

Data Transfer Instructions

Computer Engineering 1

**CT Team: A. Gieriet, J. Gruber, B. Koch, M. Loeser, M. Meli,
M. Rosenthal, M. Ostertag, A. Rüst, J. Scheier, T. Welti**

■ How do I "move" data?



Type	Distribution
Data transfers	43%
Control flow	23%
Arithmetic	15%
Compare	13%
Logic	5%
Others	1%

- CPU → CPU register to register
- memory/IO → CPU load from memory/IO
- CPU → memory/IO store to memory/IO

- **Data Transfers**
- **Register to Register**
- **Loading Literals**
- **Loading Data**
- **Storing Data**
- **Loading/Storing Multiple Registers**
- **The C Perspective**
 - Arrays
 - Pointers

For information on individual instructions covered in these slides:
See also Quick Reference Card for Thumb 16-bit Instruction Set

At the end of this lesson you will be able

- to enumerate the 4 transfer types of the Cortex-M0
- to read a Cortex-M0 assembly program with data transfer instructions
- to write assembly programs with the major Cortex-M0 data transfer instructions
- to encode and decode data transfer instructions to/from binary machine code
- to apply the EQU assembler directive
- to explain the concept of a 'literal pool' and to apply the 'LDR Rd,=literal' pseudo instruction
- to understand PC relative and indirect addressing (including offsets)
- to explain how arrays are stored in memory and how array elements can be accessed
- to understand how a compiler translates array accesses in a C-program to assembly
- to explain how a C-compiler implements pointers and address operators in assembly language

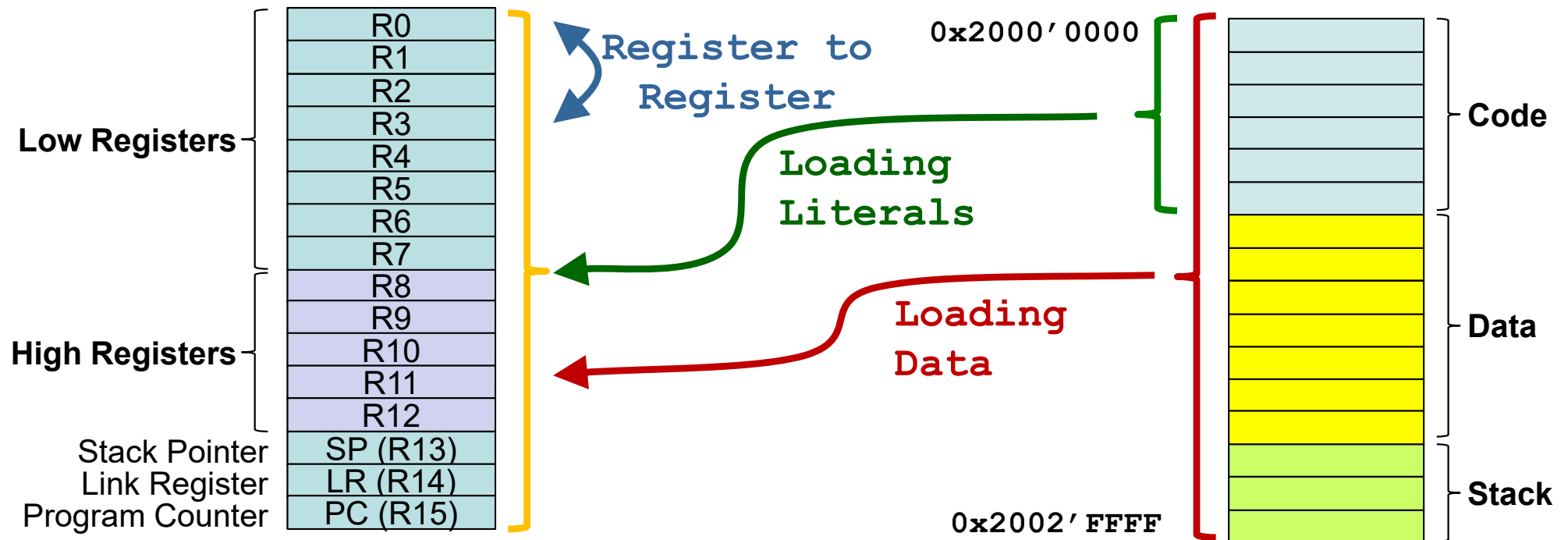
■ Load/Store Architecture (ARM Cortex-M)

- Memory accessed only with load/store operations
- Usual steps for data processing
 - **Load** operands from memory to register
 - Execute operation → result in register
 - **Store** result from register to memory

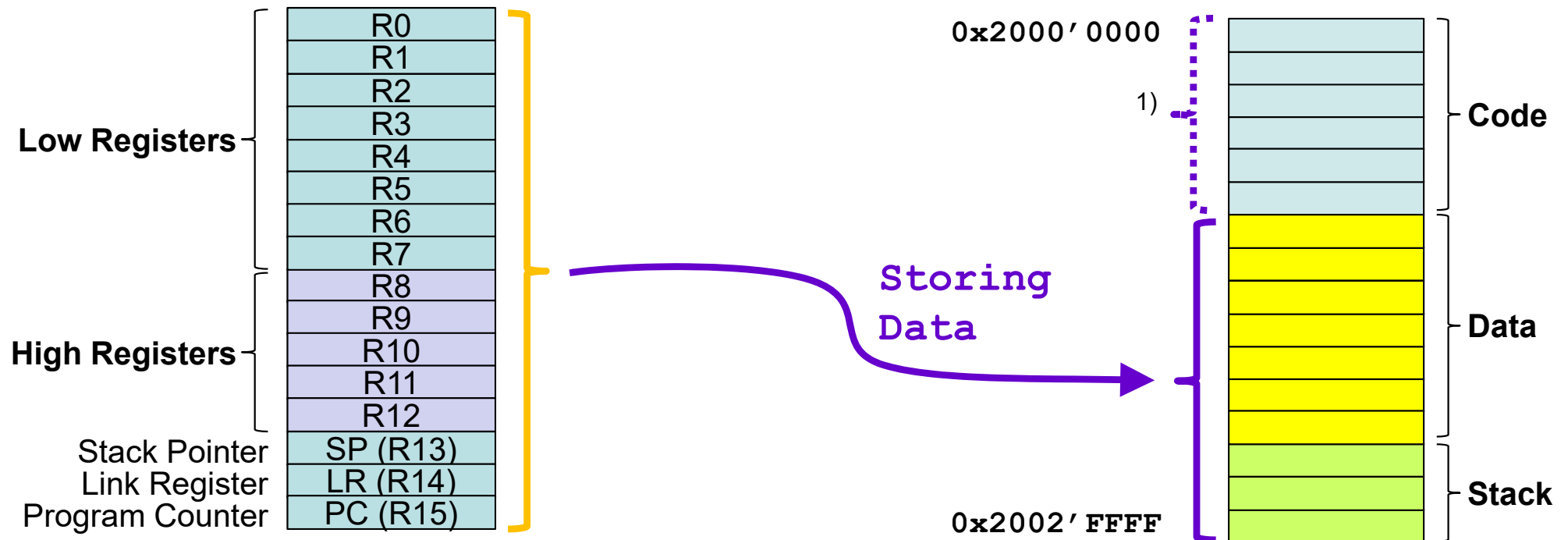
■ Register Memory Architecture e.g. Intel x86

- One of the operands can be located in memory
- Result can be directly written to memory

