3.2: Arrays of integers in MIPS assembly language

Topics:
arrays of integers
direct calculation of addresses of array elements
accessing array elements using pointers

Introduction:

In the last exercise, we saw some examples of C++ programs that accessed arrays sequentially. Now we'll study similar programs written in MIPS assembly language.

Remember that in C++, if a pointer ptr points to an array element, and we want to move it so that it points to the next array element, all we have to do is specify ptr++, for both arrays of characters and integers. So in C++, it looks like the amount by which we increment the pointer is automatically determined! This is because the compiler sees what type of variable the pointer is pointing to, and automatically adds the correct amount to the pointer. However, since there are no data types in assembly language, the programmer has to know what to add to the pointer.

Steps:

Previously, we saw arr1.cc, which performs a vector sum of two integer arrays. arr1.s is a MIPS program that does approximately the same thing:

# arr1.s: MAL program for computing vector sum of two 5-element arrays

.data
x:    .word 0:5
y:    .word 0:5
sum:  .word 0:5

prmpt1: .asciiz "Please enter 5 integers for first array:"
prmpt2: .asciiz "Please enter 5 integers for second array:"
spaces: .asciiz "  
newline:.asciiz "\n"

.text
.globl rd_loop
.globl rd_loop1
.globl loop
main:
    la   $a0,prmpt1
    li   $v0,4
    syscall

# i     $s0
# &x[0] $s1

    li   $s0,0

    la   $s1,x  # for (i=0; i<5;i++)
    rd_loop:li  $v0,5  # cin >> x[i];
        syscall
    mul  $t0,$s0,4
    add  $t0,$s1,$t0
    sw   $v0,(t0)
    add  $s0,$s0,1
    blt  $s0,5,rd_loop

    li   $s0,0
    la   $s1,y

    la   $a0,prmpt2
    li   $v0,4
    syscall

# i     $s0
# &y[0] $s1

    rd_loop1:li  $v0,5  # cin >> y[i];
        syscall
    mul  $t0,$s0,4
    add  $t0,$s1,$t0
    sw   $v0,(t0)
    add  $s0,$s0,1
    blt  $s0,5,rd_loop1

    li   $s0
# &x[0] $s1
# &y[0] $s2
# &sum[0] $s3
# &sum[i] $t2
# 4*i $t3
li $s0,0
la $s1,x
la $s2,y
la $s3,sum

loop:   mul $t3,$s0,4
        add $t0,$s1,$t3  # $t0 = &x[i];
        add $t1,$s2,$t3  # $t1 = &y[i];
        add $t2,$s3,$t3  # $t2 = &sum[i];
        lw $t4,($t0)     # sum[i] = x[i] + y[i];
        lw $t5,($t1)
        add $t4,$t5,$t4
        sw $t4,($t2)
        add $s0,$s0,1
        blt $s0,5,loop

li $s0,0
la $a0,newline
li $v0,4
syscall

la $t0,sum
wr_loop:move $a0,$s0  # for (i=0;i<5;i++)
        li $v0,1  # cout << i << " " << sum[i] <<
syscall

la $a0,spaces
li $v0,4
syscall

mul $t1,$s0,4
add $t1,$t1,$t0
lw $a0,($t1)
li $v0,1
syscall

la $a0,newline
li $v0,4
syscall

add $s0,$s0,1
blt $s0,5,wr_loop
In this version, we translate the array access code faithfully and directly calculate the address of each array element. Once we get the address of an array element, we can access it from memory with `lw` and `sw` instructions (since we have an array of integers). Recall from 256 lectures that for any integer array `x[]`, `&x[i] = &x[0] + i*4`. We will use this equation in our code to handle each array access.

