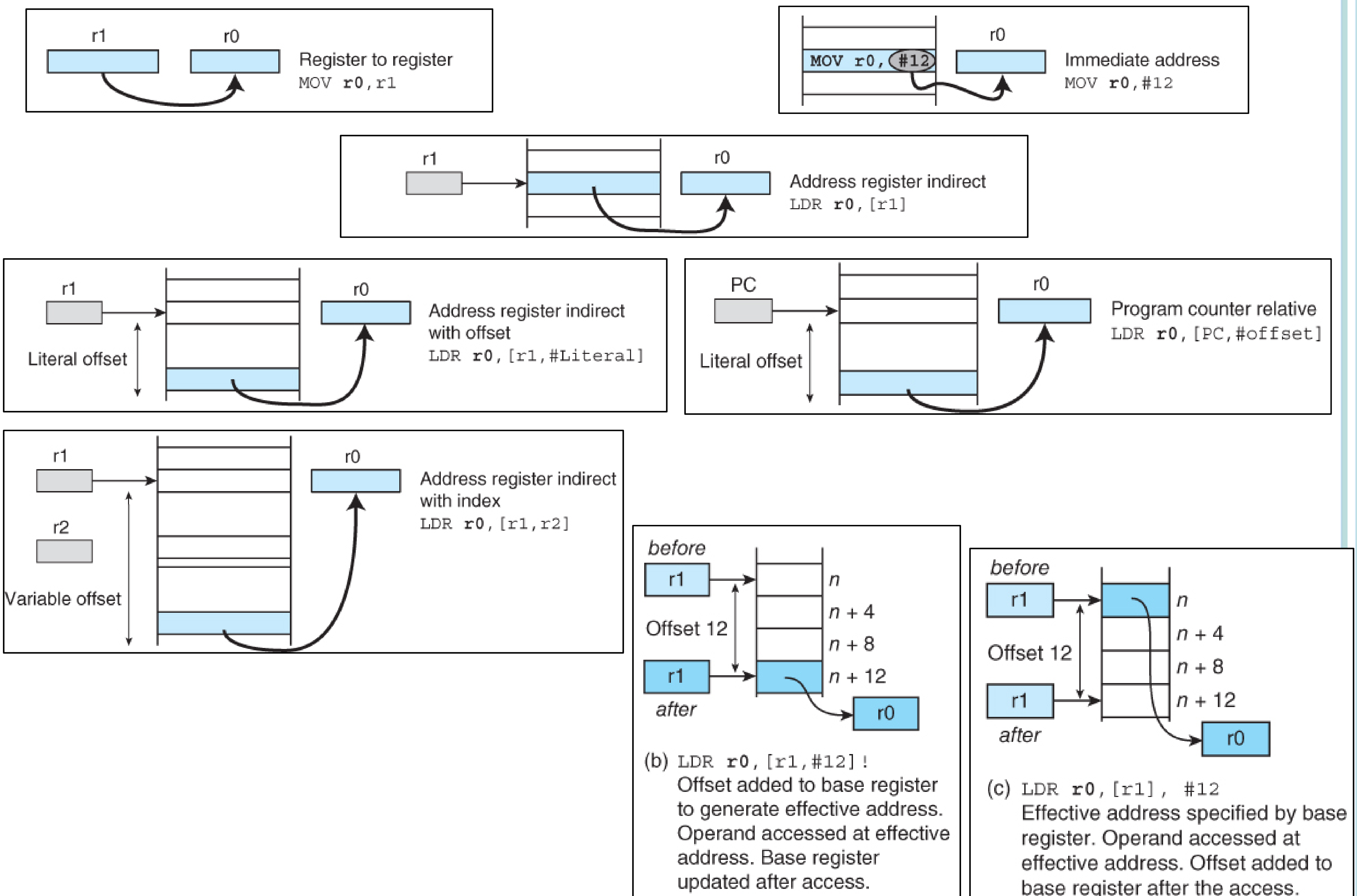
Summary of ARM Addressing Modes



Example 1: Calculating the Absolute Value

- To calculate `x ← |x|`, where `x` is a signed integer, we can implement
  `if x < 0 then x = -x`
- In ARM

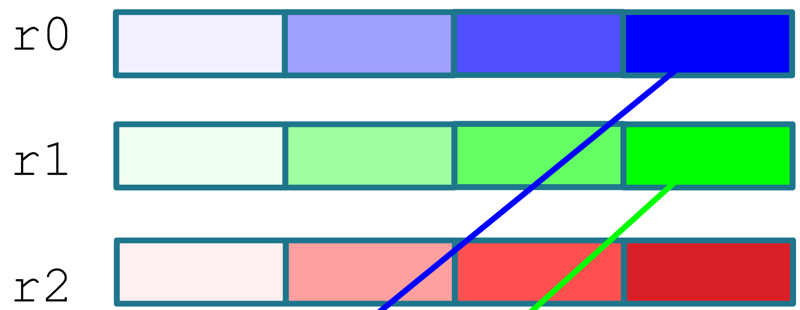
```
TEQ     r0, #0          ;compare r0 with zero
RSBMI   r0, r0, #0      ;if negative (MInus) r0 ← 0 – r0
```

- What is the difference between `TEQ` and `CMP`?

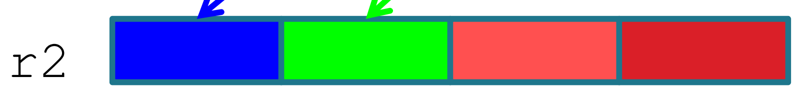
To know the difference, read slide #89

Example 2: Byte Manipulation and Concatenation

- Suppose we have `r0`, `r1`, and `r2` as follow:



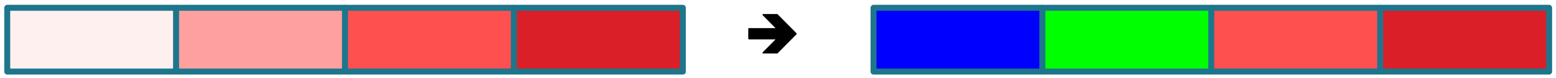
and we want to rearrange `r2` as follow:



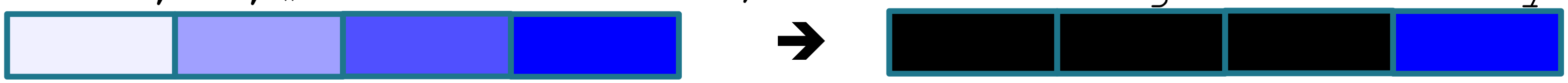
> BIC (bit clear)
> ANDing the 1st operand with the complement of the 2nd operand.
> To know more about BIC, read slide #77

```
BIC r0,r0,#0xFFFFFF00    ;clear r0 all high order 3 bytes
BIC r1,r1,#0xFFFFFF00    ;clear r1 all high order 3 bytes
ADD r2,r1,r2,LSL#16      ;LSL r2 by 2 bytes & add r1 to it
ADD r2,r2,r0,LSL#8       ;LSL r0 by 1 byte & add it to r2
MOV r2,r2,ROR#16         ;Swap the two r2 16 bits together
```

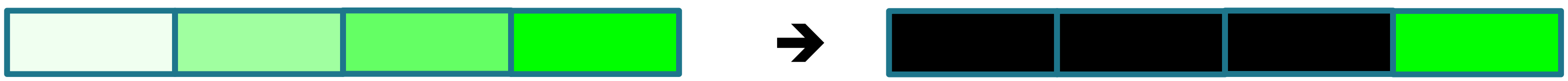
Example 2: Byte Manipulation and Concatenation



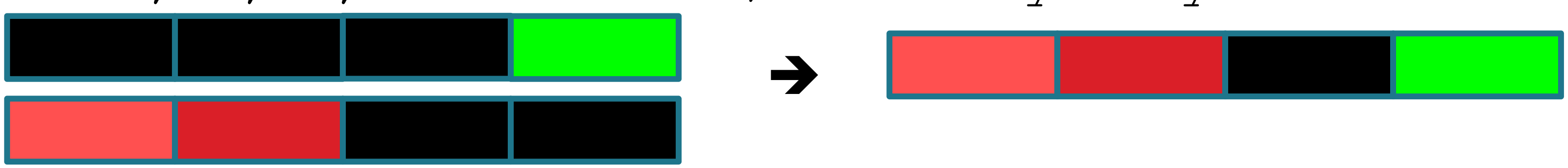
```
BIC r0,r0,#0xFFFFFF00      ;clear all high order 3 bytes
```



