Chapter 6: Memory Access and Stack

- Indexed Addressing Mode
- **Usual** Format: LDR Rd, [Rx] or STR Rx, [Rd]
- [Rx] in LDR & [Rd] in STR: "index register" which holds the pointer i.e., the base address
- **Advanced** Indexed Addressing: allows **modification of the value** in the index register ("base + **offset**" addressing mode)
- Indexed Addressing Modes (WB* = "Write back")

Addressing Mode	Syntax	Effective Address of Memory	Rm Value After Execution
Pre-index	LDR Rd, [Rm, #k]	Rm + #k	Rm
Pre-index with WB*	LDR Rd, [Rm, #k]!	Rm + #k	Rm + #k
Post-index	LDR Rd, [Rm], #k	Rm	Rm + #k

- Offset of Fixed value vs. Offset of Shifted Register

Offset	Syntax	Pointing Location
Fixed value	LDR Rd, [Rm, #k]	Rm + #k
Shifted register	LDR Rd, [Rm, Rn, <shift>]	Rm + (Rn shifted <shift>)

Pre-index mode with fixed offset

- **FORMAT:** **LDR Rd, [Rm, #k]** // immediate number
  - Effective Address: EA = Rm + k
  - After instruction: Rm ← Rm

Example)Write a code which stores contents of R5 to the location 0x10000 0000 to 0X1000 000F using pre-indexed addressing mode with fixed offset. R5 = 0x55667788.

```
1  //Example6-8.s
2  //Pre-index mode exercise
3  // LDR Rd, [Rm, #k]
4  // Address = Rm+#k
5  // Rm unchanged
6  .global _start
7  _start:
8      LDR r5, =0x55667788  //4-byte data
9      LDR r1, =0x10000000  //base address
10     STR r5, [r1]  // ususal store
11     STR r5, [r1, #4] //Pre-index mode
12                      //store at base+4
13                      //r1 unchanged
14     STR r5, [r1, #8] //store at base + 8
15     STR r5, [r1, #0x0c] //store at base + 0x0C (12)
16
17 stop:
18     B stop
19
20 .end
21
```

r0	00000000	
r1	10000000	
r2	00000000	
r3	00000000	
r4	00000000	
r5	55667788	
r6	00000000	
r7	00000000	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	00000018	
cpsr	000001d3	NZCVI SVC
spsr	00000000	NZCVI ?

0ffffff0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
10000000	55667788	55667788	55667788	55667788
10000010	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
10000020	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
10000030	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa

Pre-index Writeback mode

- **FORMAT: LDR Rd, [Rm, #k]! // exclamation mark (!)**
  - Effective Address: EA = Rm + k
  - After instruction: Rm ← Rm + k

Example)Write a code which stores contents of R5 to the location 0x20000 0000 to 0X2000 000F using pre-index writeback mode.
R5 = 0x99AABBCC

```
1  //Example6-9.s
2  //Pre-index writeback exercise
3  // LDR Rd, [Rm, #k]!    *note the exclamation (!)
4  // Address = Rm+#k
5  // Rm <-- Rm + #k
6  .global _start
7  _start:
8      LDR r5, =0x99AABBCC  //4-byte data
9      LDR r1, =0x20000000  //base address
10     STR r5, [r1]  // ususal store
11     STR r5, [r1, #4]! //store at base+4
12                       //r1 = r1 +4
13     STR r5, [r1, #4]! //store at base + 8
14                       //r1 = r1 + 4
15     STR r5, [r1, #4]! //store at base + 0x0C (12)
16                       //r1 = 2000 000C now
17 stop:
18     B stop
19
20 .end
21
22
```

Post-index mode

- **FORMAT:** **LDR Rd,[Rm], #k**
  - Effective Address: EA = Rm
  - After instruction: Rm ← Rm + k

Example)Write a code which stores contents of R5 to the location 0x20000 0000 to 0X2000 000F using post-index mode. R5 = 0x99AABBCC

