Computer Organisation & Program Execution 2021

4

Data Structures

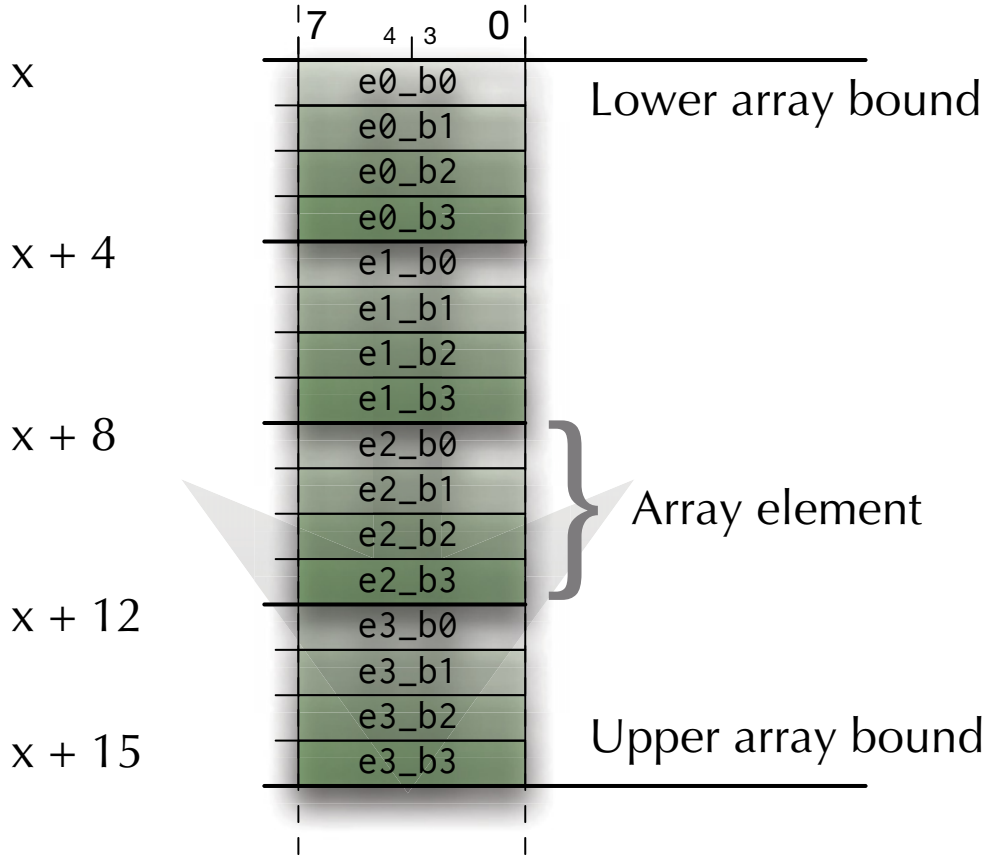
Uwe R. Zimmer - The Australian National University

