CS-200
Computer Architecture
—
Part 1d. Instruction Set Architecture
Arrays and Data Structures

Paolo Ienne
<paolo.ienne@epfl.ch>

Arrays in High-Level Languages

- Java:

```
short[] myData = {10407, -16533, -22715, 29796, 18956,…};
```

- Python:

```
myData = [10407, -16533, -22715, 29796, 18956,…]
```

or

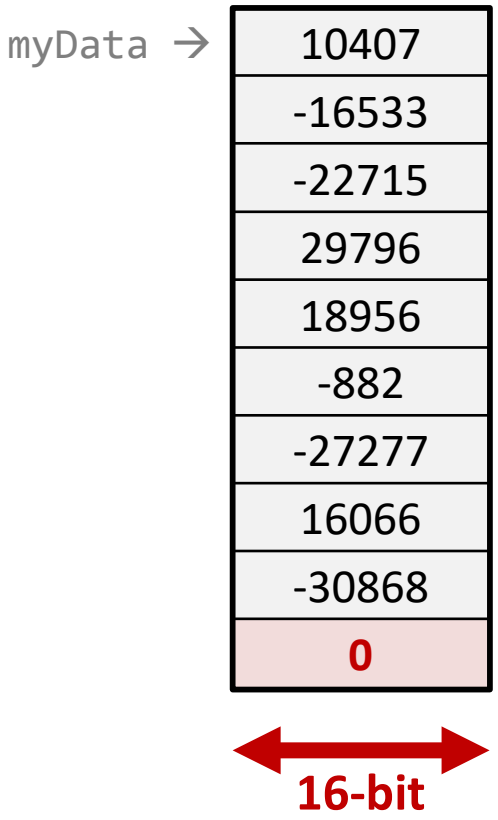
```
import array
myData = array.array('h', [10407, -16533, -22715, 29796, 18956,…])
```

- Scala:

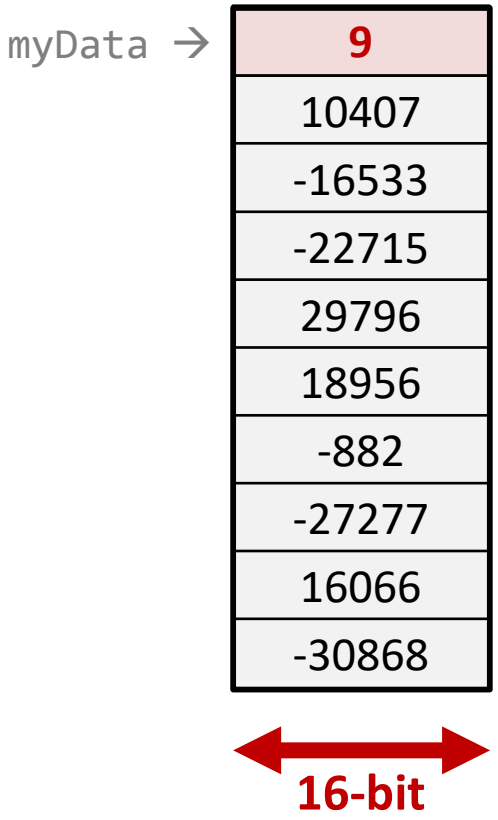
```
val myData: Array[Short] = Array(10407, -16533, -22715, 29796, 18956,…)
```

