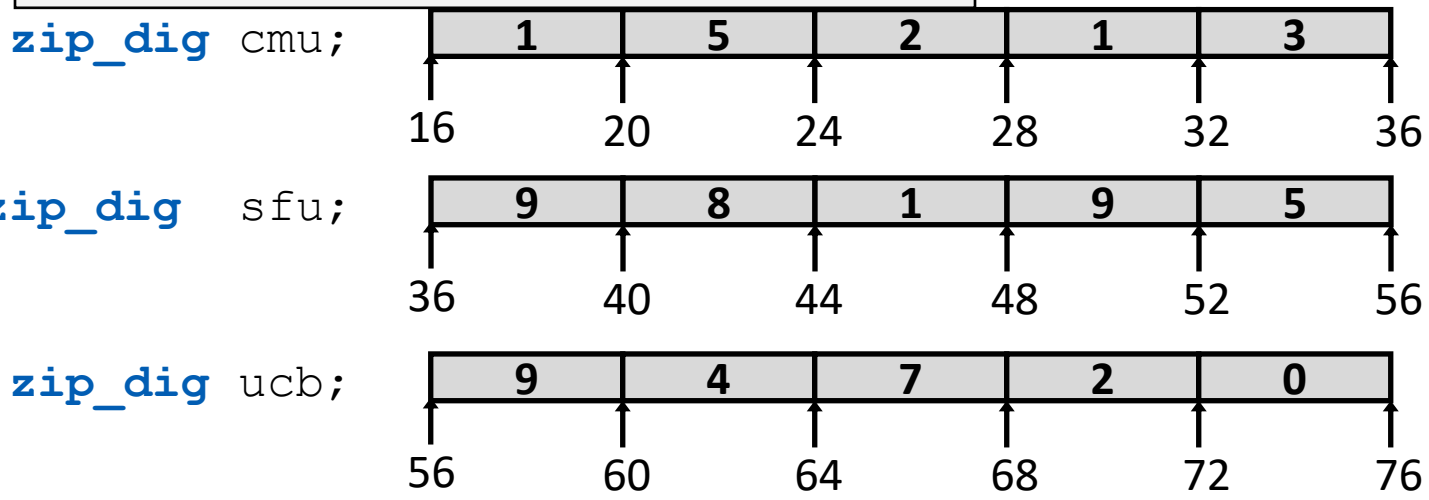
Data Structures in Assembly

- **Arrays**
  - **One-dimensional**
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig sfu  = { 9, 8, 1, 9, 5
};
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

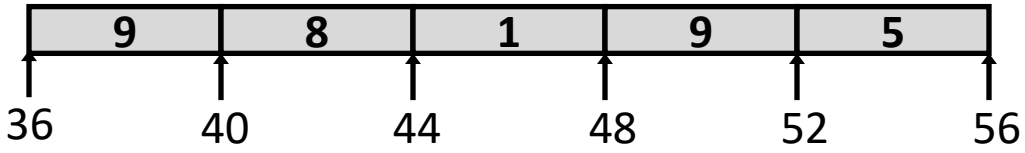


- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

```
typedef int zip_dig[5];
```