References for this chapter

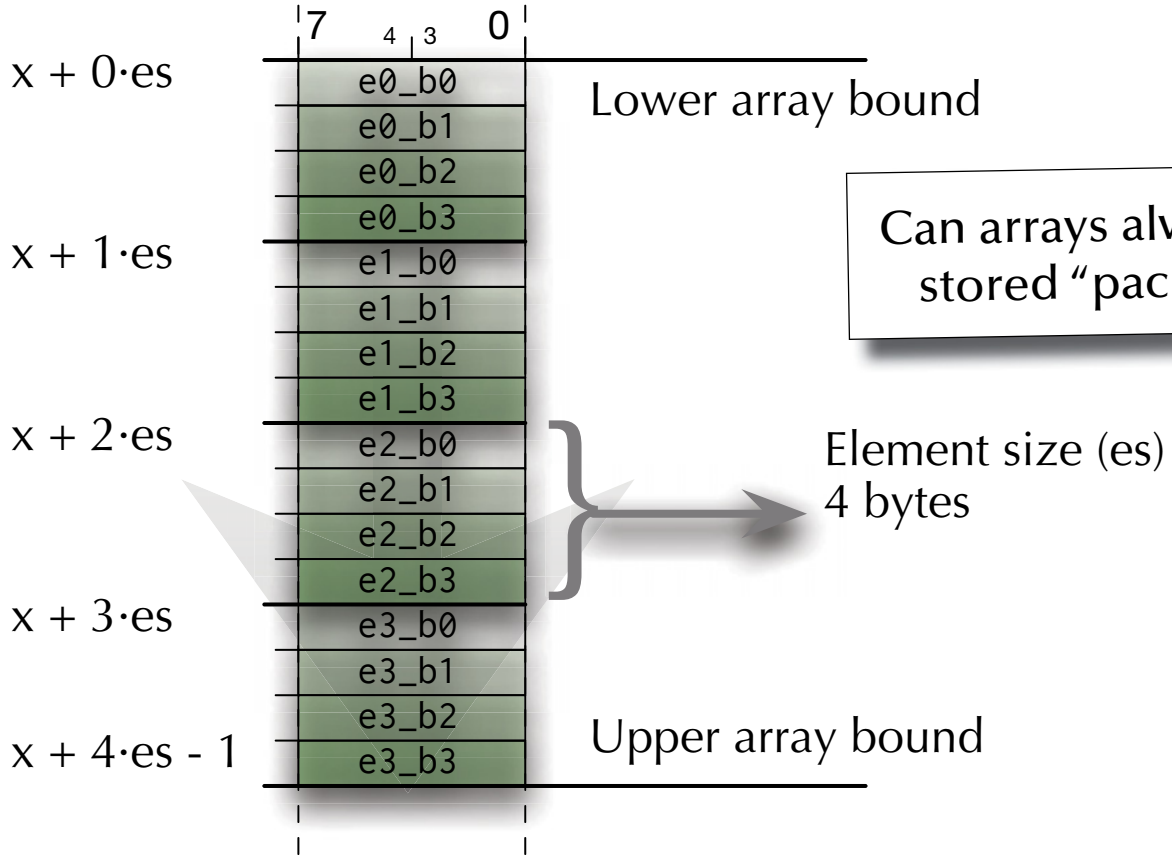
[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 "Instructions: Language of the Computer",
Chapter 5 "Large and Fast: Exploiting Memory Hierarchy"
ARM edition, Morgan Kaufmann 2017

Array layout

Elements of equal size sequentially in memory



Array addressing



$x + i \cdot es$

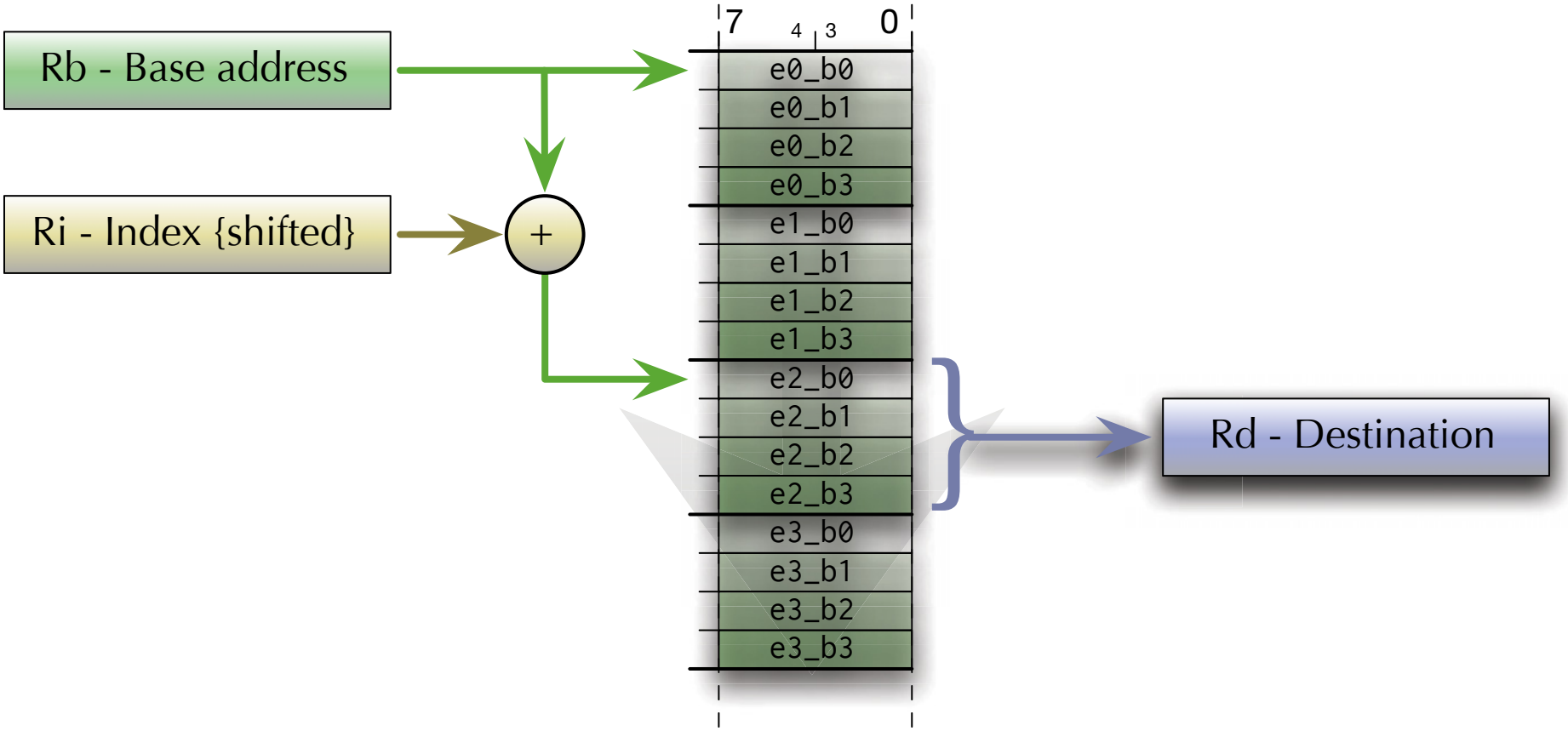
Can arrays always be stored "packed"?