Different Ways to Store Arrays

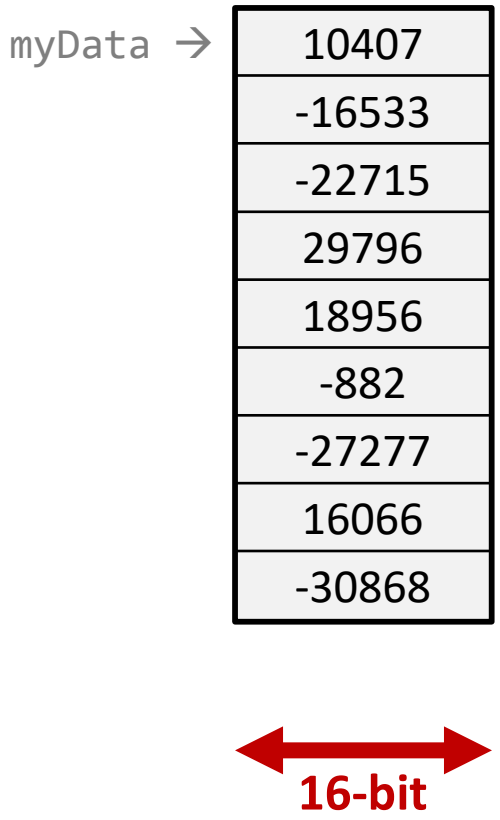
A



B



C



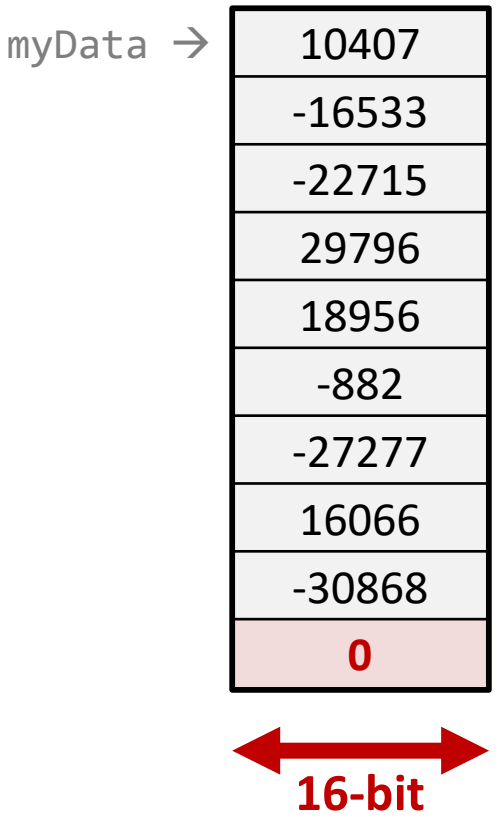
Adding Positive Elements

- Add all positive elements in an array of signed 16-bit integers
  - At call time, `a0` points to the array
  - At return time, `a0` contains the result
- Write it three times, for arrays of type **A**, **B**, and **C**

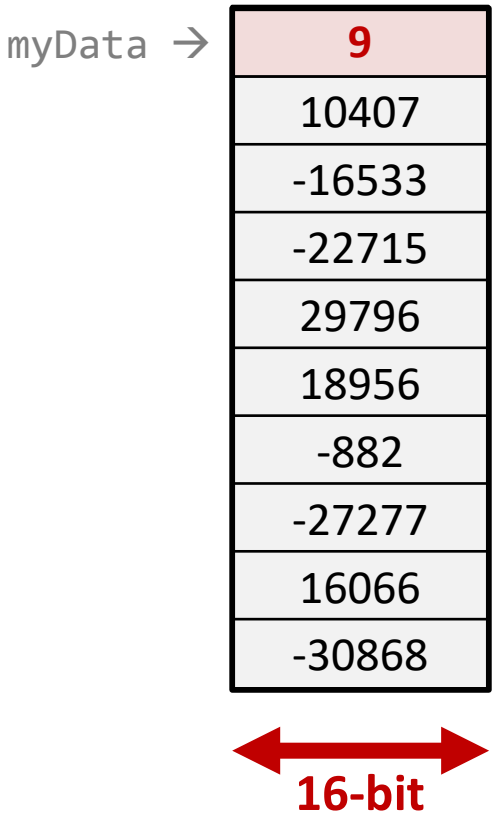
Type C

```
short sum = 0;
int i;
for (i = 0; i < N; i++) {
  if (myData[i] > 0) {
    sum += myData[i];
  }
}
```