■ Transfer Types



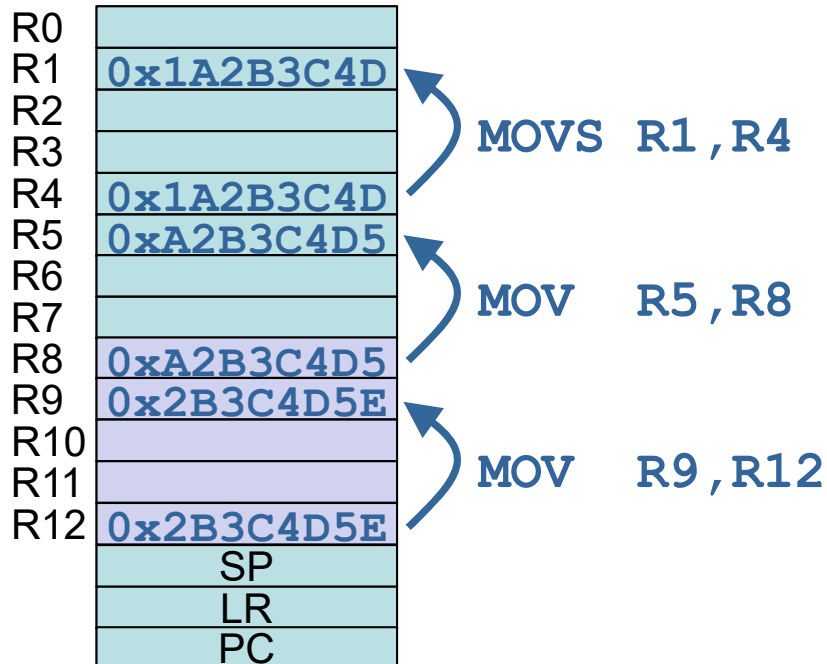
■ Transfer Types (continued)



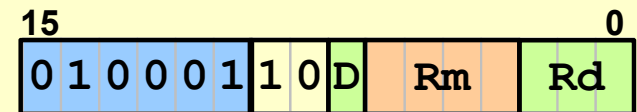
¹⁾ Code region is typically read-only

■ MOV / MOVS (register)

- Copy register value to other register
- MOV → low and high registers
- MOV**S** → only low registers
S = update of flags ³⁾

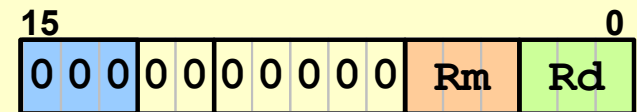


MOV <Rd>, <Rm> ¹⁾ T1



RD = D : Rd → RD = Rm

MOVS <Rd>, <Rm> ²⁾ T2



Rd = Rm

3-bit → low registers only

1) Instruction group "special data processing"
see table in lecture 2

2) Instruction group "shift by immediate, move register"
see table in lecture 2

3) 'S' stands for status register

■ Examples MOV / MOVS

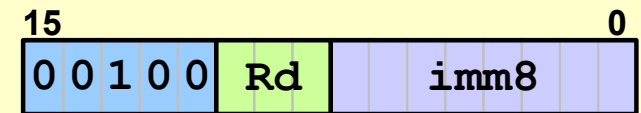
- Copy value of register Rm to Rd
 - MOV <Rd>, <Rm> high and low registers allowed
 - MOVS <Rd>, <Rm> restricted to low registers

address	opcode	instruction		comment
00000002	4621	MOV	R1, R4	; low reg to low reg
00000004	4641	MOV	R1, R8	; high reg to low reg
00000006	4688	MOV	R8, R1	; low reg to high reg
00000008	46C8	MOV	R8, R9	; high reg to high reg
0000000A	0021	MOVS	R1, R4	; low reg to low reg
		;MOVS	R1, R8	; not possible: high reg
		;MOVS	R8, R1	; not possible: high reg
		;MOVS	R8, R9	; not possible: high reg

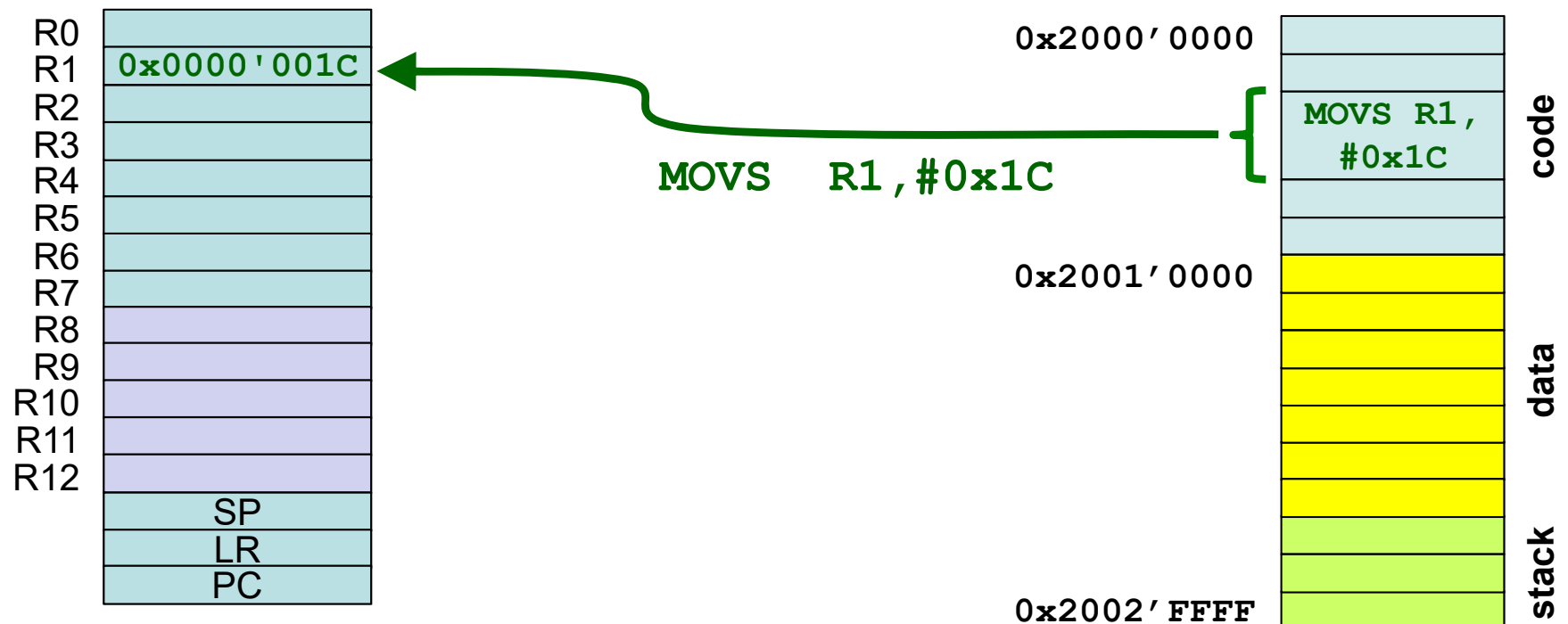
■ MOVS (immediate data)

- Copy immediate 8-bit value (literal) to register (only low registers)
- 8-bit literal is part of opcode (imm8)
- Register-bits 31 to 8 set to 0