Maybe be good to have $es = 2^n$?

What happens if array bounds are violated?

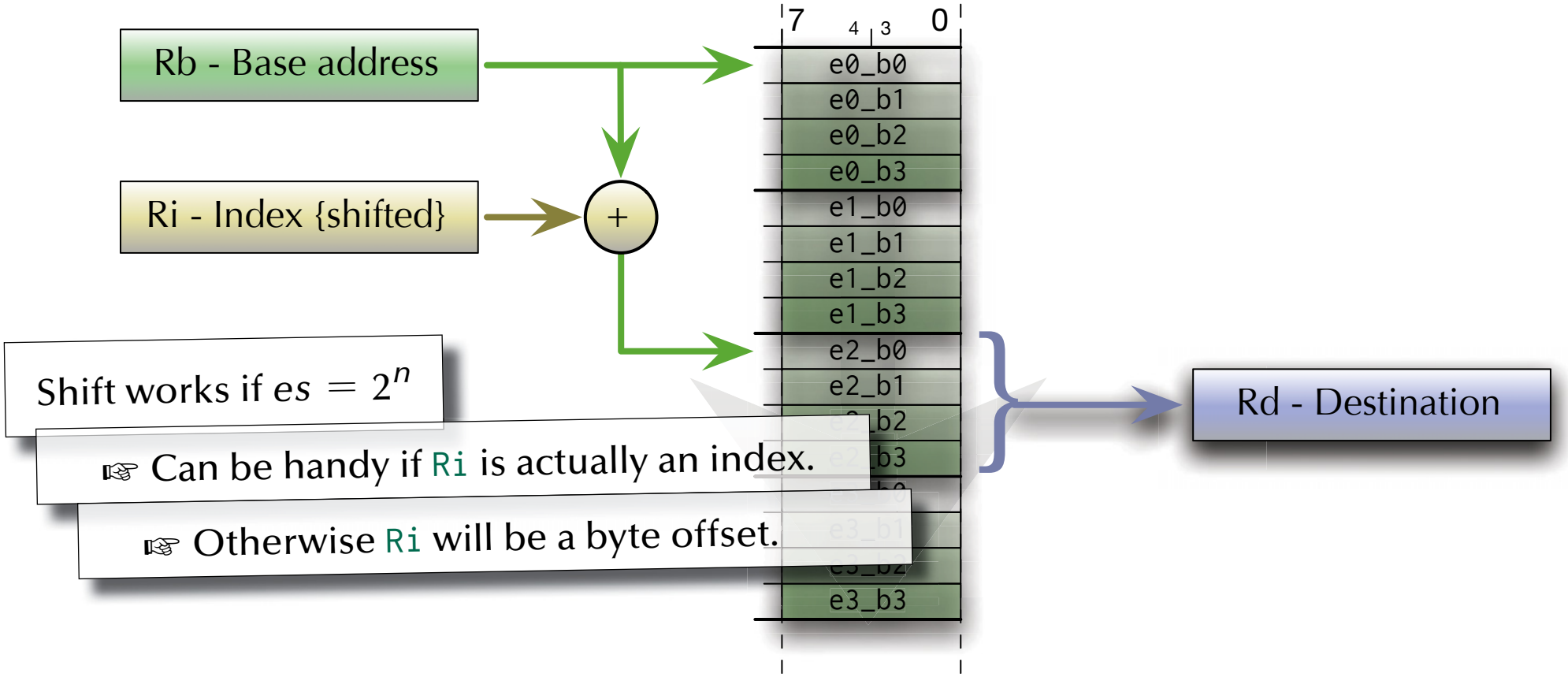
☞ And who is checking that?

