Xi'an Jiaotong-Liverpool University
西交利物浦大學

Assemblers and Instruction Encoding

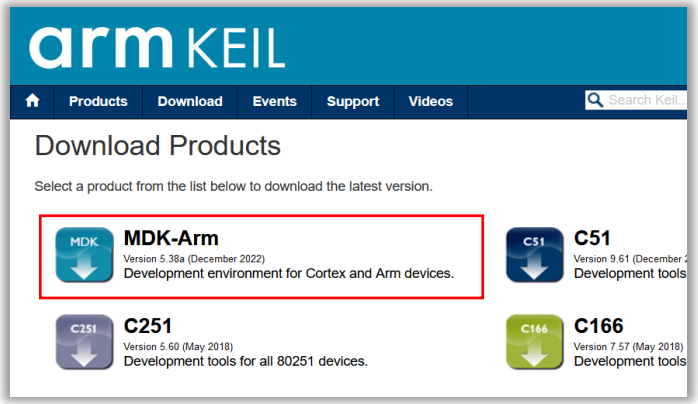
Jianjun Chen

Contents

- ARM assemblers
- Instruction encoding
- Assembler directives

Keil MDK

- The content of this lecture can be verified using the Keil MDK: https://www.keil.com/download/product/



You can compile and run small programs for free. Full version is quite expensive for personal use.

- A tutorial from past is also uploaded to LMO to guide you to setup the project.
  - Replace the "main.c" with an assembly file "main.s"
  - Make sure "__main" is defined and exported.

```
AREA    MY_PROG,CODE,READONLY
        EXPORT __main
__main
        …
```

Assemblers and Instruction Encoding

What are assemblers?

Encoding examples

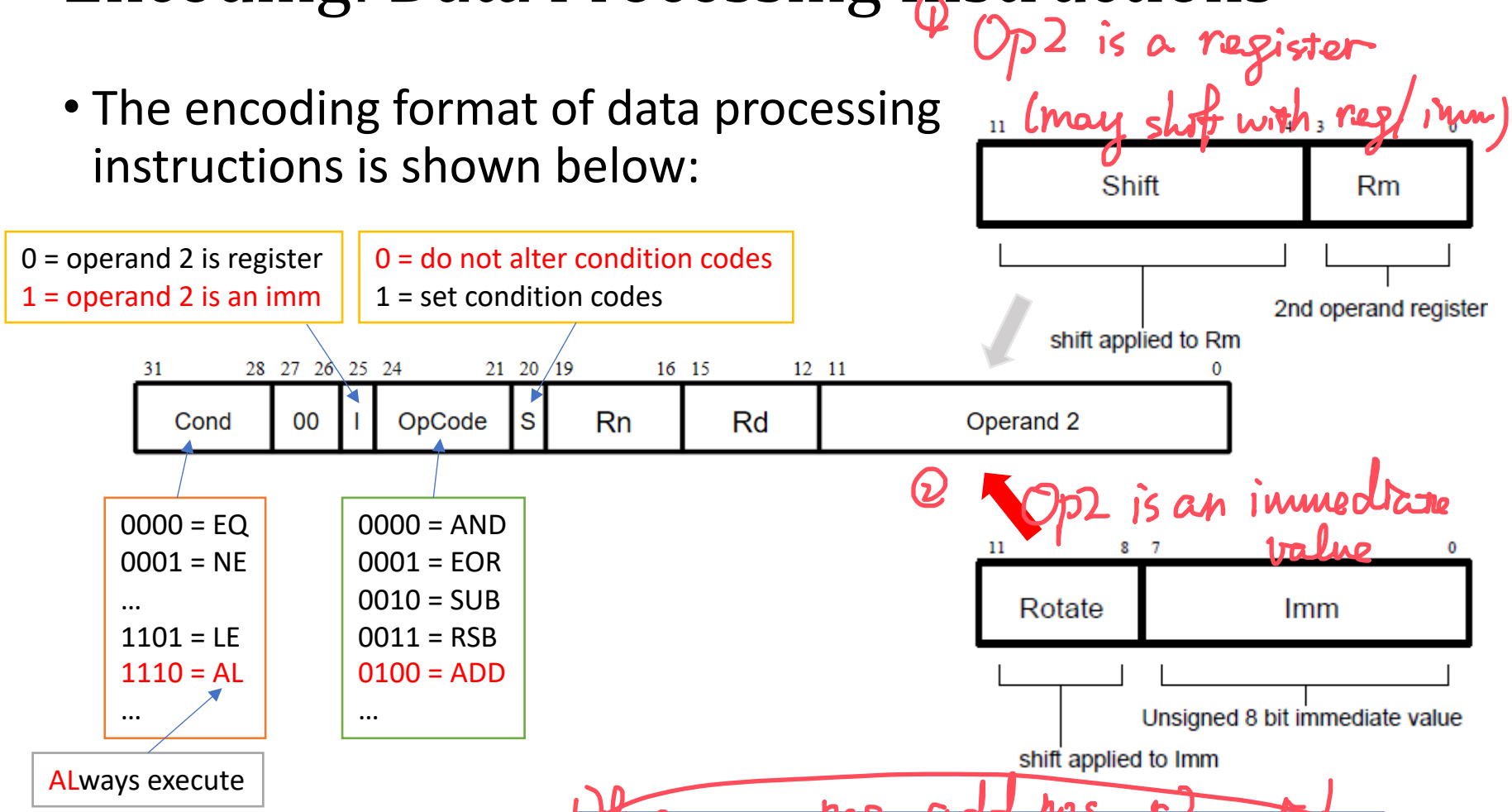
Assembler directives

Assembler

- The VisUAL emulator can run simple programs written in ARM *assembly language*.
  - However, it is only an emulator.
- Real ARM-based computers execute *machine code*.
  - Binary sequences like “0101010…”, in the main memory.
- **Assemblers** are programs that translate assembly code into machine code.
  - **GNU toolchain for ARM**: https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads.
  - **ARM tools assembler**: https://developer.arm.com/downloads/-/arm-compiler-for-embedded
    - This assembler is included in Keil MDK 5: https://www2.keil.com/mdk5

Encoding: Data Processing Instructions

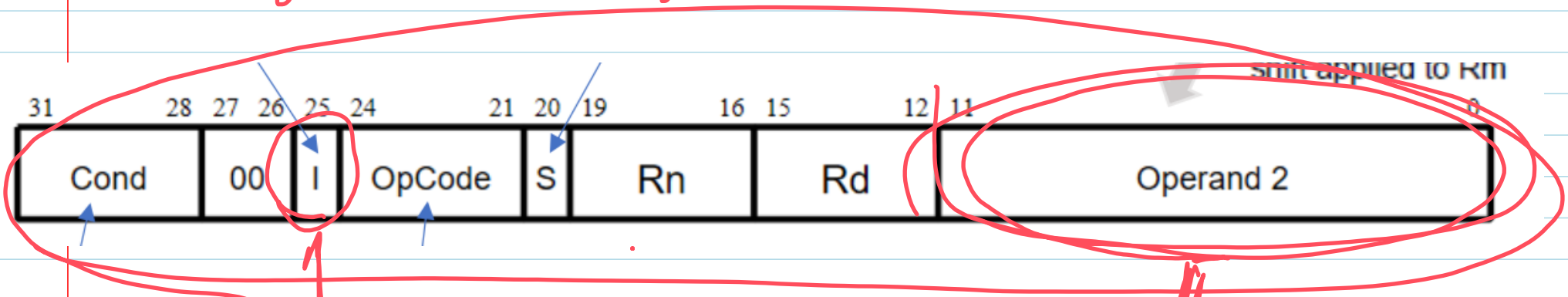
- The encoding format of data processing instructions is shown below:




```
ADD r1, r2, #1
```

```
1110 00 0 0100 0 0010 0001
0000 00000001
```

Encoding: Data Processing Instructions' Categories

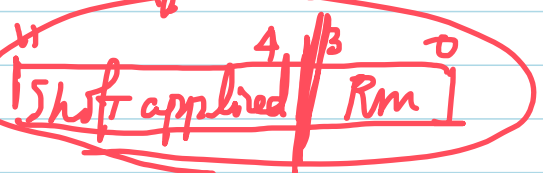


'I' decides whether Op2 is an immediate value / a register.

If I = 1 (imm).

rotations Immediate value

If I = 0 (reg).