A


myData
10407
-16533
-22715
29796
18956
-882
-27277
16066
-30868
0
16-bit

B


myData
9
10407
-16533
-22715
29796
18956
-882
-27277
16066
-30868
16-bit

Adding Positive Elements (Variation on C)

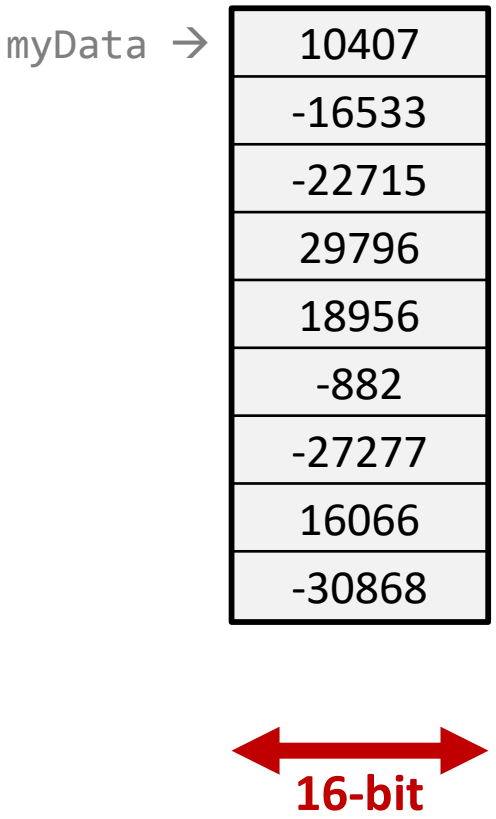
- Add all positive elements in an array of signed 16-bit integers
  - At call time, `a0` points to the array and `a1` is the length of the array
  - At return time, `a0` contains the result
- Write it by incrementing the index of the array

```
int i = 0;
while (i < N) {
  if (myData[i] > 0) {
    …
  }
  i++;
}
```

=

```
int i;
for (i = 0; i < N; i++) {
  if (myData[i] > 0) {
    …
  }
}
```

C


myData
10407
-16533
-22715
29796
18956
-882
-27277
16066
-30868
16-bit

Which is Better?

Pointer to memory

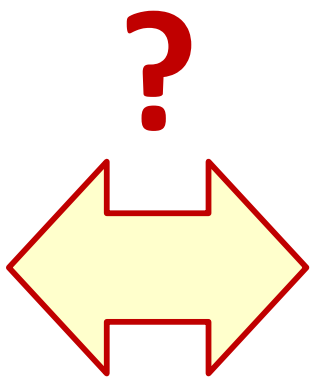
```
add_positive:
    li    t0, 0
    mv    t1, a1

next_short:
    beq   t1, zero, end
    lh    t2, 0(a0)
    bltz  t2, negative
    add   t0, t0, t2

negative:
    addi  a0, a0, 2
    addi  t1, t1, -1
    j     next_short

end:
    mv    a0, t0
    ret
```

Less instructions → Faster



Index in array

```
add_positive:
    li    t0, 0
    li    t1, 0

next_index:
    bge   t1, a1, end
    slli  t2, t1, 1
    add   t2, a0, t2
    lh    t3, 0(t2)
    bltz  t3, negative
    add   t0, t0, t3

negative:
    addi  t1, t1, 1
    j     next_index

end:
    mv    a0, t0
    ret
```

Which is Better?

Pointer to memory

```
short sum = 0;
short *ptr = myData;
short *end = myData + N;

while (ptr < end) {
  if (*ptr > 0) {
    sum += *ptr;
  }
  ptr++;
}
```

A pointer to **short** integers
= address of **myData**

Advances the pointer instead of recomputing the element address

?