Pre-index with offset register

- **FORMAT:** **LDR Rd,[Rm, Rn]**
  - Effective Address: EA = Rm + Rd
  - After instruction: Rm ← Rm

```
1  //Example6-11.s
2  //Pre-index with offset register
3  // LDR Rd, [Rm, Rd]
4  // Address = Rm + Rd
5  .text
6  .global _start
7  _start:
8      LDR r1, =mydata  //base address
9      LDR r4, [r1] //load from base address
10     MOV r2, #4
11     LDR r5, [r1,r2]  //load from base address +4
12     MOV r2,#8
13     LDR r6, [r1,r2]  //load from base address +8
14     MOV r2, #12
15     LDR r7, [r1,r2]  //load from base address +
16
17
18 stop:
19     B stop
20 .data
21 mydata:
22     .word 0x45, 0x1234, 0x12ABCD, 0x99AABBCC
23
24 .end
25
```

r0	00000000	
r1	00000028	
r2	0000000c	
r3	00000000	
r4	00000045	
r5	00001234	
r6	0012abcd	
r7	99aabbcc	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	00000020	
cpsr	000001d3	NZCVI SVC
spsr	00000000	NZCVI ?

fffffff0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
00000000	e59f101c	e5914000	e3a02004	e7915002
00000010	e3a02008	e7916002	e3a0200c	e7917002
00000020	eafffffe	00000028	00000045	00001234
00000030	0012abcd	99aabbcc	aaaaaaaa	aaaaaaaa
00000040	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
00000050	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
00000060	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
00000070	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa

Editor (Ctrl-E) | Disassembly (Ctrl-D) | Memory (Ctrl-M)

Pre-index with shifted register

- **FORMAT: LDR Rd, [Rm, Rn,<Shift>]**
  - Effective Address: EA = Rm + Rn<Shift>
  - After instruction: Rm ← Rm

r0	00000000	
r1	00000028	
r2	00000002	
r3	00000000	
r4	00000045	
r5	2489acf5	
r6	2489ac23	
r7	00000000	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	0000001c	
cpsr	000001d3	NZCVI SVC
spsr	00000000	NZCVI ?

```
1  //Example6-13.s
2  //Pre-index with shifted register
3  // LDR Rd, [Rm, Rn, <shift>]
4  // Address = Rm + (shifted Rn)
5  .text
6  .global _start
7  _start:
8      LDR r1, =mydata  //base address
9      MOV r2, #0    //index=0
10     LDR r4, [r1, r2, LSL #2] //LSL #2 --> *4
11                              //r2 = 0
12                              //at base address
13     MOV r2, #1    //index=1
14     LDR r5, [r1, r2, LSL #2] //LSL #2 --> *4
15                              //r2 = 4
16                              //at base + 4
17     MOV r2, #2    //index=2
18     LDR r6, [r1, r2, LSL #2] //LSL #2 --> *4
19                              //r2 = 8
20                              //at base + 8
21
22 stop:
23     B stop
24 .data
25 mydata:
26     .word 0x45, 0x2489ACF5, 0x2489AC23
27
28 .end
```

Index Addressing and Look-Up Table

One application of index addressing mode

Look-Up Table

- An array of pre-calculated constants
- Allows to obtain frequently used values with no complex arithmetic operations
- The cost is the memory space for the table
- Popular approach in embedded systems with lower computing power and stringent real time demand

Look-Up Table

Ex) Write a code which, for a given number x, (a) looks up a table (located in a memory) of x^5 values, and (b) stores the value in R10. Assume that x is [0 – 9]. Use a look-up table. Use subroutine approach.

$$R10(x) := x^5$$

$R10(0) = 0$
$R10(1) = 1$
$R10(2) = 32$
$R10(3) = 243$
$R10(4) = 1024$
$R10(5) = 3125$
$R10(6) = 7776$
$R10(7) = 16807$
$R10(8) = 32768$
$R10(9) = 59049$

Look-Up Table (using ADR instead of LDR)

Ex) Write a code which, for a given number x, (a) looks up a table (located in the program memory [Flash]) of x^5 values, and (b) stores the value in R10. Assume that x is [0 – 9].

Look-Up Table with Indexed Addressing Mode

Q1) Write a code which uses **x** value in R1 and returns the value of the factorial of **x** in r0. Assume that **x** is [0 – 10]. Use a look-up table in Memory . Use Subroutine approach.