MOVS <Rd>, #<imm8>



Rd = <imm8>



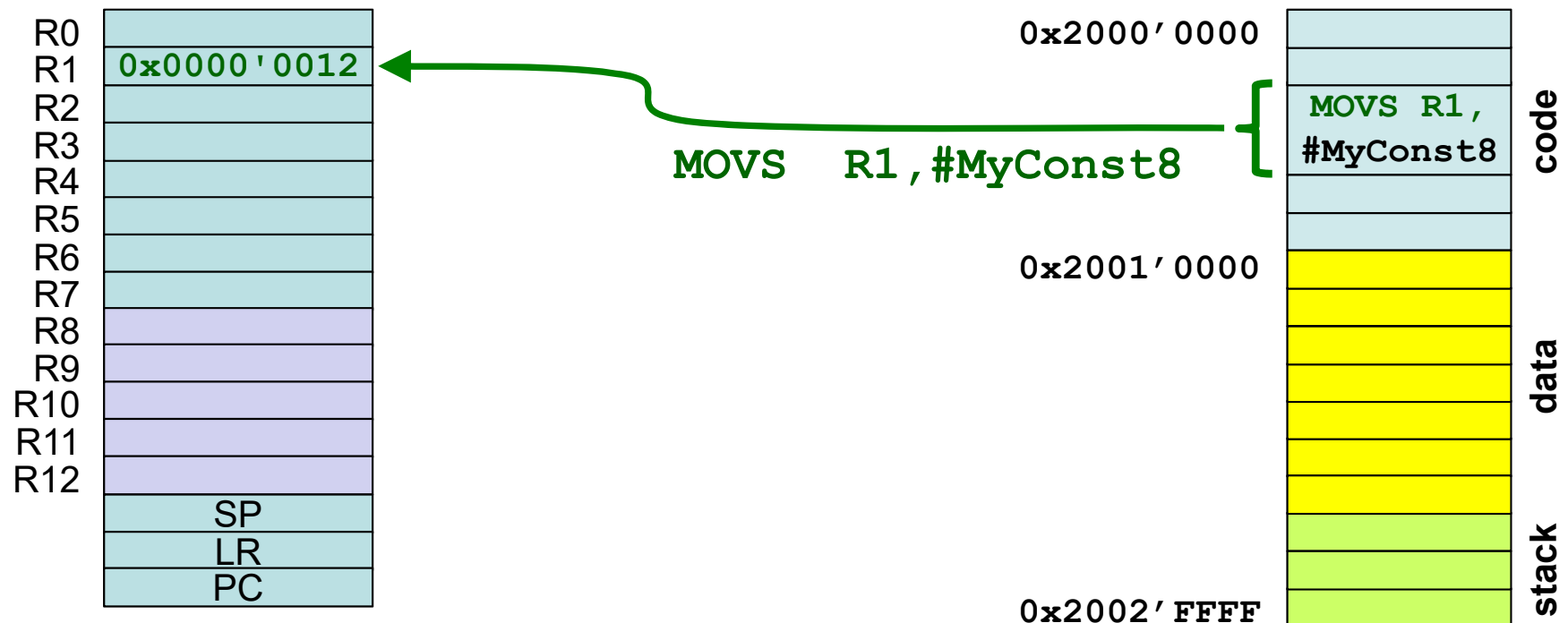
■ EQU - Assembler Directive

- Symbolic definition of literals and constants
- Comparable to `#define` in C

Example:

MOVS with symbolic definition of literal

```
MY_CONST8 EQU 0x12
          MOVS R1, #MY_CONST8
```



■ Example MOVS (immediate data)

- Immediate 8-bit → 0 to 255d

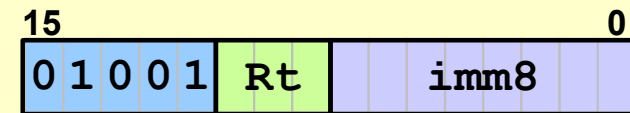
MY_CONST8	EQU	0xCD		; assembler directive
				; does not generate opcode
0000000C 21FF	MOVS	R1, #0xFF		; immediate hex to low reg
0000000E 240C	MOVS	R4, #12		; immediate dec to low reg
00000010 27CD	MOVS	R7, #MY_CONST8		; literal with symbolic name
	;MOVS	R8, #MY_CONST8		high reg not possible
	;MOVS	R1, #0x100		immediate out of range

Loading Literals

■ Load - LDR (literal)

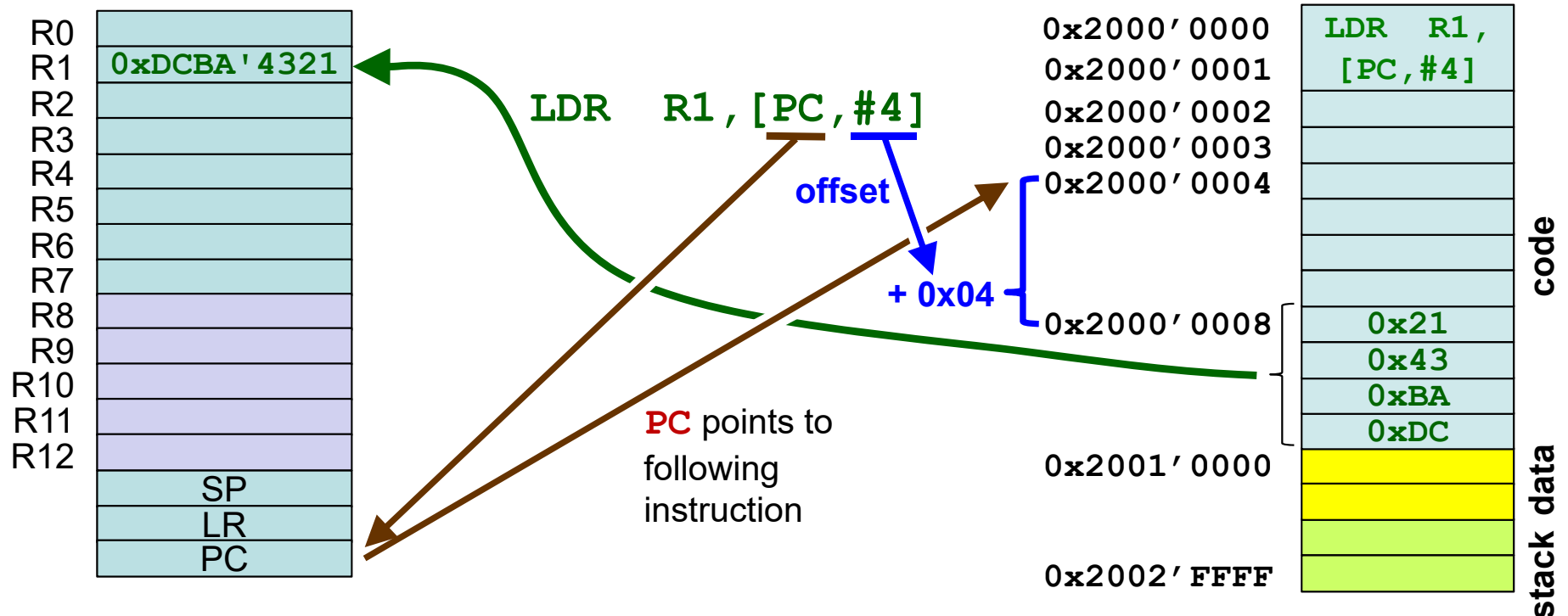
- Indirect access relative to PC ¹⁾
- PC offset <imm>
- If PC not word-aligned
 - align on next upper word-address

LDR <Rt>, [PC, #<imm>]



<imm> = imm8:00

Rt = Mem[@(PC + <imm>)]



¹⁾ offsets are positive multiples of 4 in the range of 0 to 0x3FC, i.e. 1020d

■ Example LDR (literal)

- Offset in bytes vs. imm8 in words \rightarrow $\langle \text{imm} \rangle = \text{imm8} : 00$ ¹⁾
 - Line1: assembler converts $\langle \text{imm} \rangle$ 0x08 to **imm8** = 0x02
- Literals on lines 7 and 8
- PC
 - points to following instruction
 - if not word-aligned \rightarrow align on next upper word-address

1	00000014	4902	ldr_lit	LDR	R1,[PC,#0x08]	; hex offset
2	00000016	4A03		LDR	R2,[PC,#12]	; dec offset
3	00000018	4B01		LDR	R3,myLit	
4	0000001A	4C01		LDR	R4,myLit	
5	0000001C	4D00		LDR	R5,myLit	
6	0000001E	E003		B	ldr_lit2	
7	00000020	12345678	myLit	DCD	0x12345678	
8	00000024	9ABCDEF0		DCD	0x9ABCDEF0	

Pseudo instruction:
Assembler converts to
LDR R3,[PC,#0x04]

¹⁾ offset = imm8 << 2 i.e. imm8 * 4

■ Examples LDR (literal)

same code as previous slide

- Which values are being loaded into registers R1 to R5?

```
1 00000014 4902      ldr_lit LDR    R1,[PC,#0x08] ; hex offset
2 00000016 4A03      LDR    R2,[PC,#12]   ; dec offset
3 00000018 4B01      LDR    R3,myLit
4 0000001A 4C01      LDR    R4,myLit
5 0000001C 4D00      LDR    R5,myLit
6 0000001E E003      B      ldr_lit2
7 00000020 12345678 myLit DCD    0x12345678
8 00000024 9ABCDEF0      DCD    0x9ABCDEF0
```

Use a symbol: Let assembler do the calculation

Branch prevents interpretation of literals as instructions

Line 1

PC = 0x00000016 --> word_aligned 0x00000018

Load literal from 0x00000018 + 0x08 = 0x00000020

R1 = 0x12345678

Line 2

PC = 0x00000018 --> word_aligned 0x00000018

Load literal from 0x00000018 + 12d = 0x00000024

R2 = 0x9ABCDEF0

¹⁾ Word alignment of PC causes the same offset in lines 3 and 4, although they are accessing the same literal

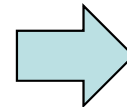
■ Pseudo Instruction

```
LDR Rd, =literal
```

- Assembler
 - creates 'literal pool' at convenient code location
 - allocates and initializes memory in 'literal pool' → DCD
 - uses LDR (literal) instruction and calculates offset ¹⁾

assembly code as written
by human programmer

```
LDR R1, =0x20000012
```



code generated
by assembler (tool)

```
LDR R1, [PC, #68]
```

...