Index in array

```
short sum = 0;
int i;
for (i = 0; i < N; i++) {
  if (myData[i] > 0) {
    sum += myData[i];
  }
}
```

Better code

We Need Good Compilers!

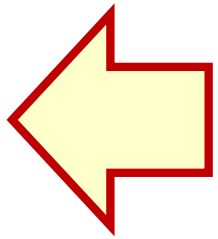
Pointer to memory

```
add_positive:
    li    t0, 0
    mv    t1, a1

next_short:
    beq   t1, zero, end
    lh    t2, 0(a0)
    bltz  t2, negative
    add   t0, t0, t2

negative:
    addi  a0, a0, 2
    addi  t1, t1, -1
    j     next_short

end:
    mv    a0, t0
    ret
```

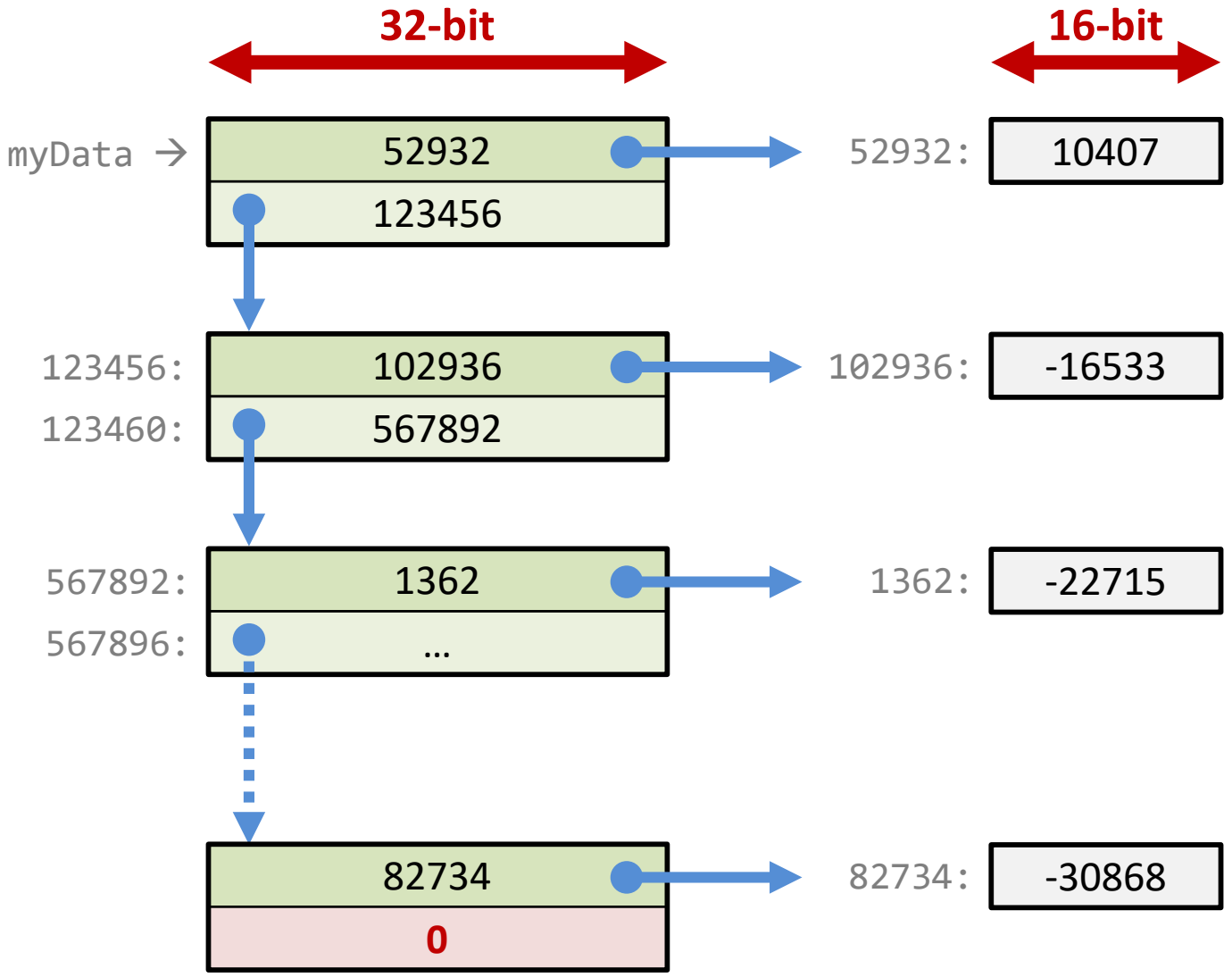


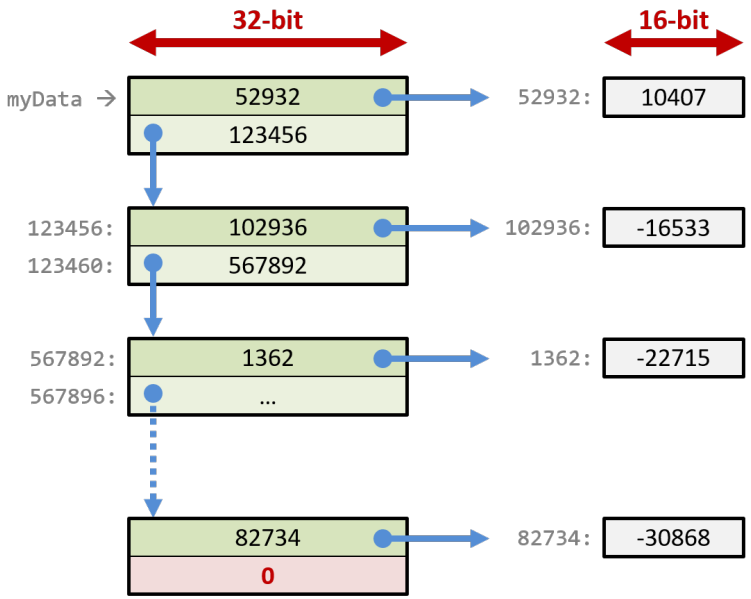
Index in array

```
short sum = 0;
int i;
for (i = 0; i < N; i++) {
  if (myData[i] > 0) {
    sum += myData[i];
  }
}
```

Write **better code** but get the **best performance**!

One More Way to Store Arrays




32-bit
16-bit
myData
52932
123456
52932:
10407
123456:
102936