The first loop reads in `x[]`. On iteration `i` of the loop, an integer is read after the `syscall` instruction and appears in `$v0`. The address of `x[i]` is calculated by applying the equation above (`$s1 contains &x[0]`, and `$t0 contains i*4`), and placed in `$t0`. Finally, the input integer is stored in `x[i]` with the `sw` instruction.

```assembly
    li      $s0,0
    la      $s1,x
    rd_loop: li      $v0,5           #    cin >> x[i];
              syscall
              mul     $t0,$s0,4
              add     $t0,$s1,$t0
              sw      $v0,($t0)
              add     $s0,$s0,1
              blt     $s0,5,rd_loop
```

Similarly, the second loop reads in `y[]`:

```assembly
    # i     $s0
    # &y[0] $s1
    rd_loop1: li      $v0,5           #    cin >> y[i];
              syscall
              mul     $t0,$s0,4
              add     $t0,$s1,$t0
              sw      $v0,($t0)
              add     $s0,$s0,1
              blt     $s0,5,rd_loop1
```
Next we look at the code for implementing \( \text{sum}[i] = x[i] + y[i] \). In each iteration, we need to compute \( &x[i] \), \( &y[i] \), and \( &\text{sum}[i] \). We first place \( &x[0] \), \( &y[0] \) and \( &\text{sum}[0] \) in some registers (and initialize \( i \), which is \$s0\):

\[
\begin{align*}
\text{li} & \quad \text{\$s0,0} \\
\text{la} & \quad \text{\$s1,x} \\
\text{la} & \quad \text{\$s2,y} \\
\text{la} & \quad \text{\$s3,sum}
\end{align*}
\]

In each loop iteration, we first compute \( 4*i \), then calculate \( &x[i] \) (\$t0 \), \( &y[i] \) (\$t1 \) and \( &\text{sum}[i] \) (\$t2 \):

\[
\text{# for (i=0; i<5; i++)}{
\begin{align*}
\text{mul} & \quad & \text{\$t3,\$s0,4} \\
\text{add} & \quad & \text{\$t0,\$s1,\$t3} \quad \# \quad \text{\$t0 = \&x[i];} \\
\text{add} & \quad & \text{\$t1,\$s2,\$t3} \quad \# \quad \text{\$t1 = \&y[i];} \\
\text{add} & \quad & \text{\$t2,\$s3,\$t3} \quad \# \quad \text{\$t2 = \&\text{sum}[i];}
\end{align*}
\}
\]

Then we load \( x[i] \) and \( y[i] \) from memory into registers, add them, and store the result into \( \text{sum}[i] \):

\[
\begin{align*}
\text{lw} & \quad \text{\$t4,(\$t0)} \quad \# \quad \text{sum}[i] = x[i] + y[i]; \\
\text{lw} & \quad \text{\$t5,(\$t1)} \\
\text{add} & \quad \text{\$t4,\$t5,\$t4} \\
\text{sw} & \quad \text{\$t4,(\$t2)} \\
\text{add} & \quad \text{\$s0,\$s0,1} \\
\text{blt} & \quad \text{\$s0,5,loop} \quad \#}
\end{align*}
\]

The output loop \text{wr_loop} is similar in principle, though it’s much more cluttered because there are a lot of little substrings to print. Look through it yourself to make sure you follow what’s going on.

Run the program yourself to make sure it works. Then we'll trace through the program using breakpoints:

\[
\begin{align*}
\text{(spim)} & \quad \text{lo "arr1.s"} \\
\text{(spim)} & \quad \text{bre loop} \\
\text{(spim)} & \quad \text{run}
\end{align*}
\]

Please enter 5 integers for first array: 1
2
3
4
5
Please enter 5 integers for second array:
6
7
8
9
10
Breakpoint encountered at 0x004000b8
(spim)

We're stopped at the first breakpoint loop. Both x[] and y[] have been read. $s0 (i) is initialized to 0, $s1 has &x[0], $s2 has &y[0], and $s3 has &sum[0]:

(spim) pr $s0
Reg 16 = 0x00000000 (0)
(spim) &x[0] = 0x10010000
Reg 17 = 0x10010000 (268500992)
(spim) &y[0] = 0x10010014
Reg 18 = 0x10010014 (268501012)
(spim) &sum[0] = 0x10010028
Reg 19 = 0x10010028 (268501032)