DCD

...

DCD

0x20000012

DCD

...

'Literal Pool' at end of code block

'literal pool' → group of literals

¹⁾ on M3/M4 the assembler may use a MOV instruction with an immediate value instead

■ Pseudo Instruction Examples

- Warning: difference between lines 7 and 8

1	CONST_A	EQU	0x000000AA
2	CONST_B	EQU	0xBBCCDDEE
3			
4	0000003C 4F03	LDR	R7,=0x12
5	0000003E 4F04	LDR	R7,=CONST_A
6	00000040 4F04	LDR	R7,=CONST_B
7	00000042 4D01	LDR	R5,mylita
8	00000044 4D04	LDR	R5,=mylita
9	00000046 E02F	B	wherever
10			
11	00000048 FF001122 mylita	DCD	0xFF001122
12			
13	0000004C 00000012		
14	00000050 000000AA		
15	00000054 BBCCDDEE		
16	00000058 00000000		

Literal Pool

load 0xFF001122 from address 0x00000048

load address of mylita from address 0x00000058

CONST_A and CONST_B

space where the address of mylita will be stored after linking

■ Pseudo Instruction Examples

LDR R5,mylita → LDR R5,[PC,#...]

The **value** 0xFF001122 at label **mylita** is loaded into R5

LDR R5,=0x20003000 → LDR R5,[PC,#...]

Space is allocated in literal pool. The **value** 0x20003000 is stored into this location and loaded into R5 from there.

LDR R5,=CONST_A → LDR R5,[PC,#...]

Space is allocated in literal pool. The **value** 0x000000AA, defined through **EQU**, is stored into this location and loaded into R5 from there.

LDR R5,=mylita → LDR R5,[PC,#...]

Space is allocated in literal pool. The address of **mylita** is stored into this location and loaded into R5 from there.

```
CONST_A EQU 0x000000AA

AREA example, CODE, ...

Start

mylita B wherever
      DCD 0xFF001122
```

Literal pool will be created here

= sign tells the compiler to allocate and initialize space in literal pool. → Programmer does not have to type DCD lines.

■ C Example

C-Code

```
static uint32_t g;  
  
void lit_example(void) {  
    uint32_t a;  
    uint32_t b;  
    const uint32_t c = 0x04;  
    uint32_t *p;  
  
    a = 0x05;  
    b = 0xABCDEF12; 1)  
    p = &g;  
    ...  
}
```

■ C Example

C-Code

```
static uint32_t g;  
  
void lit_example(void) {  
    uint32_t a;  
    uint32_t b;  
    const uint32_t c = 0x04;  
    uint32_t *p;  
  
    a = 0x05;  
    b = 0xABCDEF12; 1)  
    p = &g;  
    ...  
}
```

Compiler assigns variables to registers	a → R0 p → R2 b → R1 c → R3
---	------------------------------------

Assembly

```
AREA MyCode, CODE, ...  
...  
000002    2304    MOVS    R3,#4  
000004    2005    MOVS    R0,#5  
000006    4904    LDR     R1,lit1  
000008    4a04    LDR     R2,adr_g 2)  
...  
000018    lit1    DCD     0xabcdef12  
00001C    adr_g   DCD     g 3)  
  
AREA MyData, DATA, ...  
g         DCD     0x00000000
```

4904	→	LDR R1,[PC,#0x10]
4a04	→	LDR R2,[PC,#0x10]

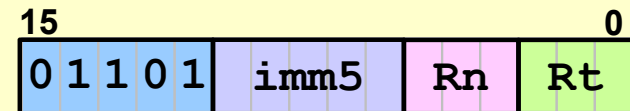
- 1) Compiler translates `b = 0xABCDEF12;` to `LDR R1,lit1`
Assembler then translates `LDR R1,lit1` to `LDR R1,[PC,#0x10]`
- 2) loads the address of `g`
- 3) DCD stores the address of `g` (not the content of `g`)
Compiler does not use the pseudo-instruction `LDR R2,=g`, which would mean the same.

Loading Data

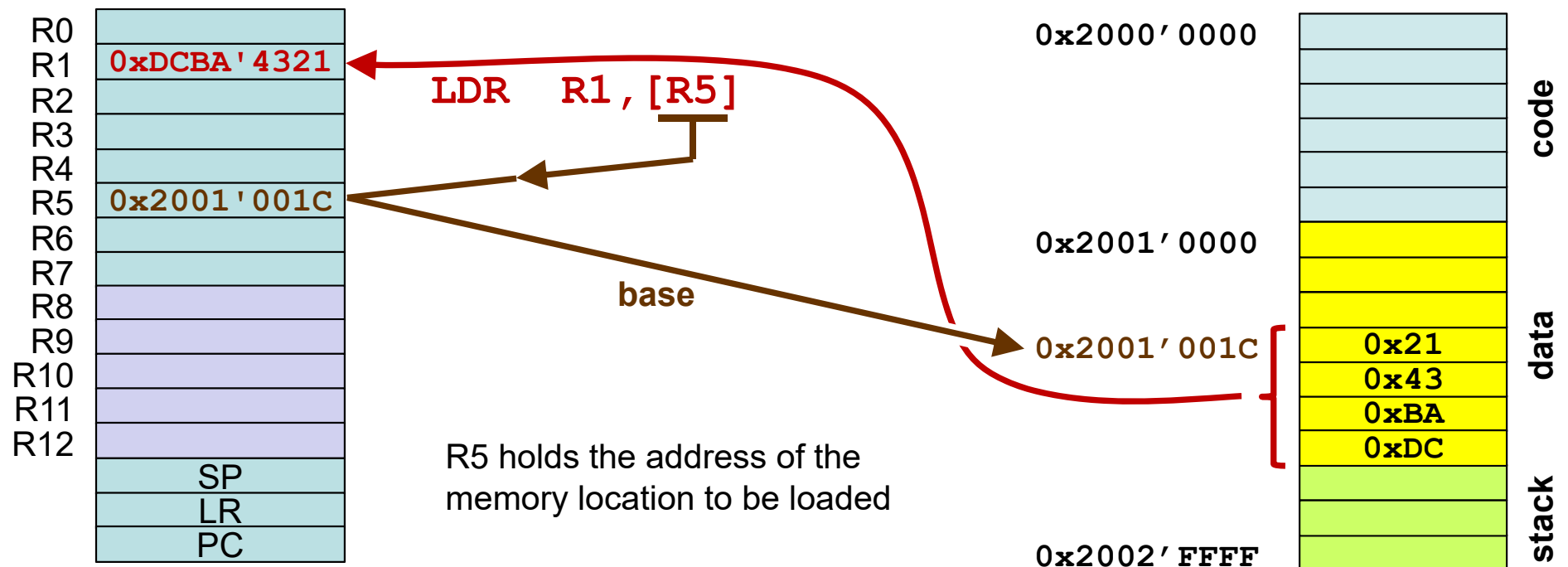
■ LDR (immediate offset) – $\text{imm5} = 0$ ¹⁾

- Indirect addressing \rightarrow [...]
 - $R_n \rightarrow$ memory address
- Only low registers

