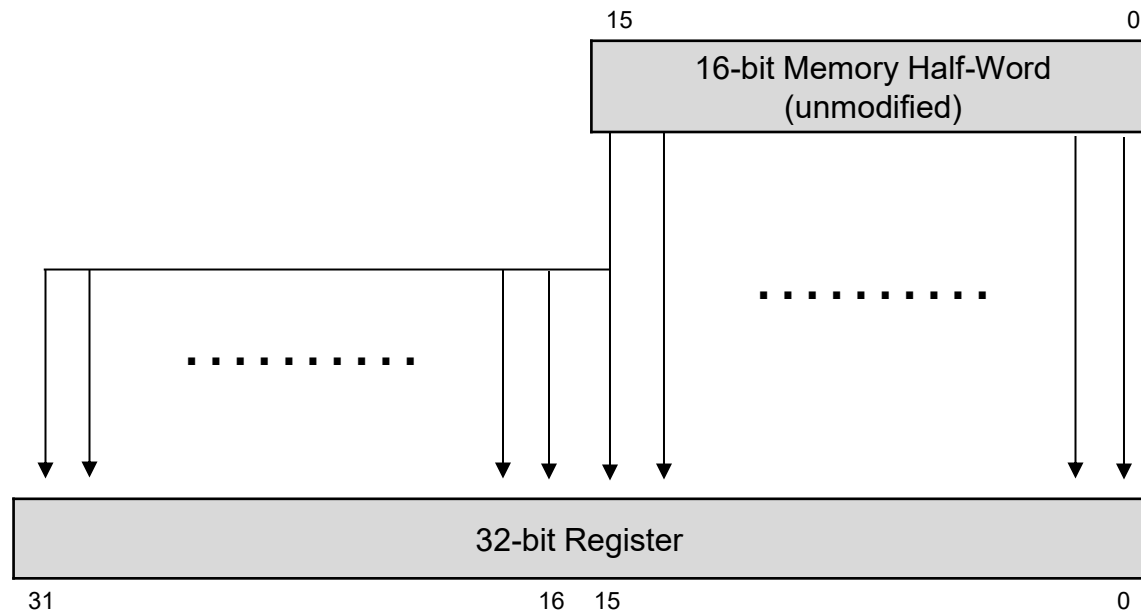
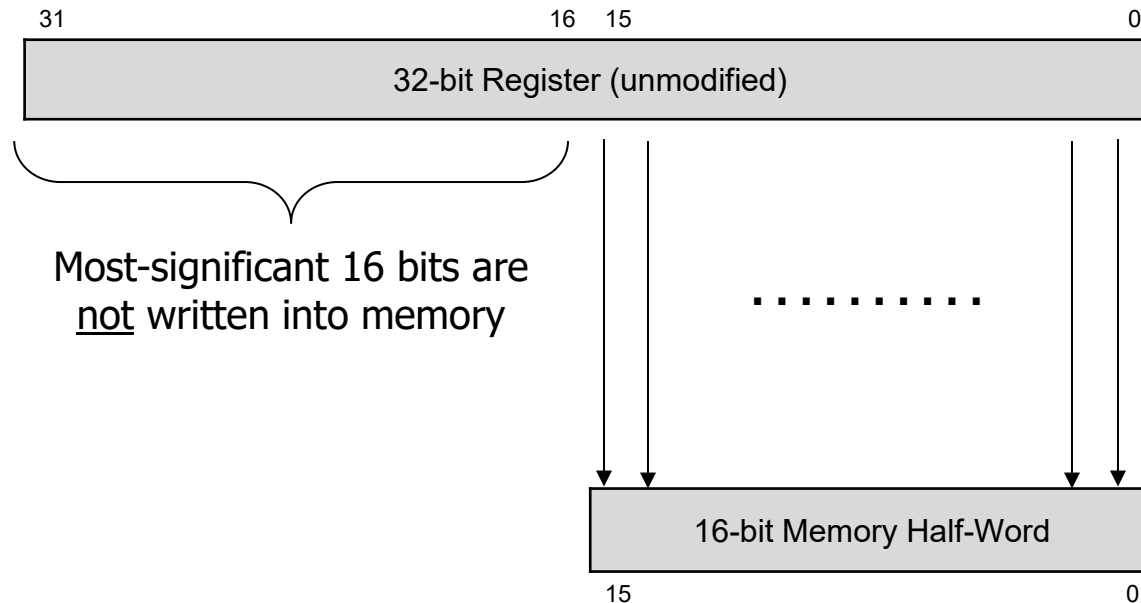Signed operands less than 32 bits wide must fill the extra bit positions with copies of their sign bit