(spim) pr 0x10010000
Reg 16 = 0x00000000 (0)
(spim) x[0] = 1
Data seg @ 0x10010000 (268500992) = 0x00000001 (1)
(spim) x[1] = 2
Data seg @ 0x10010004 (268500996) = 0x00000002 (2)
(spim) x[2] = 3
Data seg @ 0x10010008 (268501000) = 0x00000003 (3)
(spim) x[3] = 4
Data seg @ 0x1001000c (268501004) = 0x00000004 (4)
(spim) x[4] = 5
Data seg @ 0x10010010 (268501008) = 0x00000005 (5)

(spim) pr 0x10010014
Reg 16 = 0x00000000 (0)
(spim) y[0] = 6
Data seg @ 0x10010014 (268501012) = 0x00000006 (6)
(spim) y[1] = 7
Data seg @ 0x10010018 (268501016) = 0x00000007 (7)
(spim) y[2] = 8
Data seg @ 0x1001001c (268501020) = 0x00000008 (8)
(spim) y[3] = 9
Data seg @ 0x10010020 (268501024) = 0x00000009 (9)
(spim) y[4] = 10
Data seg @ 0x10010024 (268501028) = 0x0000000a (10)

In iteration 0, we compute i*4 = 0, and calculate &x[i], &y[i] and &sum[i] by adding i*4 to &x[0], &y[0], and &sum[0] (note that the mul instruction splits into two machine language instructions:
Then we load \( x[0] \) and \( y[0] \), compute the sum, store in \( \text{sum}[0] \), and move on to the next iteration. We’ll check that \( \text{sum}[0] \) (at \( 0x10010028 \)) = 1 + 6:

```
(spim) step
[0x004000c4] 0x024b4820 add $9, $18, $1       ; 73: add
$\text{t1},$s2,$\text{t3}     # $\text{t1} = \&y[i];
(spim)
[0x004000c8] 0x026b5020 add $10, $19, $11      ; 74: add
$\text{t2},$s3,$\text{t3}     # $\text{t2} = \&\text{sum}[i];
```

```
Data seg @ 0x10010028 (268501032) = 0x00000007 (7)
(spim) pr 0x10010028
```

Then we load \( x[0] \) and \( y[0] \), compute the sum, store in \( \text{sum}[0] \), and move on to the next iteration. We’ll check that \( \text{sum}[0] \) (at \( 0x10010028 \)) = 1 + 6:
In iteration 1, we compute i*4 = 4, and calculate \&x[i], \&y[i] and \&sum[i] by adding i*4 to \&x[0], \&y[0], and \&sum[0]:

```
(spim) step
[0x004000b8] 0x34010004 ori $1, $0, 4 ; 71:
mul $t3,$s0,4
(spim) [0x004000bc] 0x72015802 mul $11, $16, $1
(spim) [0x004000c0] 0x022b4020 add $8, $17, $11 ; 72:
add $t0,$s1,$t3 # $t0 = \&x[i];
(spim) [0x004000c4] 0x024b4820 add $9, $18, $11 ; 73:
add $t1,$s2,$t3 # $t1 = \&y[i];
(spim) [0x004000c8] 0x026b5020 add $10, $19, $11 ; 74:
add $t2,$s3,$t3 # $t2 = \&sum[i];
(spim) pr $t0
Reg 8 = 0x10010004 (268500996)
(spim) pr $t1
Reg 9 = 0x10010018 (268501016)
(spim) pr $t2
Reg 10 = 0x1001002c (268501036)
(spim)
```