LDR <Rt>, [<Rn>]

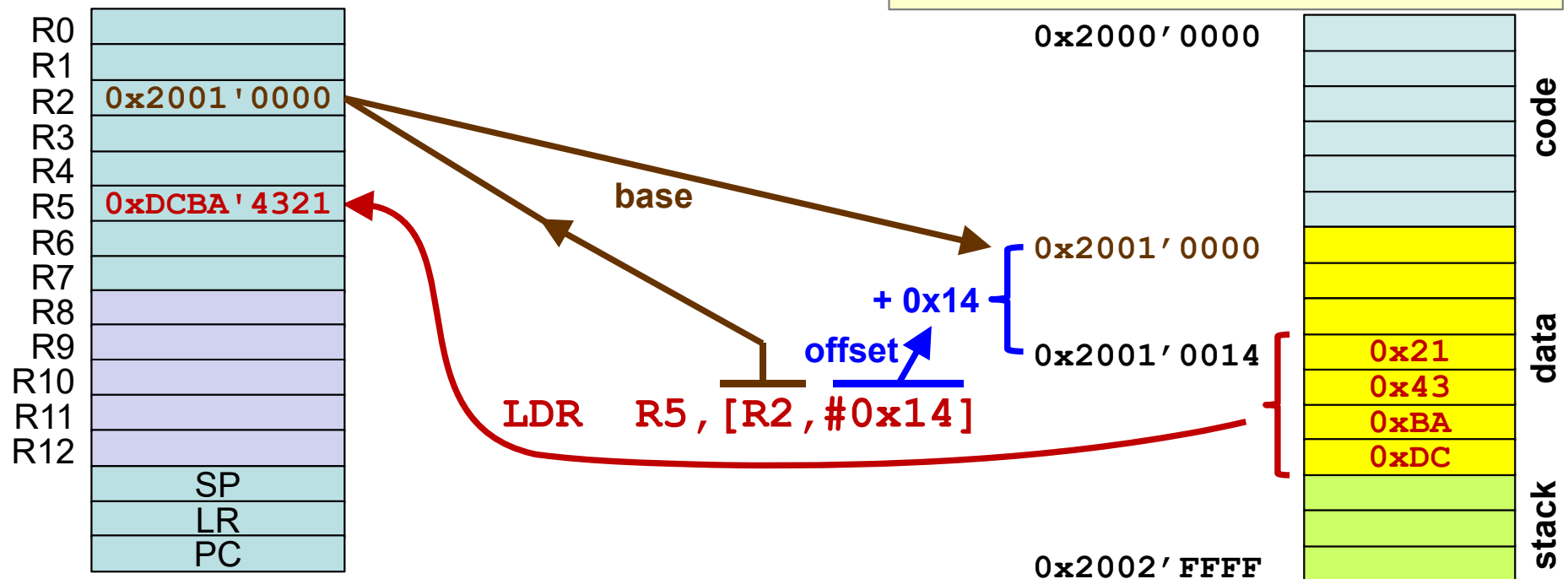
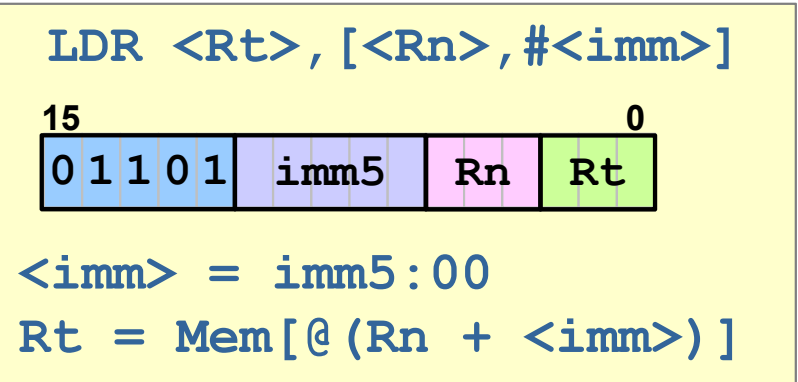


$R_t = \text{Mem}[@R_n]$



■ LDR (immediate offset) – general

- Indirect addressing
 - Immediate offset <imm>
 - Offset range 0 - 124d (0x7C) ¹⁾
- Only low registers

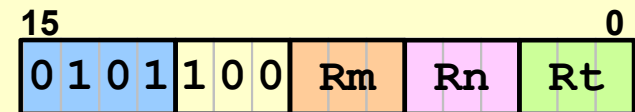


¹⁾ positive values, multiples of 4 only

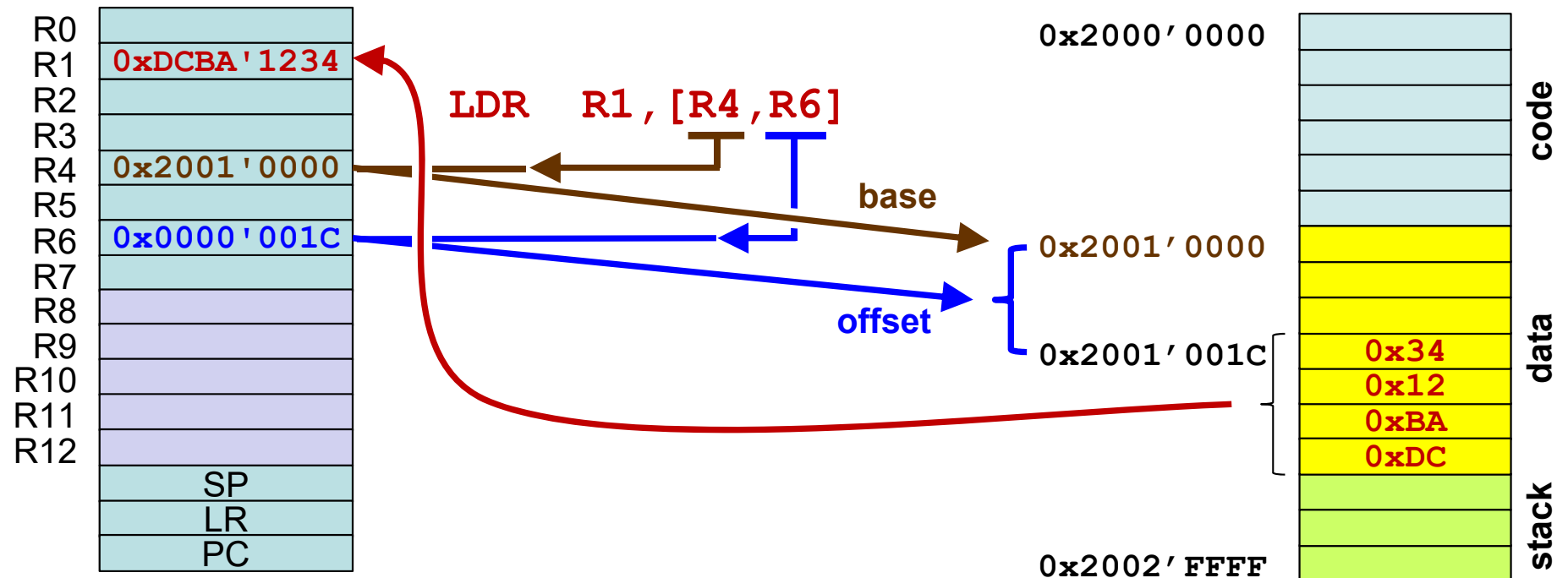
■ LDR (register offset)

- Indirect addressing with offset register
 - offset = unsigned
- Only low registers

LDR <Rt>, [<Rn>, <Rm>]