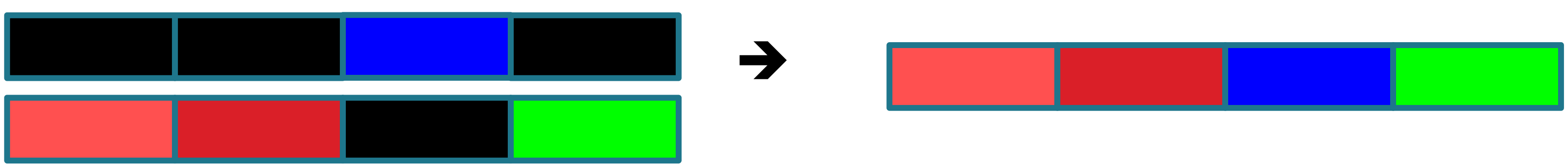
```
BIC r1,r1,#0xFFFFFF00      ;clear all high order 3 bytes
```



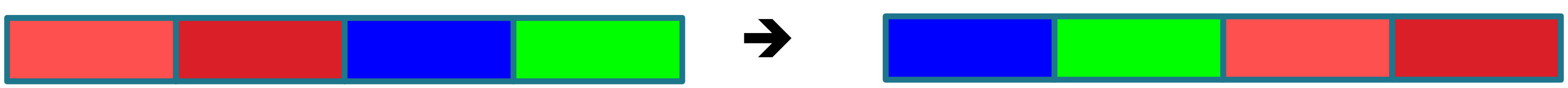
```
ADD r2,r1,r2,LSL#16        ;LSL r2 by 2 bytes & add r1 to it
```



```
ADD r2,r2,r0,LSL#8         ;LSL r0 by 1 byte & add it to r2
```



```
MOV r2,r2,ROR#16           ;Swap the two r2 16 bits together
```

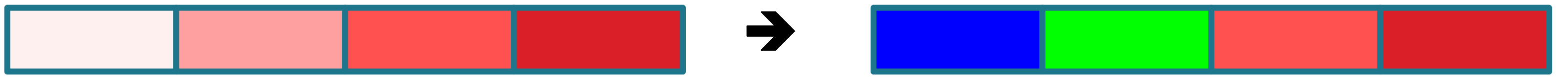


Example 2: Byte Manipulation and Concatenation

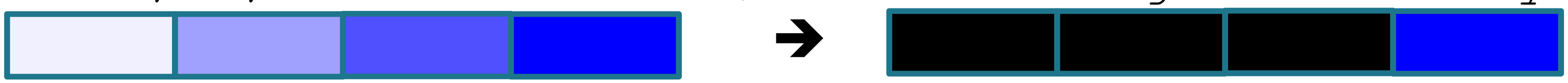
- Suppose we have `r0`, `r1`, and `r2` as follow:


```
BIC r0,r0,#0xFFFFFF00     ;clear r0 all high order 3 bytes
BIC r1,r1,#0xFFFFFF00     ;clear r1 all high order 3 bytes
BIC r2,r2,#0xFFFF0000     ;clear r2 all high order 2 bytes
ADD r2,r2,r1,LSL#16       ;LSL r1 by 2 bytes & add it to r2
ADD r2,r2,r0,LSL#24       ;LSL r0 by 3 bytes & add it to r2
```

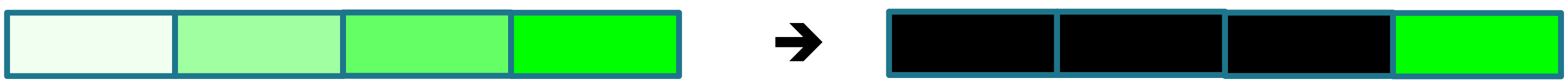
Example 2: Byte Manipulation and Concatenation



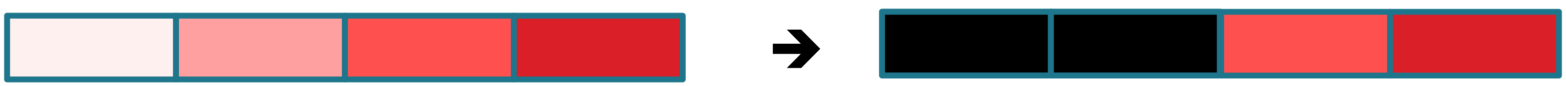
- BIC **r0**,r0,#0xFFFFFF00 ;clear all high order 3 bytes



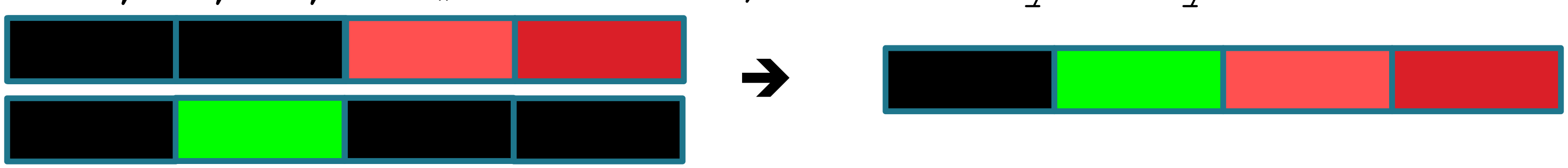
BIC **r1**,r1,#0xFFFFFF00 ;clear all high order 3 bytes



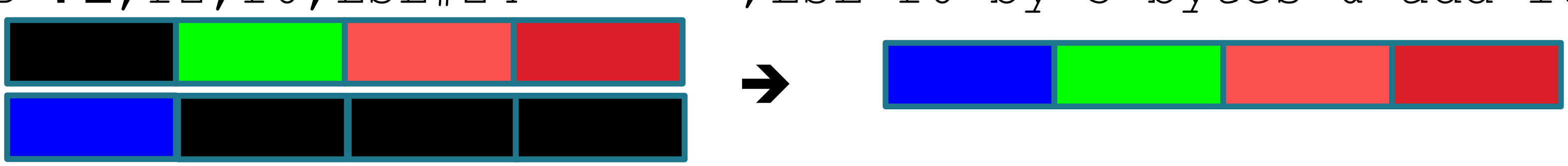
BIC **r2**,r2,#0xFFFF0000 ;clear r2 all high order 2 bytes



ADD **r2**,r2,r1,LSL#16 ;LSL r1 by 2 bytes & add it to r2



ADD **r2**,r2,r0,LSL#24 ;LSL r0 by 3 bytes & add it to r2



Example 3: Byte Reversal (Big-endian ➔ Little-endian)

- Suppose that **0xAB CD EF GH** is stored in `r0`
- We want to reverse the content of `r0`, `i.e.,` store **0xGH EF CD AB** in `r0`
- We will use `r1` as a working register

A	B	$C = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

- Let us review the XOR truth table
  - $x \oplus x = 0$
  - $x \oplus 0 = x$
  - $x \oplus y \oplus y = x$

```
EOR r1,r0,r0,ROR#16    ;A⊕E,B⊕F,C⊕G,D⊕H,E⊕A,F⊕B,G⊕C,H⊕D
BIC r1,r1,#0x00FF0000  ;A⊕E,B⊕F,   0,   0,E⊕A,F⊕B,G⊕C,H⊕D
MOV r0,r0,ROR#8        ; G  ,  H  , A , B , C  , D , E  , F
EOR r0,r0,r1,LSR#8     ;r1 after LSR#8 is
                       ; 0  , 0  ,A⊕E,B⊕F, 0  , 0 ,E⊕A,F⊕B
                       ;The final result will be
                       ; G  , H  ,A⊕A⊕E,B⊕B⊕F,C,D,E⊕E⊕A,F⊕F⊕B
                       ; G  , H  ,E    ,F    ,C,D,A    ,B
```

Example 4: Variable Swapping

- Assume that we have two variables stored in **`r0`** and **`r1`**
- We wants to swap these two variables

```
[r2] ← [r0]
[r0] ← [r1]
[r1] ← [r2]
```

- Now, we want to do the same thing without using **`r2`**
- Let us review the XOR truth table
  - x ⊕ x = 0
  - x ⊕ 0 = x
  - **x ⊕ y ⊕ y = x**

A	B	$C = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

```
EOR r0,r0,r1      ; [r0] ← [r0] ⊕ [r1]
EOR r1,r0,r1      ; [r1] ← [r0] ⊕ [r1]
                  ; [r1] ←([r0] ⊕ [r1]) ⊕ [r1]
                  ; [r1] ← [r0]
EOR r0,r0,r1      ; [r0] ← [r0] ⊕ [r1]
                  ; [r0] ←([r0] ⊕ [r1]) ⊕ [r0]
                  ; [r0] ← [r1]
```