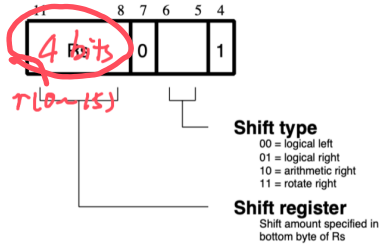


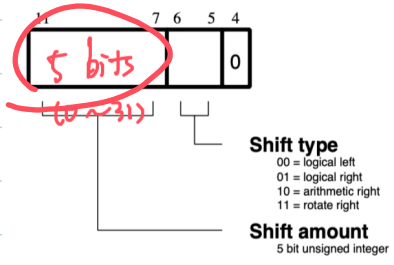
1° 4th index = 1 :

shift a reg's value

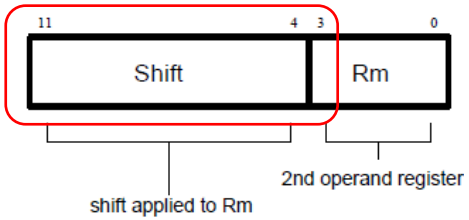
2° 4th index = 0.

shift an immediate value

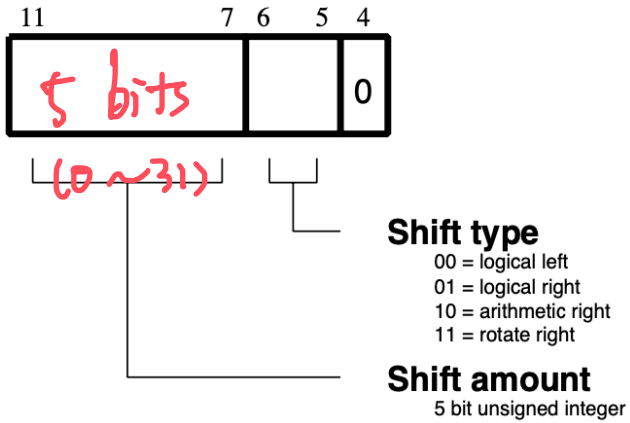


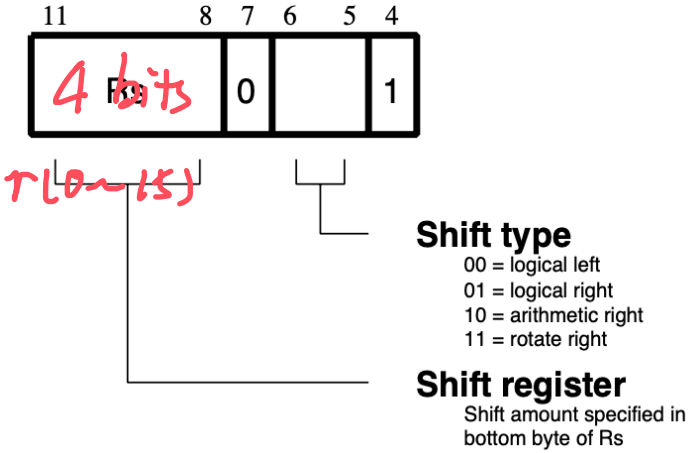


Embedded Shift Operation



- The encoding format also tells us that arithmetic operations like `ADD`, `SUB` and `RSB` supports embedded shift operations.





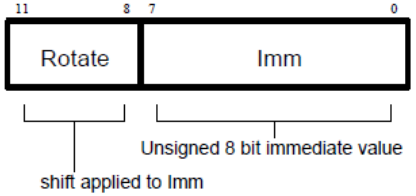
- Examples:

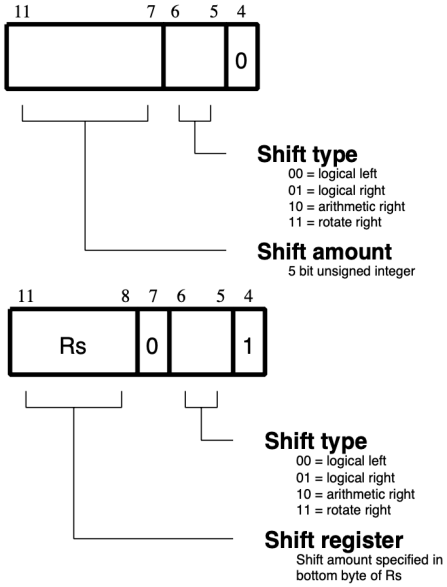
```
ADD     R0, R1, (R2, LSL #1)
ADD     R0, R1, (R2, ROR R3)
```

"LSL #1" is applied to R2

- Shift amount: The maximum number of shifts is 31.
- Shift type: Which shift operation to be applied.

Encoding Examples





- Given 3 instructions:

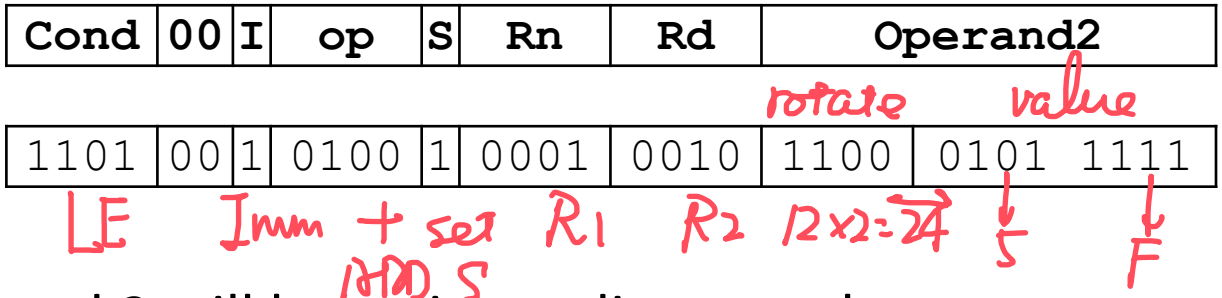
```
RSBS R9, R10, SP
ADDSLE R2, R1, #0x5f00
SUBSEQ R0, R1, R2, LSL R3
```

- Match the following machine code to the instructions above:

	Cond	00	I	op	S	Rn	Rd	Operand2
1	1101	00	1	0100	1	0001	0010	1100 0101 1111
2	0000	00	0	0010	1	0001	0000	0011 0001 0010
3	1110	00	0	0011	1	1010	1001	0000 0000 1101

LE Imm ADD R1 R2 rot=24 Imm value = 5F00
rot 24 = rot 8
EQ reg SUB R1 R0 R3
1 on index 4: shift a register
shift 0
0 on index 4: shift an immediate value
1TL reg RSB R10 R9

Answer



- `Cond` is LE.
- `I` is 1, the operand 2 will be an immediate number.
- `Op` is `ADD`.
- `S` is 1, so it will set flags.
- `Rn` is 0b0001, which means `R1`.
- `Rd` is 0b0010, which means `R2`.
- Immediate:
  - Rotation is 0b1100, which means 12*2 = 24 rotations.
  - 0b01011111 (0x5F) rotated right 24 times in a 32-bit register becomes 0x5F00.
- The answer is "**`ADDSLE R2, R1, #0x5f00`**".

ROR by 24 = ROR 8

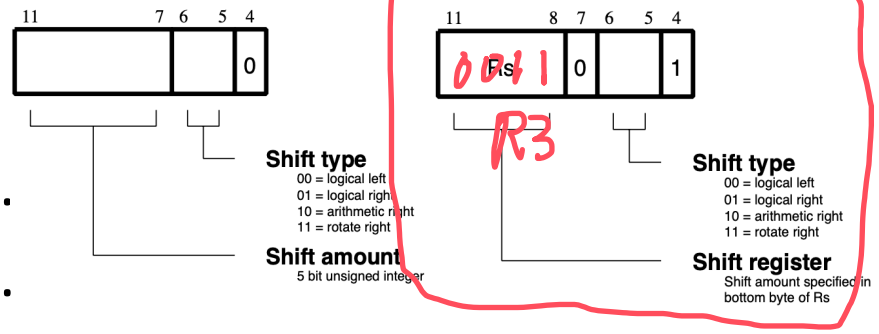
Thus 5F turns into 5F00

ROR by X = ROL by (32-X)
Ex: ROR by 30 = ROL by 2

Answer

Cond	00	I	op	S	Rn	Rd	Operand2
0000	00	0	0010	1	0001	0000	0011 0001 0010

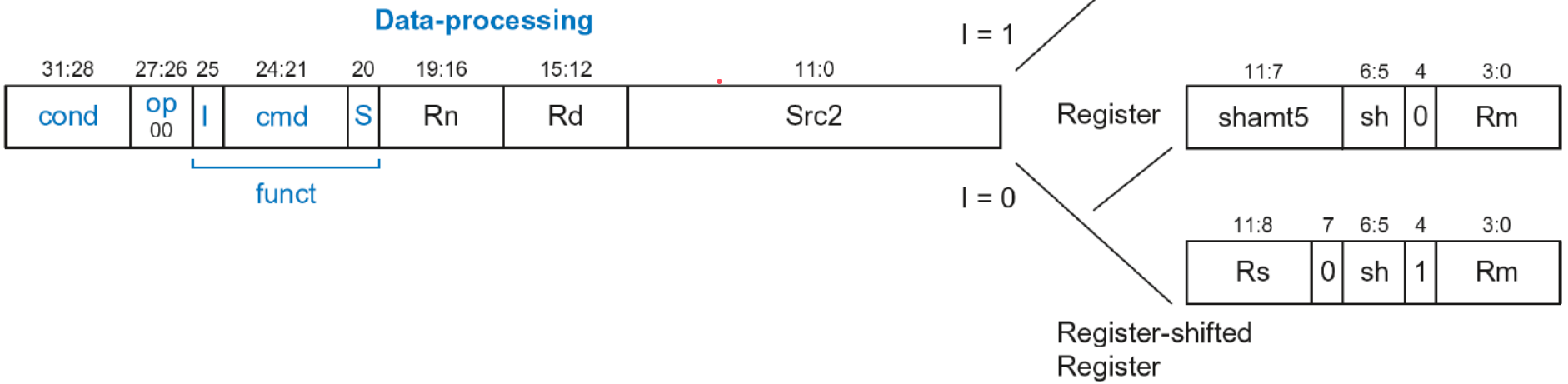
- `Cond` is EQ.
- `I` is 0, the operand 2 will be a register.
- `Op` is 0010, which is `SUB`.
- `S` is 1, so it will set flags.
- `Rn` is 0b0001, which means `R1`.
- `Rd` is 0b0000, which means `R0`.
- Operand2:
  - Register is 0010, which means `R2`.
  - The 4th bit is 1, so a shift register is used. The register is 0011 (`R3`).
  - Shift type is 00, which means `LSL`.
- The answer is "**`SUBSEQ R0, R1, R2, LSL R3`**"