$0! = 1$

$1! = 1$

$2! = 2$

$3! = 6$

$4! = 24$

$5! = 120$

$6! = 720$

$7! = 5040$

$8! = 40320$

$9! = 3.6288 \cdot 10^{5}$

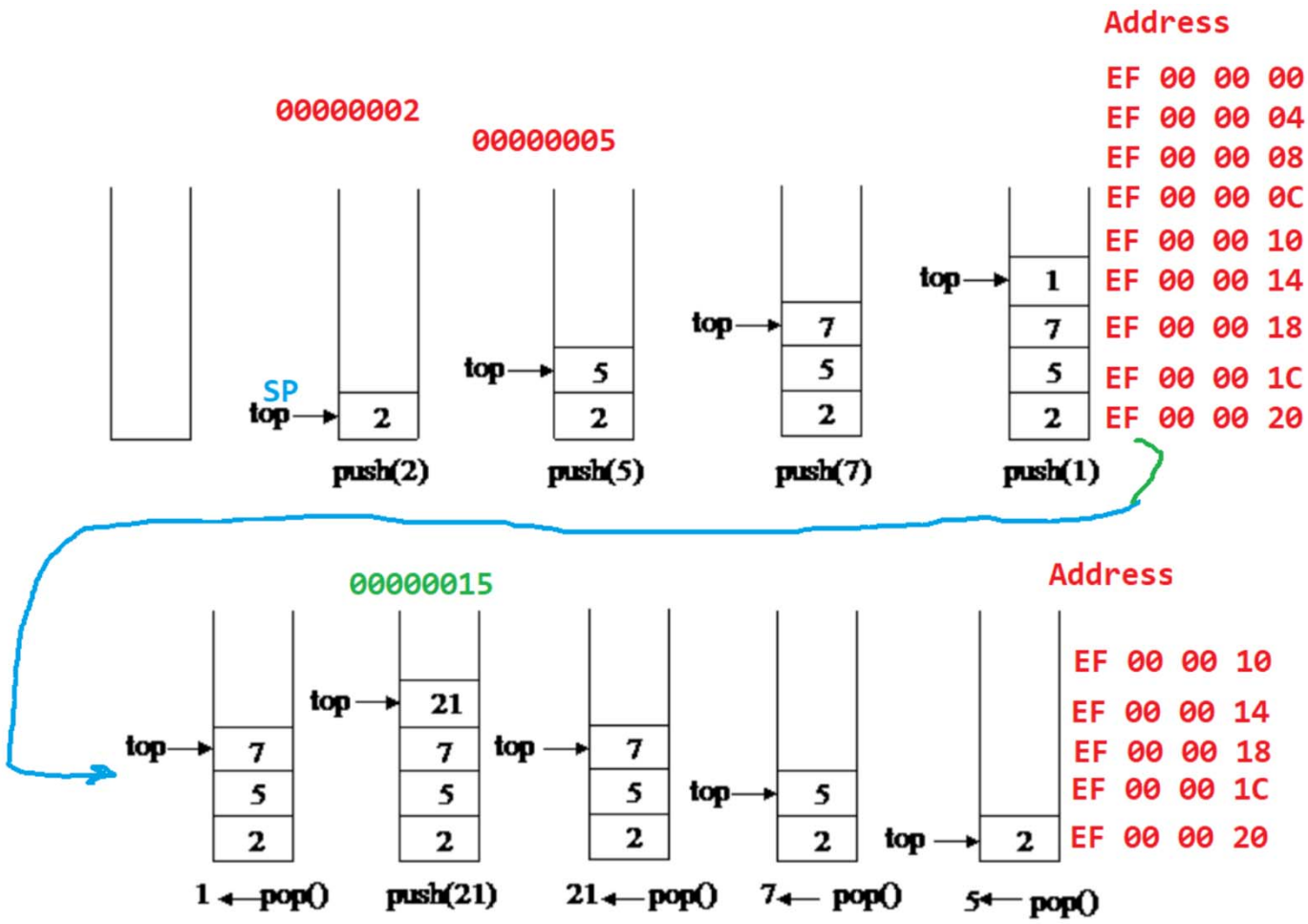
$10! = 3.6288 \cdot 10^{6}$

Look-Up Table with Indexed Addressing Mode

Q2) Write a code which calculates 10 to the power of **x** value in R2 and returns the result in R0. Assume that **x** is [0 – 9]. Use a look-up table in Memory . Use subroutine approach.

Stack

- **Last-in-first-out (LIFO) Memory Space**
  - Store/Write: Push in to the stack
  - Out/Read: Pop off the stack
- **Stack "grows" toward lower address**
  - Address at the bottom is higher
  - Address at the top is lower
- **"Stack" or "Stack pointer"**
  - The memory location where the last written data is stored
  - New data pushed are stored from the (stack – 1) location.
  - Read out data is from the (stack)
- **Usage of Stack:**
  - Pass the return address and parameters of the subroutine
  - Local variables
- **TOS (Top-of-stack)** = stack pointer (SP) = r13
- **Stack's data size** = a word = 4 bytes