X ← X ⊕ Y
Y ← X ⊕ Y
X ← X ⊕ Y

Example 5: Multiplication by $2^n - 1$, 2^n, or $2^n + 1$

- Multiplying by `2ⁿ` can be implemented using `MOV` instruction and `LSL#n`
- Example

```
MOV r2, r1, LSL#4       ; [r2]←[r1] × 2^4
```

- Multiplying by `2ⁿ + 1` can be implemented using `ADD` instruction and `LSL#n`
- Example

```
ADD r2, r1, r1, LSL#4 ; [r2]←[r1] + [r1] × 2^4
```

- Multiplying by `2ⁿ - 1` can be implemented using `RSB` instruction and `LSL#n`
- Example

```
RSB r2, r1, r1, LSL#4 ; [r2]←[r1] × 2^4  - [r1]
```

Example 5: Multiplication by $2^n - 1$, 2^n, or $2^n + 1$

- Let us translate the following C code

```
if(x > y)
  p = 17 * q;
else
{ if(x = y)
    p = 16 * q;
  else /* i.e., x < y */
     p= 15 * q;
}
```

- Assume that `x` and `y` are stored in `r2` and `r3`, and also that `p` and `q` are `r4` and `r1`

```
CMP    r2, r3                 ;Compare x and y
ADDGT r4, r1, r1, LSL#4 ; IF >, then p ← q      + q << 4
MOVEQ r4, r1, LSL#4     ; IF =, then p ← q << 4
RSBLT r4, r1, r1, LSL#4 ; IF <, then p ← q << 4 - q
```

r4 not **r1**
Not correct in
the book page
200

Example 6: Converting Capital Letter ➔ Small Letter

- Let us convert any capital letter to small letter
- Capital letters begins by `'A'` and end by `'Z'`
- Assume that the character to be converted in `r0` and `r1` is a working register

```
CMP     r0, #'A'          ;Are we in the range of the capital?
RSBGES r1, r0, #'Z'      ;If >= 'A',
                          ;then check with 'Z'
                          ;     and update the flags
ORRGE  r0, r0, #0x0020;If between 'A' and 'Z' inclusive,
                          ;then set bit 5 to force lower case
```

Example 7: If Statement in One Instruction!!

- Let us translate the following C code

```
if(x < 0)
  x = 0;
```

- Assume that `x` is stored in `r0`

```
BIC r0, r0, r0, ASR#31 ; only one instruction!!
```

- `ASR#31` will fill all bits of `r0` with the sign bit
  - If positive, the result will be `0x00000000`
  - If negative, the result will be `0xFFFFFFFF`

  Hence, if negative, all bits will be cleared, i.e., `x` ← `0`

  Otherwise, `x` will stay as it is without change

Example 8: Simple Bit-level Logical Operations

- Assume `#2_0000 0000 0000 0000 0000 0000 0000` **pqrs** is stored in `r0`
- We wish to implement the following statement

```
if ((p == 1) && (r == 1))
    s = 1;
```

- Assume that `r1` is a working register

```
ANDS   r1,r0,#0x8 ;clear all bits in r1 and copy p from r0
ANDNES r1,r0,#0x2 ;if p == 1,
                  ;  clear all bits in r1 and copy r from r0
ORRNE  r0,r0,#1   ;if r == 1, the s ← 1
```

Example 9: Hexadecimal Character Conversion

- We would like to convert **4 binary bits** to **hexadecimal digits**
- Assume that these 4 bits are stored at the LSB of `r0` and the rest of the bits are zeros
- Note that the ASCII code of
  - `'0'` is `48`, i.e., `0x30` (difference from 0000_2 is = `0x30`)
  - `'1'` is `49`, i.e., `0x31` (difference from 0001_2 is = `0x30`)
  - ...
  - `'9'` is `57`, i.e., `0x39` (difference from 1001_2 is = `0x30`)
- Note also that the ASCII code of
  - `'A'` is `65`, i.e., `0x41` (difference from 1010_2 is = `0x37`)
  - `'B'` is `66`, i.e., `0x42` (difference from 1011_2 is = `0x37`)
  - ...
  - `'F'` is `70`, i.e., `0x46` (difference from 1111_2 is = `0x37`)
- The conversion algorithm is:

```
character = the4BitBinaryValue + 0x30
  if(character > 0x39)
    character += 7
```

ADDGT not **ADDGE**
Not correct in the book page 202

```
ADD   r0,r0,#0x30;add 0x30 to convert 0 through 9 to ASCII
CMP   r0,#0x39    ;check for A to F hex values
ADDGT r0,r0,#7    ;If A to F, then add 7 to get the ASCII
```

0000 →	'0'
0001 →	'1'
0010 →	'2'
0011 →	'3'
0100 →	'4'
0101 →	'5'
0110 →	'6'
0111 →	'7'
1000 →	'8'
1001 →	'9'
1010 →	'A'
1011 →	'B'
1100 →	'C'
1101 →	'D'
1110 →	'E'
1111 →	'F'

Example 9: Hexadecimal Character Conversion

- We would like to convert **4 binary bits** to **hexadecimal digits**
- Assume that these 4 bits are stored at the LSB of `r0` and the rest of the bits are zeros
- Note that the ASCII code of
  - `'0'` is `48`, i.e., `0x30` (difference from $\texttt{0000}_2$ is = `0x30`)
  - `'1'` is `49`, i.e., `0x31` (difference from $\texttt{0001}_2$ is = `0x30`)
  - ...
  - `'9'` is `57`, i.e., `0x39` (difference from $\texttt{1001}_2$ is = `0x30`)
- Note also that the ASCII code of
  - `'A'` is `65`, i.e., `0x41` (difference from $\texttt{1010}_2$ is = `0x37`)
  - `'B'` is `66`, i.e., `0x42` (difference from $\texttt{1011}_2$ is = `0x37`)
  - ...
  - `'F'` is `70`, i.e., `0x46` (difference from $\texttt{1111}_2$ is = `0x37`)

0000	→	**'0'**
0001	→	**'1'**
0010	→	**'2'**
0011	→	**'3'**
0100	→	**'4'**
0101	→	**'5'**
0110	→	**'6'**
0111	→	**'7'**
1000	→	**'8'**
1001	→	**'9'**
1010	→	**'A'**
1011	→	**'B'**
1100	→	**'C'**
1101	→	**'D'**
1110	→	**'E'**
1111	→	**'F'**

- Another algorithm:

```
character = the4BitBinaryValue
    +(the4BitBinaryValue <= 0x9)? 0x30 : 0x37;
```

```
CMP    r0,#0x9     ;is it 0-9 or A-F hex values?
ADDLE  r0,r0,#0x30;if it is 0-9, add 0x30 to convert to ASCII
ADDGT  r0,r0,#0x37;if it is A-F, add 0x37 to convert to ASCII
```

Example 10: Multiple Selection

- Let us translate the following C code

```
switch (i)
{ case  0: do action; break;
  case  1: do action; break;
  ...
  case  N: do action; break;
  default: do something;
}
```

The case values are in order, without missing any number in the middle.

- Assume that `r0` contains the selector `i`

```
    ADR r1, TBL            ;r1 ← the address of the jump table
    CMP r0, N              ;is the switch variable in range?
    ADDLE pc,r1,r0,LSL#2   ;If OK, jump to the appropriate case
    ;The default action goes here


    ...
TBL B case0
    B case1
    ...
    B caseN
```

Example 10: Multiple Selection

- Let us translate the following C code

```
switch (i)
{ case  0: do action; break;
  case  1: do action; break;
  ...
  case  N: do action; break;
  default: do something;
}
```

The case values are in ANY order.

