3.4 ARM Addressing Modes

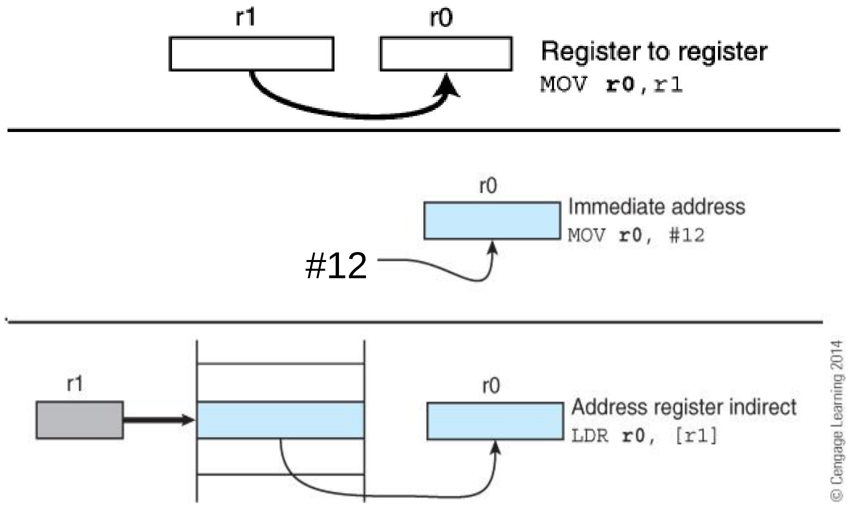
Addressing Mode

Discussed previously:

- *Register-to-register*:
  - e.g., ADD r1, r2, r3
- *Literal or immediate*: the actual value is part of the instruction
  - e.g., ADD r1, r2, #5 performs [r1] ← [r2] + 5

New:

- Register indirect



Register indirect addressing

Register indirect: a register contains the address of the operand

- e.g., LDR r1, [r0] copies to r1 the content of the memory location with address stored in r0.

FIGURE 3.31 Register indirect addressing

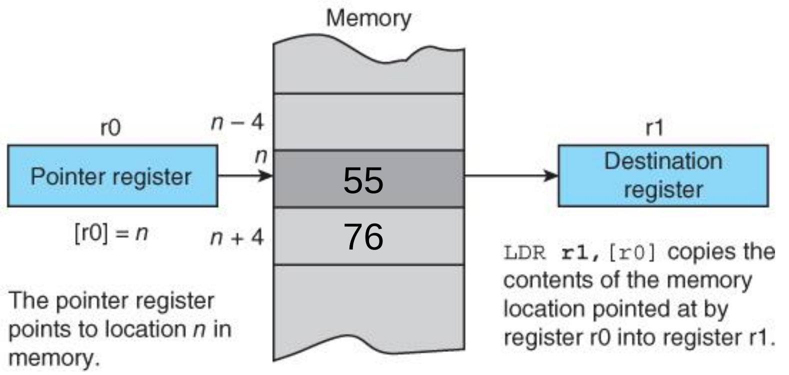
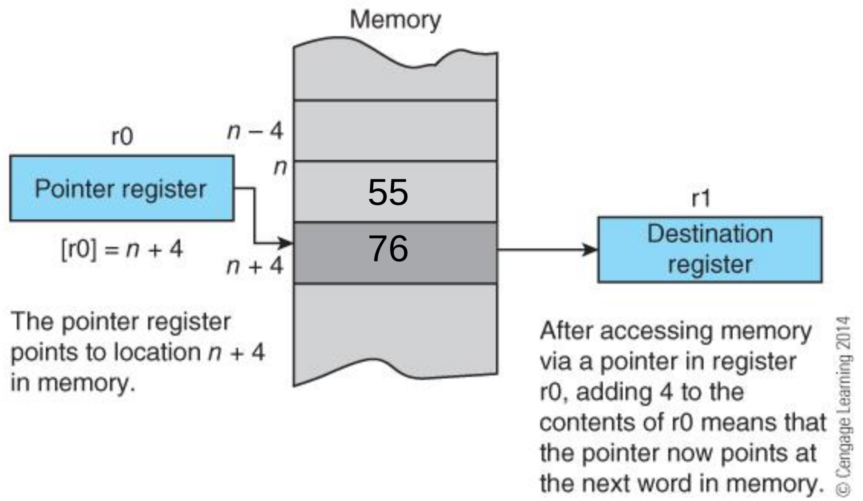

Memory
r0
Pointer register
[r0] = n
n − 4
n
n + 4
55
76
r1
Destination register
The pointer register points to location n in memory.
LDR r1,[r0] copies the contents of the memory location pointed at by register r0 into register r1.

FIGURE 3.32 Effect of incrementing the pointer register


Memory
r0
Pointer register
[r0] = n + 4
n − 4
n
n + 4
55
76
r1
Destination register
The pointer register points to location n + 4 in memory.
After accessing memory via a pointer in register r0, adding 4 to the contents of r0 means that the pointer now points at the next word in memory.
© Cengage Learning 2014

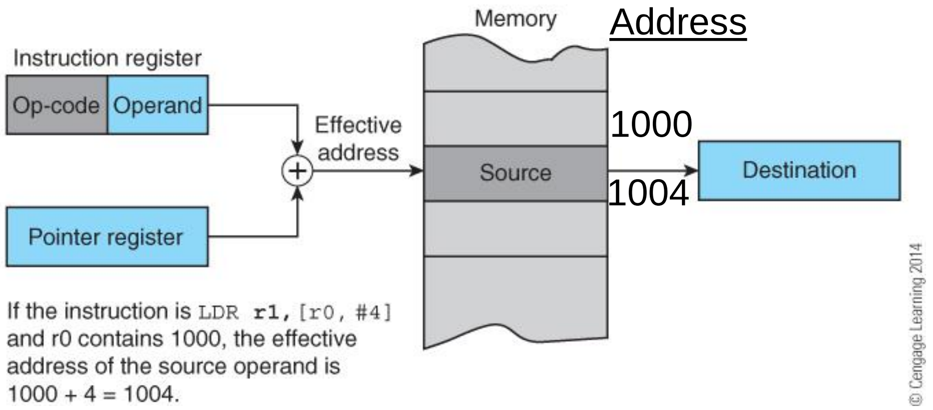
- Useful in access tables and arrays:
  - e.g., ADD r0, r0, #4 then LDR r1, [r0] moves next element of an array to r1.
  - ARM has facility for this to be done in one instruction.

Register Indirect addressing with an offset

(a) With a literal offset

- Effective address of operand = base address contained in a register + literal offset
- The literal offset is unsigned 12 bits (i.e., 0-4095 bytes)
- Example: `LDR r1, [r0, #4]`

FIGURE 3.33 Register indirect addressing with an offset



- The unsigned offset can also be subtracted from the base address: e.g., `LDR r1, [r0, #-4]`

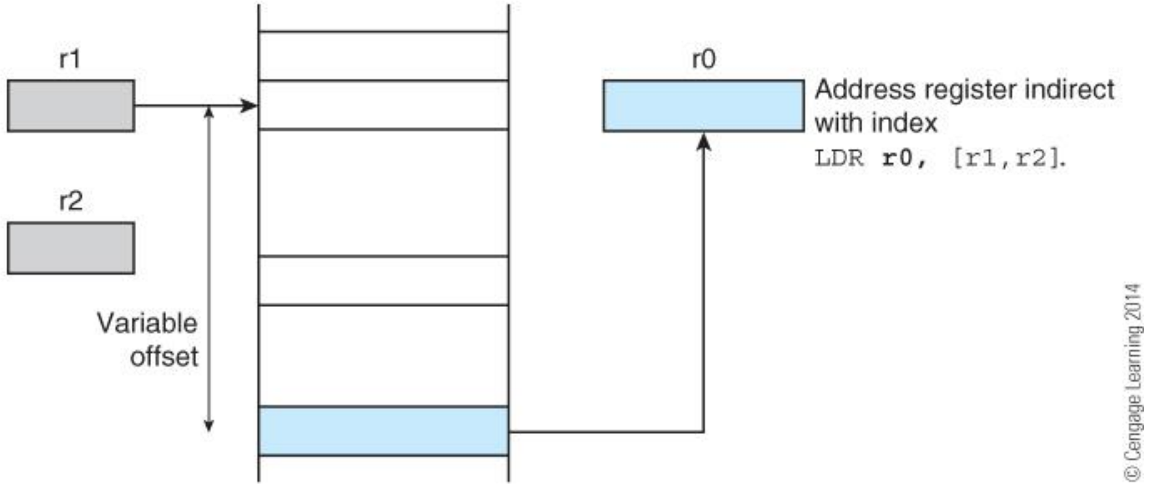
Register Indirect addressing with an offset

(b) With the of set as a second register

- `LDR r2,[r0,r1]`
  - [r2] ← [[r0] + [r1]]
  - load r2 with the location pointed at by r0 plus r1
- `LDR r2,[r0,r1,LSL #2]`
  - [r2] ← [[r0] + 4 x [r1]]
  - Register r1 is scaled by 4. This allows you to use a scaled offset when dealing with arrays.
- The register offset can also be subtracted from the base address:

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

FIGURE 3.35 Indexed addressing with a register offset

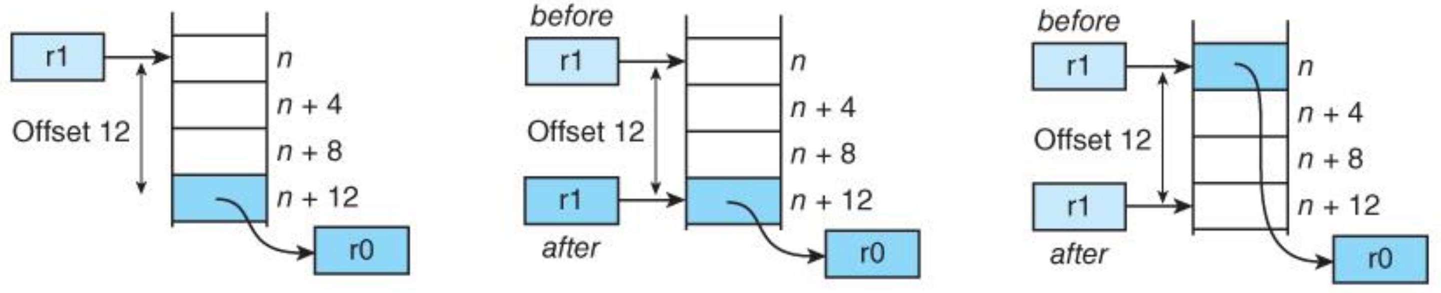


Pre-indexing and post-indexing

- Elements in an array or similar data structure are frequently accessed sequentially.
- Auto-indexing addressing modes allow pointers to be automatically adjusted to point at the next element.
- Two autoindexing modes:
  - Pre-indexed mode: Pointer updated before memory access
  - Post-indexed mode: Pointer updated after memory access

Pre-indexing and post-indexing

FIGURE 3.38 Register indirect addressing with offset



(a) `LDR r0,[r1,#12]`
Offset added to base register to generate effective address. Operand accessed at effective address. Base register remains unchanged.

(b) `LDR r0,[r1,#12]!`
Offset added to base register to generate effective address. Operand accessed at effective address. Base register updated after access.

(c) `LDR r0,[r1], #12`
Effective address specified by base register. Operand accessed at effective address. Offset added to base register after the access.

<u>Pre-indexing</u>

- Increment address then access memory

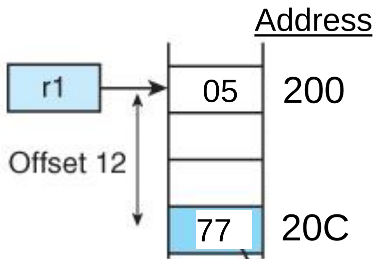
<u>Post-indexing</u>

- Access memory then increment address

Pre-indexing and post-indexing

- Before operation:

  `[r1] = 0x200, [0x200] = 0x05,`
  `[0x20C] = 0x77`



- After operation:
  - `LDR r0, [r1, #12]`

    `[r1] = 0x200, [r0] = 0x77`
  - `LDR r0, [r1, #12]!` **(Pre-indexing)**

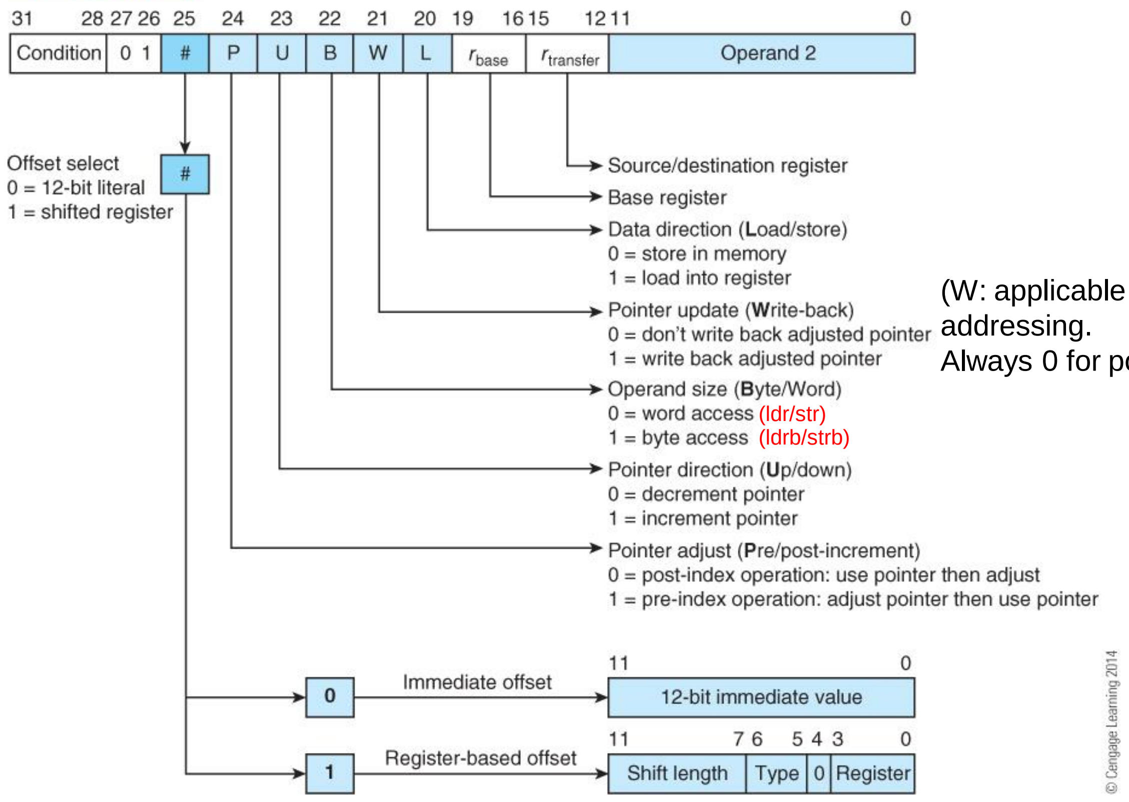
    `[r1] = 0x20C, [r0] = 0x77`
  - `LDR r0, [r1], #12` **(Post-indexing)**

    `[r1] = 0x20C, [r0] = 0x05`

ARM’s load and store encoding

- Knowing about the encoding provides an understanding of what options are available.

FIGURE 3.39 Format of ARM’s load and store instructions



(W: applicable only to pre-index addressing.
Always 0 for post-index addressing)

ARM’s load and store encoding

- W: applicable only to pre-index addressing.
- In the case of post-indexed addressing, the write back bit is *redundant* and must be set to zero.
- e.g., `ldr r0, [r1], #4`
- The non-zero offset of 4 indicates the programmer wants to increment (write-back) to address to the base address. Otherwise, he/she should have written `ldr r0, [r1]` (i.e., without the offset)
- Therefore, post-indexed data transfers always write back the modified base.