Array addressing via index register




```
ldr<c><q> <Rd>, [<Rb>, <Ri> {, lsl #<shift>}]
```

Array addressing via index register



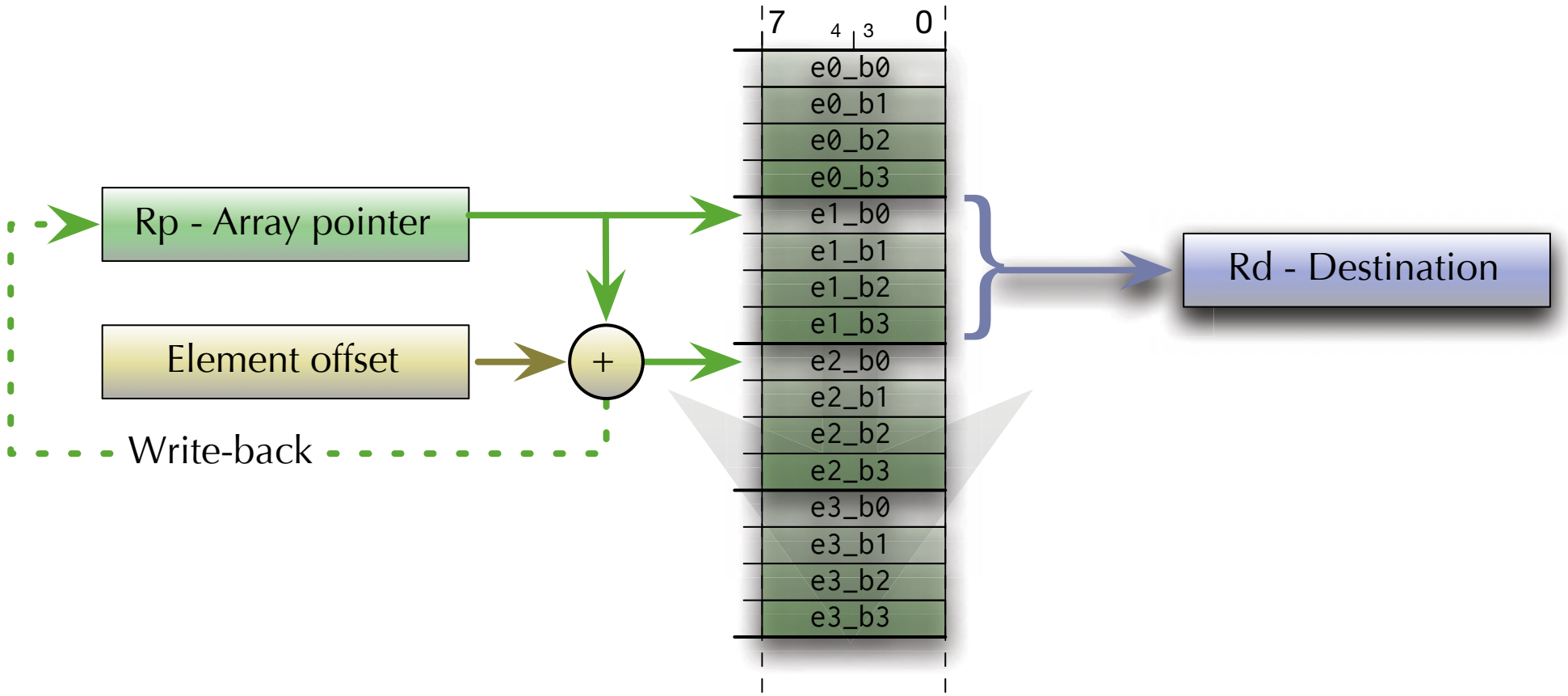
Shift works if $es = 2^n$