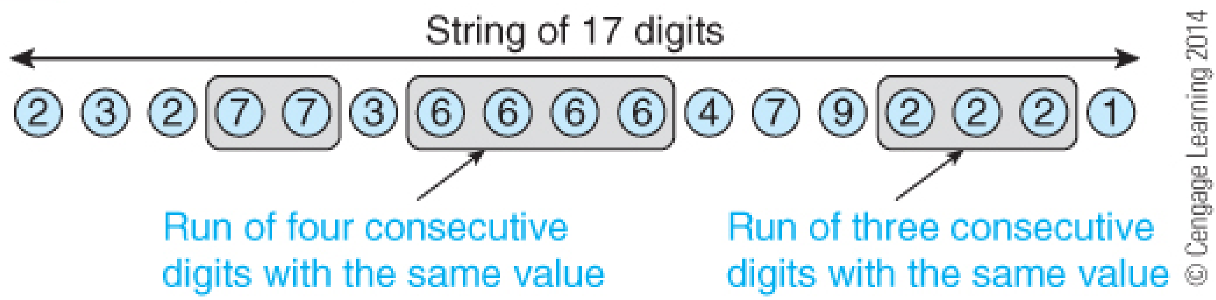
- Assume that `r0` contains the selector `i`

```
          TEQ r0, 0 ;is the switch variable == 0?
          BEQ case0 ;If i == 0, jump to the case0 code
          TEQ r0, 1 ;is the switch variable == 1?
          BEQ case1 ;If i == 1, jump to the case1 code
          ...
          TEQ r0, N ;is the switch variable == N?
          BEQ caseN ;If i == N, jump to the caseN code
          B default
case0     do action of case 0
          B AfterCase
case1     do action of case 1
          B AfterCase
          ...
caseN     do action of case N
          B AfterCase
default   do action of default
AfterCase ...
```

Example 11: Finding the Longest Sequence of Repeated Digits

- In Chapter one, we attempted to find the longest sequence of repeated digits.

FIGURE 1.7 A string of digits



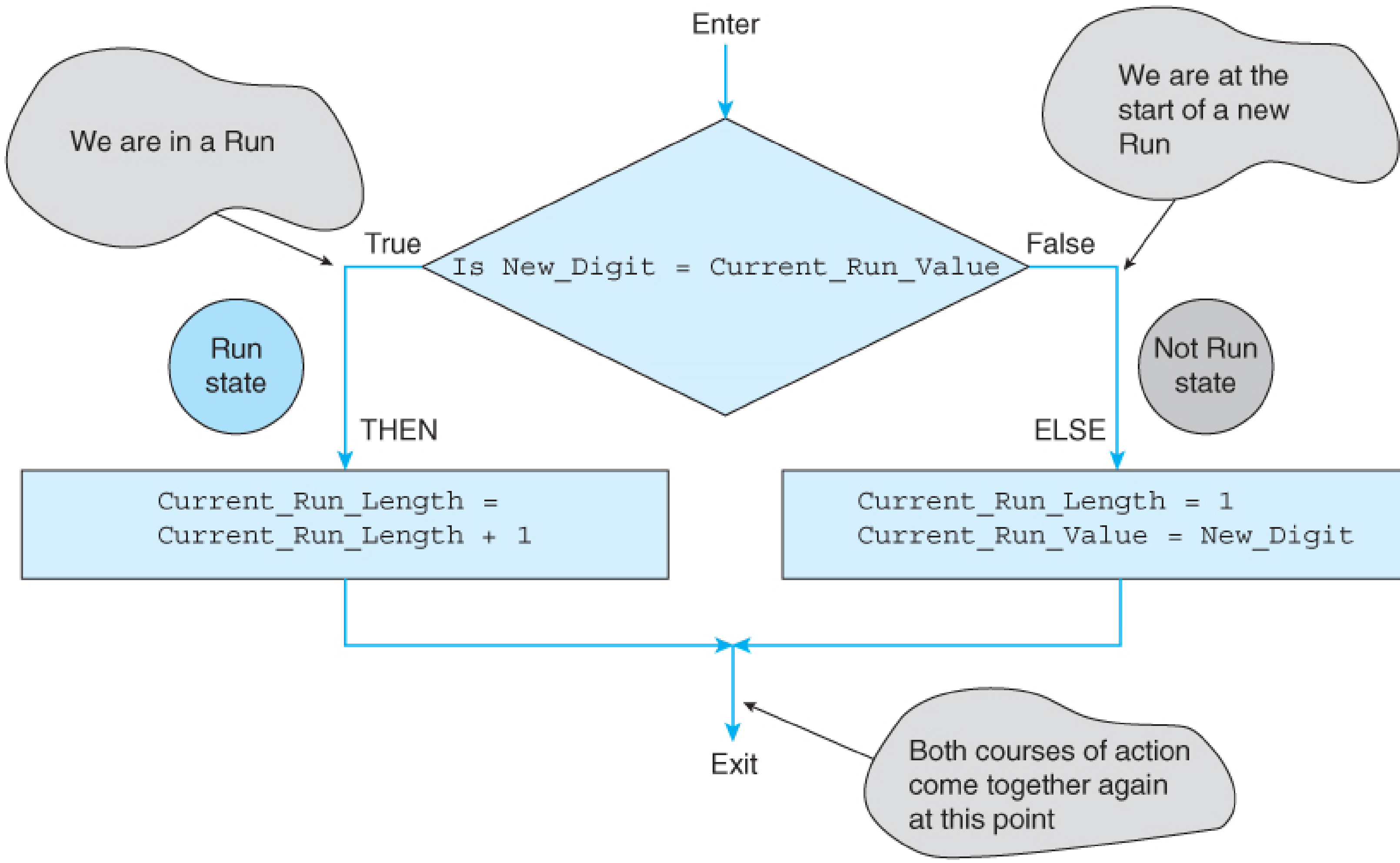
- Let us revisit this problem and implement the solution using ARM assembly language.
- If you recall, we proposed 13 steps to solve this problem:

1. Read the first digit in the string and call it New_Digit
2. Set the Current_Run_Value to New_Digit
3. Set the Current_Run_Length to 1
4. Set the Max_Run to 1
5. REPEAT
6.   Read the next digit in the sequence (i.e., read a New_Digit)
7.   IF its value is the same as Current_Run_Value
8.   THEN Current_Run_Length = Current_Run_Length + 1
9.   ELSE {Current_Run_Length = 1
10.   Current_Run_Value = New_Digit}
11.   IF Current_Run_Length > Max_Run
12.   THEN Max_Run = Current_Run_Length
13. UNTIL The last digit is read

Example 11: Finding the Longest Sequence of Repeated Digits

- Inside the body of the loop, there is an IF…THEN…ELSE construct that we used to test whether we are in a run or not to either increment the run length or reset it to 1

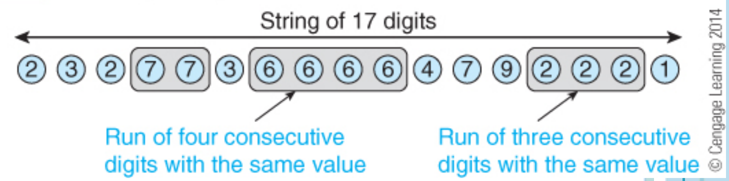
FIGURE 1.10 Illustrating the IF...THEN...ELSE construct graphically



Example 11: Finding the Longest Sequence of Repeated Digits

```
        AREA   RunLength,CODE,READONLY
        ENTRY
        ADR    r9, String ;r9 points to the sting
        LDRB   r0, EoS    ;r0 is the EoS symbol
        LDRB   r1,[r9],#1 ;Step-01: r1 is New_Digit
        MOV    r2,r1      ;Step-02: r2 is the Current_Run_Value
        MOV    r3,#1      ;Step-03: r3 is the Current_Run_Length (set to 1)
        MOV    r4,#1      ;Step-04: r4 is the Max_Run_Length (set to 1)
Repeat  LDRB   r1,[r9],#1 ;Step-05 & 06: REPEAT: Read next digit (i.e., New_Digit)
        CMP    r1,r2      ;Step-07:  Compare New_Digit and Current_Run_Value
        ADDEQ  r3,r3,#1   ;Step-08:  IF same THEN Current_Length=Current_Length+1
        MOVNE  r3,#1      ;Step-09:          ELSE Current_Run_Length = 1
        MOVNE  r2,r1      ;Step-10:               Current_Run_Value  = New_Digit
        CMP    r3,r4      ;Step-11:  IF Current_Run_Length > Max_Run
        MOVGT  r4,r3      ;Step-12:  THEN Max_Run = Current_Run_Length
        TEQ    r0,r1      ;Step-13:  Testing the end of string
        BNE Repeat        ;Step-13: UNTIL all digits tested
Park    B Park            ;parking loop
String  DCB 2,3,2,7,7
        DCB 3,6,6,6,6,4
        DCB 7,9,2,2,2,1
EoS     DCB 0xFF
        END
```