Example 1

e.g., `strpl r4, [r2, -r6, LSL #2]!`

opcode = 0101 0111 0010 0010 0100 0001 0000 0110

= 0x57224106

Bits	Description	Code
31-28	Condition PL	0101
27-26	Defines a load/store instruction	01
25	Use shift register	1
24	P: Pre-index	1
23	U: Decrement Pointer (in –r6)	0
22	B: Word	0
21	W: Write back adjusted pointer	1
20	L: Store	0
19-16	r_{base}= r2	0010 (i.e., 2 in hex)
15-12	$r_{transfer}$ = r4	0100 (i.e., 4 in hex)
11-7	Shift = 2 in LSL #2	00010 (i.e., 2 in hex)
6-5	LSL	00
4	Fixed	0
3-0	r6	0110 (i.e., 6 in hex)

Example 2

e.g., `ldr r1, [r0], #4`
opcode = 1110 0100 1001 0000 0001 0000 0000 0100
= 0xE4901004

Bits	Description	Code
31-28	Condition AL	1110
27-26	Defines a load/store instruction	01
25	Use 12-bit immediate value	0
24	P: Post-index	0
23	U: Increment Pointer	1
22	B: Word	0
21	W: Always 0 for post-indexing	0
20	L: Load	1
19-16	r_{base}= r0	0000 (i.e., 2 in hex)
15-12	$r_{transfer}$ = r1	0001 (i.e., 4 in hex)
11-0	Immediate value of 4	0000 0000 0100 (i.e., 4 in hex)

Summary of ldr/str options

- Address accessed by LDR/STR is specified by a base register plus an offset
- Offset can be:
  - An unsigned 12-bit immediate value (i.e., 0 - 4095 bytes).

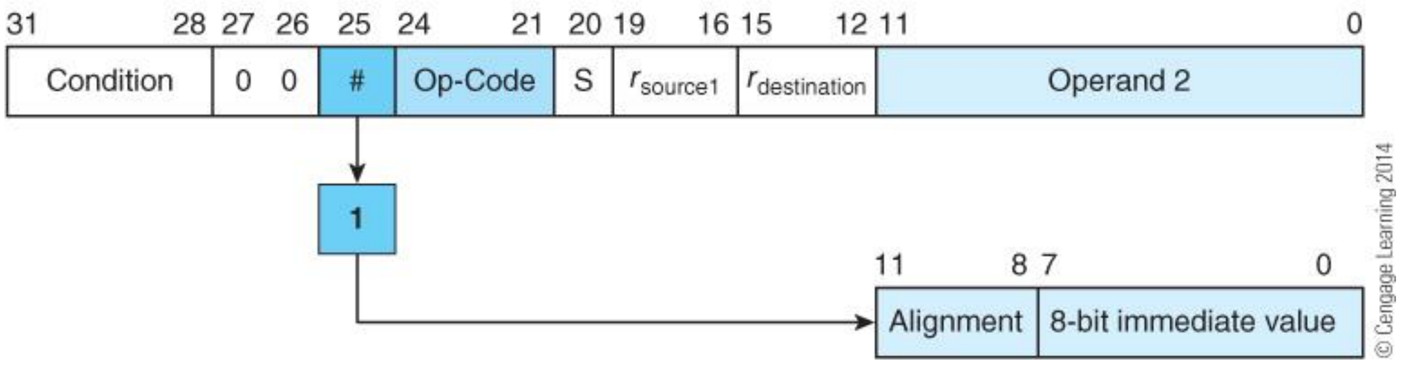
    ```
    LDR r0, [r1, #8]
    ```
  - A register, optionally shifted by an immediate value

    ```
    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]
  ```
- Choice of *pre-indexed* or *post-indexed addressing*

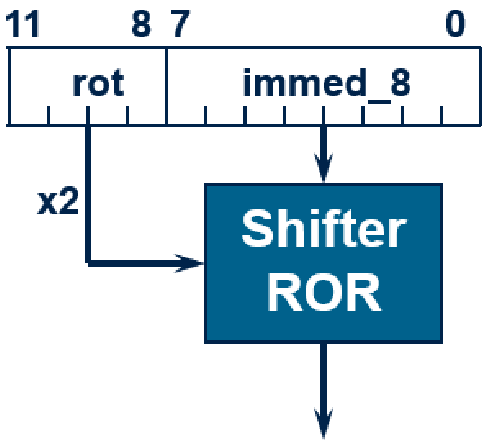
ldr pseudoinstruction

FIGURE 3.28 Diagram of ARM's literal operand encoding



Recall about literal encoding:

- A data processing instruction (including `mov`) has 12 bits available for the literal consisting of an 8-bit immediate value and number of `ror` encoded
- Error occurs when loading a literal using `mov` if the literal cannot be represented by an 8-bit value with even number of `ror`.
- How does ARM address this error so that all 32-bit values can be loaded?



ldr pseudoinstruction

`LDR rd, =const`

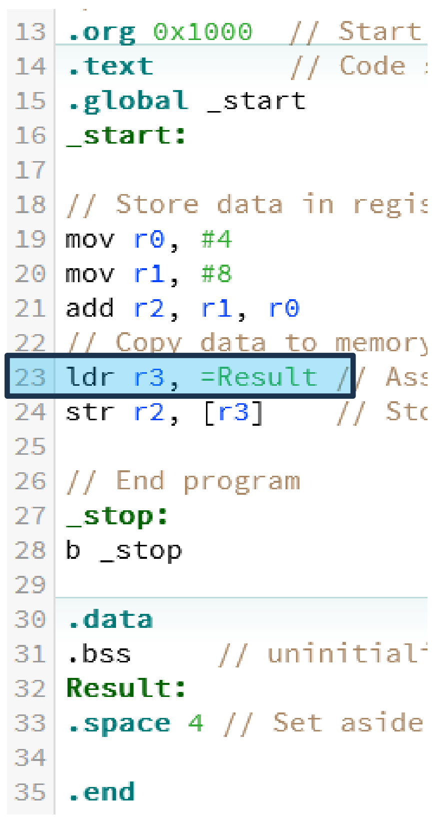
- This will either:
  - Produce a `mov` or `mvn` instruction to generate the value (if possible).

  or
  - Generate an `ldr` instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- For example

```
– LDR r0,=0xFF          =>     MOV r0,#0xFF
– LDR r0,=0x55555555    =>     LDR r0,[PC,#Imm12]
                               …
                               …
                               0x55555555
```

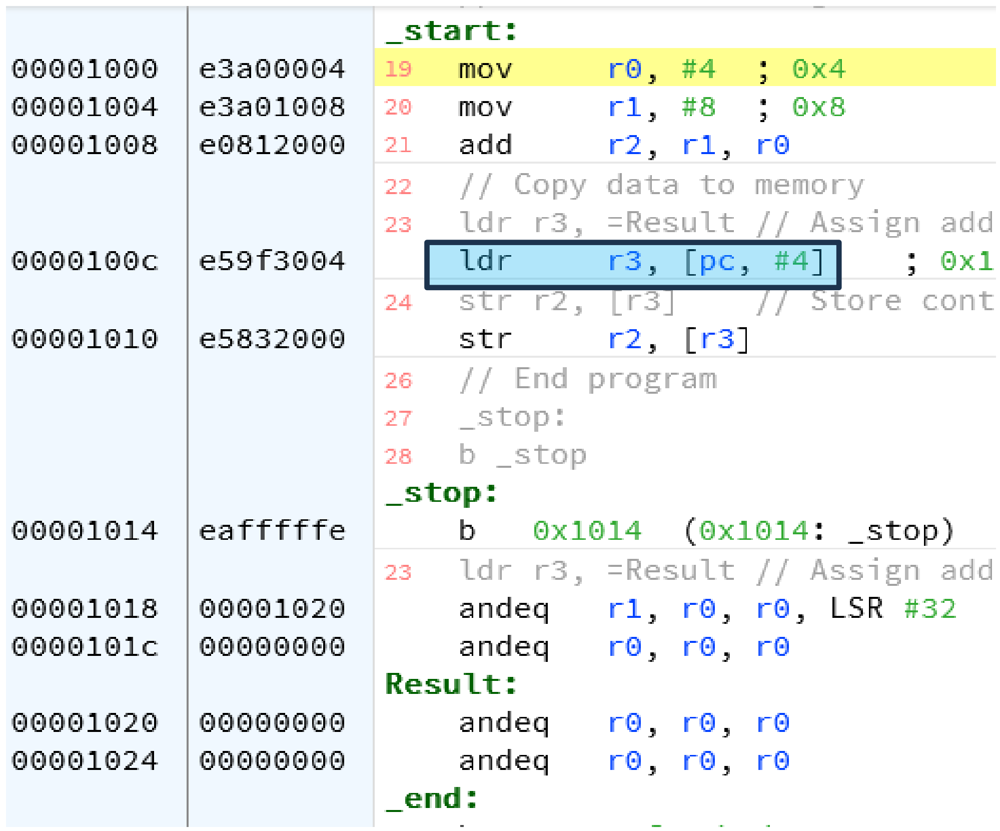
- This is the recommended way of loading constants into a register
- Called pseudoinstruction because this is not part of the processor's instruction set.

Example use in Lab 2



```
.org 0x1000   // Start
.text         // Code :
.global _start
_start:

// Store data in regis
mov r0, #4
mov r1, #8
add r2, r1, r0
// Copy data to memory
ldr r3, =Result // Ass
str r2, [r3]     // Stc