☞ Can be handy if `Ri` is actually an index.

☞ Otherwise `Ri` will be a byte offset.

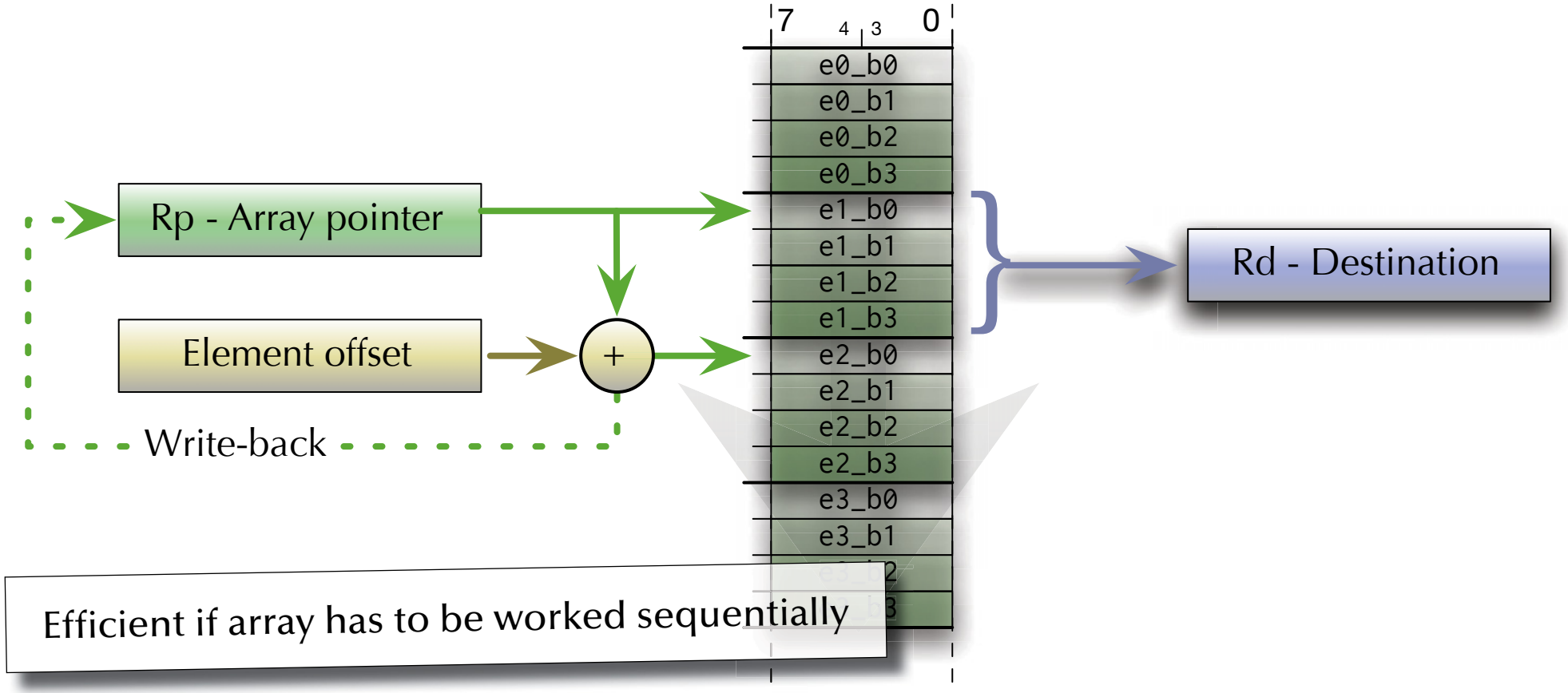
```
ldr<c><q> <Rd>, [<Rb>, <Ri> {, lsl #<shift>}]
```