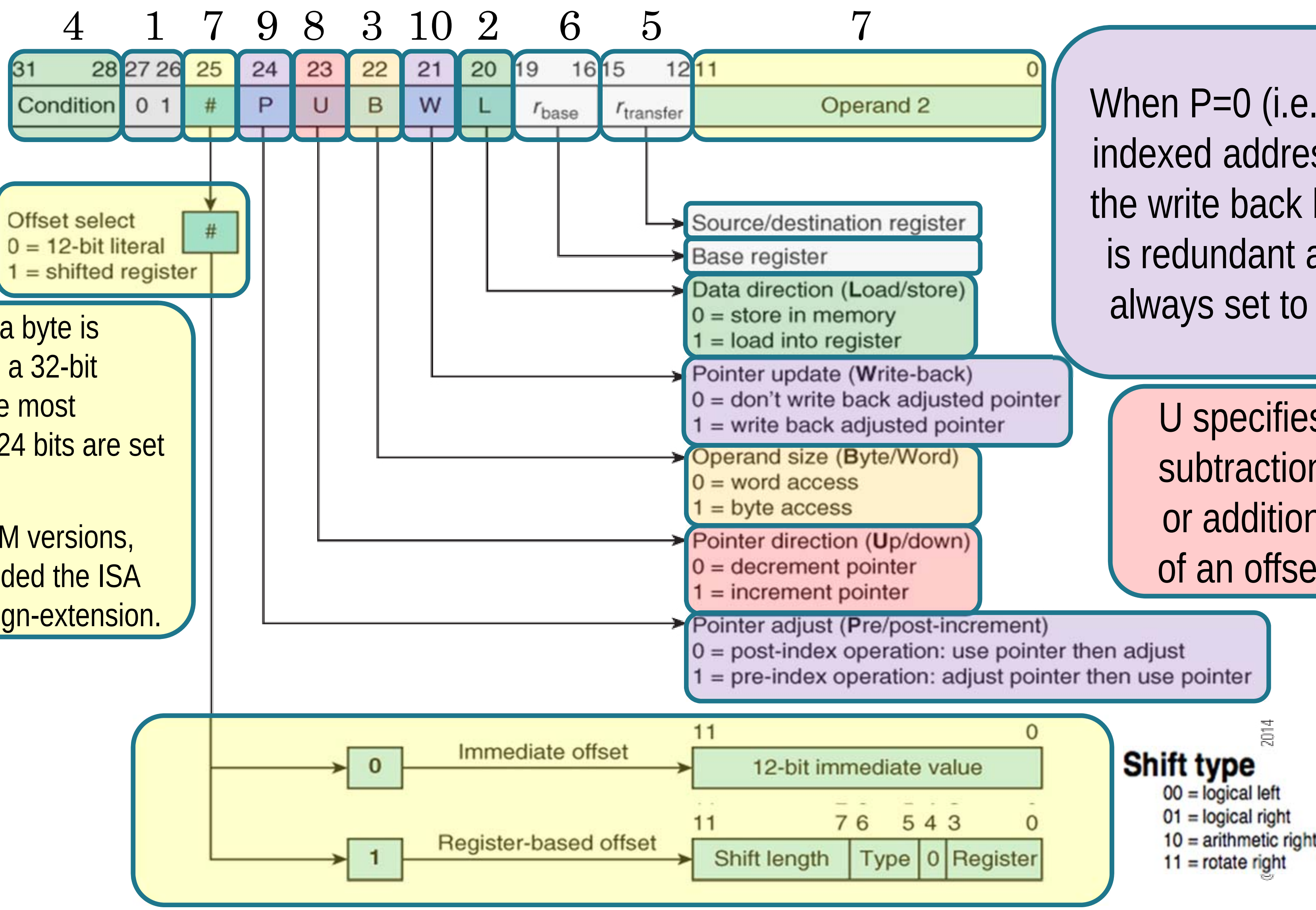
FIGURE 1.7 A string of digits



1. Read the first digit in the string and call it New_Digit
2. Set the Current_Run_Value to New_Digit
3. Set the Current_Run_Length to 1
4. Set the Max_Run to 1
5. REPEAT
6. Read the next digit in the sequence (i.e., read a New_Digit)
7. IF its value is the same as Current_Run_Value
8. THEN Current_Run_Length = Current_Run_Length + 1
9. ELSE {Current_Run_Length = 1
10. Current_Run_Value = New_Digit}
11. IF Current_Run_Length > Max_Run
12. THEN Max_Run = Current_Run_Length
13. UNTIL The last digit is read

ARM Load and Store Encoding

- The figure illustrate the format of the **ARM**'s load and store instructions.



When P=0 (i.e., post-indexed addressing), the write back bit (W) is redundant and is always set to zero.

U specifies subtraction or addition of an offset

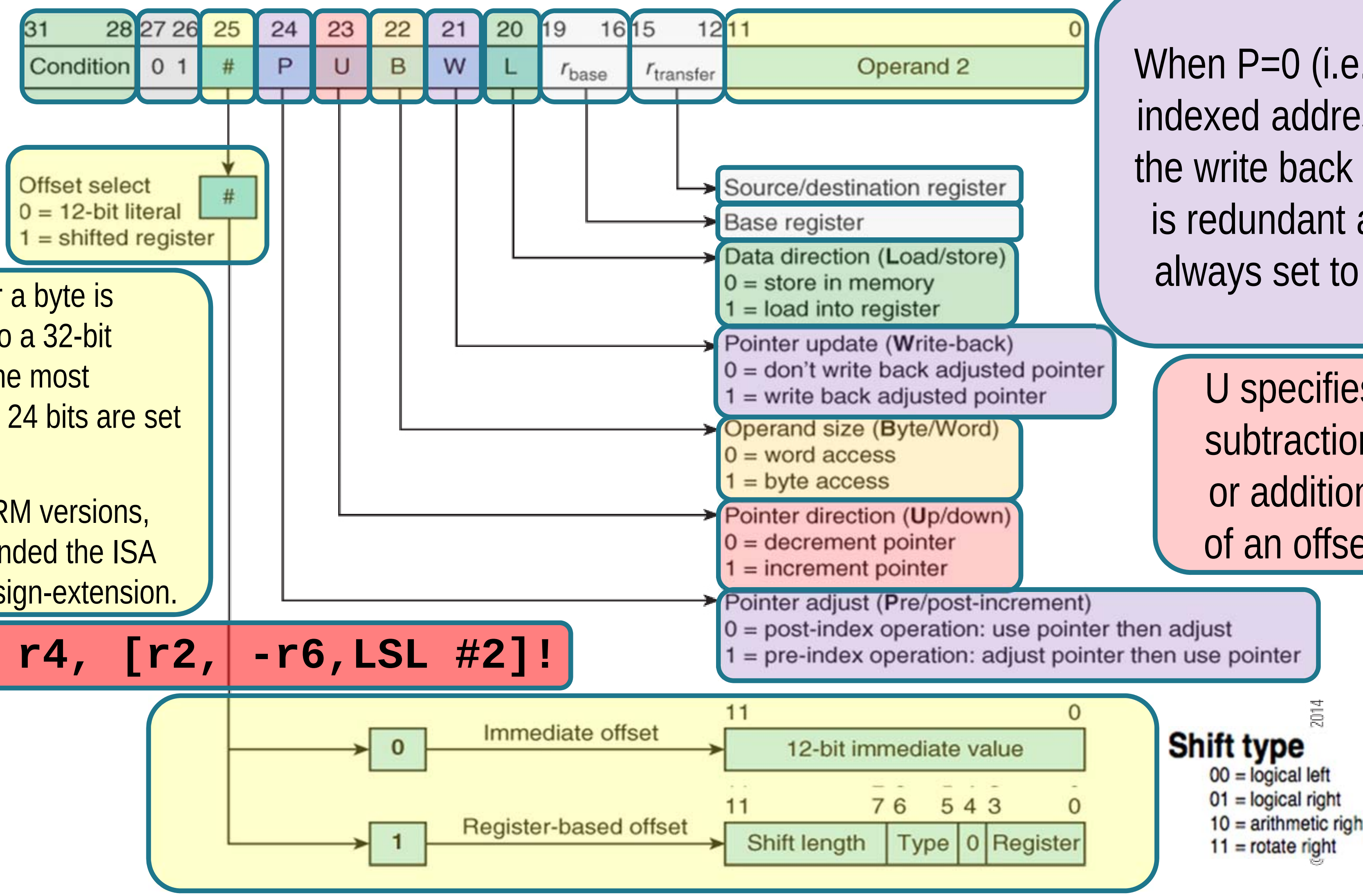
Whenever a byte is loaded into a 32-bit register, the most significant 24 bits are set to zero.

Recent ARM versions, have extended the ISA to permit sign-extension.