Array Accessing Example

```
zip_dig sfu;
```



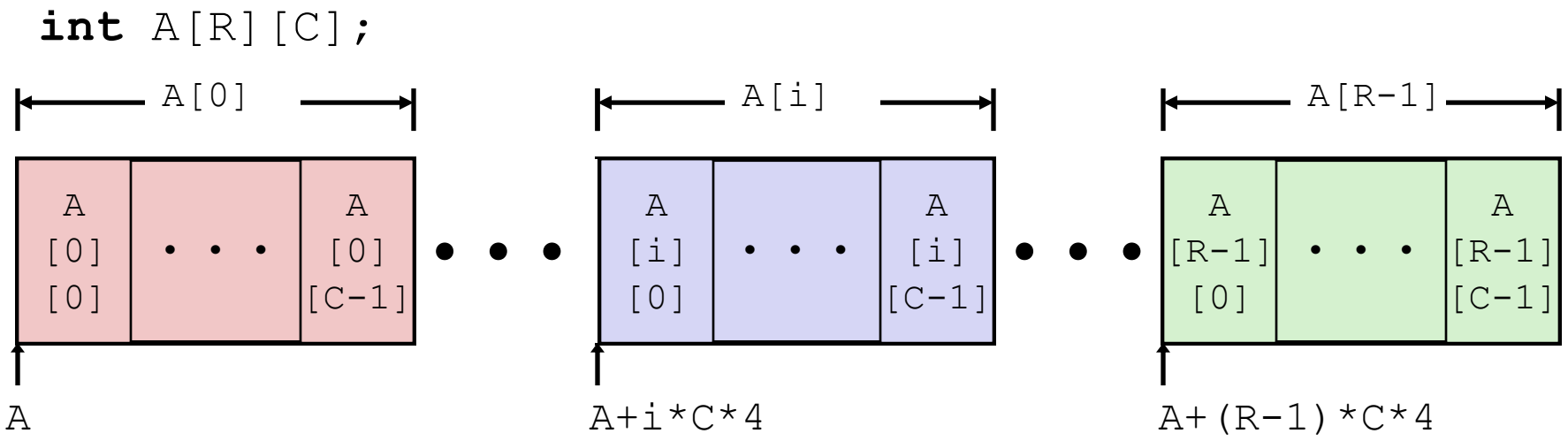

```
int get_digit(zip_dig z, int digit){
  return z[digit];
}
```

```
get_digit:
slli a1,a1,2
add a1,a0,a1
lw a0,0(a1)
ret
```

- Register `%a0` contains starting address of array
- Register `%a1` contains array index
- Desired digit at `%a0+4*%a1`

Nested Array Row Access

- Row vectors
  - Given `T A[R][C]`,
    - `A[i]` is an array of `C` elements (“row `i`”)
    - `A` is address of array
    - Starting address of row `i` = `A + i*(C * sizeof(T))`



Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
get_sea_zip(int):
        slli    a5,a0,2
        add     a5,a5,a0
        lui     a0,%hi(.LANCHOR0)
        slli    a5,a5,2
        addi    a0,a0,%lo(.LANCHOR0)
        add     a0,a0,a5
        ret
sea:
        .word   9
        .word   8
        .word   1
        .word   9
        .word   5
```

Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

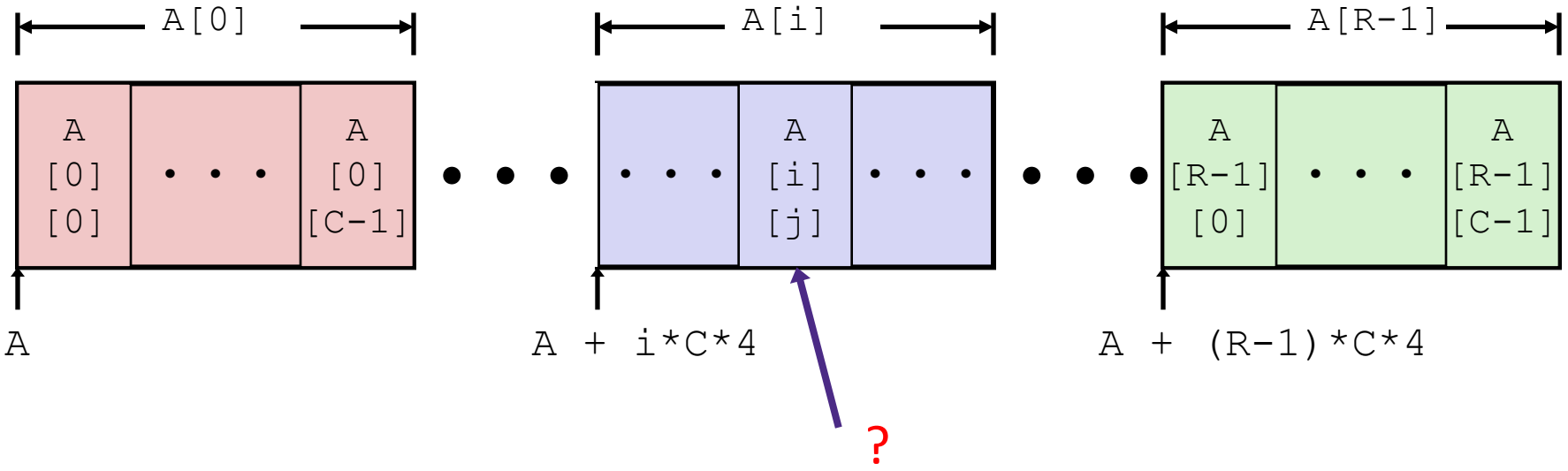
- What data type is `sea[index]`?
- What is its value?

- Row Vector
  - `sea[index]` is array of 5 `int`s
  - Starting address = `sea+20*index`
- Assembly Code
  - Computes and returns address
  - Compute as: `sea+4*(index + 4*index) = sea+20*index`

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **`T`**, which requires K bytes
  - Address of `A[i][j]` is