$Rt = Mem[@(Rn + Rm)]$



■ Examples

- Content of R4, R5, R6 after execution?

```
                AREA    my_data, DATA, READWRITE
00000000 11223344 my_array    DCD    0x11223344
00000004 55667788          DCD    0x55667788
00000008 99AABBCC          DCD    0x99AABBCC
```

```
                AREA    myCode, CODE, READONLY
                . . .

                ; load base and offset registers
0000007C 4906    LDR     R1,=my_array    ; load address of array
0000007E 4B07    LDR     R3,=0x08

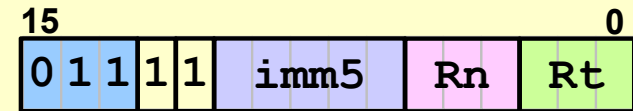
                ; indirect addressing
00000080 680C    LDR     R4, [R1]          ; base R1
00000082 684D    LDR     R5, [R1, #0x04] ; base R1, immediate offset
00000084 58CE    LDR     R6, [R1, R3]    ; base R1, offset R3
```

Not content of my_array, but address of my_array

■ LDRB (register/immediate offset)

- Load Register Byte
- Register bits 31 to 8 set to zero

LDRB <Rt>, [<Rn>, #<imm>]



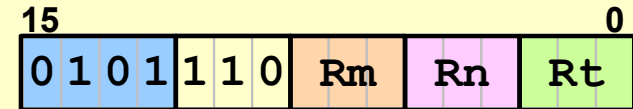
<imm> = imm5

Rt = Byte[@(Rn+<imm>)]

■ LDRH (register/immediate offset)

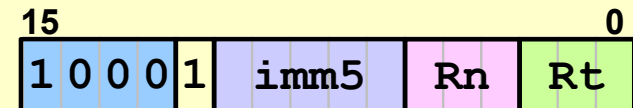
- Load Register Half-word
- Register bits 31 to 16 set to zero

LDRB <Rt>, [<Rn>, <Rm>]



Rt = Byte[@(Rn+Rm)]

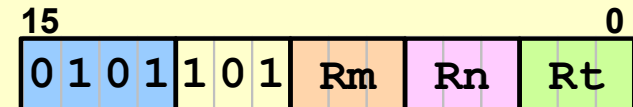
LDRH <Rt>, [<Rn>, #<imm>]



<imm> = imm5:0

Rt = Hw[@(Rn+imm5)]

LDRH <Rt>, [<Rn>, <Rm>]

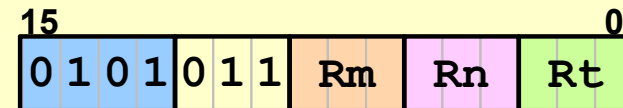


Rt = Hw[@(Rn+Rm)]

■ LDRSB

- Load Register Signed Byte
- Sign extend
 - Register bits 31 to 8 set or reset depending on bit 7

LDRSB <Rt>, [<Rn>, <Rm>]

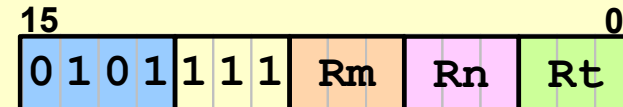


$Rt = \text{sign_extend}(\text{Byte}[\text{@}(\text{Rn} + \text{Rm})])$

■ LDRSH

- Load Register Signed Half-word
- Sign extend
 - Register bits 31 to 16 set or reset depending on bit 15

LDRSH <Rt>, [<Rn>, <Rm>]



$Rt = \text{sign_extend}(\text{Hw}[\text{@}(\text{Rn} + \text{Rm})])$

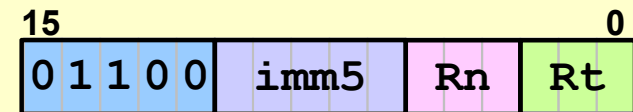
■ Sign Extension

- See slides on casting

■ STR (immediate offset)

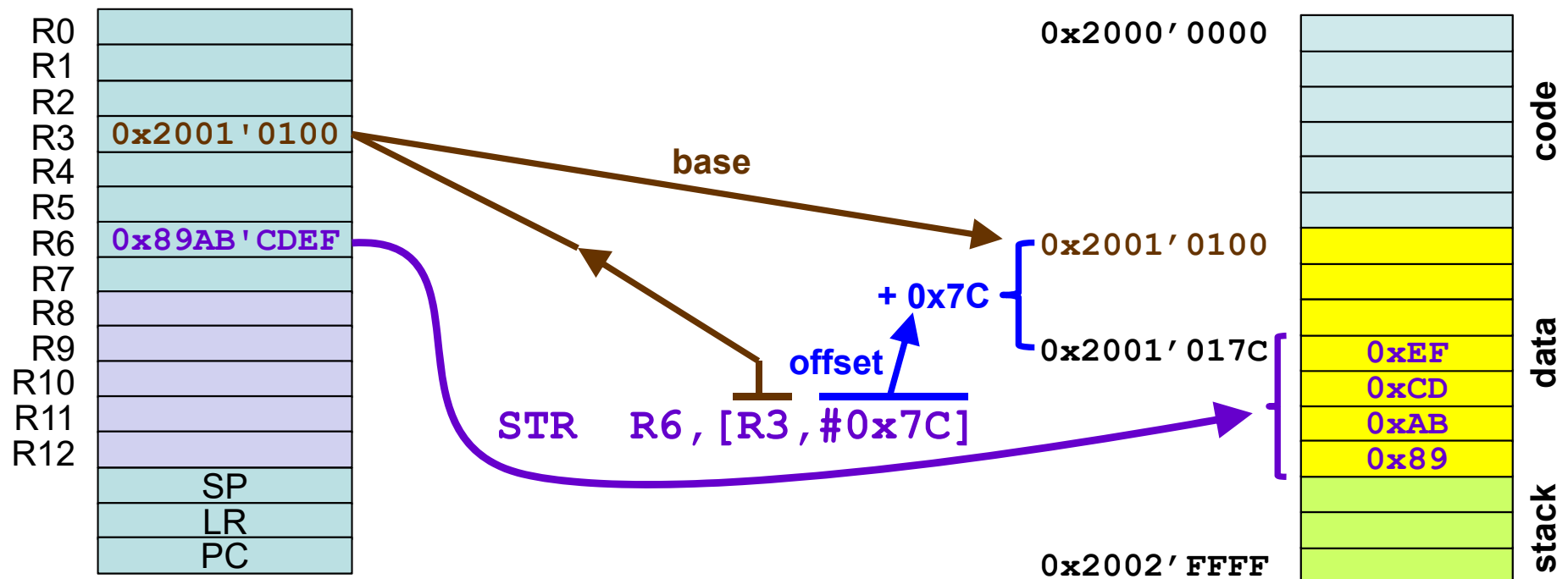
- Indirect addressing with immediate offset
 - Offset range 0 - 124d (0x7C) ¹⁾
- Only low registers

STR <Rt>, [<Rn>, #<imm>]