To summarize the encoding format of **data processing** instructions

Data-processing *Src2* Variations

- *Src2* can be:
  - Immediate
  - Register
  - Register-shifted register



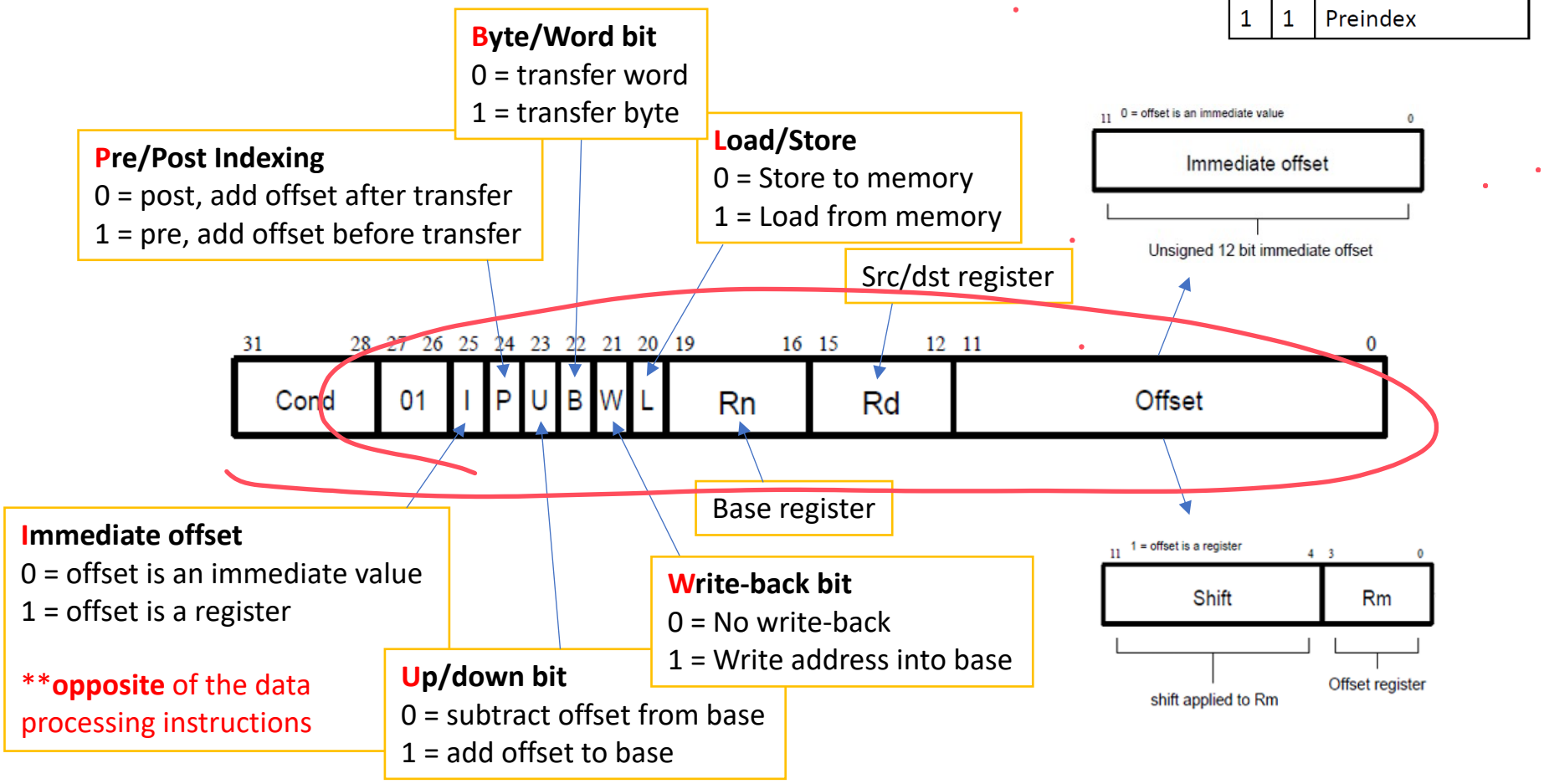
Lecture 2

Encoding: LDR and STR

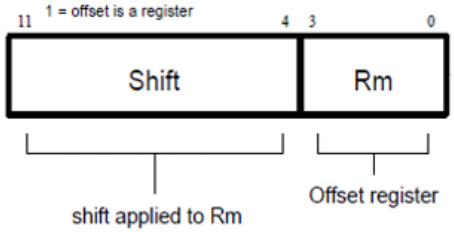
Indexing Mode

P	W	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

- The encoding format of LDR, STR is shown below:



Offset Register in LDR and STR



- `LDR` and `STR` support adding/subtracting offsets stored in another register:

  `LDR/STR``{cond}  Rt, [Rn, ±Rm{, shift}]`

  `LDR/STR``{cond}  Rt, [Rn, ±Rm{, shift}]!`

  `LDR/STR``{cond}  Rt, [Rn], ±Rm{, shift}`

- Examples:
- `LDR r0, [r1, r2, LSL #2]`
  - r1 stores the base address, offset is r2's value shifted left twice.
- `LDRB r0, [r1, r2, LSR #1]!`
  - Pre-indexed access, the offset is r2's value shifted right once.
- `LDR r0, [r1], r2, LSL #3`
  - Post-indexed access, the offset is r2's value shifted right thrice.

Symbols + and – are not supported in VisUAL. They can be used in Keil IDE

Refer to the next page for detailed explanations

Offset Register in LDR and STR

Symbol	Address	Value
my_data	0×200	0×AABBCCDD
	0×204	0×2
	0×208	0×3
	0×20C	0×4

R1	0×200
R2	0×2

0b10 → 0b1000 0x2 left shifted 2 bits as offset ⇒ the offset is #0x8.

```
LDR r0, [r1, r2, LSL #2] #8.
Same as
LDR r0, [r1, #0x8]
```

```
r0 = 0x3
r1 = 0x200
```

0b10 → 0b01 0x2 right shifted 1 bit as offset ⇒ the offset is #0x1.

```
LDRB r0, [r1, r2, LSR #1]!
```

pre-index ⇒ r1 is also updated with r1+offset, i.e. 0x200 → 0x201

```
r0 = 0xCC
r1 = 0x201
```

0x2 left shifted 3 bits as offset ⇒ the offset is #0x10.

```
LDR r0, [r1], r2, LSL #3
```

post-index

0b10 → 0b1000 = 0x10

```
r0 = 0xAABBCCDD
r1 = 0x210
```

Detailed Explanations:

Offset Register in LDR and STR

Symbol	Address	Value
my_data	0×200	0×AABBCCDD
	0×204	0×2
	0×208	0×3
	0×20C	0×4

R1	0×200
R2	0×2

① `LDR r0, [r1, r2, LSL #2]`
Same as
`LDR r0, [r1, #0x8]`
→ r0 = 0x3, r1 = 0x200

② pre-indexing: Change the address first
`LDRB r0, [r1, r2, LSR #1]!`
→ r0 = 0xCC, r1 = 0x201

③ post-indexing: Get the value first
`LDR r0, [r1], r2, LSL #3`
→ r0 = 0xAABBCCDD, r1 = 0x210

items between {} are optional

① **LDR**{type}{cond} Rt, [Rn {, #offset}]
② **LDR**{type}{cond} Rt, [Rn, #offset]!
③ **LDR**{type}{cond} Rt, [Rn], #offset

① [r1, r2, LSL #2]
offset is 0x2 being left shifted for 2 bits, which is
#0b10 → #0b1000
= #0x8
∴ LDR r0, [r1, #0x8]
It displays 0x3 on r0.
with r1 staying the same.

② [r1, r2, LSR #1]!
Offset is r2 being shifted for 1 bit which is #0b10 → #0b1, since it's a pre-indexed one, r1's value also changes from 0x200 to 0x201 and the value of 0x201 address (CC) is displayed on r0.

