## 2.15 Arrays versus pointers

This section has been set as optional by your instructor.

A challenge for any new C programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and LEGv8 assembly versions of two procedures to clear a sequence of doublewords in memory: one using array indices and one with pointers. The figure below shows the two C procedures.

Figure 2.15.1: Two C procedures for setting an array to all zeros (COD Figure 2.31).

`clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by `&`, and the object pointed to by a pointer is indicated by `*`. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the bottom part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate LEGv8 instructions to increment p by eight, the number of bytes in an LEGv8 integer. The assignment in the loop places 0 in the object pointed to by `p`.

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

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

The purpose of this section is to show how pointers map into LEGv8 instructions, and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

### Array version of clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `X0` and `X1`, and that `i` is allocated to register `X9`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
       MOV X9, XZR              // i = 0 (register X9 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 8 to get the byte address:

```
loop1: LSL X10, X9, #3          // X10 = i * 8
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
       ADD X11, X0, X10         // X11 = address of array[i]
```

Finally, we can store 0 in that address:

```
       STUR XZR, [X11, #0]       // array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
       ADDI X9, X9, #1          // i = i + 1
```

The loop test checks if i is less than size:

```
       CMP   X9, X1             // compare i to size
       B.LT  loop1              // if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the LEGv8 code for clearing an array using indices:

```
       MOV   X9, XZR             // i = 0
loop1: LSL   X10, X9, #3         // X10 = i * 4
       ADD   X11, X0, X10        // X11 = address of array[i]
       STUR  XZR, [X11, #0]      // array[i] = 0
       ADDI  X9, X9, #1          // i = i + 1
       CMP   X9, X1              // compare to size
       B.LT  loop1               // if (i < size) gotoloop1
```

(This code works as long as `size` is greater than 0; ANSI C requires a test of size before the loop, but we'll skip that legality here.)

### Pointer version of clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `X0` and `X1` and allocates `p` to register `X9`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
       MOV X9, X0              // p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into `p`:

```
loop2: STUR XZR, [X9,#0]       // Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes `p` to point to the next doubleword:

```
       ADDI X9, X9, #8         // p = p + 8
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers declared as `long long int`, each of which uses 8 bytes, the compiler increments p by 8.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 8 to get its byte address:

```
       LSL X10, X1, #3         // X10 = size * 8
```

and then we add the product to the starting address of the array to get the address of the first doubleword *after* the array:

```
       ADD X11, X0, X10        // X11 = address of array[size]
```

The loop test is simply to see if `p` is less than the last element of `array`:

```
       CMP X9, X11             // compare p to &array[size]
       B.LT loop2              // if (p < &array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
       MOV X9, X0              // p = address of array[0]
loop2: STUR XZR, [X9,#0]       // Memory[p] = 0
       ADDI X9, X9, #8         // p = p + 8
       LSL X10, X1, #3         // X10 = size * 8
       ADD X11, X0, X10        // X11 = address of array[size]
       CMP X9, X11             // compare p to &array[size]
       B.LT loop2              // if (p < &array[size]) go to loop2
```

As in the first example, this code assumes `size` is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
       MOV X9, X0              // p = address of array[0]
       LSL X10, X1, #3         // X10 = size * 8
       ADD X11, X0, X10        // X11 = address of array[size]
loop2: STUR XZR, 0[X9,#0]      // Memory[p] = 0
       ADDI X9, X9, #8         // p = p + 8
       CMP X9, X11             // compare p to &array[size]
       B.LT loop2              // if (p < &array[size]) go to loop2
```

### Comparing the two versions of clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
      MOV   X9,XZR       // i = 0                MOV   X9,X0        // p = & array[0]
loop1:LSL   X10,X9,#3    // X10 = i * 8          LSL   X10,X1,#3    // X10 = size * 8
      ADD   X11,X0,X10   // X11 = &array[i]      ADD   X11,X0,X10   // X11 = &array[size]
      STUR  XZR,[X11,#0] // array[i] = 0   loop2:STUR  XZR,[X9,#0]) // Memory[p] = 0
      ADDI  X9,X9,#1     // i = i + 1            ADDI  X9,X9,#8     // p = p + 8
      CMP   X9,X1        // compare i to size    CMP   X9,X11       // compare p to &array[size]
      B.LT  loop1        // if () go to loop1    B.LT loop2  // if (p < &array[size]) go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index. The memory pointer version on the right increments the pointer `p` directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from six to four. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops). COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) describes these two and many other optimizations.

#### Elaboration

*As mentioned earlier, a C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a branch just before the first instruction of the loop to the `CMP` instruction.*

**PARTICIPATION ACTIVITY** 2.15.1: Using pointers in assembly language.

1) In C, a possible way to declare an integer array is: int *array.

○ True
○ False

2) If int* array is allocated to register X0 and int* p is allocated to register X9, the proper way to get p to point to array[0] is: `MOV X0, X9`.

○ True
○ False

3) The instruction `ADDI X9, X9, #1` can be used either to increment an

integer or to increment a pointer to an integer.

○ True

○ False

4) If an array's size is 5 elements, and int size has a value of 5, &array[size] returns the address of the first doubleword after the array.

○ True

○ False

5) If int *p is allocated to register X9 and &array[size-1] is allocated to register X11, then the proper loop test involves `CMP X9, X11`.

○ True

○ False

## Understanding program performance

People were once taught to use pointers in C to get greater efficiency than that available with arrays: "Use pointers, even if you can't understand the code". Modern optimizing compilers can produce code for the array version that is just as good. Most programmers today prefer that the compiler do the heavy lifting.

Provide feedback on this section