<imm> = imm5:00

Mem[@ (Rn + <imm>)] = Rt

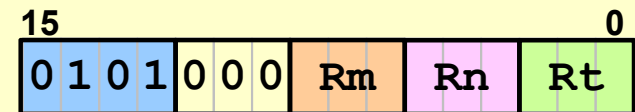


¹⁾ positive values, multiples of 4 only

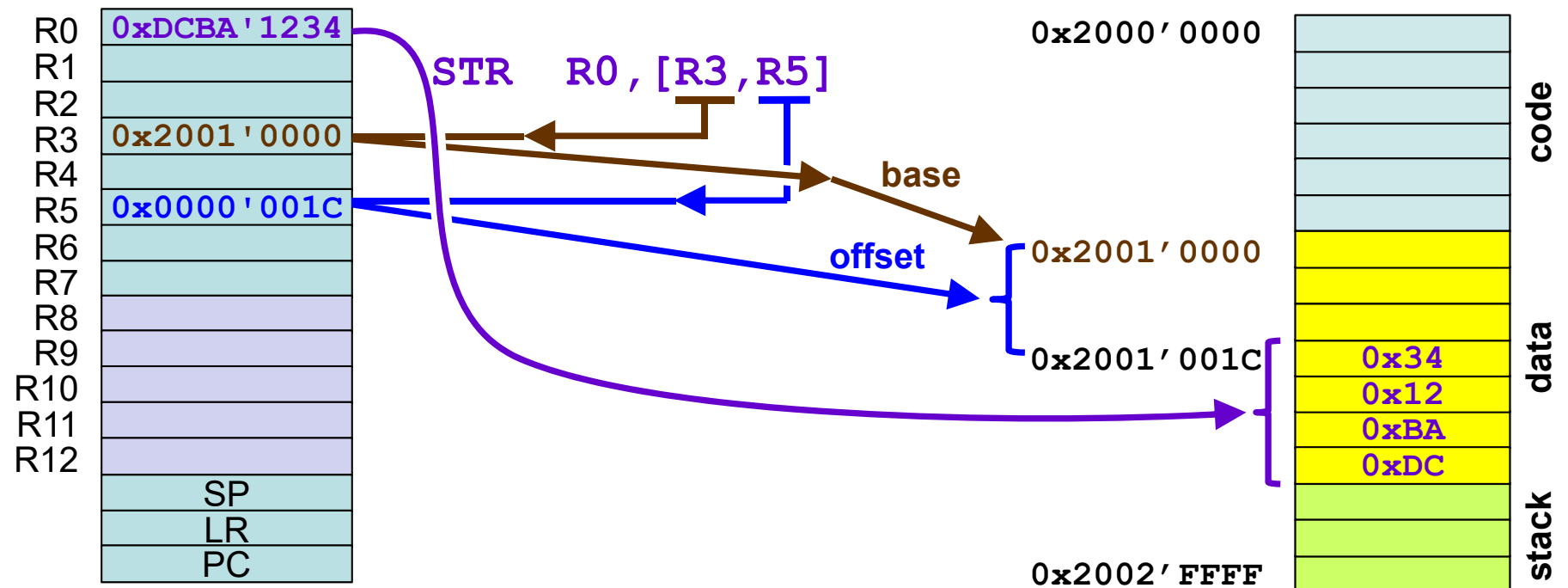
■ STR (register offset)

- Indirect addressing with offset register
 - Offset register → index
- Only low registers

STR <Rt>, [<Rn>, <Rm>]



Mem[@ (Rn+Rm)] = Rt



Storing Data

```
CONST_C EQU 0xCFFFFFFF
CONST_D EQU 0xDDDDDDDD
CONST_E EQU 0xEEEEEEEE
```

■ Example

	AREA	progData2, DATA, READWRITE	
data_array	SPACE	256	

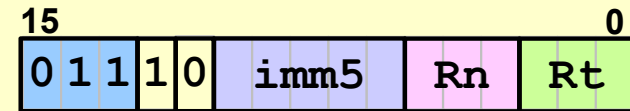
000000A4	4904	LDR	R1,=CONST_C	
000000A6	4A05	LDR	R2,=CONST_D	
000000A8	4B05	LDR	R3,=CONST_E	
000000AA	4F06	LDR	R7,=data_array	
000000AC	4E06	LDR	R6,=0x08	
000000AE	6039	STR	R1, [R7]	store CONST_C in memory at address of data_array
000000B0	607A	STR	R2, [R7, #0x04]	store CONST_D in memory at address of data_array + 0x04
000000B2	51BB	STR	R3, [R7, R6]	store CONST_E in memory at address of data_array + 0x08
000000B4	E00A	B	ldm_ex	
000000B6	0000	ALIGN	4	
000000B8	CCCCCCCC			
000000BC	DDDDDDDD			
000000C0	EEEEEEEE			
000000C4	00000000			
000000C8	00000008			storage space in literal pool for address of data_array

Storing Data

■ STRB (immediate/register offset)

- Store Register Byte
- Low 8 bits of register stored

STRB <Rt>, [<Rn>, #<imm>]



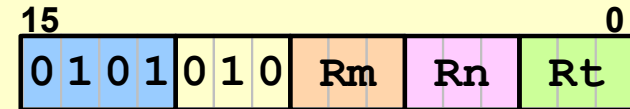
<imm> = imm5

Byte[@(Rn+<imm>)] = Rt(7:0)

■ STRH (immediate/register offset)

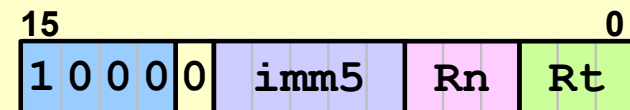
- Store Register Half-word
- Low 15 bits of register stored

STRH <Rt>, [<Rn>, <Rm>]



Byte[@(Rn+Rm)] = Rt(7:0)

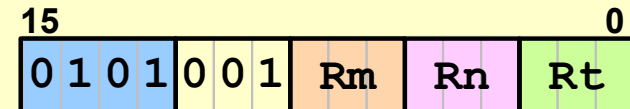
STRH <Rt>, [<Rn>, #<imm>]



<imm> = imm5:0

Hw[@(Rn+<imm>)] = Rt(15:0)

STRH <Rt>, [<Rn>, <Rm>]



Hw[@(Rn+Rm)] = Rt(15:0)

Summary Data Transfer

MOVS <Rd>, <Rm>

Register to register

MOVS <Rd>, #<imm8>

Loading literals

8-bit literal

LDR <Rt>, [PC, #<imm>]

32-bit literal, PC-relative

LDR <Rt>, [<Rn>, #<imm>]

also **LDRB** and **LDRH**

Loading data

Register indirect with immediate offset

LDR <Rt>, [<Rn>, <Rm>]

also **LDRB**, **LDRH**, **LDRSB** and **LDRSH**

Register indirect with register offset

STR <Rt>, [<Rn>, #<imm>]

also **STRB** and **STRH**

Storing data

Register indirect with immediate offset

STR <Rt>, [<Rn>, <Rm>]

also **STRB** and **STRH**

Register indirect with register offset

Loading/Storing Multiple Registers

■ LDM 1)

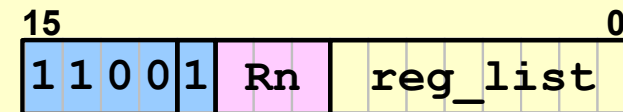
For information only

- Load Multiple Registers
- Rn: Base address

1) LDMIA (Load Multiple Increment After) and LDFM (Load Multiple from Full Descending stacks) are aliases for LDM

LDM <Rn>!, <registers>

LDM <Rn>, <registers>



Registers in reg_list
are loaded from memory
starting at address in Rn

000000CC	4A06	LDR	R2, =ldm_const
000000CE	CAE6	LDM	R2, {R1, R2, R5-R7}
000000D0	E00C	B	ldm_ex2
000000D2	0000		
000000D4	AAAAAAAA	ldm_const	
		DCD	0xAAAAAAAA
000000D8	BBBBBBBB	DCD	0xBBBBBBBB
000000DC	CCCCCCCC	DCD	0xCCCCCCCC
000000E0	DDDDDDDD	DCD	0xDDDDDDDD
000000E4	EEEEEEEE	DCD	0xEEEEEEEE
000000E8	00000000		

R1 = 0xAAAA 'AAAA
R2 = 0xBBBB 'BBBB
R5 = 0xCCCC 'CCCC
R6 = 0xDDDD 'DDDD
R7 = 0xEEEE 'EEEE

Loading/Storing Multiple Registers

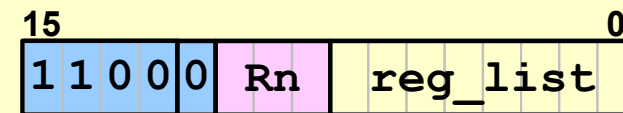
■ STM¹⁾

For information only

- Store Multiple Registers
- Rn: Base address

1) STMIA (Store Multiple Increment After) and STMEA (Store Empty Ascending) are aliases for LDM

STM <Rn>!, <registers>



Registers in reg_list
are stored to memory
starting at address in Rn

	AREA	progData2, DATA, READWRITE
data_array	SPACE	256

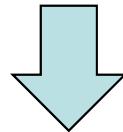
000000CE	4C01	LDR	R4, =data_array
000000D0	C4E6	STM	R4!, {R1, R2, R5-R7}
000000D2	E001	B	stm_cont
000000D4	00000000		

R1 → @data_array
R2 → @data_array + 0x04
R5 → @data_array + 0x08
R6 → @data_array + 0x0C
R7 → @data_array + 0x10