ARM Load and Store Encoding

- Decode the following **ARM** machine code instruction **0x57224106**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	0	1	0	1	1	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0



When P=0 (i.e., post-indexed addressing), the write back bit (W) is redundant and is always set to zero.

U specifies subtraction or addition of an offset

Whenever a byte is loaded into a 32-bit register, the most significant 24 bits are set to zero.

Recent ARM versions, have extended the ISA to permit sign-extension.

STRPL r4, [r2, -r6,LSL #2]!

ARM Load and Store Encoding

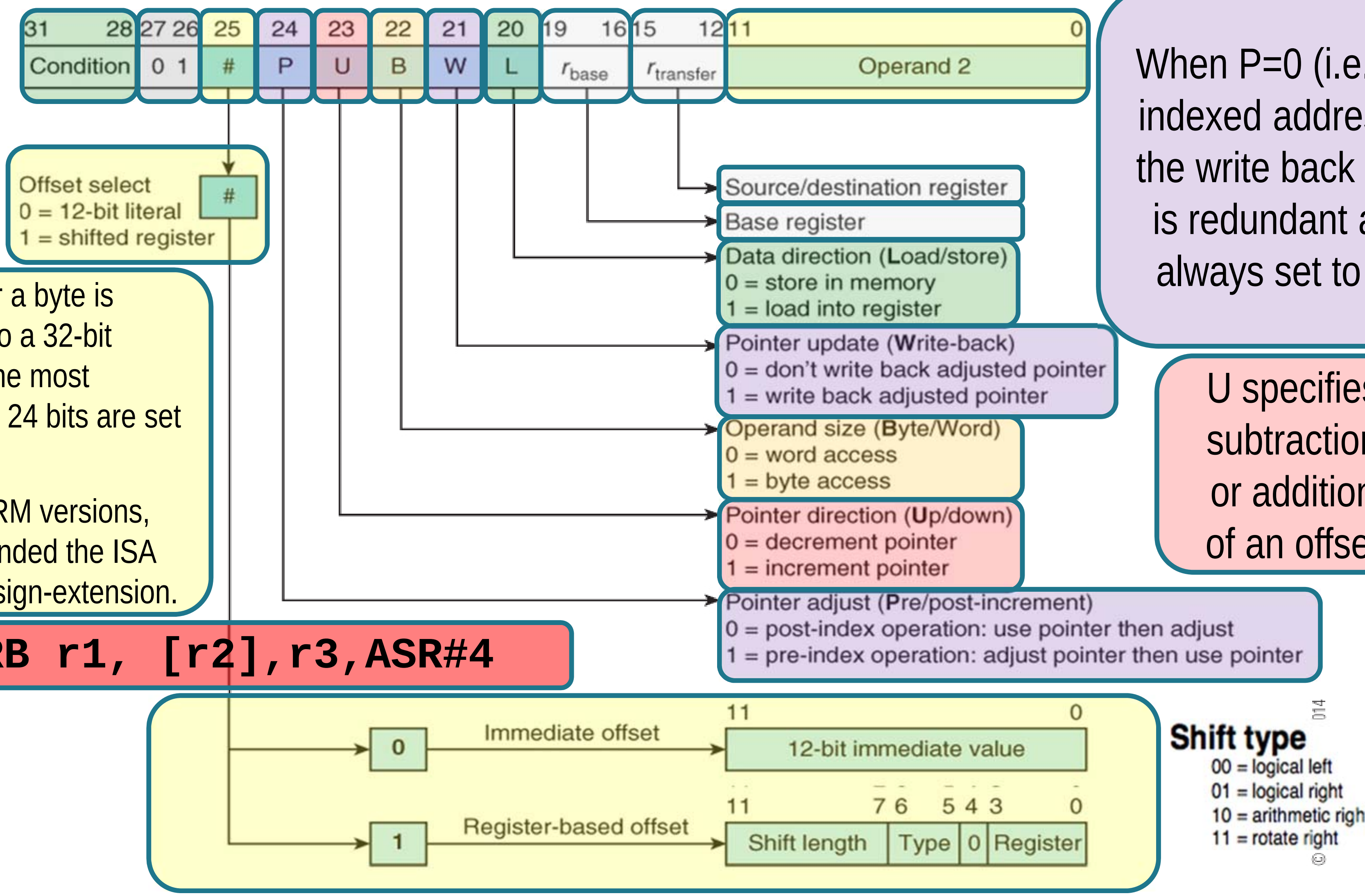
Decoding the ARM Instruction **STRPL r4,[r2,-r6,LSL#2]!**

Field Name	Value	Action	Interpretation
Condition	0101	PL	Execute on positive
OP-code	01		Defines load/store instruction
#	1	Operand 2 format	Operand is a shifted register
P	1	Pre/post adjust	Adjust pointer before using
U	0	Pointer direction	Decrement pointer
B	0	Byte/word	This is a word access
W	1	Pointer write back	Update pointer after use
L	0	Load/store	Store data in memory
r_{base}	0010	Base register	r2 is the base (pointer) register
$r_{transfer}$	0100	Source/destination	r4 is the source in this store instruction
Shift length	00010	Shift length	Shift the register 2 places
Shift type	00	Logical shift left	Logical shift left the offset in r6
Op-code	0		
Shift register	0110	Specified register to be shifted	r6 is shifted twice

ARM Load and Store Encoding

- Decode the following **ARM** machine code instruction **0xE6D21243**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1



When P=0 (i.e., post-indexed addressing), the write back bit (W) is redundant and is always set to zero.

U specifies subtraction or addition of an offset

Whenever a byte is loaded into a 32-bit register, the most significant 24 bits are set to zero.

Recent ARM versions, have extended the ISA to permit sign-extension.