Array addressing via element pointer




```
ldr<c><q> <Rd>, [<Rp>], #+/-<offset>
```

Array addressing via element pointer




```
ldr<c><q> <Rd>, [<Rp>], #+/-<offset>
```

Calculate $\sum x_i$

```
int sum (unsigned int uints [], unsigned int from, unsigned int to) {
  int i;
  int acc = 0;
  for (i = from; i <= to; i++) {
    acc += uints [i];
  }
  return acc;
}
```

```
unsigned int uints [100];
unsigned int s;
int i;
for (i = 0; i <= 99; i++) {
  uints [i] = rand ();
}
s = sum (uints, 0, 99);
```

```
type Naturals is array (Integer range <>) of Natural;
function Sum (Numbers : Naturals) return Natural is
  Acc : Natural := 0;
begin
  for n of Numbers loop
    Acc := Acc + n;
  end loop;
  return Acc;
end Sum;
```

```
Numbers : constant Naturals (1 .. 100) :=
                    (others => Random (Numbers_Generator));
Sum_of_Numbers : constant Natural := Sum (Numbers);
```

Calculate $\sum x_i$

```
int sum (unsigned int uints [], unsigned int from, unsigned int to) {
  int i;
  int acc = 0;
  for (i = from; i <= to; i++) {
    acc += uints [i];
  }
  return acc;
}
```

```
unsigned int uints [100];
unsigned int s;
int i;
for (i = 0; i <= 99; i++) {
  uints [i] = rand ();
}
s = sum (uints, 0, 99);
```

Best of luck with the array bounds!

```
type Naturals is array (Integer range <>) of Natural;
function Sum (Numbers : Naturals) return Natural is
  Acc : Natural := 0;
begin
  for n of Numbers loop
    Acc := Acc + n;
  end loop;
  return Acc;
end Sum;
```

```
Numbers : constant Naturals (1 .. 100) :=
                    (others => Random (Numbers_Generator));
Sum_of_Numbers : constant Natural := Sum (Numbers);
```

Arbitrary array indexing

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index
    mov   r3, #0        ; sum := 0
    mov   r4, #4        ; element size is 4 bytes
    mov   r5, #-1       ; first_element_offset
```

3

```
for_sum:
    cmp   r1, r2        ; i > to
    bgt   end_for_sum
    mla   r6, r1, r4, r5  ; element_offset := (i * 4) - first_element_offset
    ldr   r7, [r0, r6]  ; a [i] := [base + element_offset]
    add   r3, r7        ; sum := sum + a [i]
    add   r1, #1        ; i := i + 1
    b     for_sum
```

9

```
end_for_sum:
    mov   r0, r3        ; r0 sum over all a [from .. to]
```

1

Zero-based array indexing

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index
    mov   r3, #0        ; sum := 0
    mov   r4, #4        ; element size is 4 bytes
for_sum:
    cmp   r1, r2        ; i > to
    bgt   end_for_sum
    mul   r5, r1, r4    ; element_offset := (i * 4)
    ldr   r6, [r0, r5]  ; a [i] := [base + element_offset]
    add   r3, r6        ; sum := sum + a [i]
    add   r1, #1        ; i := i + 1
    b     for_sum
end_for_sum:
    mov   r0, r3        ; r0 sum over all a [from .. to]
```

2

8

1

Replacing multiplication with shifted index register

```
                          ; r0 base address for array a
                          ; r1 from array index
                          ; r2 to array index
    mov   r3, #0          ; sum := 0                                  1
for_sum:
    cmp   r1, r2          ; i > to
    bgt   end_for_sum
    ldr   r4, [r0, r1, lsl #2] ; a [i] := [base + element_offset]     7
    add   r3, r4          ; sum := sum + a [i]
    add   r1, #1          ; i := i + 1
    b     for_sum
end_for_sum:
    mov   r0, r3          ; r0 sum over all a [from .. to]            1
```

Replacing indices with offsets

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index
    lsl   r1, r1, #2    ; translate from index to offset
    lsl   r2, r2, #2    ; translate to index to offset
    mov   r3, #0        ; sum := 0
```

3