```
int A[R][C];
```

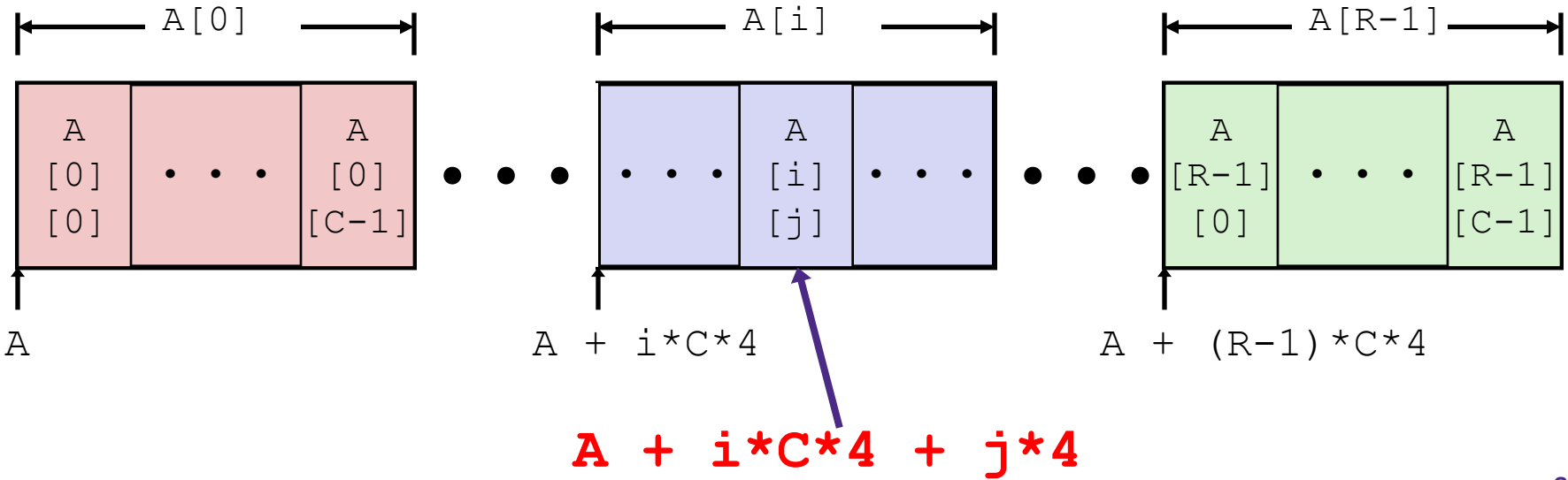


Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **`T`**, which requires *K* bytes
  - Address of `A[i][j]` is

$$\texttt{A} + \texttt{i*(C*}K\texttt{)} + \texttt{j*}K == \texttt{A} + \texttt{(i*C + j)*}K$$