// End program
_stop:
b _stop

.data
.bss        // uninitial
Result:
.space 4 // Set aside

.end
```

Source code



```
                       _start:
00001000   e3a00004       mov       r0, #4    ; 0x4
00001004   e3a01008       mov       r1, #8    ; 0x8
00001008   e0812000       add       r2, r1, r0
                          // Copy data to memory
                          ldr r3, =Result // Assign add
0000100c   e59f3004       ldr       r3, [pc, #4]       ; 0x1
                          str r2, [r3]       // Store cont
00001010   e5832000       str       r2, [r3]
                          // End program
                          _stop:
                          b _stop
                       _stop:
00001014   eafffffe       b    0x1014   (0x1014: _stop)
                          ldr r3, =Result // Assign add
00001018   00001020       andeq     r1, r0, r0, LSR #32
0000101c   00000000       andeq     r0, r0, r0
                       Result:
00001020   00000000       andeq     r0, r0, r0
00001024   00000000       andeq     r0, r0, r0
                       _end:
```

Assembled code

Example use in Lab 2

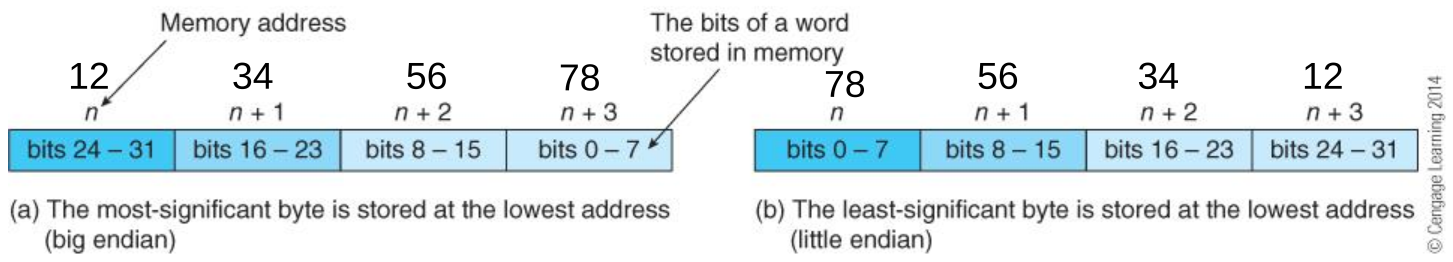
- Result is the starting address of the word we try to allocate using .space
- From the assembled code, you can see Result represents the address 0x1020
- `ldr r3, =Result` is actually `ldr r3, =0x1020`
- The assembler generate a literal 0x1020 and put it in address 0x1018 (i.e., at the end of the code section). This is an item in the literal pool.
- The instruction `ldr r3, [pc, #4]` tries to retrieve the content of the address 0x1018 and put it in r3.
- The address 0x1018 is 12 bytes under the ldr instruction (at 0x100C)
- For our example,
  - pc = 0x100C + 8 = 0x1014 (pipelining)
  - pc + 4 = 0x1018

ldr/str byte/halfword

- We can load or store byte (8-bit) or halfword (16-bit)
- We need to understand two ways of data organization:
  - Little Endian: Most significant byte (MSB) stored in the highest address
  - Big Endian: MSB stored in lowest address

Example: 12345678

FIGURE 3.52 Loading four bytes in stored *longword*



(a) The most-significant byte is stored at the lowest address (big endian)

(b) The least-significant byte is stored at the lowest address (little endian)

Big Endian

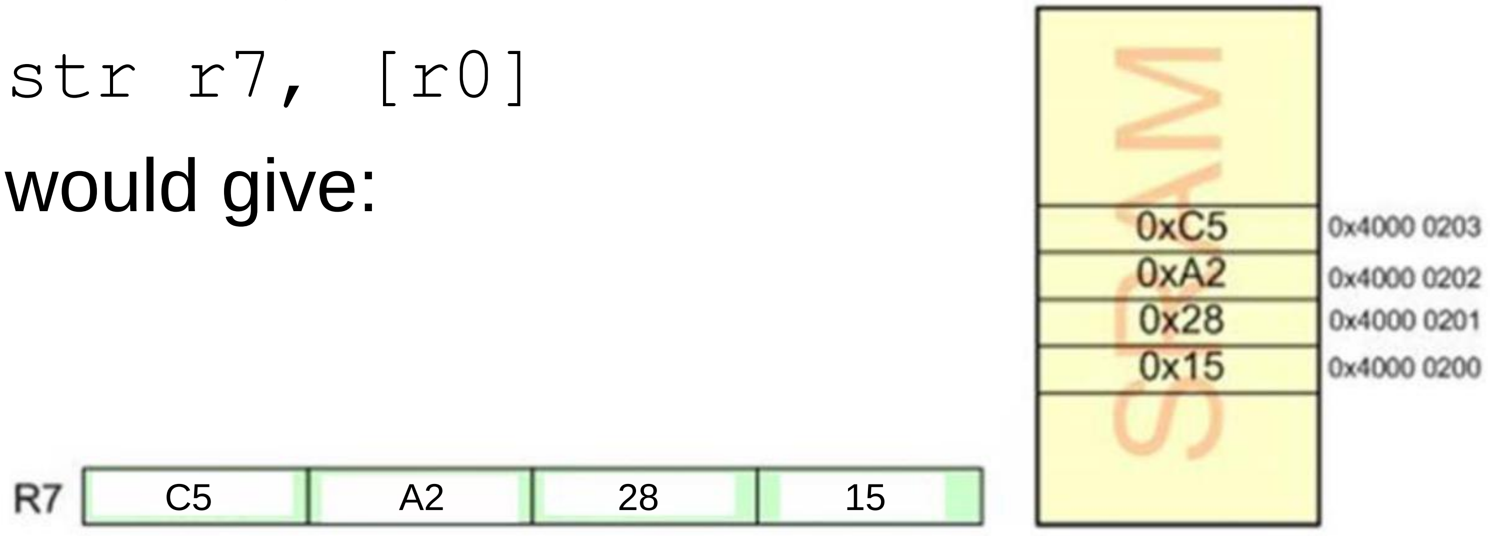
Little Endian

Little Endian

- We use little endian throughout the course.
- For example,

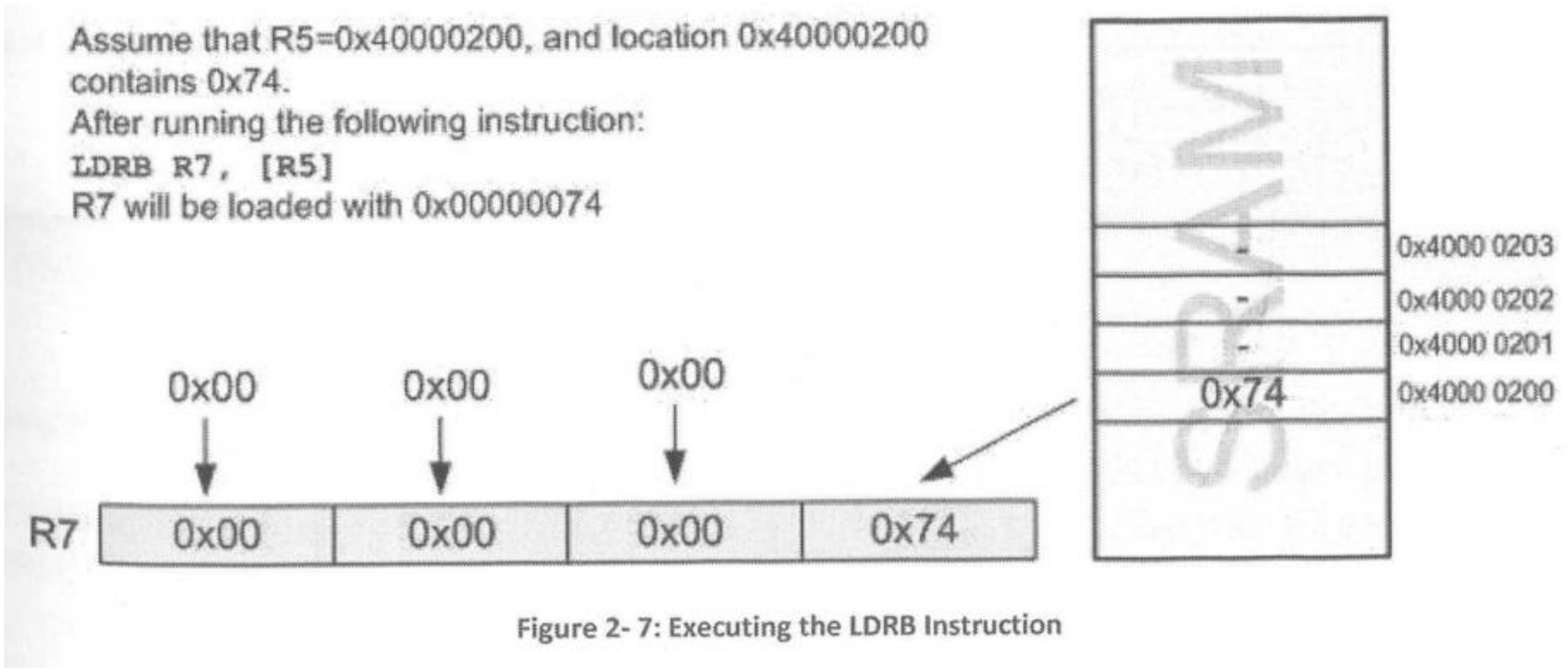
```
ldr r0, =0x40000200
str r7, [r0]
```

would give:



Byte load

- `ldrb Rd, [Rx]` loads one byte from a memory location pointed to by Rx into the least significant byte of Rd.



Assume that R5=0x40000200, and location 0x40000200 contains 0x74.
After running the following instruction:
`LDRB R7, [R5]`
R7 will be loaded with 0x00000074

Figure 2- 7: Executing the LDRB Instruction

Byte store

- `strb Rd, [Rx]` stores the least significant byte of Rx to the memory location pointed to by Rx

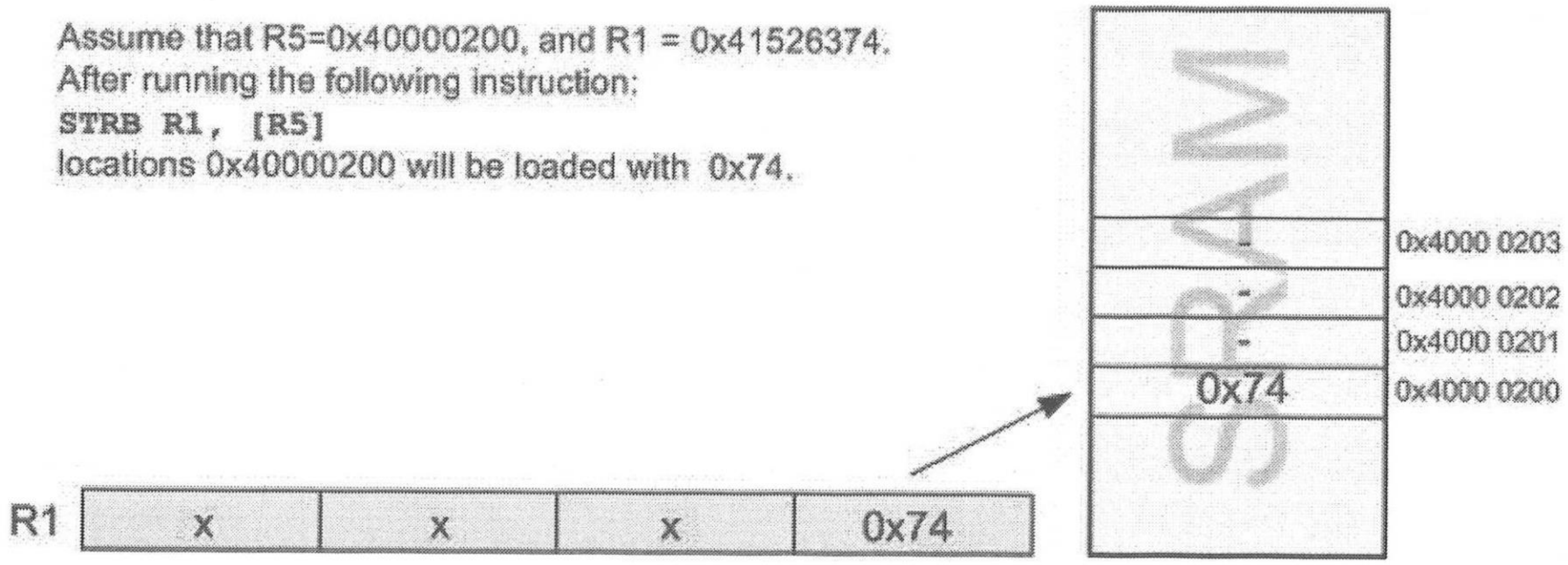
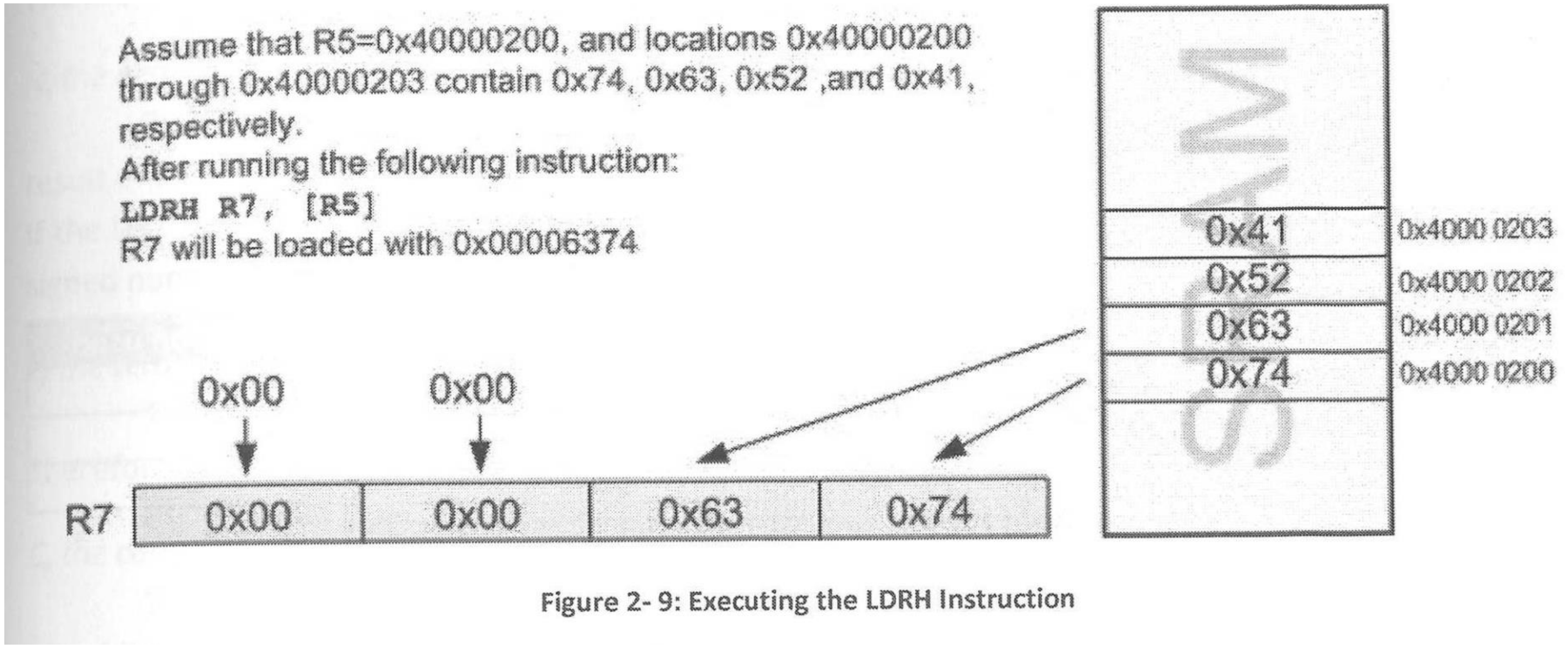

Assume that R5=0x40000200, and R1 = 0x41526374.
After running the following instruction:
STRB R1, [R5]
locations 0x40000200 will be loaded with 0x74.

Figure 2- 8: Executing the STRB Instruction

Half-word load

- `ldrh Rd, [Rx]` loads two bytes (half-word) from a memory location pointed to by Rx into the least significant byte of Rd.

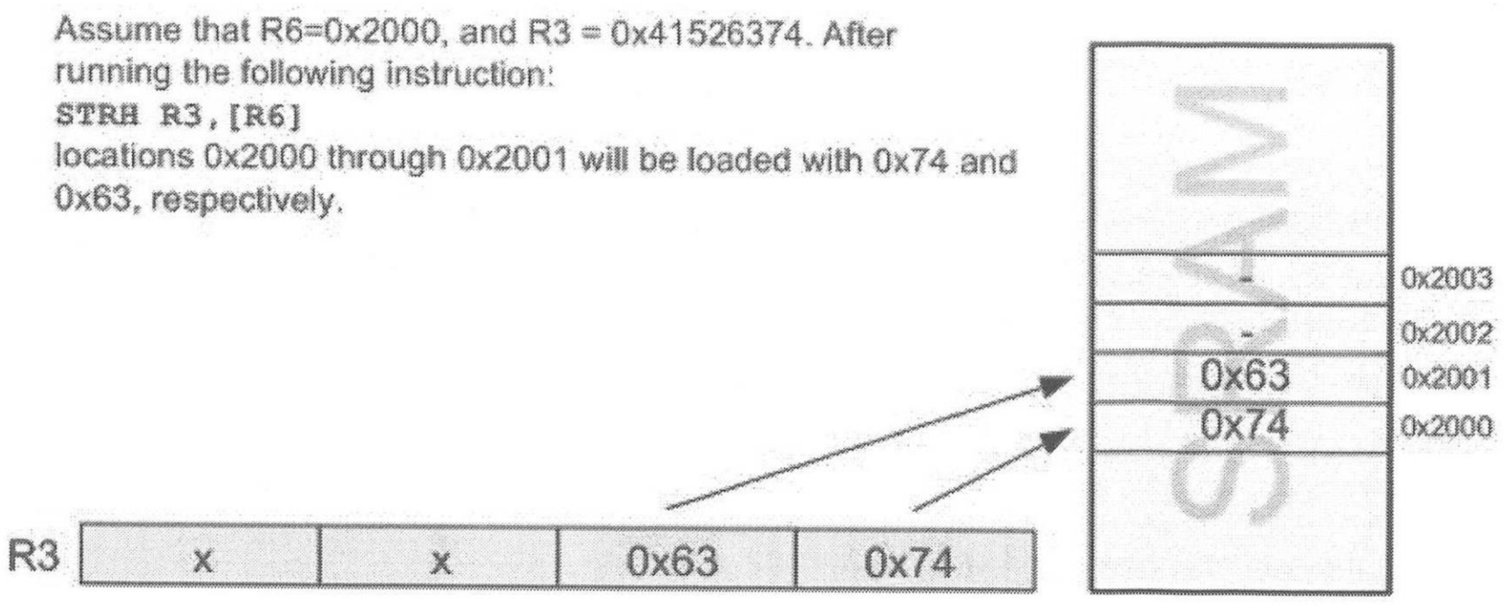


Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.
After running the following instruction:
`LDRH R7, [R5]`
R7 will be loaded with 0x00006374

Figure 2- 9: Executing the LDRH Instruction

Half-word store

- `strh Rd, [Rx]` stores the lower 16-bit contents to the memory location pointed to by Rx


Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:

```
STRH R3,[R6]
```

locations 0x2000 through 0x2001 will be loaded with 0x74 and 0x63, respectively.

Figure 2- 10: Executing the STRH Instruction

Half-word/byte load with sign extension

- Sign extension of signed number
  - Positive number: e.g., extending the 8-bit number of 56_{16} to a 32-bit number will result in 00000056_{16}
  - Negative number: e.g., extending the 8-bit number of 82_{16} to a 32-bit number will result in $FFFFFF82_{16}$
    - 82_{16} is $-7E_{16}$
    - With 32-bit, this number is represented as the 2'complement of $0000007E_{16} \rightarrow FFFFFF82_{16}$
  - Thus, signed extension involves copying the signed bit (D7) to the upper 24 bits of the 32-bit register. Similar for half-word.

Half-word/byte load with sign extension

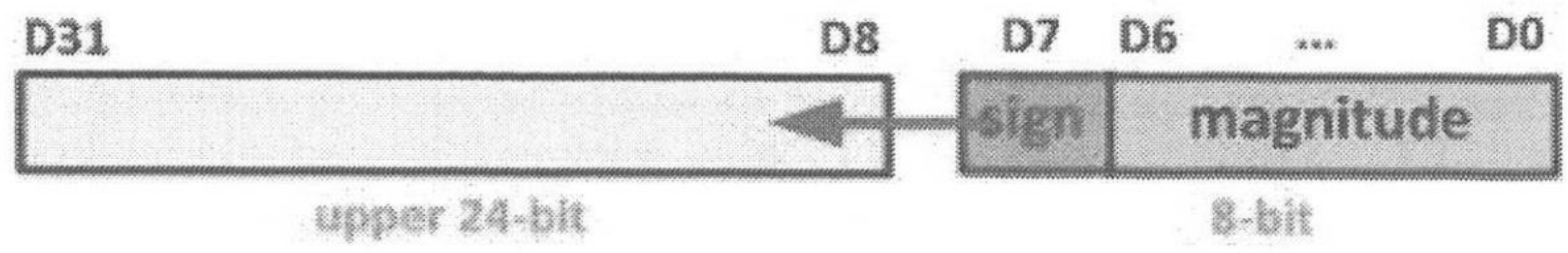


Figure 5-1: Sign Extending a Byte

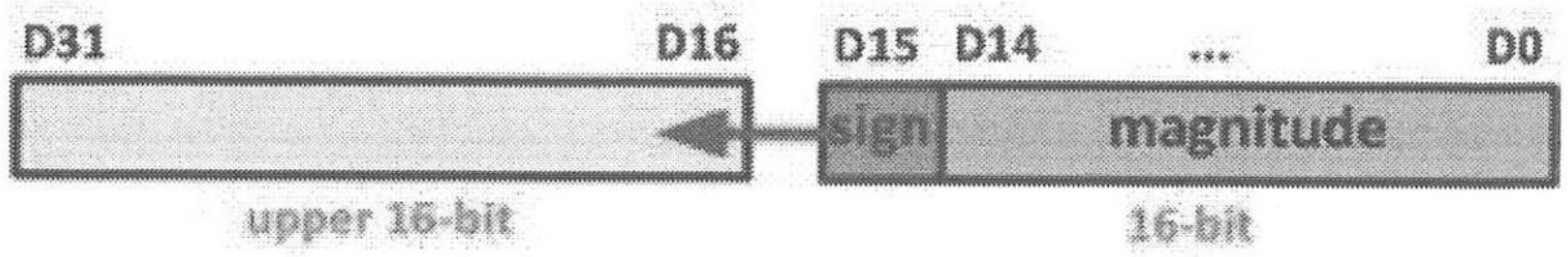
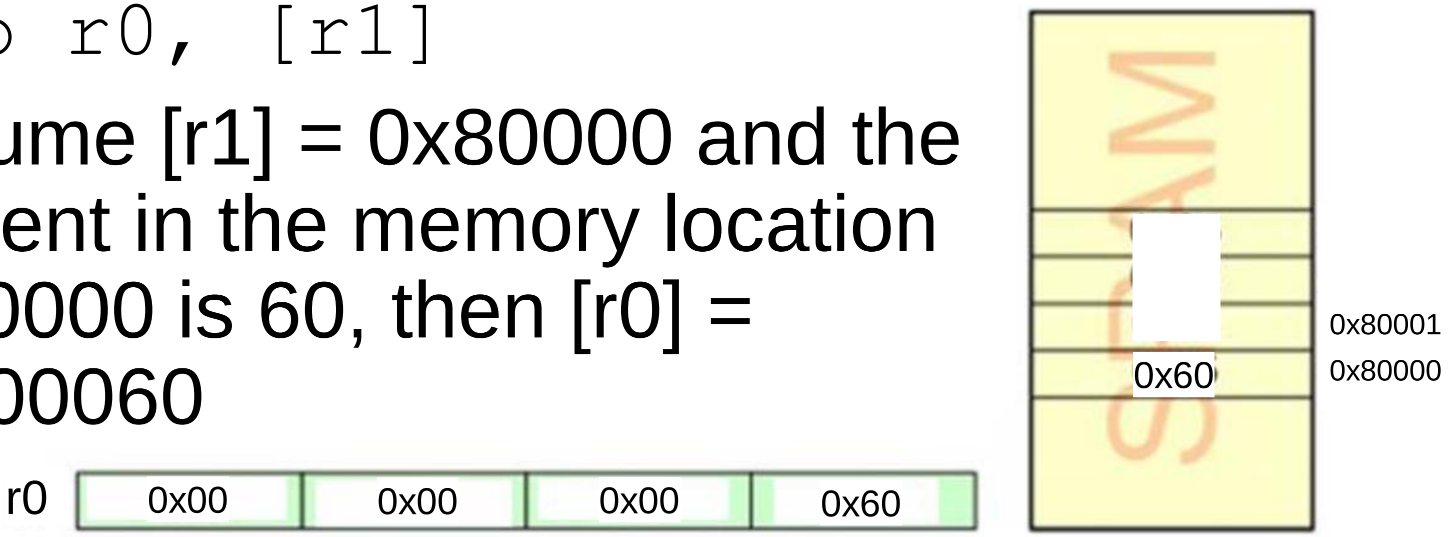


Figure 5- 2: Sign Extending a Half-word

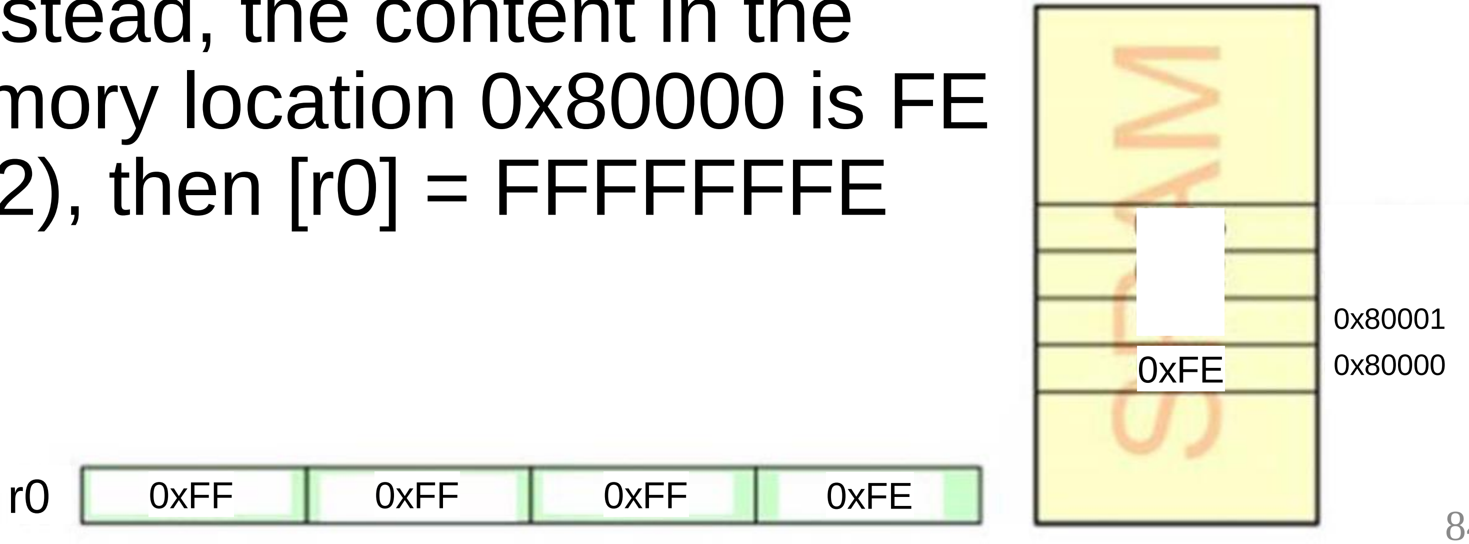
Byte load with sign extension

```
ldrsb r0, [r1]
```

- Assume [r1] = 0x80000 and the content in the memory location 0x80000 is 60, then [r0] = 00000060



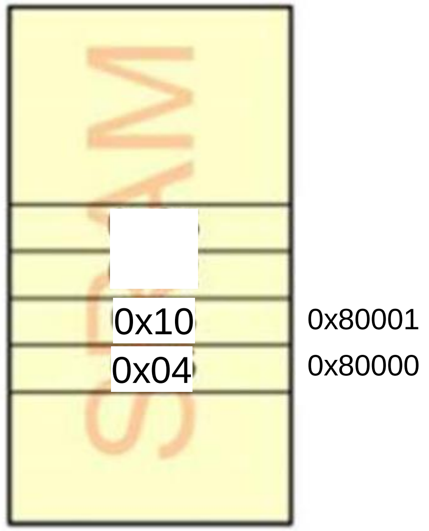
- If instead, the content in the memory location 0x80000 is FE (= -2), then [r0] = FFFFFFFE

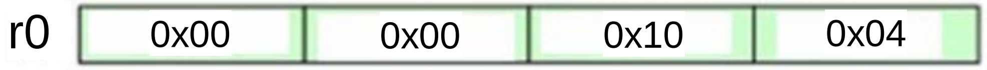


Half-word load with sign extension

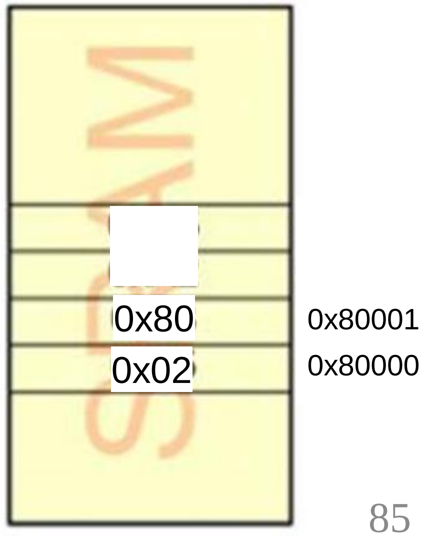
```
ldrsh r0, [r1]
```

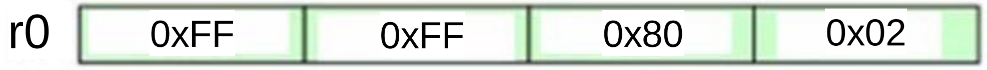
- Assume [r1] = 0x80000 and the content in the memory location 0x80000 is 0104, then [r0] = 00000104



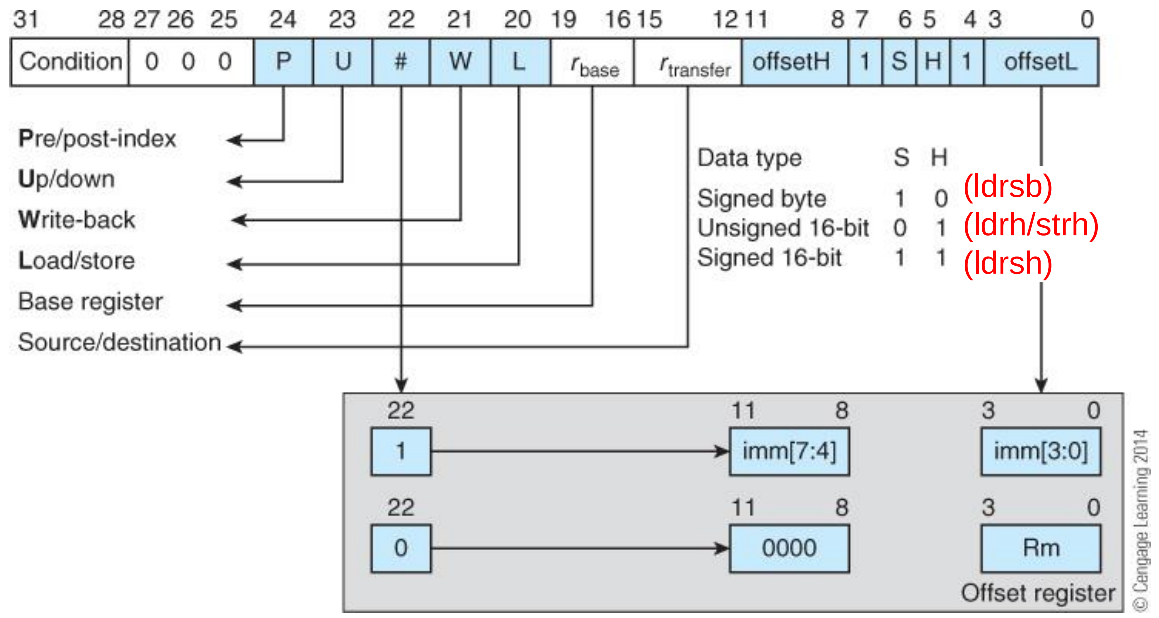


- If instead, the content in the memory location 0x80000 is 8002 (= -32766$_{10}$), then [r0] = FFFF8002



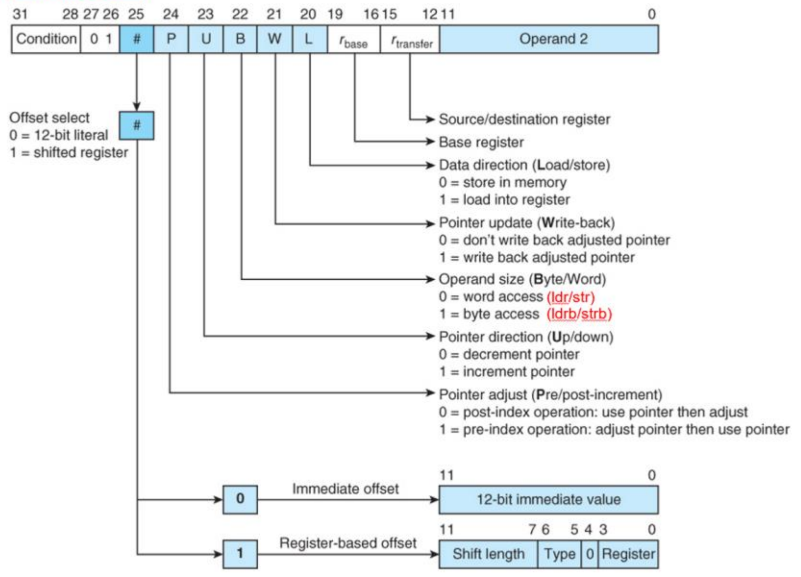
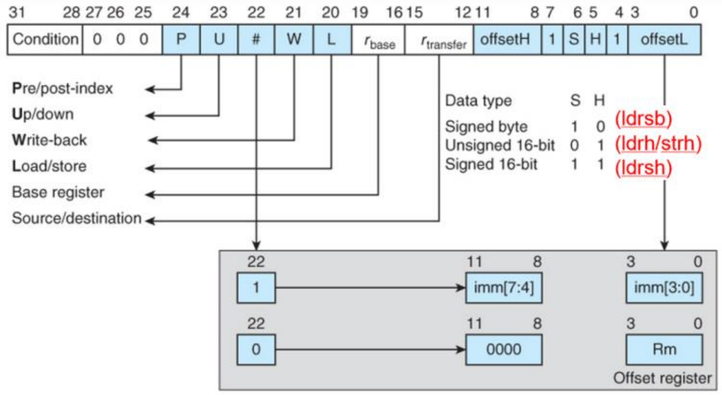


Half-word/byte load/store encoding



- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8-bit immediate value (i.e., 0-255 bytes).
  - A register (unshifted).

Regular load vs half-word/byte load

LDR/STR/LDRB/STRB	LDRH/STRH/LDRSB/LDRSH
FIGURE 3.39 Format of ARM's load and store instructions  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 0 Condition 0 1 # P U B W L r_base r_transfer Operand 2 Offset select 0 = 12-bit literal 1 = shifted register Source/destination register Base register Data direction (Load/store) 0 = store in memory 1 = load into register Pointer update (Write-back) 0 = don't write back adjusted pointer 1 = write back adjusted pointer Operand size (Byte/Word) 0 = word access (ldr/str) 1 = byte access (ldrb/strb) Pointer direction (Up/down) 0 = decrement pointer 1 = increment pointer Pointer adjust (Pre/post-increment) 0 = post-index operation: use pointer then adjust 1 = pre-index operation: adjust pointer then use pointer Immediate offset 0 11 0 12-bit immediate value Register-based offset 1 11 7 6 5 4 3 0 Shift length Type 0 Register	 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 Condition 0 0 0 P U # W L r_base r_transfer offsetH 1 S H 1 offsetL Pre/post-index Up/down Write-back Load/store Base register Source/destination Data type S H Signed byte 1 0 (ldrsb) Unsigned 16-bit 0 1 (ldrh/strh) Signed 16-bit 1 1 (ldrsh) 22 11 8 3 0 1 imm[7:4] imm[3:0] 22 11 8 3 0 0 0000 Rm Offset register
Offset can be:	
An unsigned 12-bit immediate value (i.e., 0 - 4095 bytes). • e.g., LDR r0, [r1, #x], where x ranges from 0 to 4095 bytes	An unsigned 8-bit immediate value (i.e., 0-255 bytes). • e.g., LDRH r0, [r1, #x], where x ranges from 0 to 255 bytes
A register, optionally shifted by an immediate value • Unshifted: e.g., LDR r0, [r1, r2] • Shifted: e.g., LDR r0, [r1, r2, LSL #x], where x ranges from 0 to 31 bits	Unshifted register: • e.g., LDRH r0, [r1, r2] • Shifting is not allowed
Other options are the same, including	
(a) Offset can be added or subtracted:	
LDR r0, [r1, #-8] LDR r0, [r1, -r2]	LDRH r0, [r1, #-8] LDRH r0, [r1, -r2]
Post-indexing and pre-indexing options are the same	

Regular ldr/str vs half-word/byte ldr/str

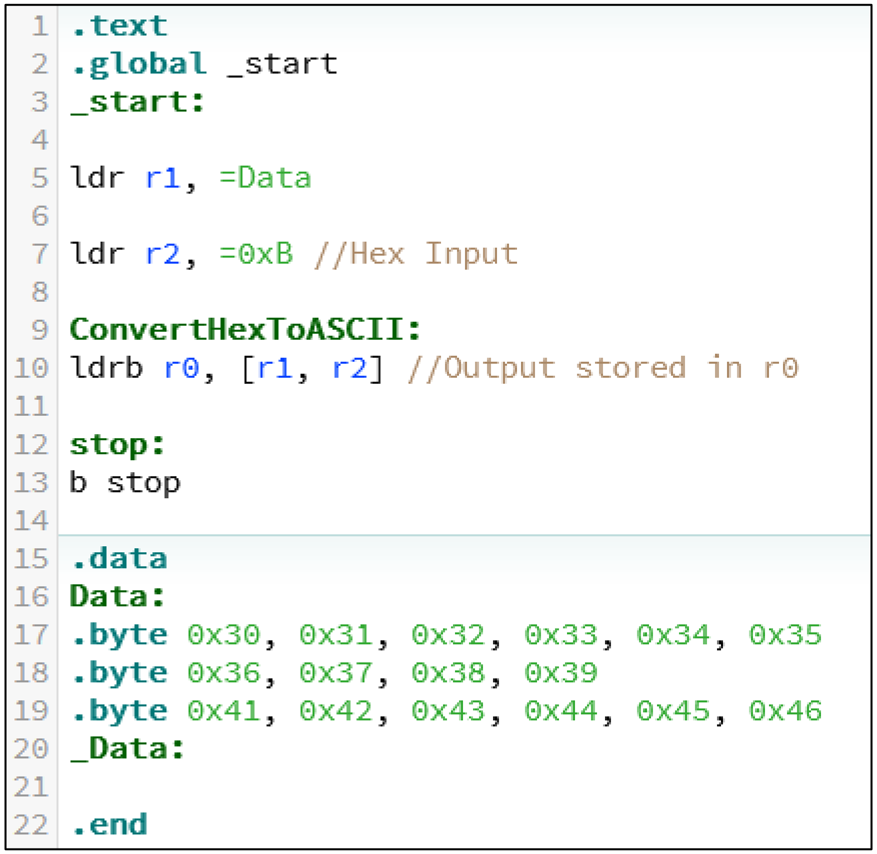
Legal	Illegal
ldr r0, [r1, #4000] ldrb r0, [r1, #4000] ldr r0, [r1, r2] ldrh r0, [r1, r2] ldr r0, [r1, r2, LSL #30]	ldrh r0, [r1, #4000] (offset is limited to 0 to 255) ldrh r0, [r1, r2, LSL #30] (no shifting is allowed for ldrh) ldr r0, [r1, r2, LSL r3] ldrh r0, [r1, r2, LSL r3] (no dynamic shift offset is allowed for all ldr/str operations) ldr r0, [r1, r2, LSL #40] (shift is limited to 0 to 31)

Example: Lookup Table

This example converts a hexadecimal digit to its corresponding ASCII code according to the following lookup table:

Hexadecimal	ASCII
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39
A	0x41
B	0x42
C	0x43
D	0x44
E	0x45
F	0x46

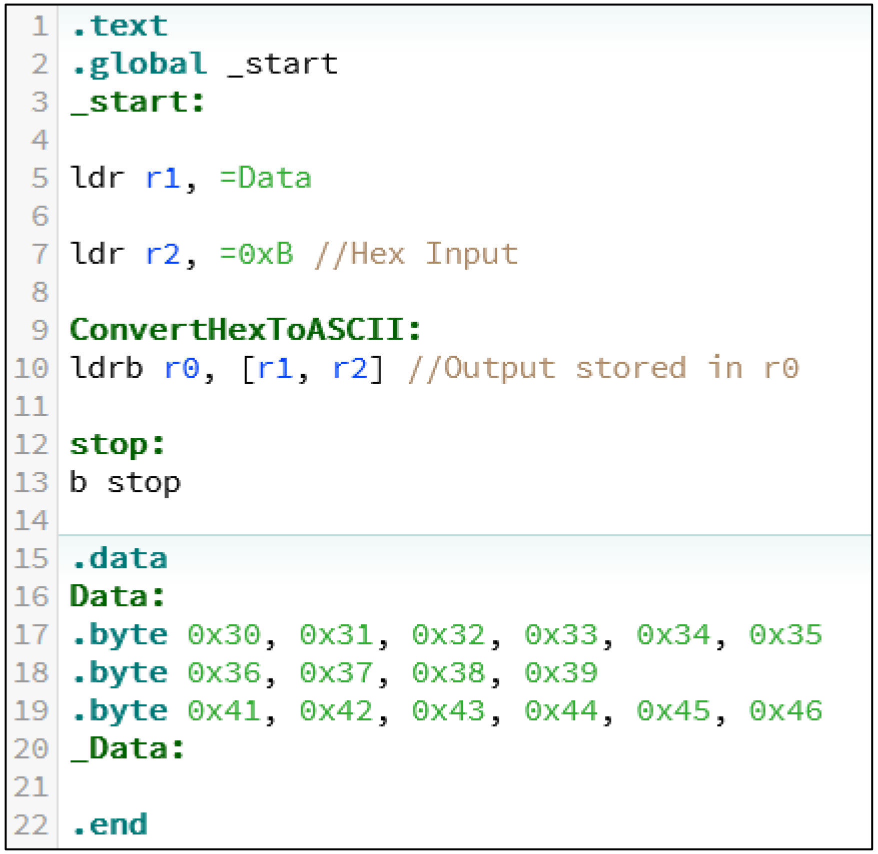
Example: Lookup Table