```
for_sum:
    cmp   r1, r2        ; i > to
    bgt   end_for_sum
    ldr   r4, [r0, r1]  ; a [i] := [base + offset]
    add   r3, r4        ; sum := sum + a [i]
    add   r1, #4        ; offset := offset + 4
    b     for_sum
```

7

```
end_for_sum:
    mov   r0, r3        ; r0 sum over all a [from .. to]
```

1

Assuming non-empty arrays

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index >= from index
    lsl   r1, r1, #2    ; translate from index to offset
    lsl   r2, r2, #2    ; translate to index to offset
    mov   r3, #0        ; sum := 0
for_sum:
    ldr   r4, [r0, r1]  ; a [i] := [base + offset]
    add   r3, r4        ; sum := sum + a [i]
    add   r1, #4        ; offset := offset + 4
    cmp   r1, r2        ; i <= to
    ble   for_sum
end_for_sum:
    mov   r0, r3        ; r0 sum over all a [from .. to]
```

3

6

1

Replacing offsets with addresses

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index >= from index
    lsl   r1, r1, #2    ; translate from index to offset
    lsl   r2, r2, #2    ; translate to index to offset
    add   r1, r0        ; translate from index to address -> i_addr
    add   r2, r0        ; translate to index to address -> to_addr
    mov   r0, #0        ; sum := 0
for_sum:
    ldr   r3, [r1], #4  ; a [i] := [i_addr]; i_addr += 4
    add   r0, r3        ; sum := sum + a [i]
    cmp   r1, r2        ; i_addr <= to_addr
    ble   for_sum
end_for_sum:
                        ; r0 sum over all a [from .. to]
```

5

5

Array Slices

```
numbers      = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]

# numbersSlice equals [1, 2, 3]
```

```
numbers      := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]
```

```
type Naturals is array (Integer range <>) of Natural;

Numbers         : constant Naturals (-50 .. 50) := (others => Random (Generator));

Numbers_Slice_1 : constant Naturals             := Numbers (-10 .. 10);
Numbers_Slice_2 : constant Naturals (  1 .. 10) := Numbers ( 11 .. 20);
Numbers_Slice_3 :          Naturals             := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
```

Array Slices

```
numbers      = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]
# numbersSlice equals [1, 2, 3]
```

Are those copy or reference affairs?

```
numbers      := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]
```

```
type Naturals is array (Integer range <>) of Natural;

Numbers         : constant Naturals (-50 .. 50) := (others => Random (Generator));

Numbers_Slice_1 : constant Naturals             := Numbers (-10 .. 10);
Numbers_Slice_2 : constant Naturals (  1 .. 10) := Numbers ( 11 .. 20);
Numbers_Slice_3 :          Naturals             := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
```

Copy array slice

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index >= from index
                        ; r3 base address for array b
    lsl   r1, r1, #2    ; translate from index to offset
    lsl   r2, r2, #2    ; translate to index to offset
    add   r1, r0        ; translate from index to address -> i_addr
    add   r2, r0        ; translate to index to address -> to_addr
for_copy:
    ldr   r4, [r1], #4  ; a [i]    := [i_addr]; i_addr += 4
    str   r4, [r3], #4  ; [j_addr] := a [i]   ; j_addr += 4
    cmp   r1, r2        ; i_addr <= to_addr
    ble   for_copy
end_for_copy:
                        ; b [] := a [from .. to]
```

4

6

Copy array slice

```
                        ; r0 base address for array a
                        ; r1 from array index
                        ; r2 to array index >= from index
                        ; r3 base address for array b
  lsl   r1, r1, #2      ; translate from index to offset
  lsl   r2, r2, #2      ; translate to index to offset
  add   r1, r0          ; translate from index to address -> i_addr
  add   r2, r0          ; translate to index to address -> to_addr
for_copy:
  ldr   r4, [r1], #4    ; a [i]    := [i_addr]; i_addr += 4
  str   r4, [r3], #4    ; [j_addr] := a [i]   ; j_addr += 4
  cmp   r1, r2          ; i_addr <= to_addr
  ble   for_copy
end_for_copy:
                        ; b [] := a [from .. to]
```

4

6

Moving blocks of memory can be done even/much faster with special hardware.

☞ **DMA controllers**

Summary

Data Structures

- **Arrays**
  - Structure
  - Alignment
  - Addressing
  - Iterators
  - Copy procedures