Detailed Explanations (②):

Offset Register in LDR and STR

Symbol	Address	Value
my_data	0×200	0×AABBCCDD
	0×204	0×2
	0×208	0×3
	0×20C	0×4

R1	0×200
R2	0×2

①
```
LDR r0, [r1, r2, LSL #2]
Same as
LDR r0, [r1, #0x8]
```
→ r0 = 0x3, r1 = 0x200

② pre-indexing: Change the address first
```
LDRB r0, [r1, r2, LSR #1]!
```
→ r0 = 0xCC, r1 = 0x201

③ post-indexing: Get the value first
```
LDR r0, [r1], r2, LSL #3
```
→ r0 = 0xAABBCCDD, r1 = 0x210

① items between {} are optional
```
LDR{type}{cond} Rt, [Rn {, #offset}]
LDR{type}{cond} Rt, [Rn, #offset]!
LDR{type}{cond} Rt, [Rn], #offset
```

②

[r1], r2, LSL, #3

offset

post-indexed

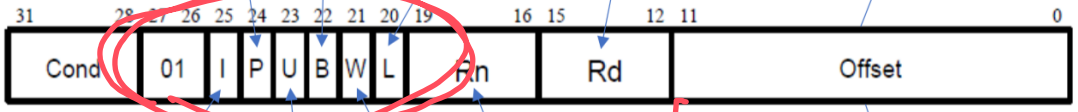
r2=2, left shift 3 times

0b10000 = 0x10

∴ Get the value stored at the original R1 and then update it by 10

∴ r0 = 0x11223344

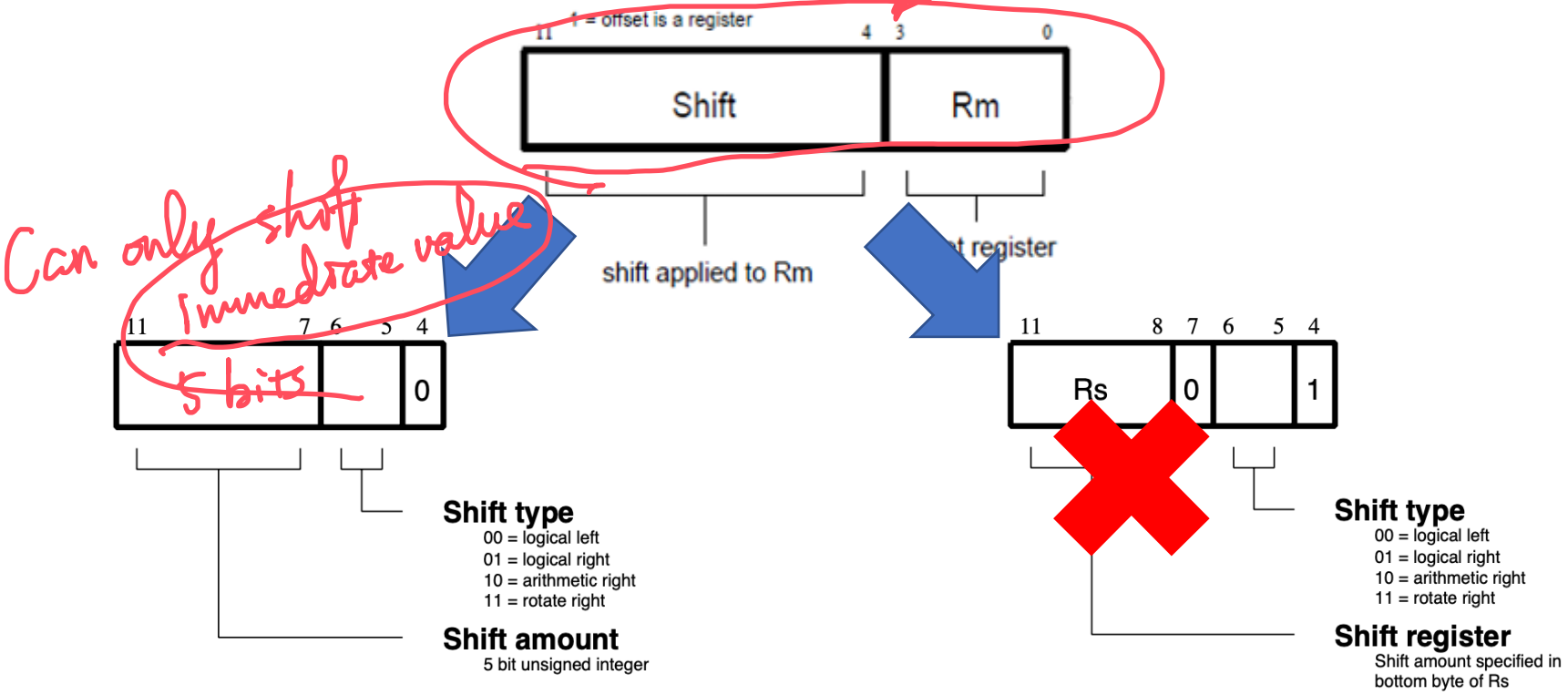
r1 = 0x210



Offset Register in LDR and STR

- Note that register-specified shift is not supported in LDR and STR. Thus, you cannot write:

```
LDR r0, [r1, r2, LSL r3]
```



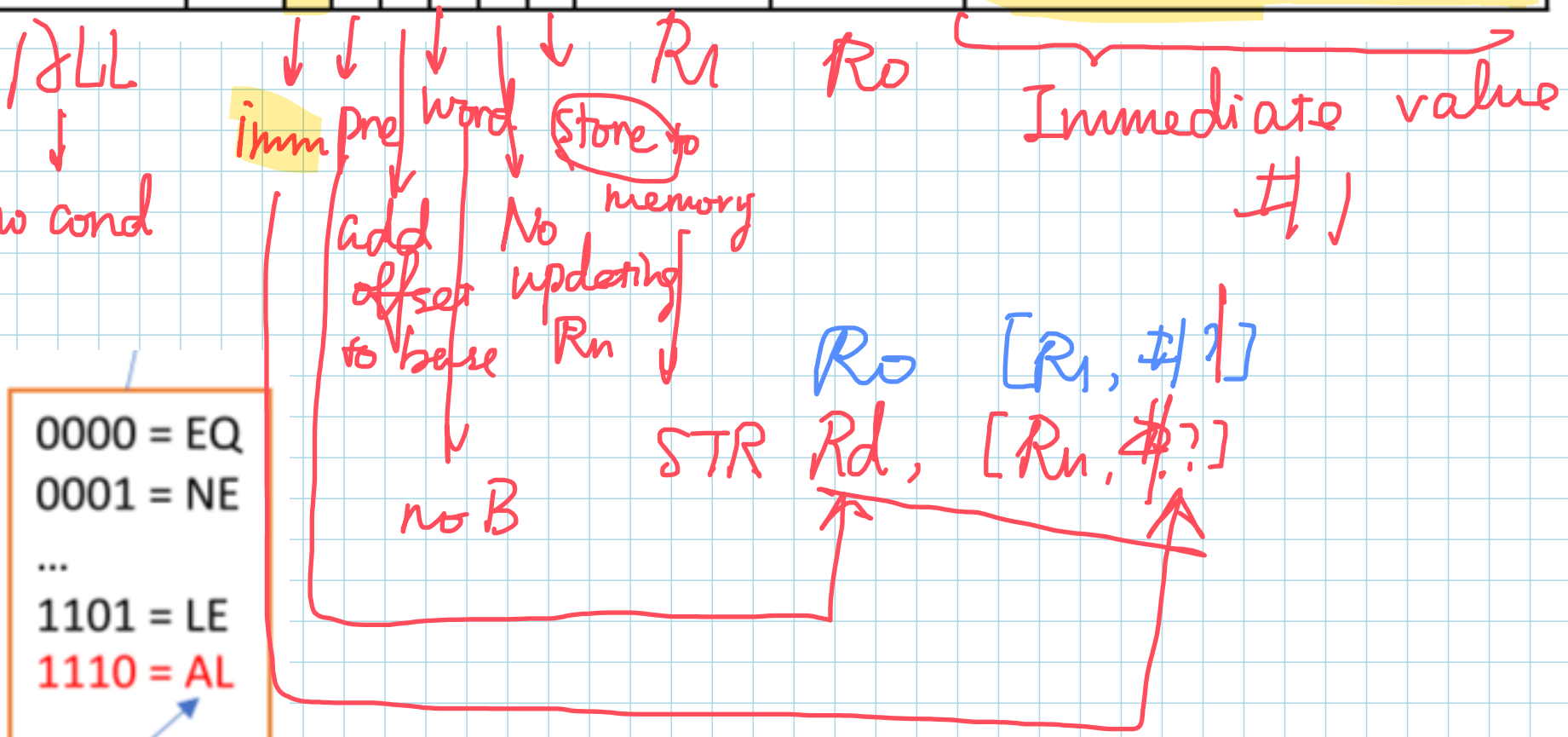
Encoding Examples

- What do the following machine code do?

	Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1	1110	01	0	1	1	0	0	0	0001	0000	0000 0000 0001
2	1110	01	0	1	1	0	1	1	0001	0000	0000 0000 0001
3	1110	01	1	0	1	1	0	1	0001	0000	0011 0010 0010
4	1110	01	0	1	0	0	1	1	0001	0000	0000 0000 0001

1.

Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1110	01	0	1	1	0	0	0	0001	0000	0000 0000 0001



2.

Cond	01	I	P	U	B	W	L	Rn	Rd	Offset

1110	01	0	1	1	0	1	1	0001	0000	0000 0000 0001

1011

0000 = EQ
0001 = NE
...
1101 = LE
1110 = AL
...

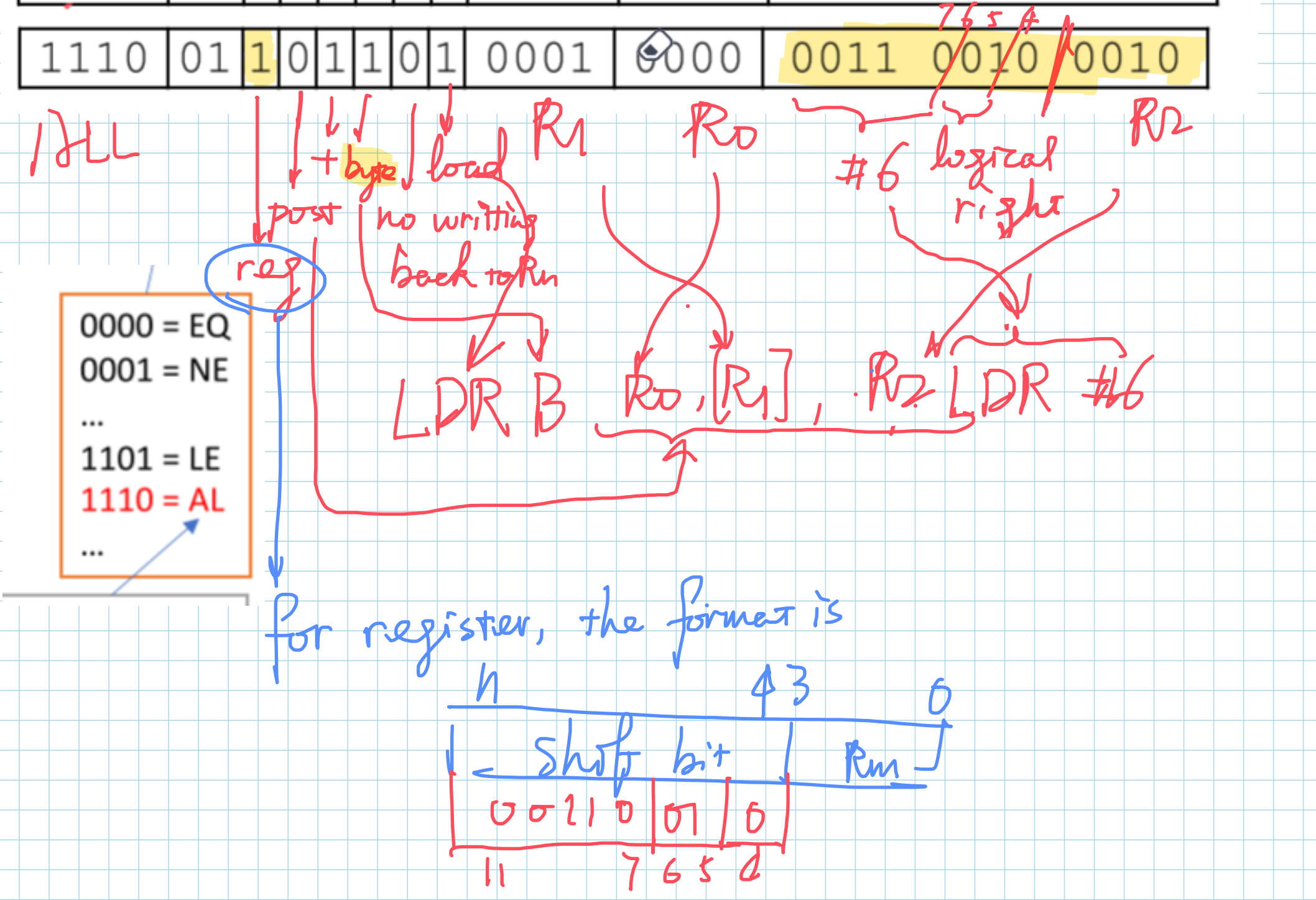
pre
wd
LDR
R1
R0
of immediate value
#1
imm
Op2
update Rn

LDR[] R0, [R1, #1]!

No B

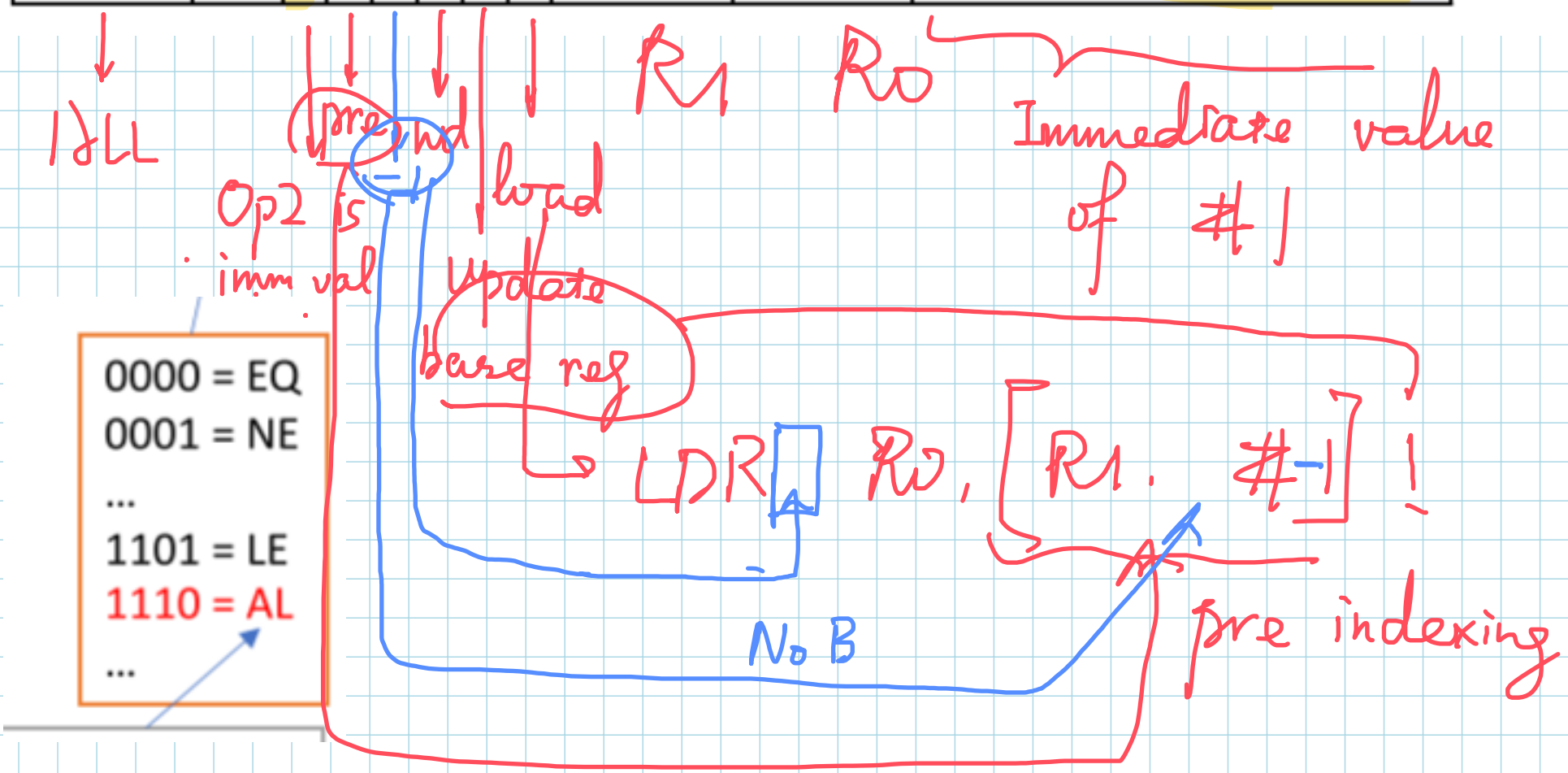
3.

Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1110	01	1	0	1	1	0	1	0001	0000	0011 0010 0010



4.

Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1110	01	0	1	0	0	1	1	0001	0000	0000 0000 0001



Answers

Step	Instruction
`L` is 0	`STR` …
`B` is 0, transfer a whole word.	`STR` …
`Cond` is 0b1110, which means "always".	`STR` …
`P` is 1, pre-indexing	`STR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`STR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`STR R0, [R1, ??]`
`W` is 0, do not update the base address (`Rn`)	`STR R0, [R1, ??]`
`I` is 0, the operand 2 will be an immediate number.	`STR R0, [R1, #?]`
`Offset` is 1.	`STR R0, [R1, #1]`
`U` is 1, adding offset to base (offset is a positive immediate).	`STR R0, [R1, #1]`

Answers

Step	Instruction
`L` is 1.	`LDR …`
`B` is 0, transfer a whole word.	`LDR …`
`Cond` is 0b1110, which means "always".	`LDR …`
`P` is 1, pre-indexing	`LDR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`LDR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`LDR R0, [R1, ??]`
`W` is 1, update the base address (`Rn`)	`LDR R0, [R1, ??]!`
`I` is 0, the operand 2 will be an immediate number.	`LDR R0, [R1, #??]!`
`Offset` is 1.	`LDR R0, [R1, #1]!`
`U` is 1, adding offset to base (offset is a positive immediate).	`LDR R0, [R1, #1]!`

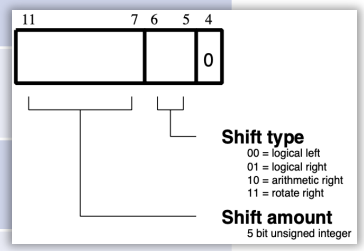
Answers

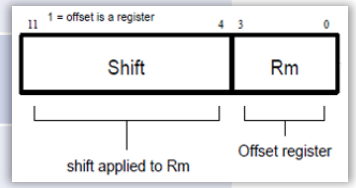
Step	Instruction
`L` is 1.	`LDR …`
`B` is 0, transfer a whole word.	`LDR …`
`Cond` is 0b1110, which means "always".	`LDR …`
`P` is 1, pre-indexing	`LDR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`LDR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`LDR R0, [R1, ??]`
`W` is 1, update the base address (`Rn`)	`LDR R0, [R1, ??]!`
`I` is 0, the operand 2 will be an immediate number.	`LDR R0, [R1, #??]!`
`Offset` is 1.	`LDR R0, [R1, #1]!`
`U` is 0, subtracting offset (offset is a negative immediate).	`LDR R0, [R1, #-1]!`

Answers

Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1110	01	1	0	1	1	0	1	0001	0000	0011 0010 0010

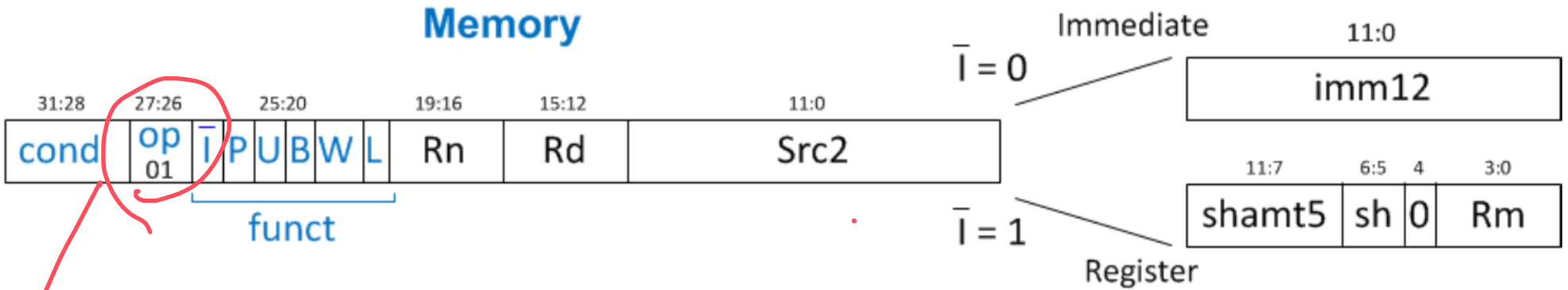
Step	Instruction
L is 1.	LDR …
B is 1, transfer a byte.	LDRB …
Cond is 0b1110, which means "always".	LDRB …
P is 0, post-indexing	LDRB Rd, [Rn], ??
Rn is 0b0001, which means R1.	LDRB Rd, [R1], ??
Rd is 0b0000, which means R0.	LDRB R0, [R1], ??
W is 0, do not update the base address. (ignored as Post-indexing always updates the base address)	LDRB R0, [R1], ??
I is 1, the operand 2 will be a register.	LDRB R0, [R1], Rm, ?????
Offset register is 0b0010 (R2), shift type is LSR (01), shift amount is 0b00110 (6)	LDRB R0, [R1], R2, LSR #6
U is 1, adding offset to base (offset is a positive immediate).	LDRB R0, [R1], R2, LSR #6





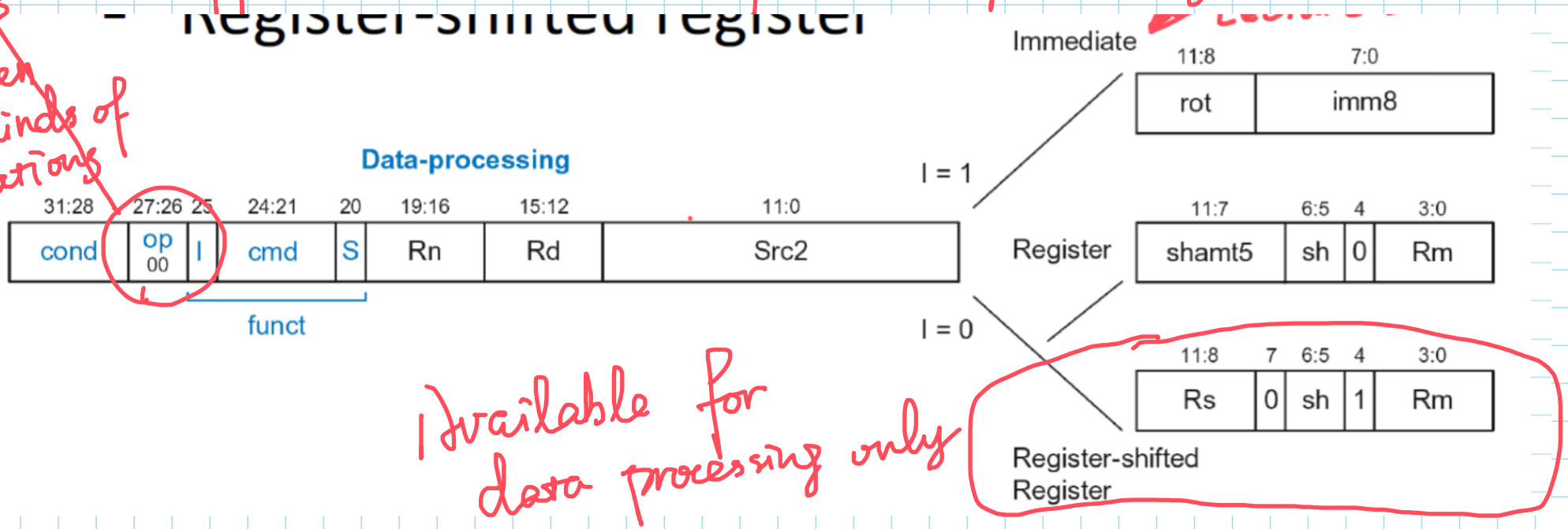
If U is 0: LDRB R0, [R1], -R2, LSR #6
This syntax is only available on Keil.

To summarize the encoding format of **storing** instructions



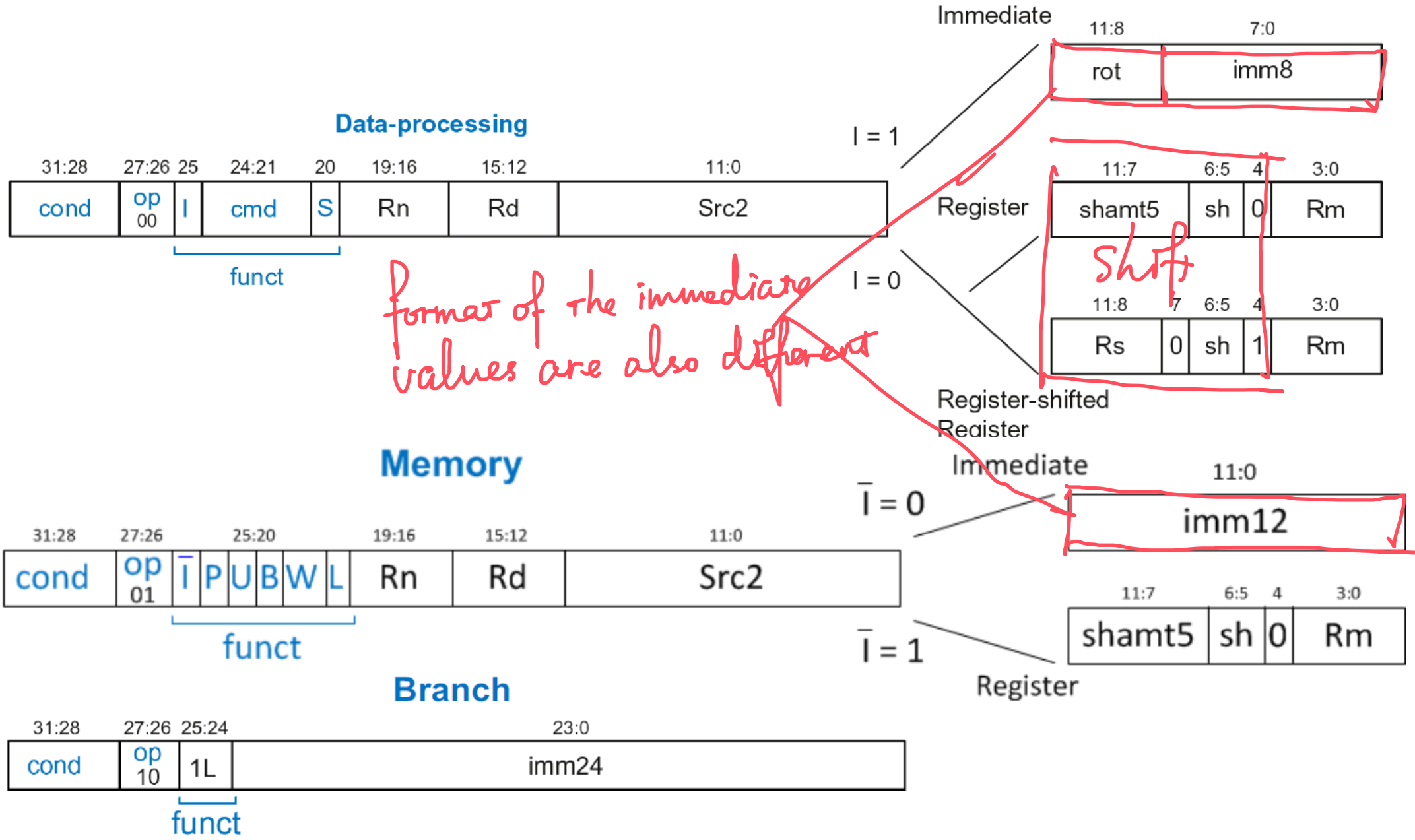
op differs between two kinds of operations