```
.text
.global _start
_start:

ldr r1, =Data

ldr r2, =0xB //Hex Input

ConvertHexToASCII:
ldrb r0, [r1, r2] //Output stored in r0

stop:
b stop

.data
Data:
.byte 0x30, 0x31, 0x32, 0x33, 0x34, 0x35
.byte 0x36, 0x37, 0x38, 0x39
.byte 0x41, 0x42, 0x43, 0x44, 0x45, 0x46
_Data:

.end
```

- *.text*: Identifies the code section of the program.
- *.data*: Defines a section of code that contains initialized data and variables rather than executable code.
- The list begins at the address labeled *Data*
- The list values are comma-separated and are byte-sized (as defined by *.byte*)

Example: Lookup Table



```
.text
.global _start
_start:

ldr r1, =Data

ldr r2, =0xB //Hex Input

ConvertHexToASCII:
ldrb r0, [r1, r2] //Output stored in r0

stop:
b stop

.data
Data:
.byte 0x30, 0x31, 0x32, 0x33, 0x34, 0x35
.byte 0x36, 0x37, 0x38, 0x39
.byte 0x41, 0x42, 0x43, 0x44, 0x45, 0x46
_Data:

.end
```

Hexadecimal	ASCII
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39
A	0x41
B	0x42
C	0x43
D	0x44
E	0x45
F	0x46

Example: Convert the hex digit B to its corresponding ASCII code

- Set [r2] $\leftarrow$ 0xB
- Set base address to be the address of the label *Data*, store it in r1.
- The desired ASCII code is located at address [r1]+[r2]
- Use ldrb r0, [r1, r2] to retrieve the desired ASCII code.

3.5 Looping and Branching

Looping in ARM

- Loop – repeating a sequence of instructions a certain number of times.
- e.g., To add 9 to r0 five times, I could do:

```
  mov r0, #0
  mov r1, #9
add r0, r0, r1
add r0, r0, r1
add r0, r0, r1
add r0, r0, r1
add r0, r0, r1
```

- Disadvantage: Too much code space is needed if repeating 100 times
- Use looping

Three steps in writing a *for* loop

- Step 1: Initialize variables
  - Count: specifies how many times you want to repeat the loop. (e.g., [r2] = 5 in previous example)
  - Other variables depending on application.
- Step 2: Identify statement(s) to repeat
  - In the previous example, it would be: `add r0, r0, r1`
- Step 3: Determine whether to stop looping
  - Use `subs r2, r2, #1` to decrement [r2] after one iteration (repetition) of the loop, and check whether [r2] is 0. Two possibilities:
    - If [r2] = 0 (i.e., finished the desired number of iterations), exit loop.
    - If [r2] ≠ 0, branch back to Step 2.

Example 1

- e.g., Write a program to (a) clear r0, (b) add 9 to r0 a thousand times.

```
mov r0, #0
ldr r2, =1000; Step 1
AddMore:
add r0, r0, #9; Step 2
subs r2, r2, #1; Step 3
bne AddMore
```

Z=0

General format of a for loop

```
MOV   r0,#10   // Step 1
Loop:
code ...       // Step 2
SUBS  r0,r0,#1 // Step 3
BNE   Loop
Post loop ...  ;fall through on zero count
```

Example 2

- Write a program to place the value 0x55 into 100 consecutive bytes starting from address 0x1500.
- Step 1: Initialization
  - [r1] = 0x55555555 (each time write 4 bytes)
  - [r0] = 25
  - [r2] = 0x1500
- Step 2: Statements to repeat
  - str r1, [r2], #4
- Step 3: Termination conditions
  - [r0] = [r0] – 1
  - Stop repetition if [r0] = 0

Example 2

```
mov r0, #25
ldr r1, =0x55555555
ldr r2, =0x1500

Again:
str r1, [r2], #4
subs r0, r0, #1
bne Again

Here:
b Here
```

While loop

- Pseudocode:

```
while expression
      statement(s)
end
```

- If *expression* is true, run *statement(s)* once.
- Assembly code:

```
Loop:
cmp r0, #0
beq WhileExit
…. statement(s)
b Loop
WhileExit:
Post loop ....; Exit
```

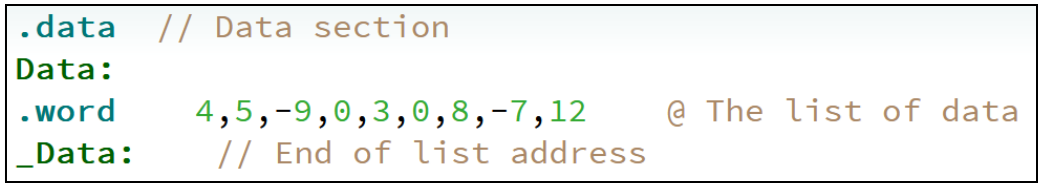
Example 1 using while loop

- e.g., Write a program to (a) clear r0, (b) add 9 to r0 a thousand times.

```
mov r0, #0
ldr r2, =1000 // Step 1
AddMore:
cmp r2, #0
beq WhileExit
add r0, r0, #9 // Step 2
sub r2, r2, #1 // Step 3
b AddMore
WhileExit: Post loop; Exit
```

Example on while loop from Lab 4

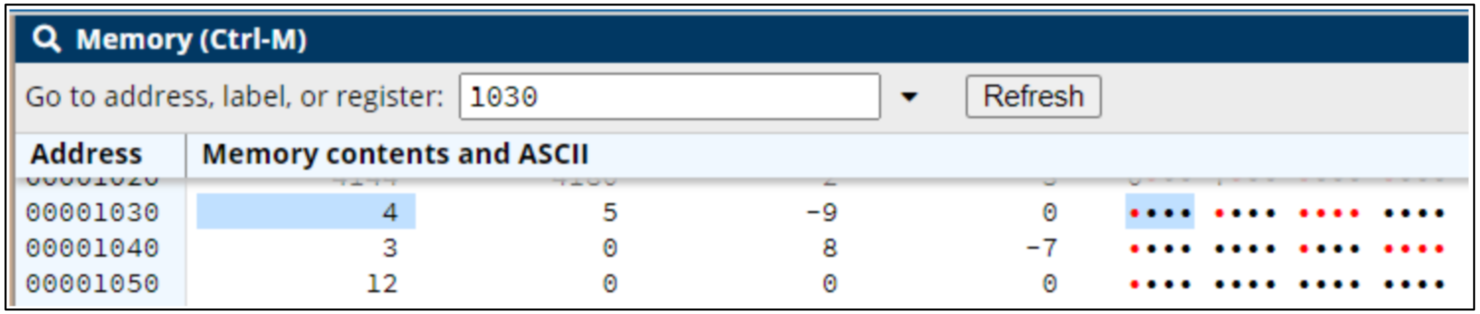
- This example traverses a list and operates on each value of the list.
- Let's have a good understanding of the list first:



```
.data   // Data section
Data:
.word     4,5,-9,0,3,0,8,-7,12     @ The list of data
_Data:     // End of list address
```

- The list begins at the address labeled *Data*
- The list values are comma-separated and are word-sized (as defined by *.word*)
- The address of the end of the list (i.e., immediately after the last item) is labeled as *_Data*. This is important as otherwise we have no way to know where the list ends.

Example on while loop from Lab 4

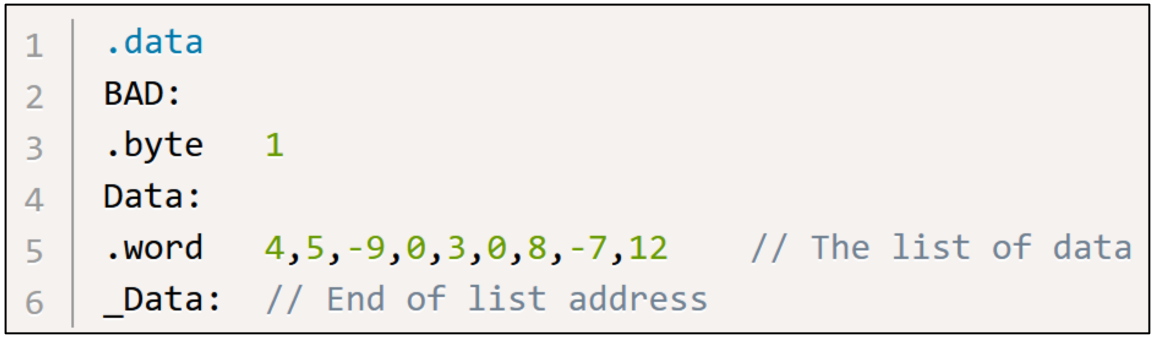


Memory (Ctrl-M)

Go to address, label, or register: 1030 Refresh

```
Address   Memory contents and ASCII
00001030           4          5         -9          0     .... .... .... ....
00001040           3          0          8         -7     .... .... .... ....
00001050          12          0          0          0     .... .... .... ....
```

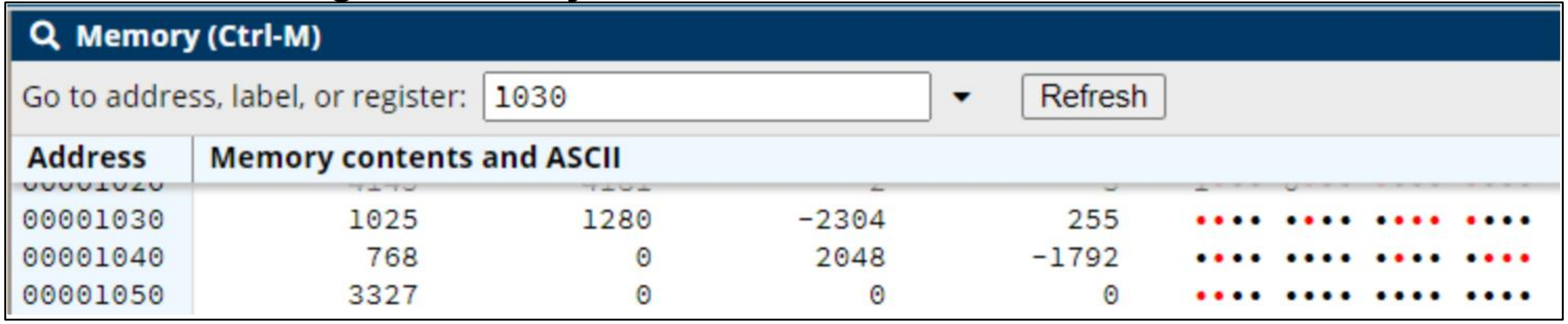
- Data items are stored correctly as displayed above because the list *Data* starts on the word boundary (address divisible by 4).
- However, this is not guaranteed. For example, the list would be unaligned by adding a single byte with a value of 1 in front of the *Data* list



```
.data
BAD:
.byte   1
Data:
.word   4,5,-9,0,3,0,8,-7,12    // The list of data
_Data:  // End of list address
```

Example on while loop from Lab 4

The resulting memory looks like:



```
Address    Memory contents and ASCII
00001030        1025      1280     -2304       255    •••• •••• •••• ••••
00001040         768         0      2048     -1792    •••• •••• •••• ••••
00001050        3327         0         0         0    •••• •••• •••• ••••
```

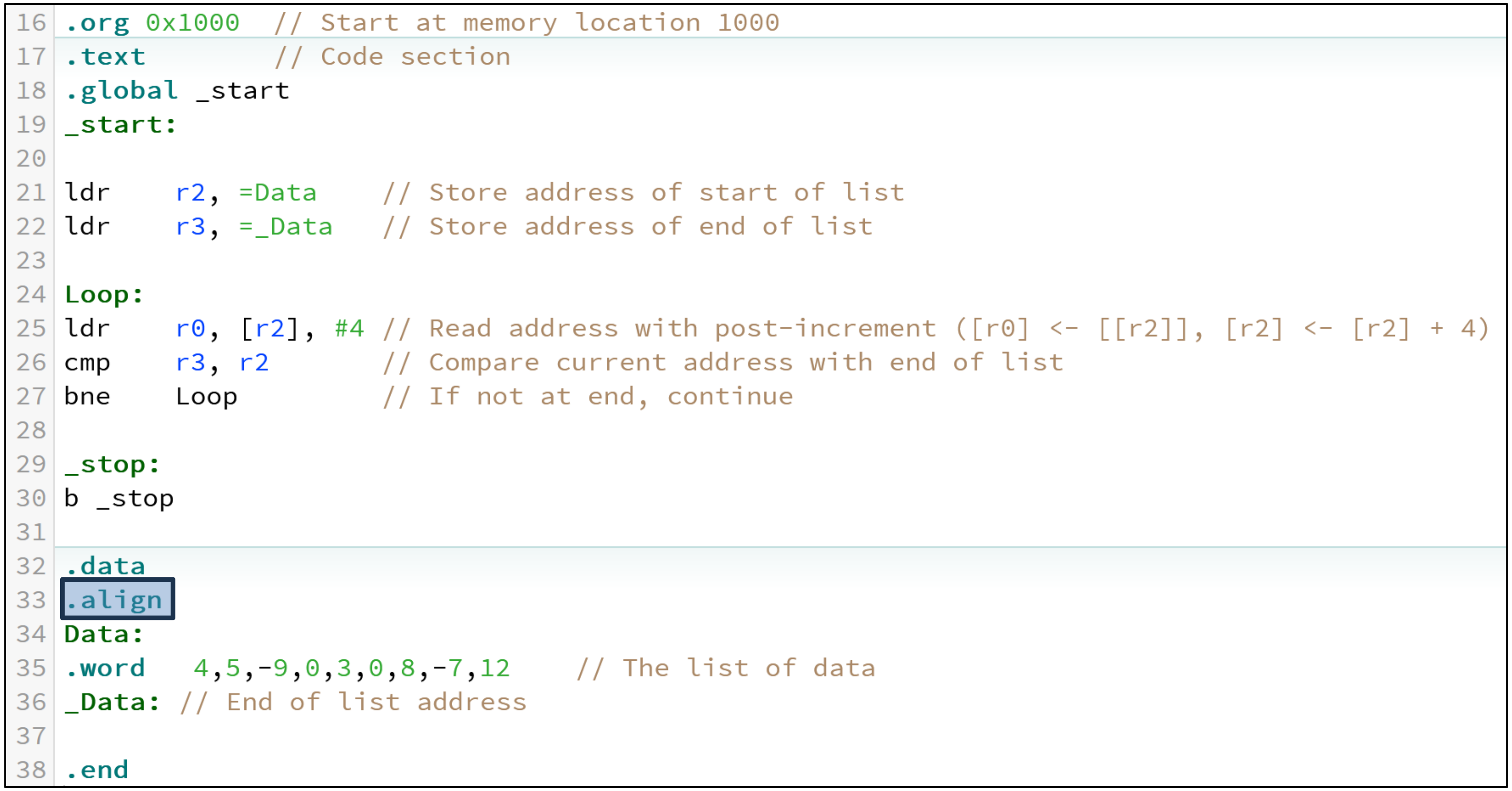
- The original list values are still there, but because they are now shifted by a byte, they no longer make sense in decimal.
- We can correct this with the *.align* directive:

```
.data
BAD:
.byte    1
.align   // Force list to start on a word boundary
Data:
.word    4,5,-9,0,3,0,8,-7,12     // The list of data
_Data:   // End of list address
```

.align: Fill address 1031 to 1033 by 0x00
Start writing the first word in address 1034

Example on while loop from Lab 4

This code traverses the list:



```
.org 0x1000   // Start at memory location 1000
.text         // Code section
.global _start
_start:

ldr     r2, =Data     // Store address of start of list
ldr     r3, =_Data    // Store address of end of list

Loop:
ldr     r0, [r2], #4 // Read address with post-increment ([r0] <- [[r2]], [r2] <- [r2] + 4)
cmp     r3, r2        // Compare current address with end of list
bne     Loop          // If not at end, continue

_stop:
b _stop

.data
.align
Data:
.word   4,5,-9,0,3,0,8,-7,12    // The list of data
_Data: // End of list address

.end
```

Example on while loop from Lab 4

```
ldr     r2, =Data     // Store address of start of list
...
ldr     r0, [r2], #4 // Read address with post-increment ([r0] <- [[r2]], [r2] <- [r2] + 4)
```

does the following:

- loads the value at the address stored in r2 to r0. This is initially the first element of the *Data* list.
- The value of r2 is then incremented by 4. r2 now points to the second element of the *Data* list.

```
.data   // Data section
Data:
.word    4,5,-9,0,3,0,8,-7,12    @ The list of data
_Data:     // End of list address
```

- This continues until some end condition is reached.

Example on while loop from Lab 4

The end condition in this case is to compare the current address in r3 against the address immediately following the end of the list, i.e., the value in _*Data*:

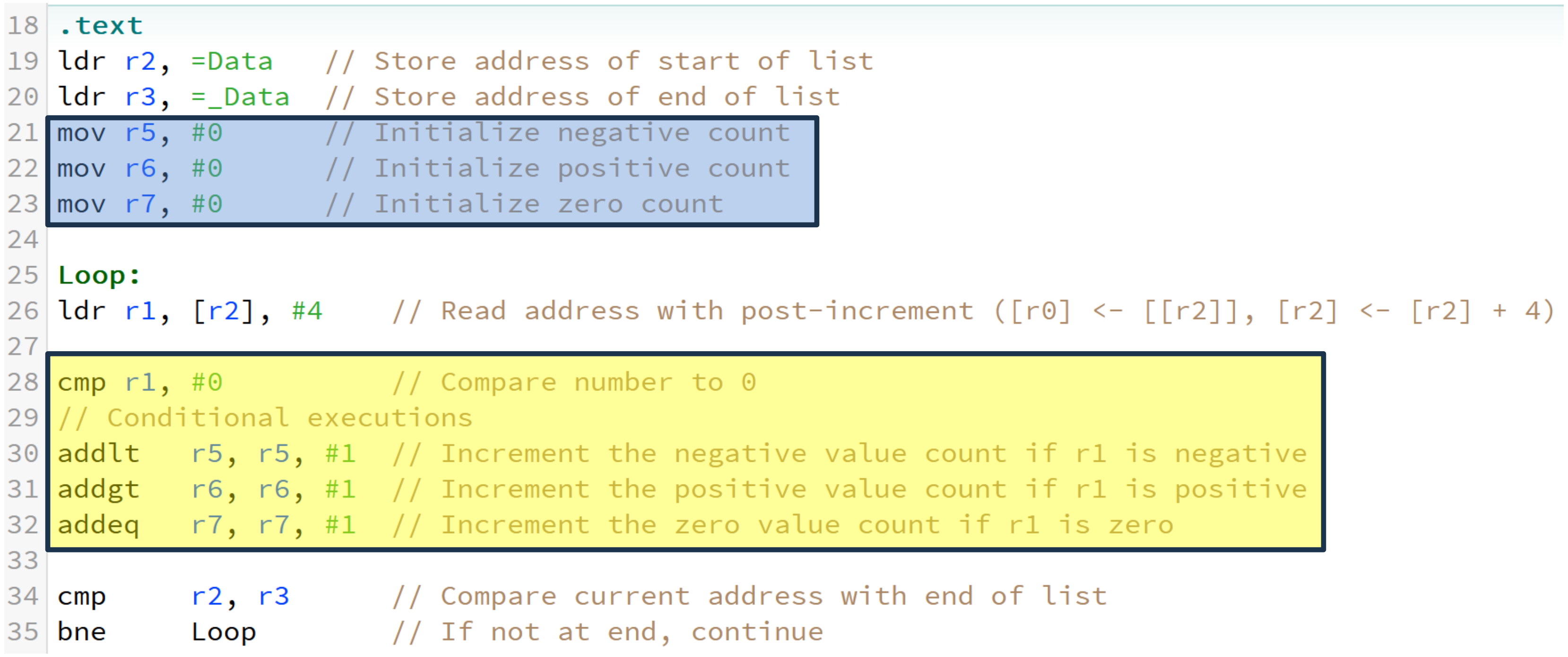
```
ldr r2, =Data       // Store address of start of list
ldr r3, =_Data      // Store address of end of list
…
Loop:
ldr    r0, [r2], #4 // Read address with post-increment ([r0] <- [[r2]], [r2] <- [r2] + 4)
…
cmp r3, r2          // Compare current address with end of list
bne Loop            // If not at end, continue
```

- If [r3]>[r2], end has not been reached, Z = 0, the branch bne is taken.
- If [r3]=[r2], end is reached, Z = 1, bne not taken.

Example on while loop from Lab 4

- Suppose we want to process a list of values, and count how many positive, negative, and zero values are in the list.
- To do it:
  1. We would use three registers to store the number of positive, negative and zero values.
  2. For each value we traverse using the above program, we test whether it is greater than, equal to or less than 0 and add 1 to the appropriate register.

Example on while loop from Lab 4



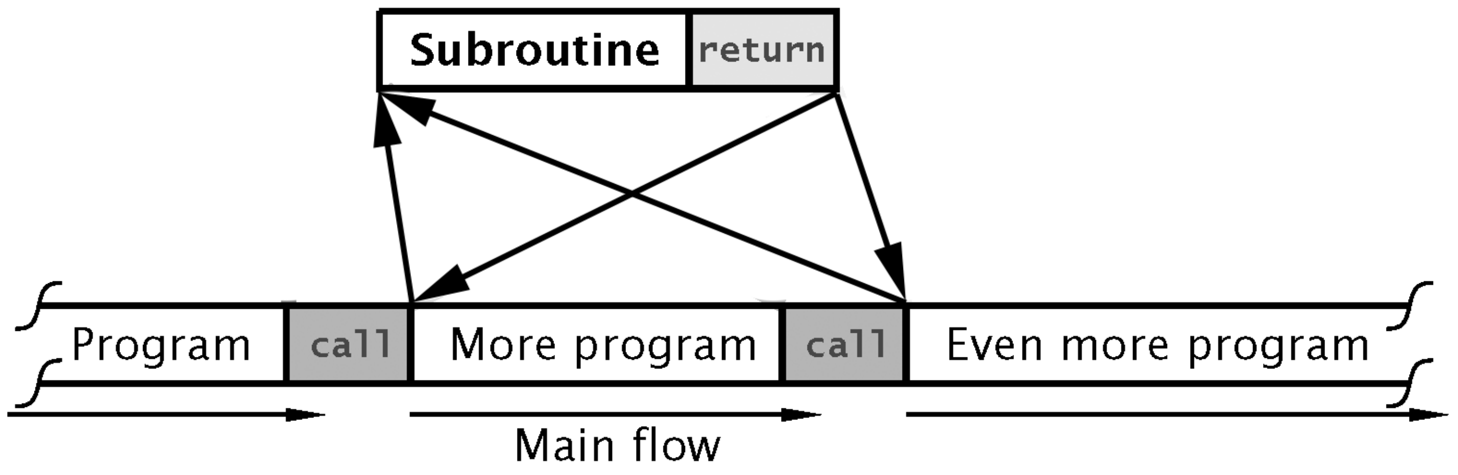
```
.text
ldr r2, =Data    // Store address of start of list
ldr r3, =_Data   // Store address of end of list
mov r5, #0       // Initialize negative count
mov r6, #0       // Initialize positive count
mov r7, #0       // Initialize zero count

Loop:
ldr r1, [r2], #4     // Read address with post-increment ([r0] <- [[r2]], [r2] <- [r2] + 4)

cmp r1, #0           // Compare number to 0
// Conditional executions
addlt    r5, r5, #1  // Increment the negative value count if r1 is negative
addgt    r6, r6, #1  // Increment the positive value count if r1 is positive
addeq    r7, r7, #1  // Increment the zero value count if r1 is zero

cmp      r2, r3      // Compare current address with end of list
bne      Loop        // If not at end, continue
```

3.6 Subroutine call and return

Subroutine

- A subroutine is a sequence of instructions that can be called from different places in a program.
- Two reasons for creating subroutines:
  - The problem is too big: Easier to divide the problem into smaller sub-problems
  - There are several places in a program that need to perform the same operation

Calling subroutine: Is it just branching?



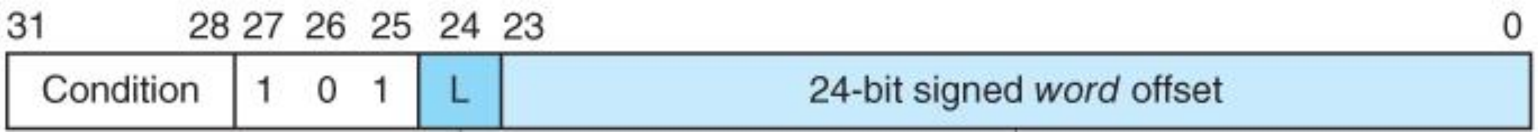
- Calling subroutine involves the “jumping” part, which is the same as branching.
- But we need to get back to the main program after the subroutine returns.
- Thus, we need a location in which the return address can be stored.

Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

ARM support for subroutine

- Use `BL` (branch with link)
  - Performing branching and
  - Store the return address (i.e., the address of the instruction immediately below the bl instruction) into `r14` (also called `lr`) register.

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions



- At the end of the subroutine, use `BX lr` to return [lr] to [pc].

Example 1

Suppose that you want to evaluate
if x > 0 then x = 16x + 1 else x = 32x
several times in a program. Assuming that x is in r0, we can write:

```
Func1:
cmp r0, #0 //test for x > 0
movgt r0, r0, lsl #4 // if x > 0 x = 16x
addgt r0, r0, #1 // if x > 0 then x = 16x + 1
movle r0, r0, lsl #5 // ELSE if x < 0 THEN x = 32x
bx lr // [pc] <- [lr]
```

Example 1

```
.text
.global _start
_start:

ldr r0, =-1

bl Func1

ldr r4, =0x1000
str r0, [r4] // store the result in address 1000

Here:
b Here

Func1:
cmp r0, #0 //test for x > 0
movgt r0, r0, lsl #4 // if x > 0 x = 16x
addgt r0, r0, #1 // if x > 0 then x = 16x + 1
movle r0, r0, lsl #5 // ELSE if x < 0 THEN x = 32x
bx lr // [pc] <- [lr]

.end
```

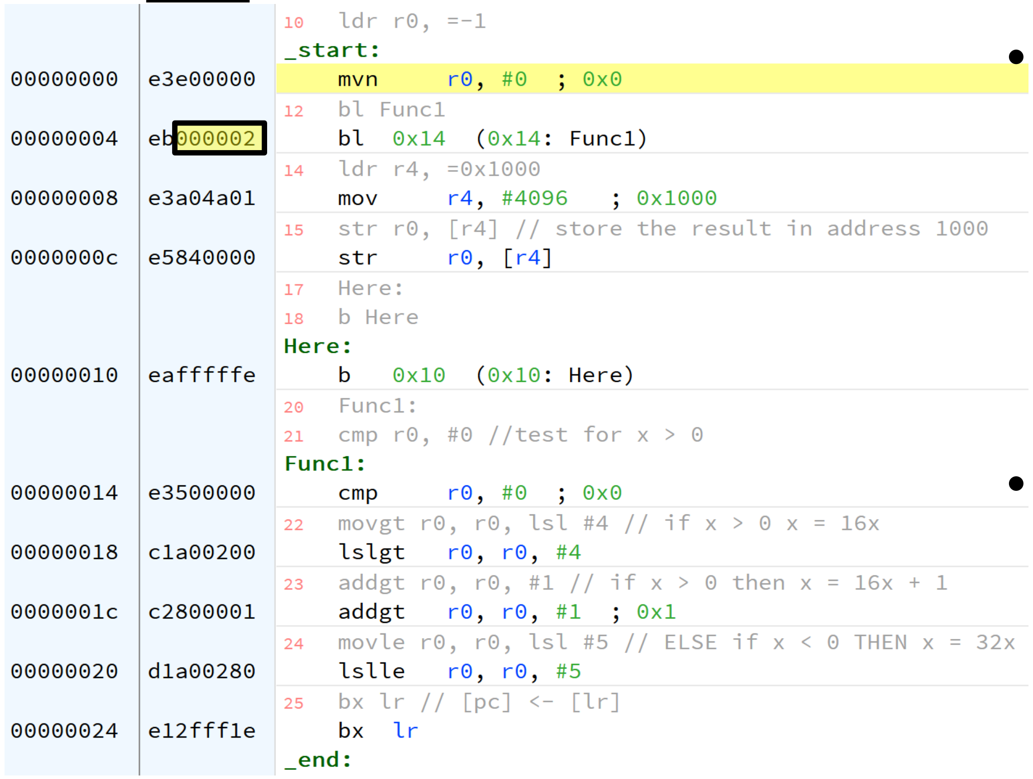
Instructions calling the subroutine Func1:

- Set up r0 as the input parameter
- After Func1 generates an output stored in r0, copy the output to address 1000

Note that the subroutine appears after the `Here: b Here` instruction. This guarantees that the subroutine will not be executed by mistake.

Example 1

Address **Machine Code**



