Topic V15

Array Indexing vs Pointers

Reading: (Section 2.14)

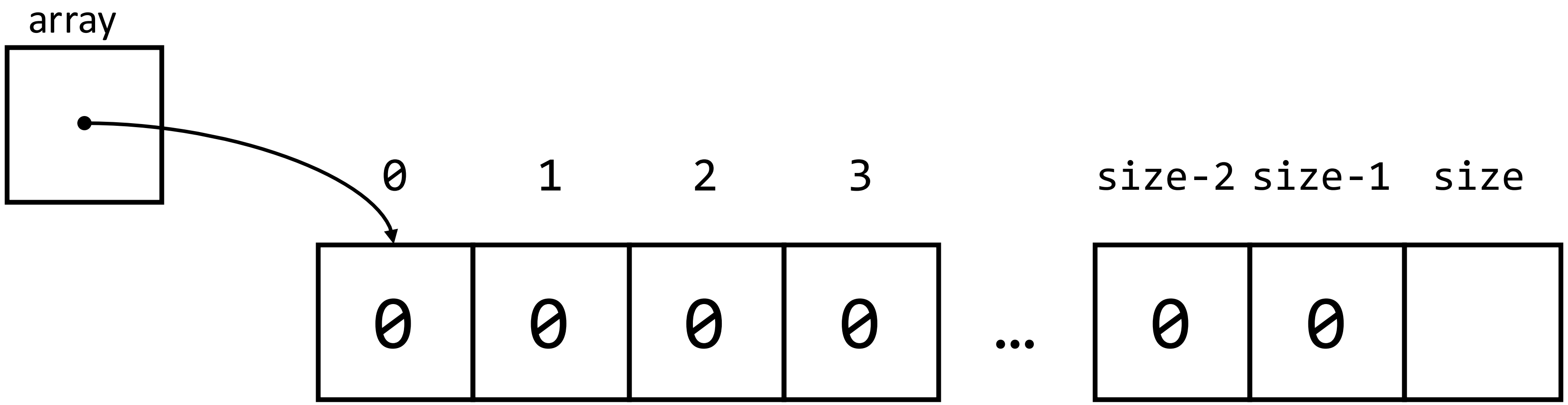
Example of a Task

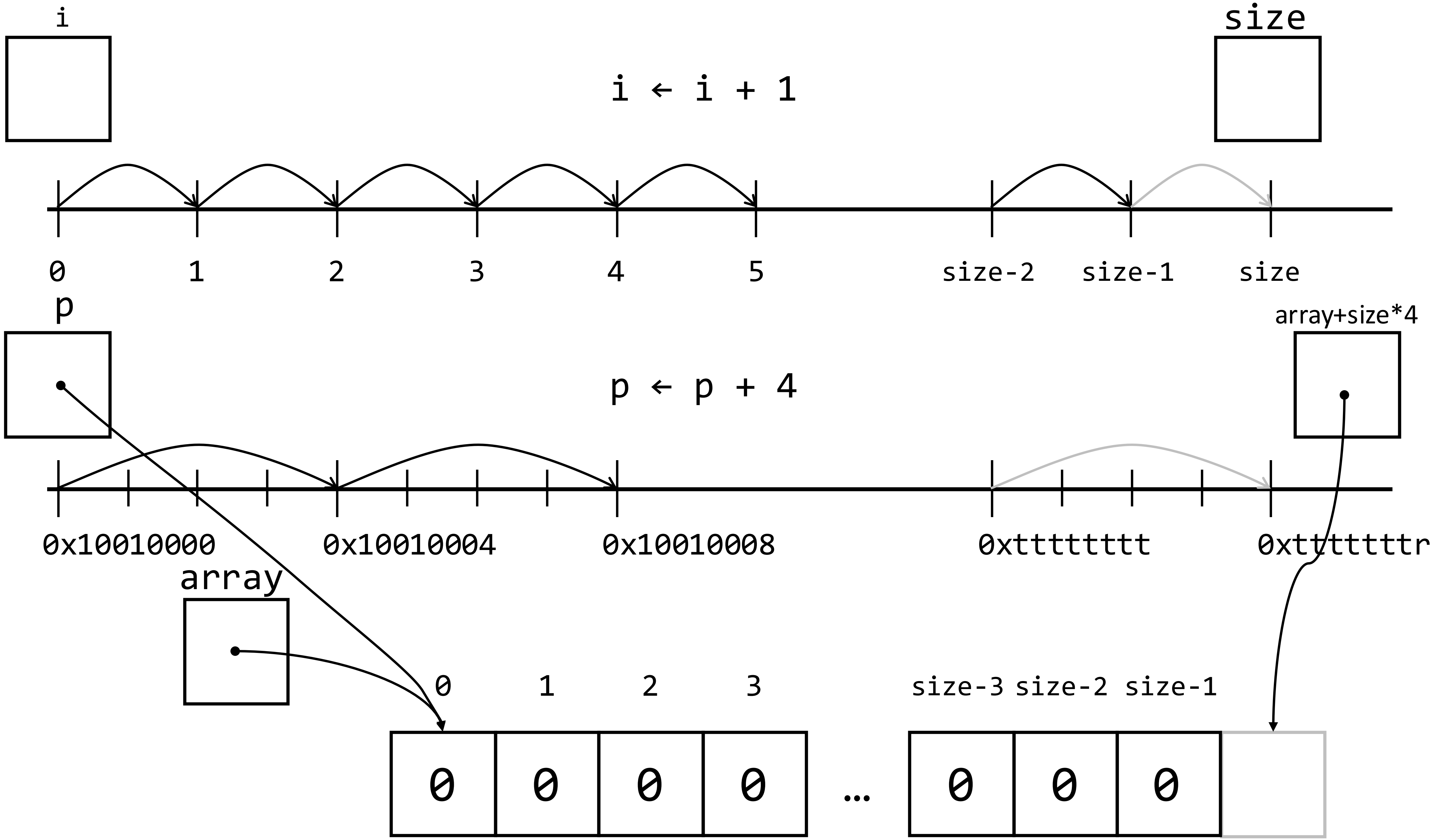
`clear`: writes zero in all elements of an integer array