Stack Initialization and PUSH & POP

- Initial value of 32-bit size R13(SP) when powered up is 0x00
- 32-bit address of RAM as a stack section
- Convention: Initialize SP to the uppermost RAM memory region (because stack grows toward lower address)
- Different ways of storing (push) and retrieving (pop)

  - **PUSH**
  - **POP**

Stack Initialization and PUSH & POP

```
1  //stack_simple.s
2  .global _start
3  _start:
4      MOV r3, r13 //Load SP to see the stack
5      //initial stack = 00000000
6      LDR r0, =0x01020304 //r0 value
7      PUSH {r0}
8      LDR r2, =#0x21222324
9      PUSH {r2}
10     POP {r4}
11     POP {r5}
12
13 //Change the Stacktop
14 //And do the same
15     LDR r13, =0xFFFFFF00
16     PUSH {r0}
17     LDR r2, =#0x21222324
18     PUSH {r2}
19     POP {r4}
20     POP {r5}
21
22 stop:
23     B stop
24
25 .end
26
```

0xE0001FE5				
0xE0001FE6				
0xE0001FE7				
0xE0001FE8				
0xE0001FE9				
0xE0001FEA				
0xE0001FEB				
0xE0001FEC				
0xE0001FED				
0xE0001FEE				
0xE0001FEF				
0xE0001FF0				
0xE0001FF1				
0xE0001FF2				
0xE0001FF3				
0xE0001FF4				
0xE0001FF5				
0xE0001FF6				
0xE0001FF7				
0xE0001FF8				
0xE0001FF9				
0xE0001FFA				
0xE0001FFB				
0xE0001FFC				
0xE0001FFD				
0xE0001FFE				
0xE0001FFF				
0xE0002000				

PUSH/POP operation with Multiple Registers

- When multiple registers are pushed to and popped from the stack:
  - Lower register is stored in the lower address in the stack → top of the stack
  - The order in { } for multiple registers is irrelevant

```
1  //stack_multi.s
2  // in multiple stack and pop
3  //regardless the order inside { }
4  //lowest register will be stored in the lowest address
5  //the next resgitsr to the next address etc
6  .global _start
7  _start:
8      LDR r1, =#0x11CC11DD
9      LDR r2, =#0x22EE22FF
10
11     LDR SP, =0xE0002000
12     PUSH {r2, r1}   // PUSH {r1, r2} woudld be the same
13     // r1 in the top and r2 just below it
14     POP {r6,r5}  // r5 gets r1 and r6 gets r2
15
16 //test with 4 registers
17 //Stack rule is to do with even number
18     LDR SP, =0xE0001FE0
19     LDR r3, =0x11223344
20     LDR r4, =0x55AA66BB
21     PUSH {r3,r1,r4,r2} //irregular oder
22     //register number order --> stored location order
23     POP {r7,r5,r6,r8} // irregular order
24     //register number order --> retrieving location order
25     //r5 = r1, r6 = r2, etc
26 stop:
27     B stop
28
29 .end
30
```

Stack operation with PUSH and POP

```
1  //Example6-19.s
2  //Stack operation using PUSH and POP
3  //SP itself is changed
4  .text
5  .equ stack_top, 0xE0002000
6  .global _start
7  _start:
8      LDR r13, =stack_top
9      LDR r0, =0x125
10     LDR r1, =0x144
11     MOV r2, #0x56
12     BL stack_sub
13     ADD r3, r1, r0
14     ADD r4, r3, r2
15 stop:
16     B stop
17 stack_sub:
18     PUSH {r0-r2}          //save r0, r1, r2 on stack
19     //or STMDB r13! {r0, r1, r2}
20         //before they are used by a loop
21     MOV r0, #0
22     MOV r1, #0
23     MOV r2, #0
24
25     POP {r0-r2}  //Restore the original from stack
26     //or LDMIA r13! {r0, r1, r2}
27     BX lr
28 .end
```

0xE0001FE5					
0xE0001FE6					
0xE0001FE7					
0xE0001FE8					
0xE0001FE9					
0xE0001FEA					
0xE0001FEB					
0xE0001FEC					
0xE0001FED					
0xE0001FEE					
0xE0001FEF					
0xE0001FF0					
0xE0001FF1					
0xE0001FF2					
0xE0001FF3					
0xE0001FF4					
0xE0001FF5					
0xE0001FF6					
0xE0001FF7					
0xE0001FF8					
0xE0001FF9					
0xE0001FFA					
0xE0001FFB					
0xE0001FFC					
0xE0001FFD					
0xE0001FFE					
0xE0001FFF					
0xE0002000					