Load (from memory) Instructions

Load/Store Memory	Operation	Notes
LDR r_d,<mem>	$r_d \leftarrow mem_{32}[address]$	
LDRB r_d,<mem>	$r_d \leftarrow mem_8[address]$	Zero fills
LDRH r_d,<mem>	$r_d \leftarrow mem_{16}[address]$	Zero fills
LDRSB r_d,<mem>	$r_d \leftarrow mem_8[address]$	Sign extends
LDRSH r_d,<mem>	$r_d \leftarrow mem_{16}[address]$	Sign extends
LDRD r_t,r_{t2},<mem>	$r_{t2} \cdot r_t \leftarrow mem_{64}[address]$	

STRH (Store Halfword)

31   16   15   0

32-bit Register (unmodified)

Most-significant 16 bits are <u>not</u> written into memory

. . . .

16-bit Memory Half-Word

15   0

Writing a half-word result to address 104: The two memory bytes that should be written are at addresses 104 and 105. However, the 32-bit memory data bus is actually 4 bytes wide, corresponding to addresses 104, 105, 106, and 107. Each byte of the memory data bus has its own write enable; during the memory write cycle, the write-enable signals for addresses 106 and 107 are disabled so that only the bytes at addresses 104 and 105 are modified.

Store (to memory) Instructions

Load/Store Memory	Operation	Memory Byte Addresses Actually Written			
STR r_d,<mem>	$r_d \rightarrow mem_{32}[addr]$	addr+3	addr+2	addr+1	addr
STRB r_d,<mem>	$r_d \rightarrow mem_8[addr]$				addr
STRH r_d,<mem>	$r_d \rightarrow mem_{16}[addr]$			addr+1	addr
STRD r_t, r_{t2},<mem>	$r_{t2}.r_t \rightarrow mem_{64}[addr]$				

LDR and STR

LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword

STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword

Assume Little-Endianness

▸ For the rest of the questions, assume Little-Endian ordering by default unless specified otherwise

Question: Load a Byte, Half-word, Word

What does r1 contain: Load a Byte

`LDRB r1, [r0]`

r1 | | | | |
31   0

What does r1 contain: Load a Halfword

`LDRH r1, [r0]`

r1 | | | | |
31   0

What does r1 contain: Load a Word

`LDR r1, [r0]`

r1 | | | | |
31   0

0x02000003	**0x87**
0x02000002	**0x65**
0x02000001	**0xE3**
0x02000000	**0xE1**

Little Endian

Assume
r0 = 0x02000000

Use zero-filling for
remaining Bytes

Question: Sign Extension

Load a Signed Byte. Sign-bit of 0xE1 (11100001) is its leftmost bit (1), so the left 3 Bytes of r1 are filled with all 1's

`LDRSB r1, [r0]`

r1 | | | | |
31   0

Load a Signed Halfword. Sign-bit of 0xE3E1
(1110001111100001) is its leftmost bit (1),so the left 3
Bytes of r1 are filled with all 1's

`LDRSH r1, [r0]`

r1 | | | | |
31   0

0x20000003	**0x87**
0x20000002	**0x65**
0x20000001	**0xE3**
0x20000000	**0xE1**

Little Endian

Assume
r0 = 0x02000000

Use sign-filling for remaining Bytes
Facilitate subsequent 32-bit signed arithmetic!

Address

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes).
    ```
    LDR r0,[r1,#8]
    ```
  - A register, optionally shifted by an immediate value
    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```
- This can be either added or subtracted from the base register:
  ```
  LDR r0,[r1,#-8]
  LDR r0,[r1,-r2]
  LDR r0,[r1,-r2,LSL#2]
  ```
- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0-255 bytes).
  - A register (unshifted).
- Choice of *pre-indexed* or *post-indexed* addressing

Addressing Modes

- Offset addressing (most important):

  `LDR R1,[R0]` ;Load R1 from memory address R0

  `LDR R1,[R0,#16]` ;Load R1 from memory address R0+16

- Auto-indexing with pre-indexed addressing mode:

  `LDR R1,[R0,#16]!` ;Load from memory address R0+16, then
  ;update R0 = R0+16

- Auto-indexing with post-indexed addressing mode:

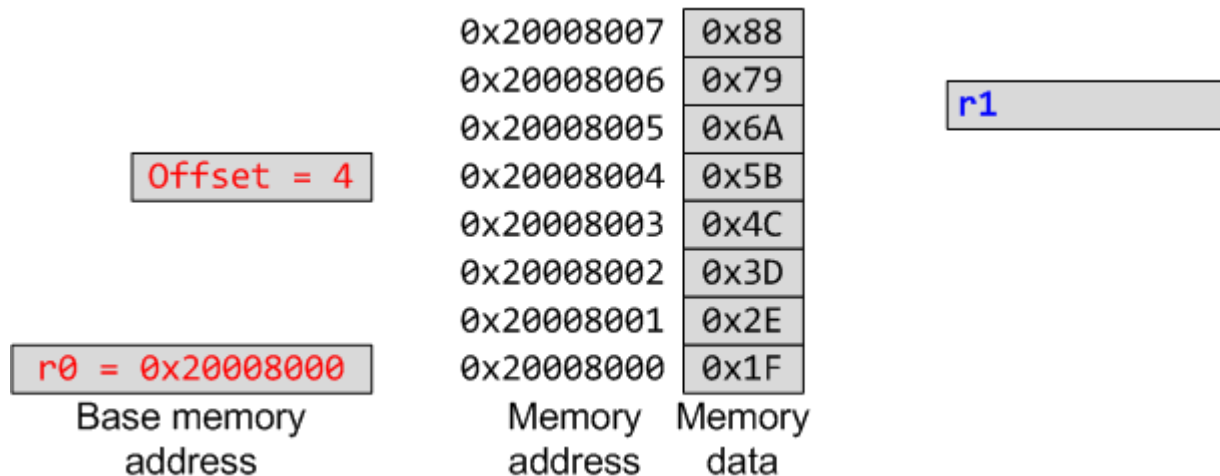 `LDR R1,[R0],#16` ;Load R0 from memory address R0, then
  ;update R0 = R0+16

Addressing Modes

Index Format	Example	Equivalent
offset	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	LDR r1, [r0], #4	r1 ← memory[r0] r0 ← r0 + 4

Offset range is -255 to +255

Question: LDR w/ Offset

What do r0 and r1 contain after: LDR r1, [r0, #4]

Base memory address	Memory address	Memory data		r1
	0x20008007	0x88		
	0x20008006	0x79		
	0x20008005	0x6A		
Offset = 4	0x20008004	0x5B		
	0x20008003	0x4C		
	0x20008002	0x3D		
	0x20008001	0x2E		
r0 = 0x20008000	0x20008000	0x1F		

Offset range is -255 to +255
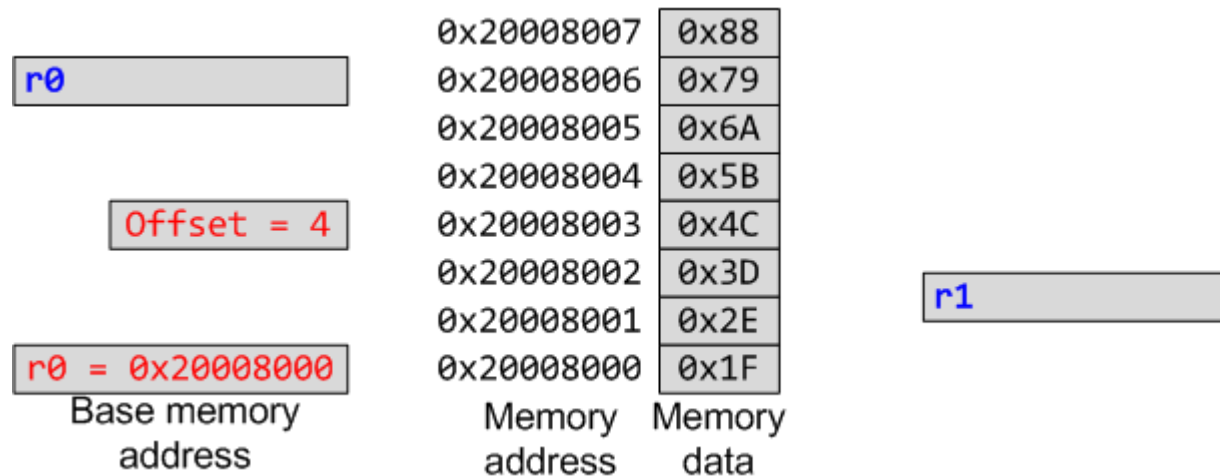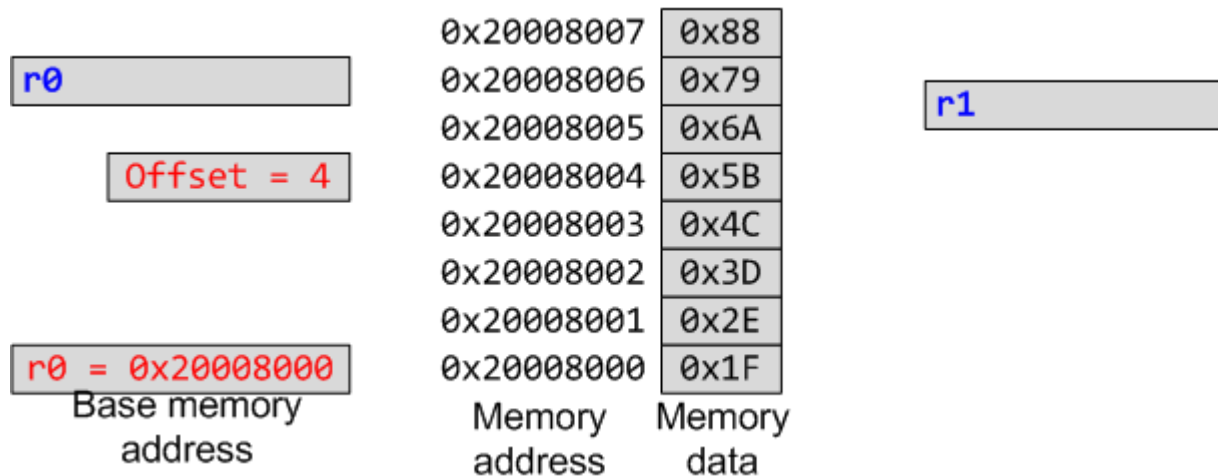
Question: LDR w/ Post-index