Parameters:

`array`: address of first array element

`size`: number of elements in array





i ← i + 1

p ← p + 4

i
size
0
1
2
3
4
5
size-2
size-1
size
p
array+size*4
0x10010000
0x10010004
0x10010008
0xtttttttt
0xtttttttr
array
0
1
2
3
size-3
size-2
size-1
0
0
0
0
…
0
0
0

Array Indexing vs. Pointers

Array Indexing:

```
void clear1(int array[], int size){
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
t0 ← 0
while (t0 < size){
  t2 ← array+4*t0
  M[t2] ← zero
  t0 ← t0 + 1
}
```

a0 array

t0 i

Array Indexing vs. Pointers

Array Indexing:

```
void clear1(int array[], int size){
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
t0 ← 0
while (t0 < size){
  t1 ← 4*t0
  t2 ← array+t1
  M[t2] ← zero
  t0 ← t0 + 1
}
```

Must multiply by 4

and add to base

in every loop iteration

a0 array

t0 i

Pointer Indexing:

```
void clear2(int *array, int size){
  int *p;
  for (p = &array[0]; p < &array[size]; p += 1)
    *p = 0;
}
```

```
t0 ← array
t2 ← &array[size]
while (t0 < t2){
  M[t0] ← zero
  t0 ← t0 + 4
}
```

Pointer corresponds
to memory address

a0 array

t0 p

```
void clear1(int array[], int size){
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
       ble   a1, zero, done # if n <= 0
       mv    t0, zero       # i ← 0
loop1: slli  t1, t0, 2      # t1 ← i * 4
       add   t2, a0, t1     # t2 ← &array[i]
       sw    zero, 0(t2)    # array[i] ← 0
       addi  t0, t0, 1      # i ← i + 1
       blt   t0, a1, loop1  # if i < size
                                      goto loop1
```

```
void clear2(int *array, int size){
  int *p;
  for (p = &array[0]; p < &array[size]; p += 1)
    *p = 0;
}
```

```
       ble   a1, zero, done # if n <= 0
       mv    t0, a0         # p ← &array[0]
       slli  t1, a1, 2      # t1 ← size * 4
       add   t2, a0, t1     # t2 ← &array[size]
loop2: sw    zero, 0(t0)    # M[p] ← 0
       addi  t0, t0, 4      # p ← p + 4
       blt   t0, t2, loop2  # if p<&array[size]
                                      goto loop2
```

`add`, `addi`, `slli`: 1 cycle;   `sw`: 5 cycles;   `blt`, `ble`: 2 cycles

Assume size is very large. Compute CPI. Which is faster? By how much?

# of instructions = 5 * size + 2

# of cycles = 1 + 2 + size * (1 + 1 + 5 + 1 + 2)

= 10 * size + 3

CPI $= \dfrac{10 * \text{size} + 3}{5 * \text{size} + 2} \approx 2$

```
        ble    a1, zero, done  # if n <= 0          2
        mv     t0, zero        # i ← 0              1
loop1:  slli   t1, t0, 2       # t1 ← i * 4         1
        add    t2, a0, t1      # t2 ← &array[i]     1
        sw     zero, 0(t2)     # array[i] ← 0       5
        addi   t0, t0, 1       # i ← i + 1          1
        blt    t0, a1, loop1   # if i < size        2
                                    goto loop1
```

# of instructions = 3 * size + 4

# of cycles = 5 + size * (5 + 1 + 2)

= 8 * size + 5

CPI $= \dfrac{8 * \text{size} + 5}{3 * \text{size} + 4} \approx 2.67$

```
        ble    a1, zero, done  # if n <= 0          2
        mv     t0, a0          # p ← &array[0]      1
        slli   t1, a1, 2       # t1 ← size * 4      1
        add    t2, a0, t1      # t2 ← &array[size]  1
loop2:  sw     zero, 0(t0)     # M[p] ← 0           5
        addi   t0, t0, 4       # p ← p + 4          1
        blt    t0, t2, loop2   # if p<&array[size]  2
                                    goto loop2
```

# of cycles(index) = 10 * size + 3

# of cycles(ptr) = 8 * size + 5

Speedup $= \dfrac{10 * \text{size} + 3}{8 * \text{size} + 5} \approx 1.25$

`add`, `addi`, `slli`: 1 cycle; `sw`: 5 cycles; `blt`, `ble`: 2 cycles

Assume size is very large. Compute CPI. Which is faster? By how much?

Comparison of Array Indexing vs. Pointers

Array indexing requires shift to be inside loop

- It is part of index calculation for incremented `i`
- Recalculate address in each iteration

Compiler can achieve same effect as manual use of pointers

- Induction variable elimination
- Better to make program clearer and safer