as opposed to the one of data processing instructions.



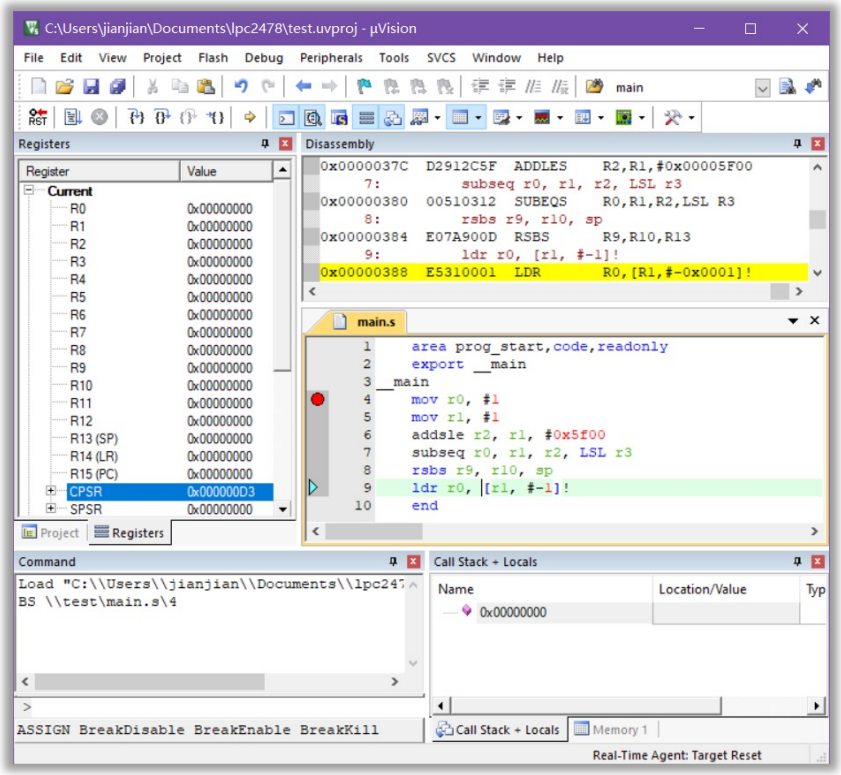
Available for data processing only

Review: Instruction Formats



More Examples

- The encodings on the right are obtained in Keil IDE.
- VisUAL might give errors.



str r0, [r1, #1]	0xE5810001
str r0, [r1, #1]!	0xE5A10001
str r0, [r1], #1	0xE4810001
ldr r0, [r1, #1]	0xE5910001
ldr r0, [r1, #1]!	0xE5B10001
ldr r0, [r1], #1	0xE4910001

str r0, [r1, r2, LSL #1]	0xE7810082
strb r0, [r1, r2, LSR #2]!	0xE7E10122
str r0, [r1], r2, ASR #3	0xE68101C2
ldrb r0, [r1, r2, ROR #4]	0xE7D10262
ldr r0, [r1, r2, LSL #5]!	0xE7B10282
ldrb r0, [r1], r2, LSR #6	0xE6D10322

Encoding: ADR

- ADR is converted into ADD or SUB.
- Symbols num1 and num2 are stored in the instruction memory.

Symbols will be converted into PC-relative values by assemblers

Memory address

value

```
        AREA    MY_PROG,CODE,READONLY
        EXPORT  __main
num1    DCD     0x1, 0x2, 0x3, 0x4
__main
        ADR     R6, num1
        ADR     R7, num2
        ADD     R5, R6, R7
        ADD     R5, R5, #0x45
num2    DCD     0x7, 0x8, 0x9, 0xA
        END
```

```
0x00000550  00000001  ANDEQ   R0,R0,R1
0x00000554  00000002  ANDEQ   R0,R0,R2
0x00000558  00000003  ANDEQ   R0,R0,R3
0x0000055C  00000004  ANDEQ   R0,R0,R4
     5:         ADR    R6, num1
0x00000560  E24F6018  SUB     R6,PC,#0x00000018
     6:         ADR    R7, num2
0x00000564  E28F7004  ADD     R7,PC,#0x00000004
     7:         ADD    R5, R6, R7
0x00000568  E0865007  ADD     R5,R6,R7
     8:         ADD    R5, R5, #0x45
0x0000056C  E2855045  ADD     R5,R5,#0x00000045
0x00000570  00000007  ANDEQ   R0,R0,R7
0x00000574  00000008  ANDEQ   R0,R0,R8
0x00000578  00000009  ANDEQ   R0,R0,R9
0x0000057C  0000000A  ANDEQ   R0,R0,R10
```

num1

num2

0x00000568 - 0x00000018 = 0x00000550

0x0000056C + 0x00000004 = 0x00000570

If converted into instruction

Encoding: LDR Pseudo Instruction

- LDR pseudo instruction allows you to assign 32-bit constants to registers.
  - But instructions are 32-bit long already.
- The assembler stores the constant in the text segment close to the referencing instruction
- Then references the value using (usually) PC-relative addressing.

```
        AREA prog,CODE,READONLY
        EXPORT __main
num1    DCD  0x1, 0x2, 0x3, 0x4
__main
        LDR  R0, =num1
        LDR  R1, =0xaabbccdd
        END
```

Memory address	Value	If converted into instruction	
0x00000374	00000001	ANDEQ	R0,R0,R1
0x00000378	00000002	ANDEQ	R0,R0,R2
0x0000037C	00000003	ANDEQ	R0,R0,R3
0x00000380	00000004	ANDEQ	R0,R0,R4
5:		LDR R0,	=num1
0x00000384	E59F0000	LDR	R0,[PC]
6:		LDR R1,	=0xaabbccdd
0x00000388	E59F1000	LDR	R1,[PC]
0x0000038C	00000374	ANDEQ	R0,R0,R4,ROR R3
0x00000390	AABBCCDD	BGE	0xFEEF370C

Encoding: LDR Pseudo Instruction

- When the PC register is involved in instructions like `LDR` and `MOV`, its value is the address of the next instruction + 4.
- For instruction: LDR R0, [PC] (= current + 8)
  - The address of the next instruction: 0x388
  - 0x388 + 0x4 = 0x38C, which points to the value 0x374 (address of num1)
- This design is to preserve compatibility for programs written for early ARM processors.

```
        AREA prog,CODE,READONLY
        EXPORT __main
num1    DCD  0x1, 0x2, 0x3, 0x4
__main
        LDR  R0, =num1
        LDR  R1, =0xaabbccdd
        END
```

Memory address	Value	If converted into instruction	
0x00000374	00000001	ANDEQ	R0,R0,R1
0x00000378	00000002	ANDEQ	R0,R0,R2
0x0000037C	00000003	ANDEQ	R0,R0,R3
0x00000380	00000004	ANDEQ	R0,R0,R4
5:		LDR R0,	=num1
0x00000384	E59F0000	LDR	R0,[PC]
6:		LDR R1,	=0xaabbccdd
0x00000388	E59F1000	LDR	R1,[PC]
0x0000038C	00000374	ANDEQ	R0,R0,R4,ROR R3
0x00000390	AABBCCDD	BGE	0xFEEF370C