LDRB r1, [r2],r3,ASR#4

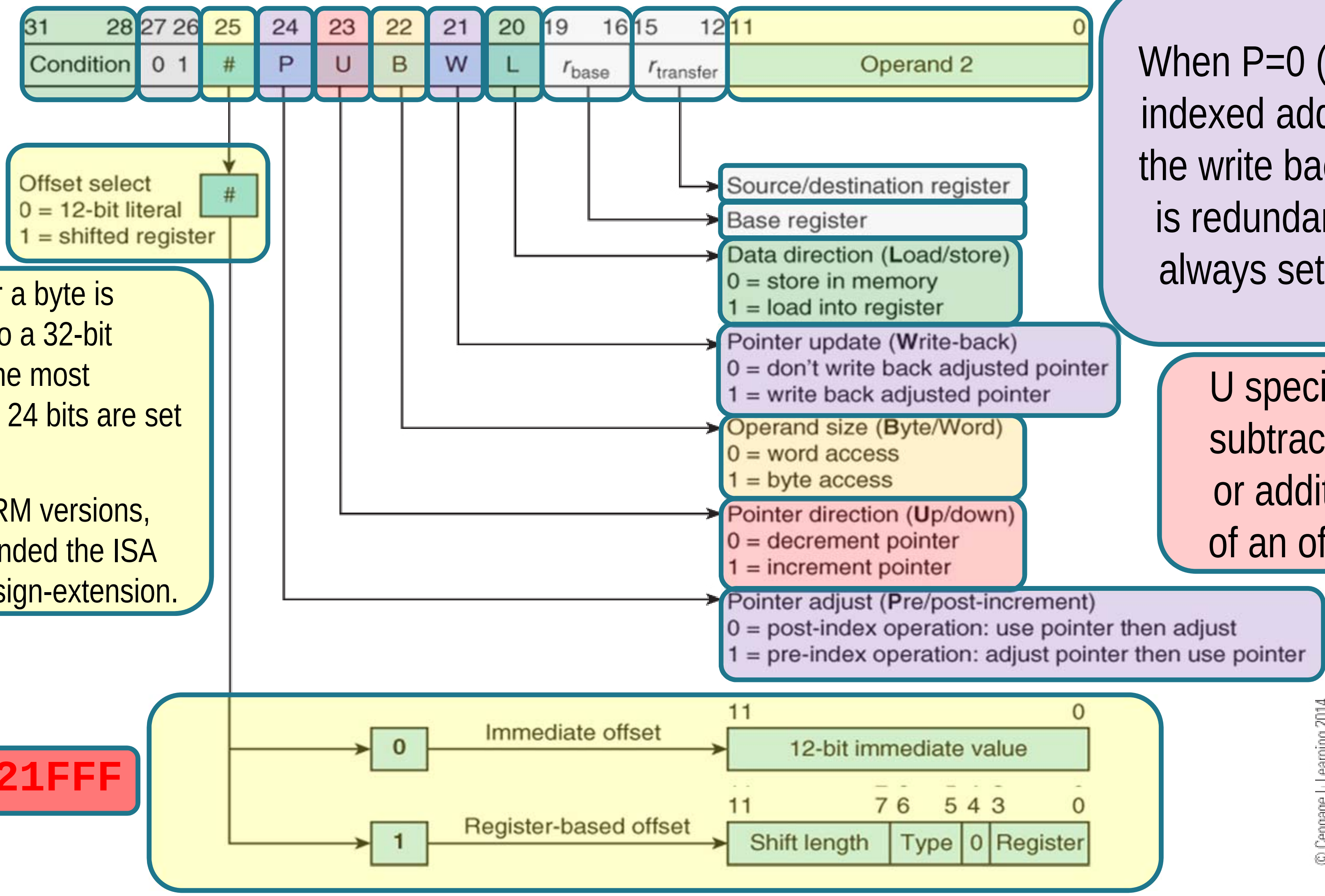
ARM Load and Store Encoding

Decoding the ARM Instruction **LDR r1, [r2],r3,ASR#4**

Field Name	Value	Action	Interpretation
Condition	1110	AL	Always (default)
OP-code	01		Defines load/store instruction
#	1	Operand 2 format	Operand is a shifted register
P	0	Pre/post adjust	Adjust pointer after using
U	1	Pointer direction	Increment pointer
B	0	Byte/word	This is a word access
W	0	Pointer write back	As P=0, W is redundant and always=0
L	1	Load/store	Load data from memory
r_{base}	0010	Base register	r2 is the base (pointer) register
$r_{transfer}$	0001	Source/destination	r1 is the destination in this load instruction
Shift length	00100	Shift length	Shift the register 4 places
Shift type	10	Logical shift left	Arithmetic shift right the offset in r3
Op-code	0		
Shift register	0011	Specified register to be shifted	r3 is shifted four times

ARM Load and Store Encoding

- Encode the following **ARM** instruction **STRGT r1,[r2,#-0xFFF]**



When P=0 (i.e., post-indexed addressing), the write back bit (W) is redundant and is always set to zero.

U specifies subtraction or addition of an offset

Whenever a byte is loaded into a 32-bit register, the most significant 24 bits are set to zero.

Recent ARM versions, have extended the ISA to permit sign-extension.

0xC5021FFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

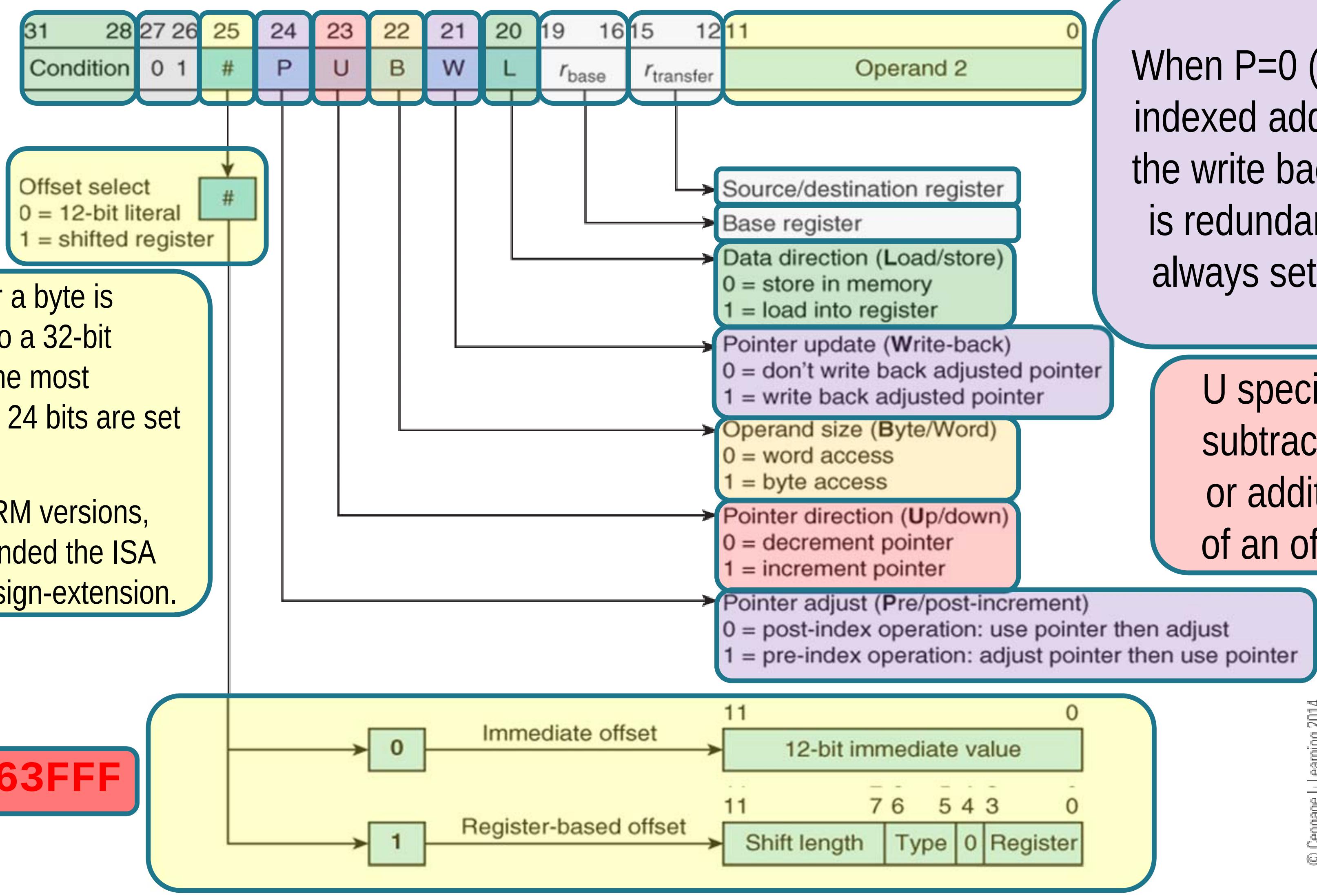
ARM Load and Store Encoding

Decoding the ARM Instruction **STRGT r1,[r2,#-0xFFF]**

Field Name	Value	Action	Interpretation
Condition	1100	GT	Execute on greater than
OP-code	01		Defines load/store instruction
#	0	Operand 2 format	Operand is immediate
P	1	Pre/post adjust	Adjust pointer before using
U	0	Pointer direction	Decrement pointer
B	0	Byte/word	This is a word access
W	0	Pointer write back	Update pointer before use
L	0	Load/store	Store data in memory
r_{base}	0010	Base register	r2 is the base (pointer) register
$r_{transfer}$	0001	Source/destination	r1 is the source in this store instruction
Immediate offset	111111111111	Shift length	Offset value = 0xFFF

ARM Load and Store Encoding

- Encode the following ARM instruction **LDREQ r3,[r6],#-0xFFF**



When P=0 (i.e., post-indexed addressing), the write back bit (W) is redundant and is always set to zero.

U specifies subtraction or addition of an offset

Whenever a byte is loaded into a 32-bit register, the most significant 24 bits are set to zero.

Recent ARM versions, have extended the ISA to permit sign-extension.

0x04163FFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

ARM Load and Store Encoding

Decoding the ARM Instruction **LDREQ r3,[r6],#-0xFFF**

Field Name	Value	Action	Interpretation
Condition	0000	EQ	Execute on equal
OP-code	01		Defines load/store instruction
#	0	Operand 2 format	Operand is immediate
P	0	Pre/post adjust	Adjust pointer after using
U	0	Pointer direction	Decrement pointer
B	0	Byte/word	This is a word access
W	0	Pointer write back	Update pointer before use
L	1	Load/store	Load data from memory
r_{base}	0110	Base register	r6 is the base (pointer) register
$r_{transfer}$	0011	Source/destination	r3 is the destination in this load instruction
Immediate offset	111111111111	Shift length	Offset value = 0xFFF

ARM Load and Store Encoding

- Encode the following **ARM** instructions?