123460:
567892
102936:
-16533
567892:
1362
567896:
...
1362:
-22715
82734
0
82734:
-30868

Not Much More Complex…

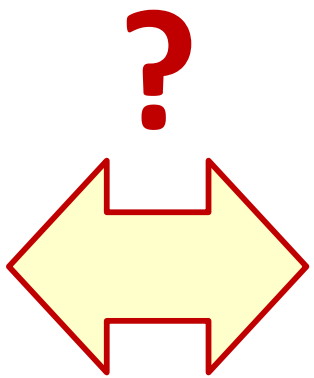
Pointer to memory

```
add_positive:
    li    t0, 0
    mv    t1, a1

next_short:
    beq   t1, zero, end
    lh    t2, 0(a0)
    bltz  t2, negative
    add   t0, t0, t2

negative:
    addi  a0, a0, 2
    addi  t1, t1, -1
    j     next_short

end:
    mv    a0, t0
    ret
```



Linked list

```
add_positive:
    li    t0, 0

next_element:
    beqz  a0, end
    lw    t1, 0(a0)
    lh    t1, 0(t1)
    bltz  t1, negative
    add   t0, t0, t1

negative:
    lw    a0, 4(a0)
    j     next_index

end:
    mv    a0, t0
    ret
```

Multiple noncontiguous loads

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 2** and, in particular, **Sections 2.9**, **2.13**, and **2.14**