Assembler Directives

Common directives

GNU Assembler VS ARM tools Assembler

- There are a few differences between the GNU assembler and the ARM assembler.
  - Comments: @ and ;
  - Labels: GNU uses colon (:)
  - Directives: GNU starts with a period (.)
  - Different set of directives supported.

```
        .text
entry:  b start                 @ Skip over the data
arr:    .byte 10, 20, 25        @ Read-only array of bytes
eoa:                            @ Address of end of array + 1

        .align
start:
        ldr   r0, =eoa          @ r0 = &eoa
        ldr   r1, =arr          @ r1 = &arr
        mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r1], #1      @ r2 = *r1++
        add   r3, r2, r3        @ r3 += r2
        cmp   r1, r0            @ if (r1 != r
        bne   loop              @    goto loop
stop:   b stop
```

GNU Assembler

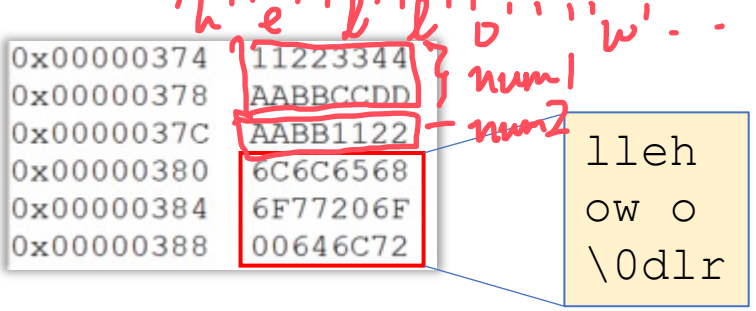
ARM tools Assembler

```
        AREA    MY_PROG,CODE,READONLY
num1    DCD 0x1, 0x2, 0x3, 0x4
        EXPORT __main
__main
        MOV R2, PC          ; step 1
        LDR R0, =num1       ; pc-relative
        LDR R1, =0xaabbccdd
        END
```

Assembler Directives

- Assembler directives tell the assembler to do something.
- Define constant directives:
  - DCB: byte sized data
  - DCW: half-word sized data
  - DCD: word sized data

```
num1    DCD 0x11223344, 0xAABBCCDD
num2    DCW 0x1122, 0xAABB
msg     DCB "hello world",0
```



- The EQU directive lets you assign names to address or data values.

```
twelve  EQU 0x12
        LDR R3, =twelve
```

- END is used to denote the end of the assembly language source program

<=> Entry: denote the start of the program (optional)

Assembler Directives

- The `SPACE` directive reserves a zeroed block of memory.
- The `FILL` directive reserves a block of memory to fill with a given value.

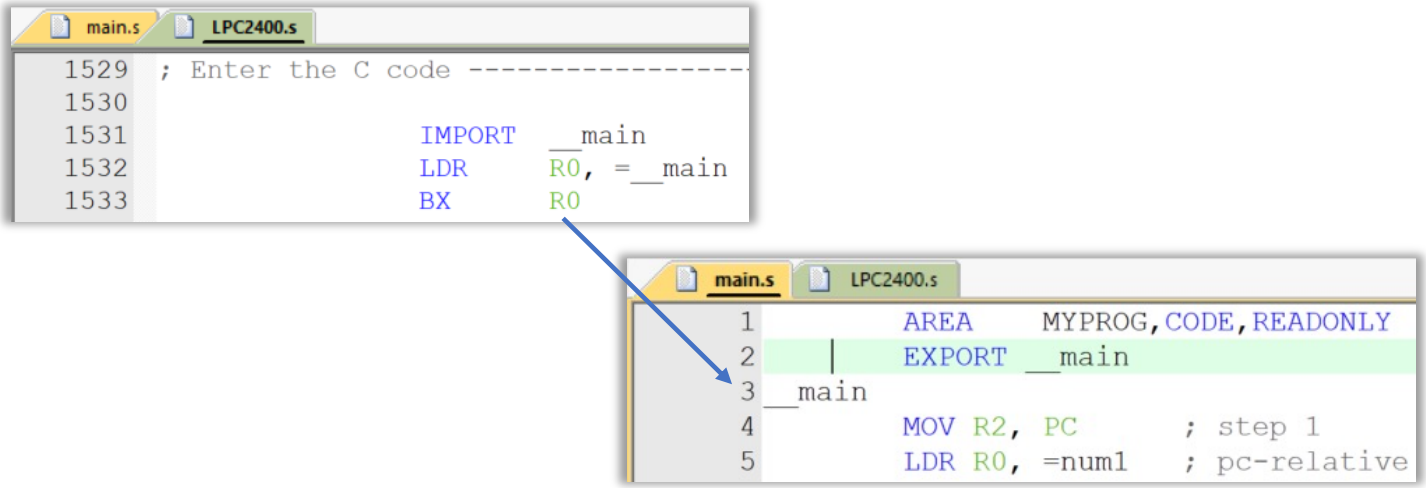
```
label SPACE expr
label FILL expr{,value{,valuesize}}
```

  - `expr`: number of bytes reserved
  - `value`: the value to fill the reserved bytes.
  - `valuesize`: is the size, in bytes, of value. It can be any of 1, 2, or 4. valuesize is optional and if omitted, it is 1.

```
; defines 255 bytes of zeroed store
data1  SPACE 255
; defines 50 bytes containing 0xAB
data2  FILL  50,0xAB,1
```

Assembler Directives

- `EXPORT`: gives code in other files access to symbols in the current file
- `IMPORT`: provide the assembler with a name that is not defined in the current assembly file.



ELF Sections and the AREA Directive

- AREA: instructs the assembler to assemble a new code or data section.
- The example below defines two AREAs.
  - MyData: stores data and is read-write accessible.
  - MyCode: stores instructions and is read-only.

```
      AREA    MyData,DATA,READWRITE
num2  DCD     0x11223344

      AREA    MyCode,CODE,READONLY
num1  DCD     0xAABBCCDD
      EXPORT  __main
__main
      LDR     R1, =num1
      MOV     R0, #65
      STR     R0, [R1]
      LDR     R1, =num2
      STR     R0, [R1]
      END
```

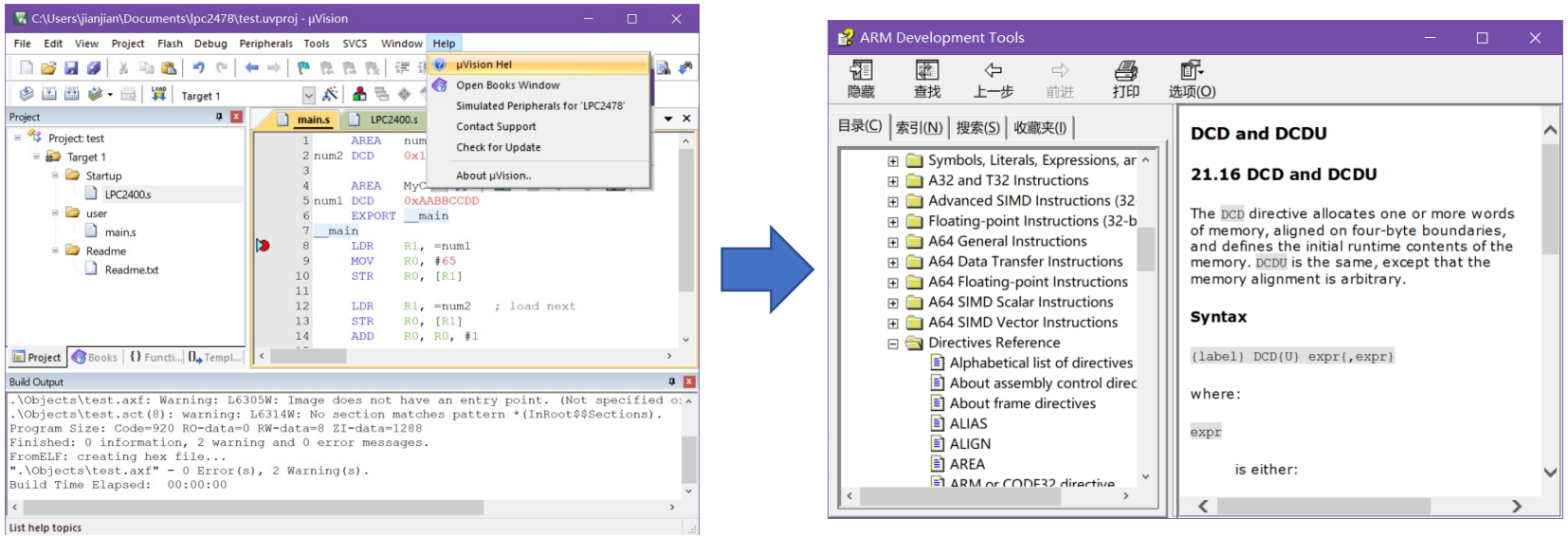
ADR cannot be used to get the value associated with a symbol in another AREA.

Fails because the memory is read-only

Works because the memory can be written

Extended Reading

- For assembler directives of the ARM tool assembler, read the help file of the Keil IDE: (menu->help->uvision hel)



- For assembler directives of the GNU toolchain: Read http://bravegnu.org/gnu-eprog/index.html.