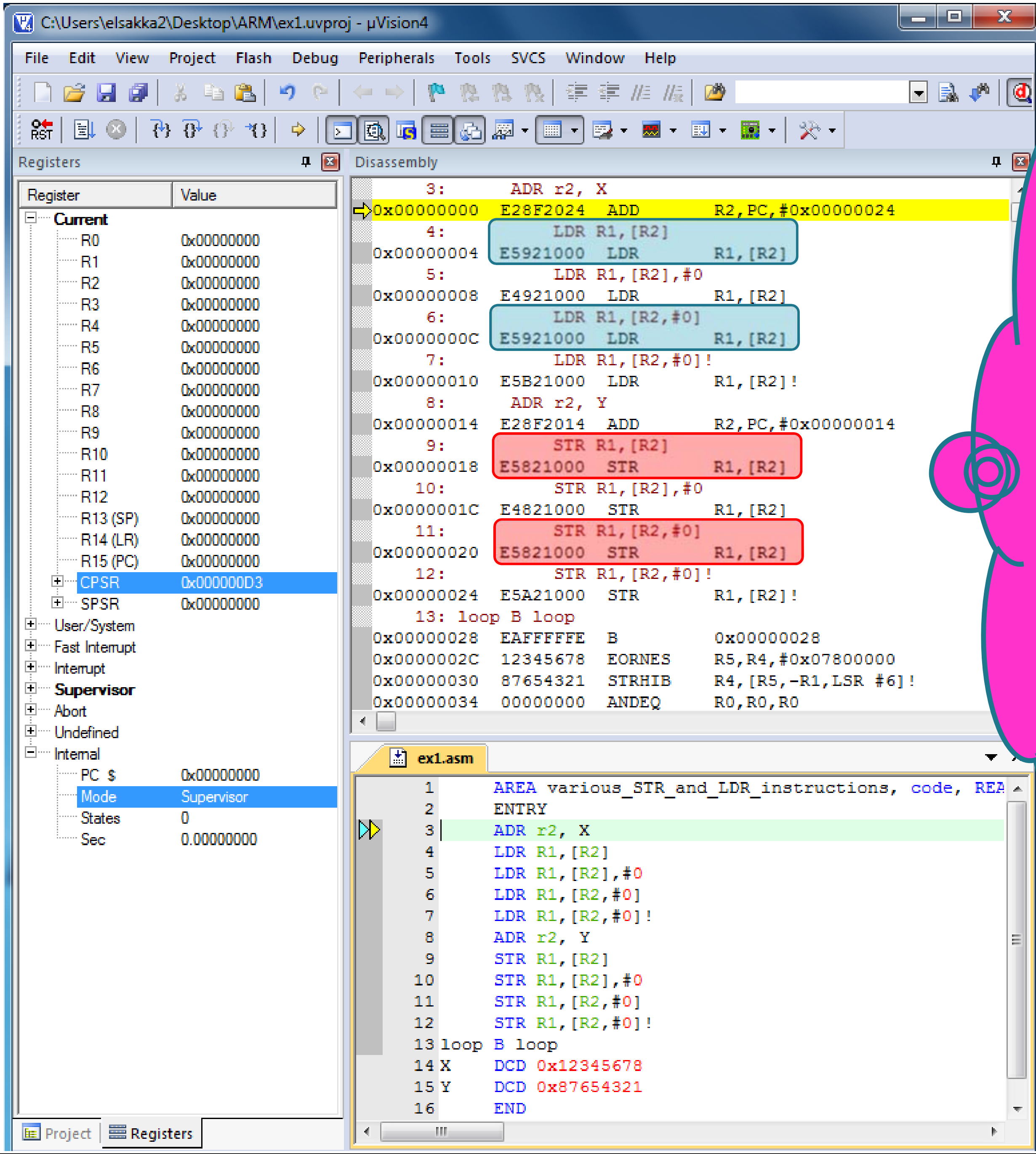
```
LDR R1,[R2]
LDR R1,[R2],#0
LDR R1,[R2,#0]
LDR R1,[R2,#0]!
```

- Is there any *effective* difference between these instructions?

ARM Load and Store Encoding

```
      AREA various_STR_and_LDR_instructions, code, READONLY
      ENTRY
      ADR r2, X
      LDR R1,[R2]
      LDR R1,[R2],#0
      LDR R1,[R2,#0]
      LDR R1,[R2,#0]!
      ADR r2, Y
      STR R1,[R2]
      STR R1,[R2],#0
      STR R1,[R2,#0]
      STR R1,[R2,#0]!
loop B loop
X     DCD 0x12345678
Y     DCD 0x87654321
      END
```

ARM Load and Store Encoding



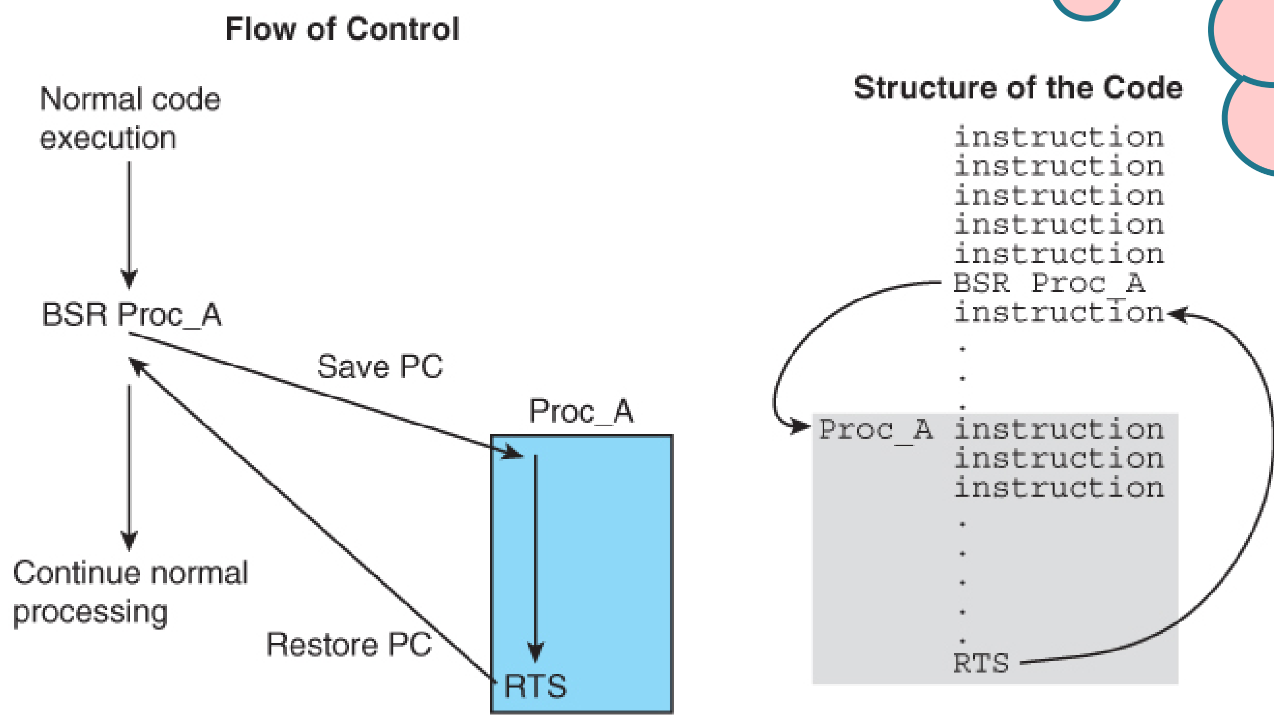
To test this program, you need to change the permission of memory locations from 0x30 to 0x33 (i.e., the location of Y) to make it read/write.

You also need to open a memory window to see the effect of the STR instructions.

Subroutine Call and Return

- The instruction *BSR Proc_A calls* subroutine *Proc_A*.
  - The processor saves the address of the next instruction to be executed in a safe place, and
  - loads the program counter with the address of the first instruction in the subroutine.
- At the end of the subroutine a *return from subroutine instruction*, *RTS*,
  - causes the processor to return to the point immediately following the subroutine call.

FIGURE 3.40 The subroutine call and return



This is not used by ARM processors

ARM Support for Subroutines

- **RISC** processors (including **ARM**) ***do not provide*** a fully automatic subroutine call/return mechanism like **CISC** processors.

This is the main difference between B and BL

- **ARM**'s *branch with link* instruction, **BL**,
  - automatically saves the return address in register `r14`.

- The branch instruction (Figure 3.41) has a 24-bit ***signed*** program counter relative offset (*word address offset*).
- The 24-bit offset is
  - shift left twice the 24-bit offset to convert the word-offset address to a byte address,
  - ***sign-extended*** to 32 bits,
  - added it to the current value of the program counter (***the result is PC ± 32 MBytes***).

Do not forget the pipelining effect

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