```
                      ldr r0, =-1
                    _start:
00000000  e3e00000      mvn      r0, #0   ; 0x0
                        bl Func1
00000004  eb000002      bl   0x14   (0x14: Func1)
                        ldr r4, =0x1000
00000008  e3a04a01      mov      r4, #4096    ; 0x1000
                        str r0, [r4] // store the result in address 1000
0000000c  e5840000      str      r0, [r4]
                        Here:
                        b Here
                    Here:
00000010  eaffffe       b    0x10   (0x10: Here)
                        Func1:
                        cmp r0, #0 //test for x > 0
                    Func1:
00000014  e3500000      cmp      r0, #0   ; 0x0
                        movgt r0, r0, lsl #4 // if x > 0 x = 16x
00000018  c1a00200      lslgt    r0, r0, #4
                        addgt r0, r0, #1 // if x > 0 then x = 16x + 1
0000001c  c2800001      addgt    r0, r0, #1   ; 0x1
                        movle r0, r0, lsl #5 // ELSE if x < 0 THEN x = 32x
00000020  d1a00280      lslle    r0, r0, #5
                        bx lr // [pc] <- [lr]
00000024  e12fff1e      bx   lr
                    _end:
```

- After execution of the `bl` instruction:
  - [lr] = 0x08
  - [pc] = 0x14
- After execution of the bx instruction inside the subroutine Func1:
  - [pc] = 0x08

thus returning to the line immediately below the `bl` instruction.

- Encoding at the `bl` instruction

$$\frac{Dest \ - Current\ PC}{4}$$

$$= \frac{0x14 \ - 0x0C}{4} = 2$$

- Encoded Address = 000002

Example 2 (Lab 5)

1. Instructions calling the swap subroutine

```
.text           // Code section
.global _start
_start:

ldr     r0, =x    // store location of x
ldr     r1, =y    // store location of y
bl      swap      // call the swap subroutine

_stop:
b       _stop
```

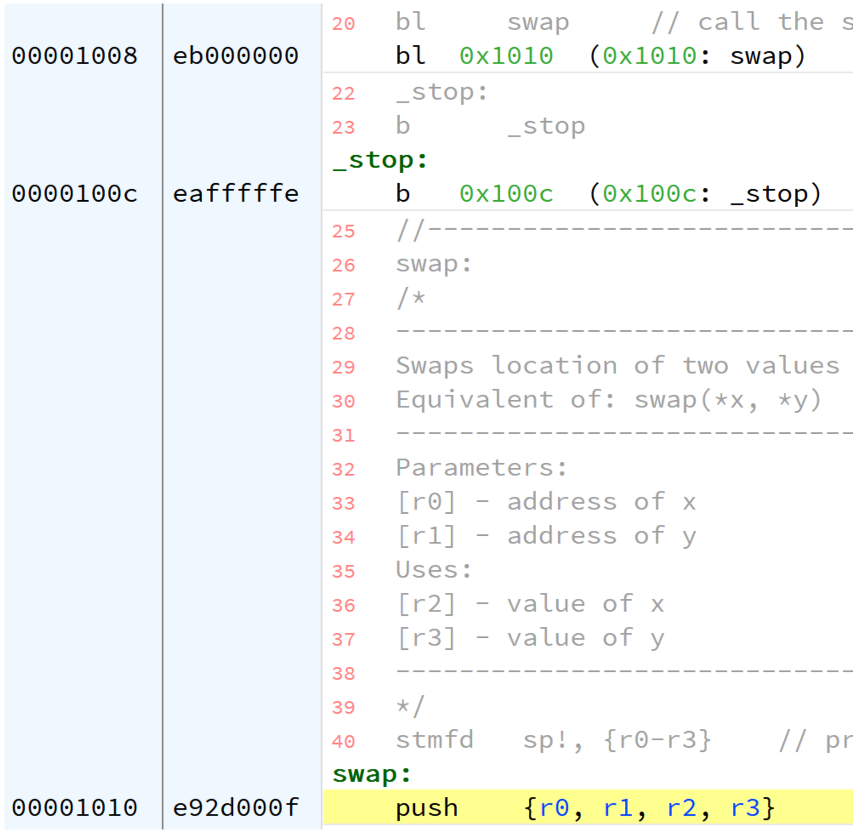
2. swap subroutine

```
swap:
/*
---------------------------------------------------
Swaps location of two values in memory.
Equivalent of: swap(*x, *y)
---------------------------------------------------
Parameters:
  [r0] - address of x
  [r1] - address of y
Uses:
  [r2] - value of x
  [r3] - value of y
---------------------------------------------------
*/
stmfd   sp!, {r0-r3}     // preserve all registers

ldr     r2, [r0]         // get value at x
ldr     r3, [r1]         // get value at y
str     r2, [r1]         // store value of x in y
str     r3, [r0]         // store value of y in x

ldmfd   sp!, {r0-r3}     // pop preserved registers
bx      lr               // return from subroutine
```

3. Data storage

```
.data
.align
x:
.word     9
y:
.word     4

.end
```

Example 2 (Lab 5)



```
                     bl      swap      // call the s
00001008  eb000000       bl   0x1010   (0x1010: swap)
                     _stop:
                     b       _stop
                 _stop:
0000100c  eafffffe       b    0x100c   (0x100c: _stop)
                     //--------------------------
                     swap:
                     /*
                     ----------------------------
                     Swaps location of two values
                     Equivalent of: swap(*x, *y)
                     ----------------------------
                     Parameters:
                     [r0] - address of x
                     [r1] - address of y
                     Uses:
                     [r2] - value of x
                     [r3] - value of y
                     ----------------------------
                     */
                     stmfd   sp!, {r0-r3}     // pr
                 swap:
00001010  e92d000f       push    {r0, r1, r2, r3}
```

- After execution of the `bl` instruction:
  - [lr] = 0x100C
  - [pc] = 0x1010
- After execution of the bx instruction inside the subroutine swap:
  - [pc] = 0x100C

Preserving Registers

- The values in registers should be the same before and after a subroutine call, except registers being updated (e.g., r0 in Example 1) or those used to return a value.
- Registers are preserved and recovered by pushing and popping their values to the stack.
- ARM provides instructions for pushing and popping multiple register values at once by stmfd and ldmfd.

Block data transfer

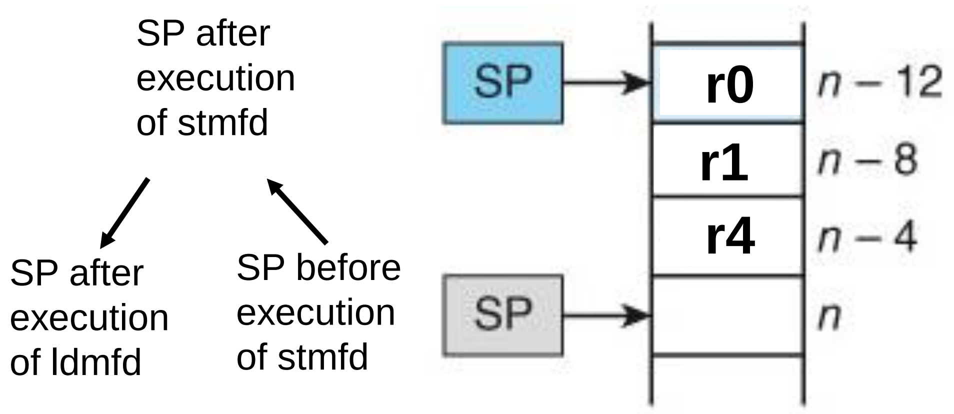
stmfd (Store Memory Full Descending stack)

- copies a block of data from registers to memory

ldmfd (Load Memory Full Descending stack)

- copies a block of data from memory to registers

```
stmfd sp!, {r0,r1,r4}
ldmfd sp!, {r0,r1,r4}
```


SP after execution of stmfd
SP after execution of ldmfd
SP before execution of stmfd
SP
r0
r1
r4
SP
n – 12
n – 8
n – 4
n

The FD option:

- Full
  - SP points to the top item of stack
- Descending
  - stack grows towards lower addresses

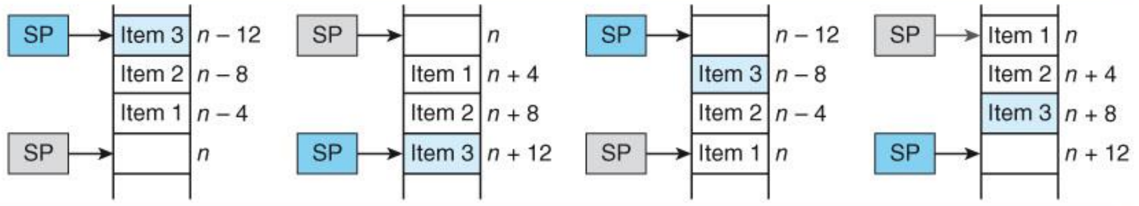
Note that sp moves only if appended with an !

Block data transfer

The full descending (FD) option is most often used in this course, but you should know about other options:

- Full vs. Empty
  - Full: SP points to the top item of the stack
  - Empty: SP points to the next empty element above the top of stack
- Descending vs. Ascending
  - The stack grows towards either lower (descending) or higher (ascending) addresses.

TABLE 3.5 Stack Types and the ARM Block Move Instruction Suffixes



Stack type	1	2	3	4
Stack growth	Descending	Ascending	Descending	Ascending
Class	Full	Full	Empty	Empty
Stack suffix	FD	FA	ED	EA
Load suffix	IA (increment after)	DA (decrement after)	IB (increment before)	DB (decrement before)
Store suffix	DB (decrement before)	IB (increment before)	DA (decrement after)	IA (increment after)

Block data transfer

stmfd = stmdb

db = decrement before

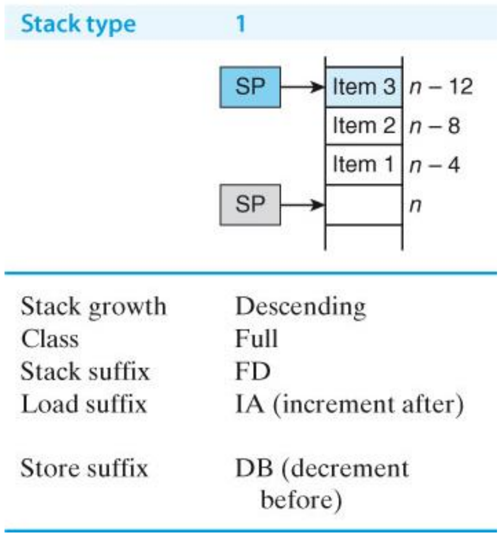
- SP decrements before storing the next element

ldmfd = ldmia

ia = increment after

- SP increments after popping each element

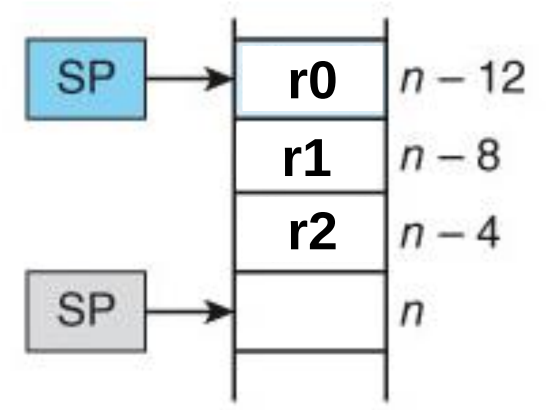
Try to understand equivalence in other types of stacks.



Stack growth	Descending
Class	Full
Stack suffix	FD
Load suffix	IA (increment after)
Store suffix	DB (decrement before)

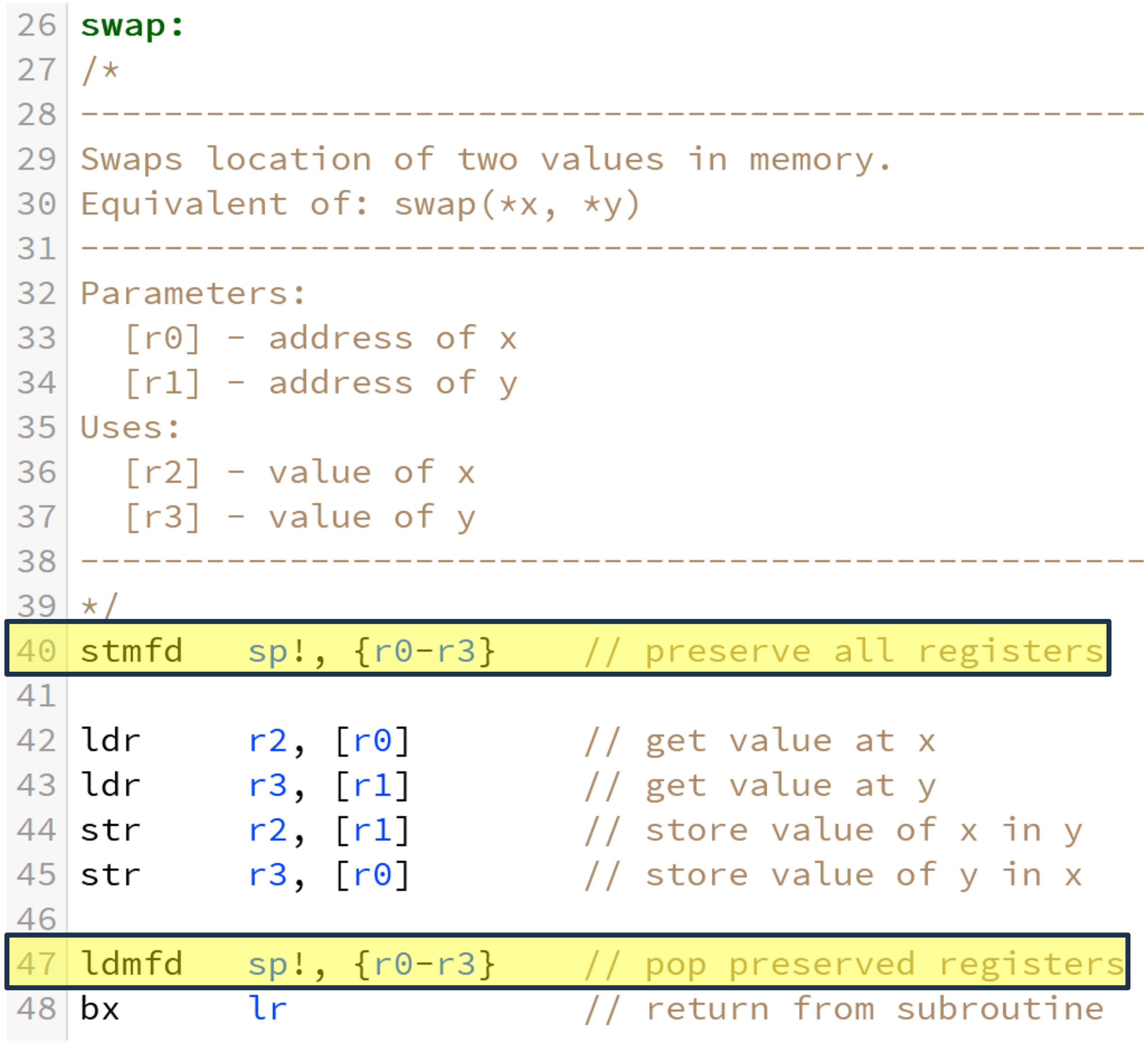
Block data transfer

```
stmfd sp!, {r0-r2}
```



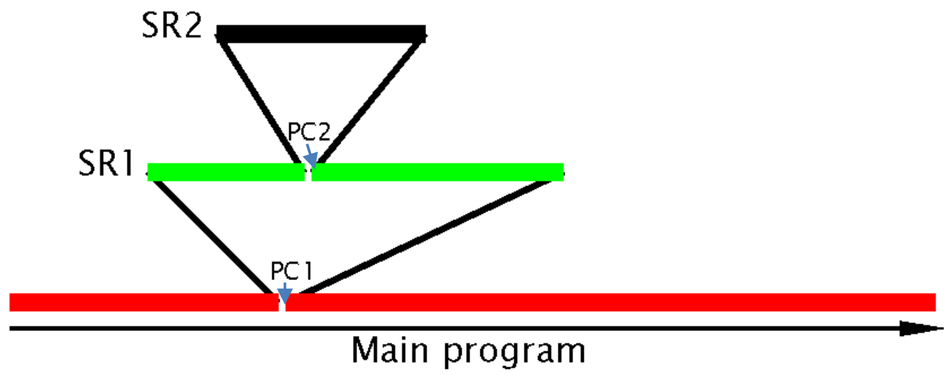
- If multiple registers are adjacent, they can be listed with a dash (-):
  - e.g., `stmfd sp!, {r0-r2}` pushes r0, r1 and r2.
- Note that the registers are pushed in right-to-left order (i.e., pushing r2, then r1 then r0)
- The same would happen even with `stmfd sp!, {r2, r0, r1}`
- If you require the stack to be in a different order, you must issue separate `stmfd` instructions.

Preserving registers in the swap subroutine