What do r0 and r1 contain after: LDR r1, [r0], #4

Offset range is -255 to +255

Question: LDR w/ Pre-index

What do r0 and r1 contain after: LDR r1, [r0, #4]!

r0

Offset = 4

r0 = 0x20008000
Base memory
address

Memory address	Memory data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r1

Offset range is -255 to +255

Question: LDRH

LDRH r1, [r0]

; r0 = 0x20008000

r1 before load

0x12345678

r1 after load

?

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Question: LDSB

What is r1 after: LDSB r1, [r0]

; r0 = 0x20008000

r1 before load

| 0x12345678 |

r1 after load

| ? |

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Question: STR w/ Post-Index

What are r0, and memory content after: 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

Question: STR with Offset

What are r0, and memory content after: STR r1, [r0, #4]

; r0 = 0x20008000, r1=0x76543210

r0 before the store

0x20008000

r0 after the store

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

Question: STR with Pre-Index

What are r0, and memory content after: 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

Question: LDR w/ Big Endian Ordering

What is r11 after load with Big-Endian ordering?

```
LDR r11, [r0]
; r0 = 0x20008000
```

r11 before load

0x12345678

r11 after load

?

Memory Address	Memory Data
0x20008003	0xEE
0x20008002	0x8C
0x20008001	0x90
0x20008000	0xA7

ARM ADR Pseudo-op

- ADR pseudo-op generates instruction required to calculate address:

  ADR R1,x ;get memory address of variable x and put it in register R1

Example 1: Assignment

- C:

  //assume x, a, b are 32-bit integer variables

  x = a - b;

- Assembler:

  ADR R4,a   ; get address for a

  LDR R0,[R4] ; get value of a

  ADR R4,b   ; get address for b, reusing R4

  LDR R1,[R4] ; get value of b

  SUB R0,R0,R1 ; subtract R1 from R0, and store result in R0

  ADR R4,x   ; get address for x

  STR R0,[R4] ; store value of x into memory

Question: Programming

▸ Q: Fill in the blanks of the following assembly program that implements the following C program snippet (assuming x, y and z are 32-bit ints):

```
//C Program
//assume x, y, z, are integer variables
z=x+y;
z=z²
```

```
%Assembler Program
ADR R4, x

...

ADR R4, y

...

ADR R4, z

...
```

Question: Programming

▸ Q: can you reduce the number of registers used in the assembler program?

```
//C Program
//assume x, y, z, are integer variables
z=x+y;
z=z²
%Assembler Program
ADR R4, x
LDR R0, [R4]
ADR R4, y
LDR R1, [R4]
ADD R2, R0, R1
MUL R3, R2, R2
ADR R4, z
STR R3, [R4]
```