```
int A[R][C];
```



Nested Array Element Access Code

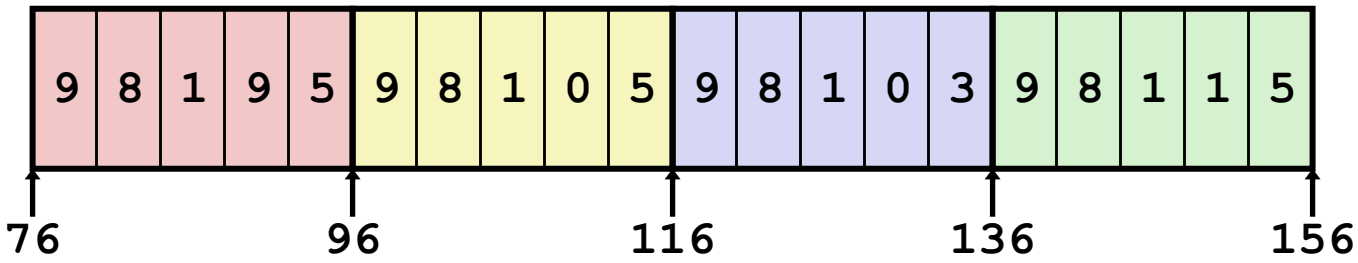
```
int get_sea_digit
  (int index, int digit)
{
  return sea[index][digit];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
get_sea_digit(int, int):
    slli    a5,a0,2    // a0 : index a1: digit. index*4.
    add     a0,a5,a0   // a0: index*5
    add     a0,a0,a1   // a0 = index*5+digit
    lui     a1,%hi(.LANCHOR0)   // Load base address
    addi    a1,a1,%lo(.LANCHOR0) // in a1
    slli    a0,a0,2    // multiply by sizeof(integer)
    add     a0,a1,a0   // Add to base address in a1
    lw      a0,0(a0)   // return value
    ret
```

Multi-Dimensional Referencing Examples

```
zip_dig sea[4];
```



Reference	Address	Value	Guaranteed?
`sea[3][3]`			
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

Multi-Level Array Declaration(s):

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  sfu[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {sfu, cmu, ucb};
```

2D Array Declaration:

```
zip_dig univ2D[3] = {
  { 9, 8, 1, 9, 5 },
  { 1, 5, 2, 1, 3 },
  { 9, 4, 7, 2, 0 }
};
```

Is a multi-level array the same thing as a 2D array?

NO

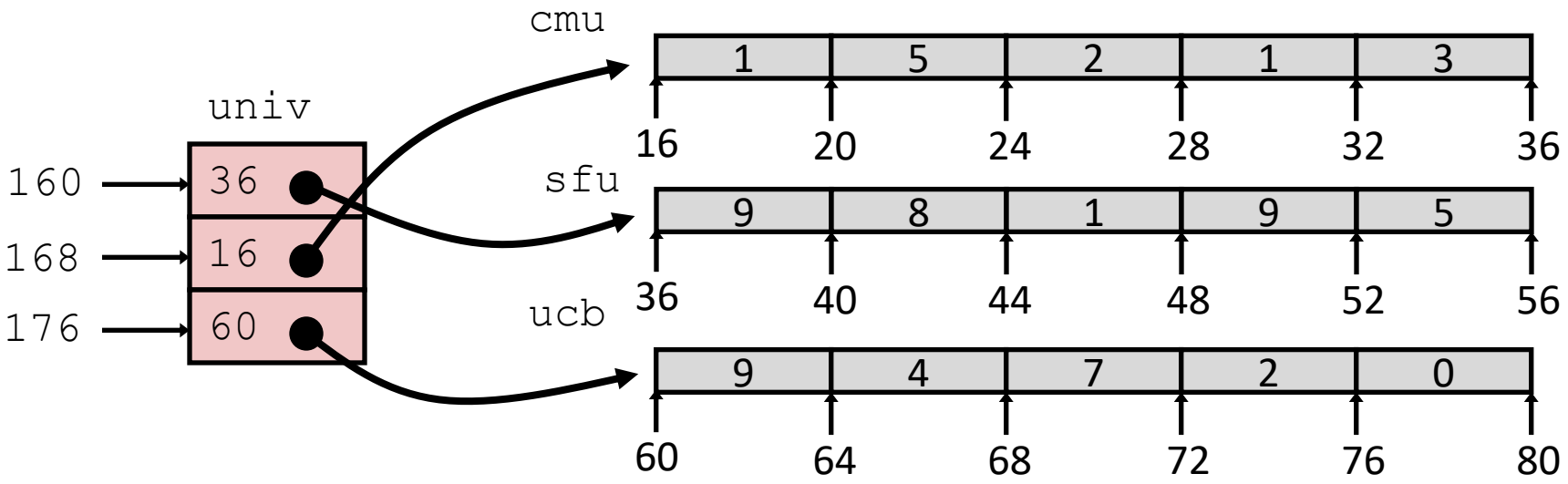
One array declaration = one contiguous block of memory

Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  sfu[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {sfu, cmu, ucb};
```

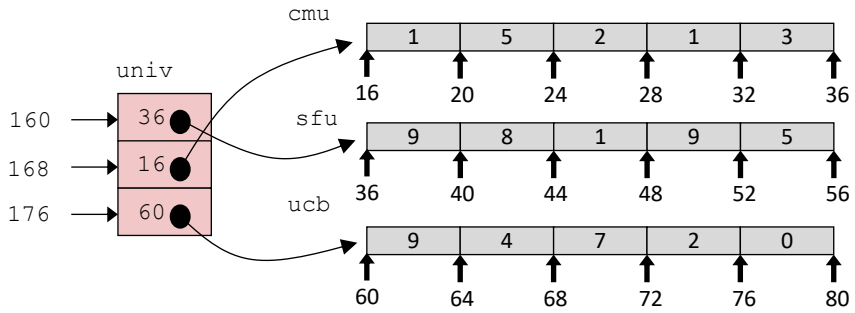
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes each
- Each pointer points to array of `ints`



Note: this is how Java represents multi-dimensional arrays

Element Access in Multi-Level Array

```
int get_univ_digit
   (int index, int digit)
{
   return univ[index][digit];
}
```



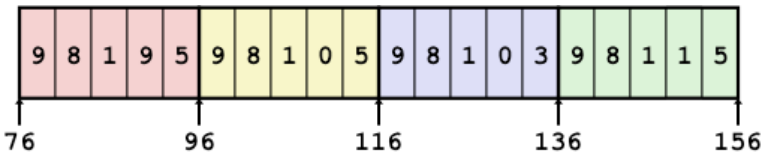

```
slli a0, a0, 3    # All registers are assumed to be 64 bit. a5 = &univ[0]
add a5,a5,a0
ld a5,0(a5)       # 64 bit load. This is because univ is a pointer array.
slli a1,a1,2      # 4*digit
add a5,a5,a1      # univ[index]+4*digit
lw a0,0(a5)       # 32 bit load
```

- Element access Mem[Mem[`univ+8*index`]+`4*digit`]
- Must do **two memory reads**
  - First get pointer to row array
  - Then access element within array
- But allows inner arrays to be different lengths (not in this example)

Array Element Accesses

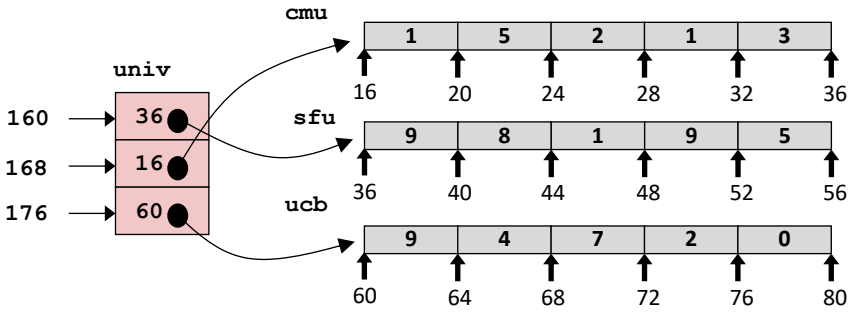
Nested array

```
int get_sea_digit
  (int index, int digit)
{
  return sea[index][digit];
}
```



Multi-level array