```
swap:
/*
-------------------------------------------------------
Swaps location of two values in memory.
Equivalent of: swap(*x, *y)
-------------------------------------------------------
Parameters:
  [r0] - address of x
  [r1] - address of y
Uses:
  [r2] - value of x
  [r3] - value of y
-------------------------------------------------------
*/
stmfd    sp!, {r0-r3}     // preserve all registers

ldr      r2, [r0]         // get value at x
ldr      r3, [r1]         // get value at y
str      r2, [r1]         // store value of x in y
str      r3, [r0]         // store value of y in x

ldmfd    sp!, {r0-r3}     // pop preserved registers
bx       lr               // return from subroutine
```

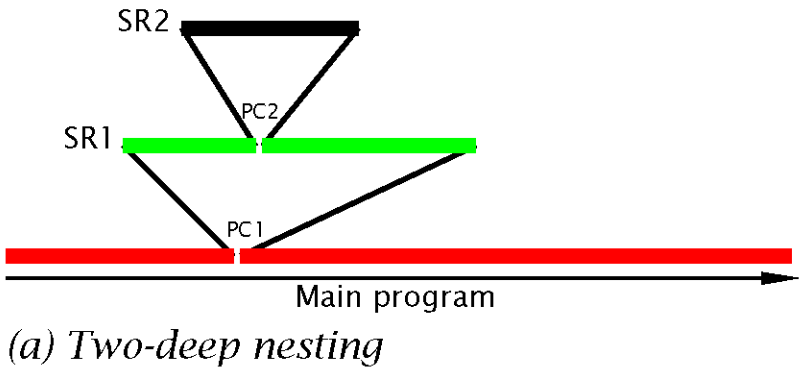
Nested subroutine



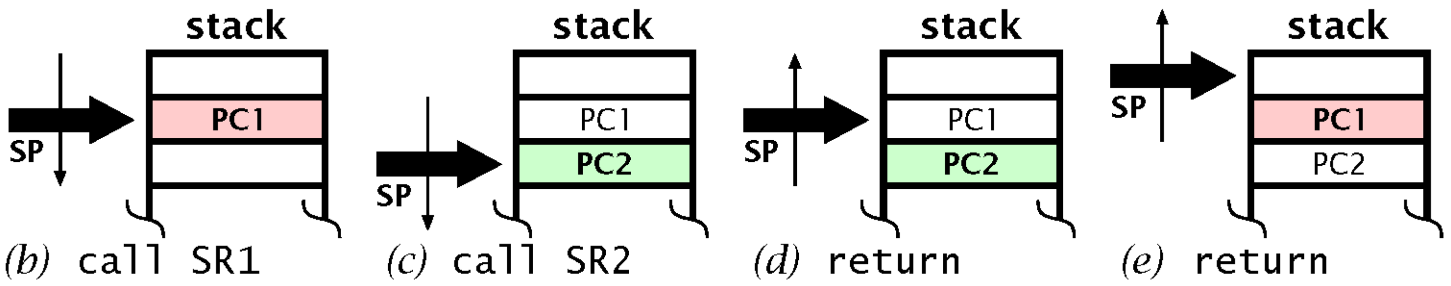
- Nested subroutine: calling a subroutine within a subroutine.
- BL call from Main saves the return address (PC1) to the link register (lr).
- BL call from SR1 saves the return address (PC2) to the lr, thereby overwriting PC1 → SR1 cannot return to Main.

Nested subroutine

Some microcontrollers have a dedicated stack to handle nested subroutines, but not ARM.



SP: Stack Pointer
TOS: Top of Stack
(The address pointed to by SP)



TOS = PC1	TOS = PC2	PC = PC2	PC = PC1
SP = 1	SP = 2	TOS = PC1	SP = 0
		SP = 1	

How does ARM handle nested subroutine calls?

- In ARM, no such stack. Need to create our own stack.
- A leaf routine doesn't call another routine (at the end of tree).
  - If you call a leaf routine with BL, the return address is saved in link register r14. A return to the calling point is made with `bx lr`
- If the routine is not a leaf routine, you cannot call another routine without first saving the link register.

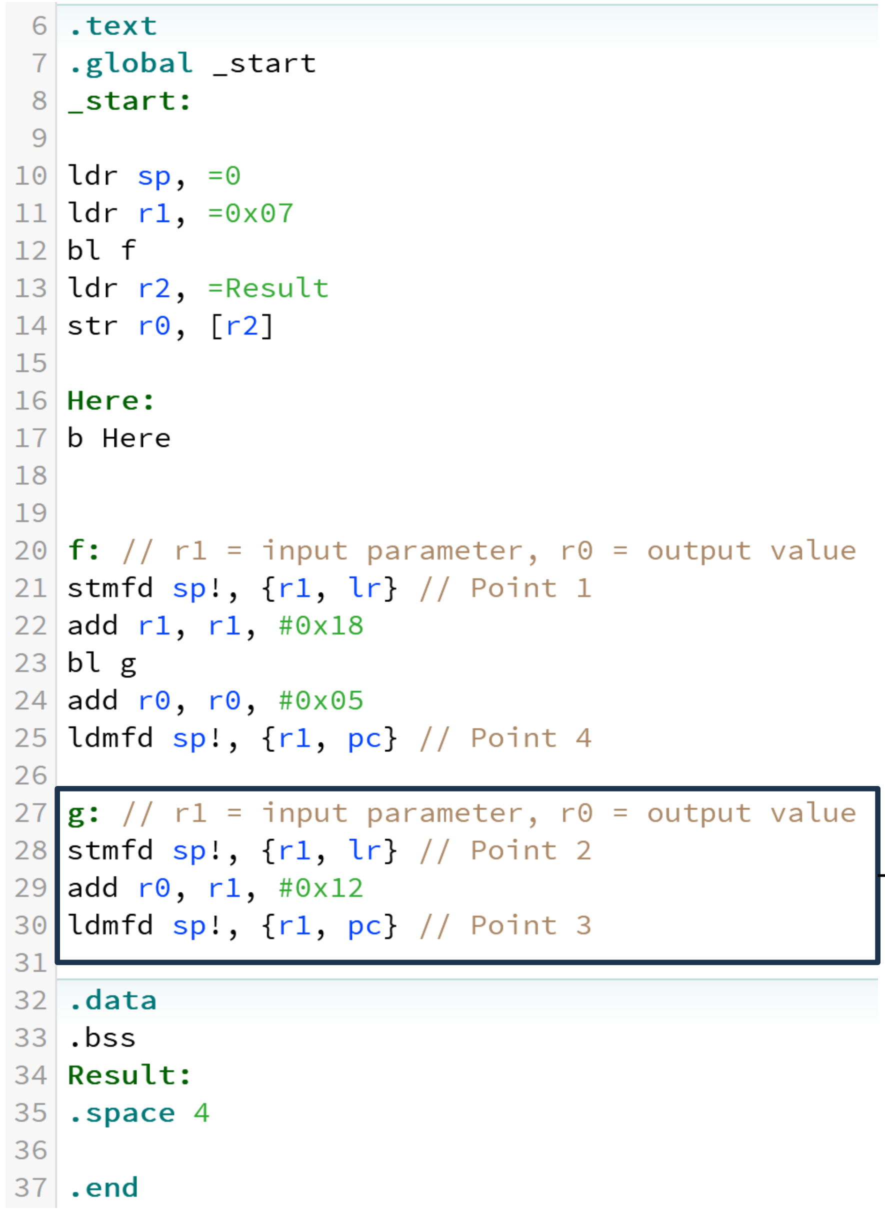
```
        BL      XYZ                 ;call a simple leaf routine
        .
        BL      XYZ1                ;call a routine that calls a nested routine
        .
XYZ     . . .                       ;code (this is the leaf routine)
        .
        bx lr                       ;copy link register into PC and return
XYZ1    STMFD   sp!,{r0-r4,lr}      ;save working registers and link register
        .
        BL      XYZ                 ;call XZY - overwrites the old link register
        .
        LDMFD   sp!,{r0-r4,pc}      ;restore registers and force a return
```

- The link register (lr) is preserved just as other registers
- At the end of the subroutine the preserved lr is popped to pc.

Example of a nested subroutine

- The following program calculates the value of f(0x07) where
  - f(x) = g(x+0x18) + 0x05
  - g(x) = x+0x12
- Determine the status of the stack at Points 1-4.

Example of a nested subroutine

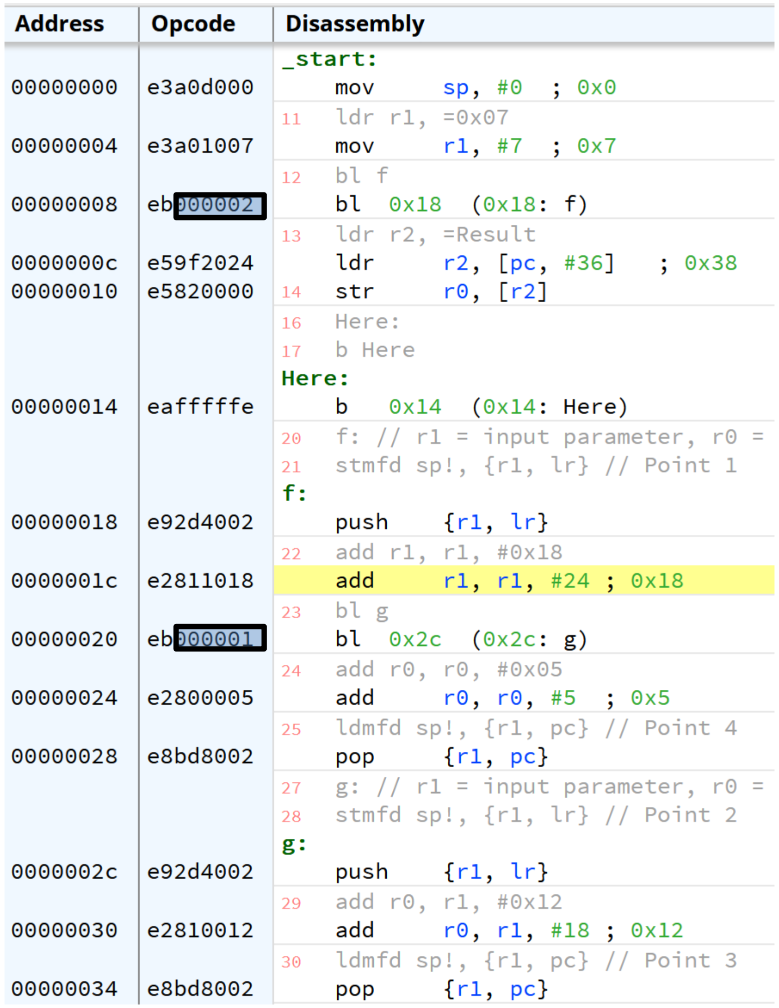


```
.text
.global _start
_start:

ldr sp, =0
ldr r1, =0x07
bl f
ldr r2, =Result
str r0, [r2]

Here:
b Here


f: // r1 = input parameter, r0 = output value
stmfd sp!, {r1, lr} // Point 1
add r1, r1, #0x18
bl g
add r0, r0, #0x05
ldmfd sp!, {r1, pc} // Point 4

g: // r1 = input parameter, r0 = output value
stmfd sp!, {r1, lr} // Point 2
add r0, r1, #0x12
ldmfd sp!, {r1, pc} // Point 3

.data
.bss
Result:
.space 4

.end
```

Could do this because g is a leaf subroutine, but saving to stack is always safe:

```
g:
stmfd sp!, {r1}
add r0, r1, #0x12
ldmfd sp!, {r1}
bx lr
```

Example of a nested subroutine



```
Address   Opcode     Disassembly
                     _start:
00000000  e3a0d000       mov      sp, #0   ; 0x0
                         ldr r1, =0x07
00000004  e3a01007       mov      r1, #7   ; 0x7
                         bl f
00000008  eb000002       bl   0x18   (0x18: f)
                         ldr r2, =Result
0000000c  e59f2024       ldr      r2, [pc, #36]    ; 0x38
00000010  e5820000       str      r0, [r2]
                         Here:
                         b Here
                     Here:
00000014  eafffffe       b    0x14   (0x14: Here)
                         f: // r1 = input parameter, r0 =
                         stmfd sp!, {r1, lr} // Point 1
                     f:
00000018  e92d4002       push     {r1, lr}
                         add r1, r1, #0x18
0000001c  e2811018       add      r1, r1, #24 ; 0x18
                         bl g
00000020  eb000001       bl   0x2c   (0x2c: g)
                         add r0, r0, #0x05
00000024  e2800005       add      r0, r0, #5   ; 0x5
                         ldmfd sp!, {r1, pc} // Point 4
00000028  e8bd8002       pop      {r1, pc}
                         g: // r1 = input parameter, r0 =
                         stmfd sp!, {r1, lr} // Point 2
                     g:
0000002c  e92d4002       push     {r1, lr}
                         add r0, r1, #0x12
00000030  e2810012       add      r0, r1, #18 ; 0x12
                         ldmfd sp!, {r1, pc} // Point 3
00000034  e8bd8002       pop      {r1, pc}
```

Point 1

	Address	Content
SP →	FFFFFFF8	00000007
	FFFFFFFC	0000000C

Point 2

	Address	Content
SP →	FFFFFFF0	0000001F
	FFFFFFF4	00000024
	FFFFFFF8	00000007
	FFFFFFFC	0000000C

Point 3

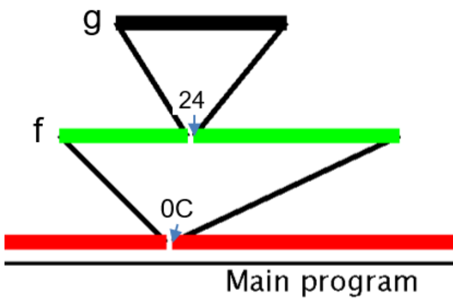
	Address	Content
SP →	FFFFFFF8	00000007
	FFFFFFFC	0000000C

[PC] = 00000024
[r1] = 0000001F

Point 4

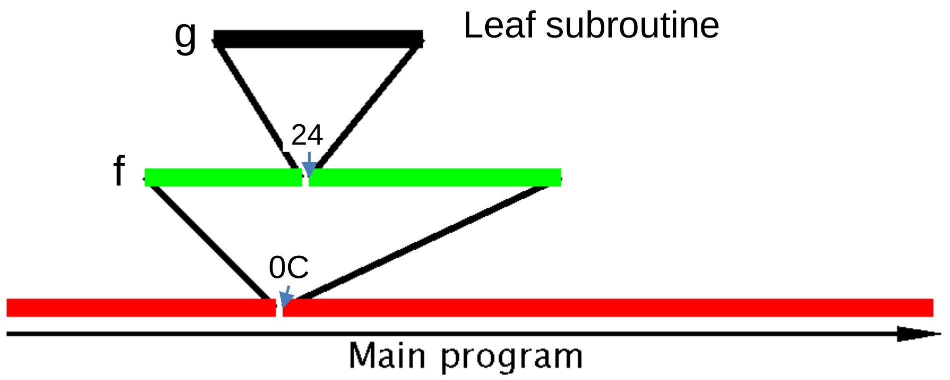
[PC] = 0000000C
[r1] = 00000007

Structure of the nested call



Note that [r1] is preserved
(i.e., the bl call does not change its value)

Further notes



- `bx lr` can be used only in leaf subroutine (e.g., `g`)
  - Saving `lr` in stack is safe for all subroutine though.
- What happens if `f` is returned by `bx lr`?
  - Upon completion of f, program should return to address 0C
  - After `f` is called, [lr] = 0C
  - After `g` is called, [lr] = 24, overwriting 0C
  - Thus, `f` cannot be returned by `bx lr`.