■ Array of Bytes

C-code

```
static uint8_t byte_array[] =  
    {0xAA, 0xBB, 0xCC, 0xDD,  
     0xEE, 0xFF};
```



assembly

```
byte_array  
    DCB    0xAA, 0xBB, 0xCC, 0xDD  
    DCB    0xEE, 0xFF
```

address		index
0x2001'0000	0xAA	0
0x2001'0001	0xBB	1
0x2001'0002	0xCC	2
0x2001'0003	0xDD	3
0x2001'0004	0xEE	4
0x2001'0005	0xFF	5

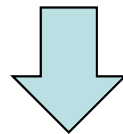
assuming that `byte_array` starts at 0x2001'0000

The C Perspective: Arrays

■ Array of Half-words

C-code

```
static uint16_t halfword_array[] =  
    {0x0011, 0x2233,  
     0x4455, 0x6677,  
     0x8899, 0xAABB};
```



assembly

```
halfword_array  
    DCW    0x0011,0x2233  
    DCW    0x4455,0x6677  
    DCW    0x8899,0xAABB
```

address		index
0x2001'0000	0x11	0
	0x00	
0x2001'0002	0x33	1
	0x22	
0x2001'0004	0x55	2
	0x44	
0x2001'0006	0x77	3
	0x66	
	0x99	4
	0x88	
	0xBB	5
	0xAA	
0x2001'000F		

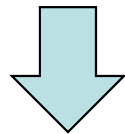
assuming that `halfword_array` starts at 0x2001'0000

The C Perspective: Arrays

■ Array of Words

C-code

```
static uint32_t word_array[] =  
    {0xFFEEDDCC,  
     0xBBAA9988,  
     0x77665544,  
     0x33221100};
```



assembly

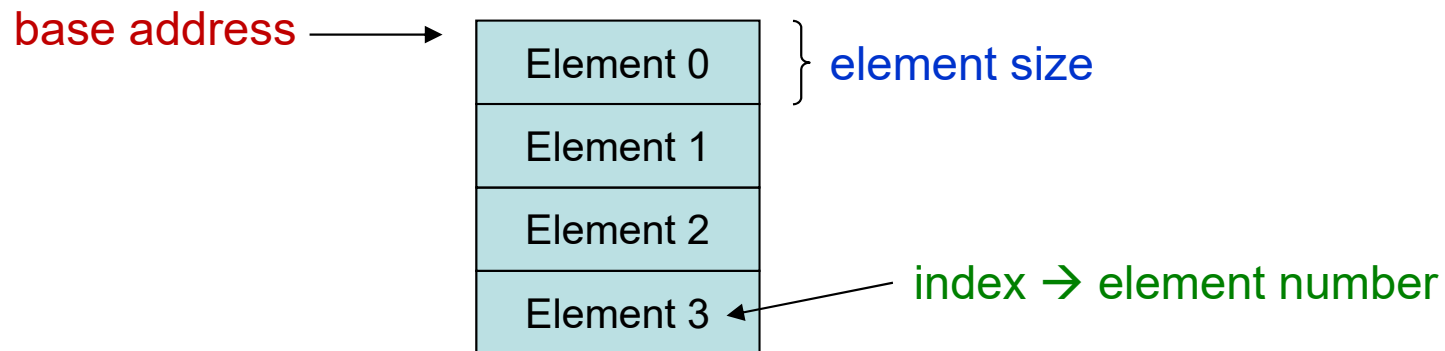
```
word_array  DCD    0xFFEEDDCC  
            DCD    0xBBAA9988  
            DCD    0x77665544  
            DCD    0x33221100
```

address		index
0x2001'0000	0xCC	0
	0xDD	
	0xEE	
	0xFF	
0x2001'0004	0x88	1
	0x99	
	0xAA	
	0xBB	
0x2001'0008	0x44	2
	0x55	
	0x66	
	0x77	
0x2001'000C	0x00	3
	0x11	
	0x22	
0x2001'000F	0x33	

assuming that `word_array` starts at 0x2001'0000

■ Accessing array elements

- element address = **base address** + **element size** • **index**



- element sizes in bytes
 - word 4
 - half-word 2
 - byte 1

■ Example: array access

C-Code

```
static uint8_t byte_array[] =
    {0xAA, 0xBB, 0xCC, 0xDD,
     0xEE, 0xFF};

void access_byte_array(void)
{
    ...
    byte_array[3] = 0x12;
    ...
}
```

- 1 Load value to be stored into R0
- 2 Load base address from label below ¹⁾
- 3 Store R0 to base address plus offset

Assembly

```
AREA MyData, DATA, READWRITE
byte_array DCB 0xaa,0xbb
           DCB 0xcc,0xdd
           DCB 0xee,0xff

AREA MyCode, CODE, READONLY
access_byte_array
    ...
    1 MOVS    r0,#0x12
    2 LDR     r1,adr_b
    3 STRB    r0,[r1,#3]
    ...

adr_b      DCD      byte_array
```

1) Compiler picks this form and not `LDR r1,=byte_array`

■ Array access (word)

C-Code

```
static uint32_t word_array[] =
    {0xFFEEDDCC,
     0xBBAA9988,
     0x77665544,
     0x33221100};

void access_word_array(void)
{
    ...
    word_array[3] = 0xAABBCCDD;
    ...
}
```

- ❶ Load literal from label → R0
- ❷ Load base address from label → R1
- ❸ Store R0 to base address plus offset
*offset (0xC) = element size (4) * index (3)*

Assembly

```
AREA MyData, DATA, READWRITE
word_array DCD 0xffeeddcc
           DCD 0xbbaa9988
           DCD 0x77665544
           DCD 0x33221100

AREA MyCode, CODE, READONLY
access_word_array
    ...
    ❶ LDR    r0, lit_1
    ❷ LDR    r1, adr_w
    ❸ STR    r0, [r1, #0xc]
    ...

lit_1      DCD 0xaabbccdd
adr_w      DCD word_array
```

■ Pointer and Address Operator

C-Code

```
void pointer_example(void)
{
    static uint32_t x;
    static uint32_t *xp;

    xp = &x;
    *xp = 0x0C;
}
```

- 1 Load address of x → R0
- 2 Load address of xp → R1
- 3 Store R0 (i.e. address of x) in xp variable (indirect memory access through R1)
- 4 Load immediate value 0x0C → R0
- 5 Load content of xp → R1
i.e. address of x is now in R1
- 6 Store R0 at address given by R1

Assembly

```
AREA MyData, DATA, READWRITE
x          DCD      0x00000000
xp         DCD      0x00000000

AREA MyCode, CODE, READONLY
pointer_example
    ...
    1 LDR      r0, adr_x
    2 LDR      r1, adr_xp
    3 STR      r0, [r1, #0]
    4 MOVS     r0, #0xc
    5 LDR      r1, [r1, #0]
    6 STR      r0, [r1, #0]
    ...

adr_x      DCD      x
adr_xp     DCD      xp
```


■ Memory Mapped I/O

- Write to LEDs on CT Board

C-Code

```
void main(void)
{
    volatile uint32_t *p;

    p = (volatile uint32_t *)
        0x60000100;

    *p = 0x1A2B3C4D;
}
```

Assembly

```
AREA MyCode, CODE, READONLY

                LDR    r0,led_adr
                LDR    r1,led_val
                STR    r1,[r0, #0]

led_val        DCD    0x1A2B3C4D
led_adr        DCD    0x60000100
```

The local variable p is kept in register r0, not in memory

■ Data transfers

- Register to Register `MOV/MOVS (register)`
- Loading Literals `MOVS (immediate data)`
`LDR (PC-relative/literal-pool)`
- Loading Data `LDR (immediate/register offset)`
- Storing Data `STR (immediate/register offset)`

■ Addressing Modes

- PC Relative `[PC, #0x12]`
- Indirect Addressing `[R1], [R2, #0x12], [R5, R6]`

■ Arrays

- Element address = **base address** + **element size** • **index**
- Accessed with data transfer instructions

■ Volatile

- Use for accessing memory mapped items