```
int get_univ_digit
  (int index, int digit)
{
  return univ[index][digit];
}
```

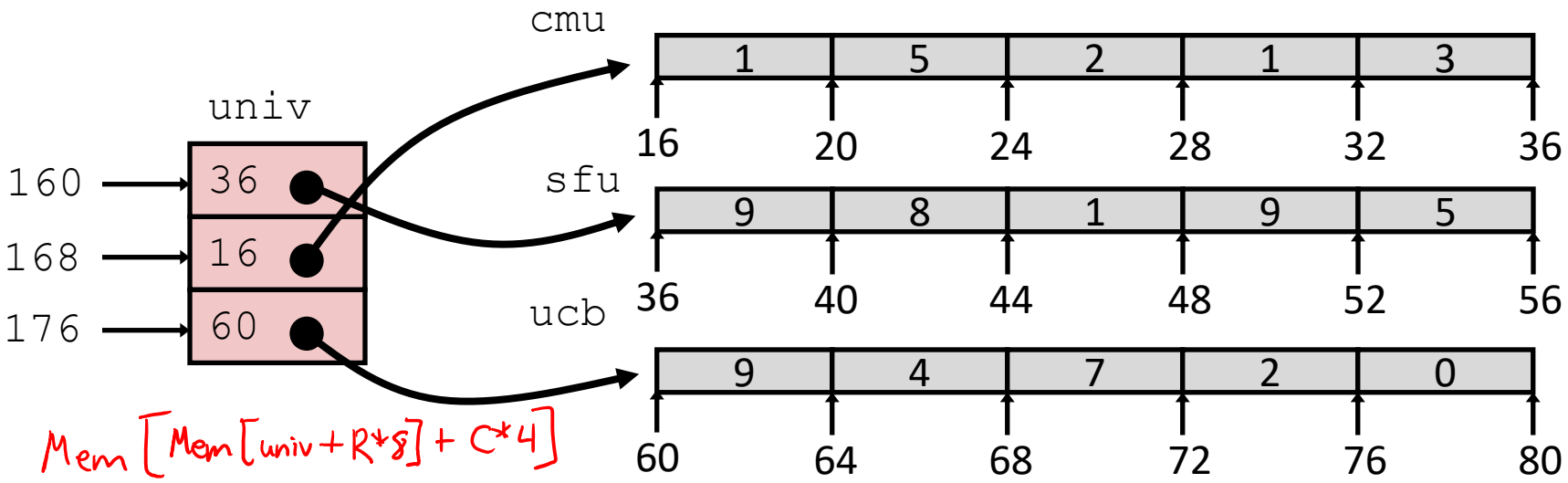


Access *looks* the same, but it isn't:

Mem[sea+20*index+4*digit]

Mem[Mem[univ+8*index]+4*digit]

Multi-Level Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	Mem[176]+3*4 = 60+12 = 72	2	Yes
`univ[1][5]`	Mem[168]+5*4 = 16+20 = 36	9	No
`univ[2][-2]`	Mem[176]+(-2)*4 = 60-8 = 52	5	No
`univ[3][-1]`	Mem[184]+(-1)*4 = ??-4 = ??	???	No
`univ[1][12]`	Mem[168]+12*4 = 16+48 = 64	4	No

- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed

```
typedef int zip_dig[5];
```

Array Loop Example

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

```
zi = 10*0 + 9 = 9
```

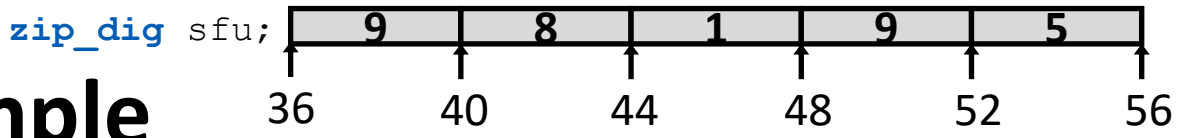
```
zi = 10*9 + 8 = 98
```

```
zi = 10*98 + 1 = 981
```

```
zi = 10*981 + 9 = 9819
```

```
zi = 10*9819 + 5 = 98195
```

9	8	1	9	5



Array Loop Example

- Original:

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- Transformed:
  - Eliminate loop variable `i`, use pointer `zend` instead
  - Convert array code to pointer code
    - Pointer arithmetic on `z`
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 5;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z < zend);
  return zi;
}
```

address just past 5th digit

Increments by 4 (size of int)

Array Loop Implementation

gcc with -O1

```
zd2int(int*): # @zd2int(int*)
a3=10,a4=20
mv a2, a0
add a4, a0, a4  # a4=zend
# a0 = Base address
# a2 = Loop variable
.LBB0_1: Loop
lw a5, 0(a2)    # *z
mul a1, a1, a3  # 10 * zi
add a1, a1, a5  # 10 * zi + *z
addi a2, a2, 4  # increment pointer
z++
bne a2, a4, .LBB0_1  # z < zend
add a0, zero, a1
ret
```

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 5;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z < zend);
  return zi;
}
```