Then we load x[1] and y[1], compute the sum, store in sum[1], and move on to the next iteration. We'll check that sum[1] (at 0x1001002c) = 2 + 7:

```
(spim) step
[0x004000cc] 0x8d0c0000 lw $12, 0($8) ; 76:
lw $t4,($t0) # \sum[i] = x[i] + y[i];
(spim) [0x004000d0] 0x8d2d0000 lw $13, 0($9) ; 77:
lw $t5,($t1)
(spim) [0x004000d4] 0x01ac6020 add $12, $13, $12 ; 78:
add $t4,$t5,$t4
(spim) [0x004000d8] 0xad4c0000 sw $12, 0($10) ; 79:
sw $t4,($t2)
(spim) pr 0x1001002c
Data seg @ 0x1001002c (268501036) = 0x00000009 (9)
```
(spim) **step**
[0x0040000c] 0x22100001  addi $16, $16, 1 ; 80:
add   $s0,$s0,1

(spim)
[0x004000e0] 0x2a010005  slti $1, $16, 5                 ; 81:
blt   $s0,5,loop # } }

(spim)
[0x004000e4] 0x1420f ff5  bne $1, $0, -44 [loop-0x004000e4]

The other iterations are very similar; we’ll just continue and finish:

(spim) **cont**
0  7
1  9
2 11
3 13
4 15

(spim) **pr** 0x10010028
Data seg @ 0x10010028 (268501032) = 0x00000007 (7)

(spim) Data seg @ 0x1001002c (268501036) = 0x00000009 (9)

(spim) Data seg @ 0x10010030 (268501040) = 0x0000000b (11)

(spim) Data seg @ 0x10010034 (268501044) = 0x0000000d (13)

(spim) Data seg @ 0x10010038 (268501048) = 0x0000000f (15)

Notice that in this program, we actually use the index $i$ to compute the address of each array element that we accessed. For example,

address of $x[i] = \text{base address of } x + i \times 4$

Then we store the address of $x[i]$ in a register, and use that register as a pointer to access the array element.

More common practice in low-level code is to use pointers to point to each array element, and access the array sequentially by incrementing (or decrementing) the pointers. We already saw this in **arr1_ptr.cc**, which we traced through in some detail.

**arr1_ptr.s** is a MIPS version of **arr1.s** that uses pointers for sequential array access. Once again, I've omitted lines of code in the main compute loop for you to fill in:

# **arr1_ptr.s**: MIPS program for computing vector sum of two 5-
element arrays,
# sequential array access using pointers

.data
x: .word 0:5
y: .word 0:5
sum: .word 0:5
prmpt1: .asciiz "Please enter 5 integers for first array:
prmpt2: .asciiz "Please enter 5 integers for second array:
spaces: .asciiz " "
newline:.asciiz "\n"

.text
main:
   la    $a0,prmpt1
   li    $v0,4
   syscall
   # i   $s0
   # ptr_x $s1
   li    $s0,0
   la    $s1,x            # ptr_x = &x[0];
rd_loop:                        # for (i=0; i<5;i++) {
   li    $v0,5           # cin >> *ptr_x;
   syscall
   sw    $v0,($s1)
   add   $s1,$s1,4       # ptr_x++;
   add   $s0,$s0,1
   blt   $s0,5,rd_loop   # }
   blt   $s0,5,rd_loop   # }
   la    $a0,prmpt2
   li    $v0,4
   syscall
   # i   $s0
   # ptr_y $s1
   li    $s0,0
   la    $s1,y
rd_loop1:  
   li    $v0,5           # cin >> *ptr_y;
syscall
sw $v0,$s1
add $s1,$s1,4  # ptr_y++;
add $s0,$s0,1
blt $s0,5,rd_loop1
# i $s0
# ptr_x $s1
# ptr_y $s2
# ptr_s $s3
li $s0,0
la $s1,x
la $s2,y
la $s3,sum

# for (i=0; i<5;i++) {
loop:

#    *ptr_s = *ptr_x + *ptr_y;

#    ptr_x++;
#    ptr_y++;
#    ptr_s++;
add $s0,$s0,1
blt $s0,5,loop
# }

# i $s0
# ptr_s $s1
la $a0,newline
li $v0,4
syscall
li $s0,0
la $s1,sum  # ptr_s = sum;
wr_loop:move $a0,$s0  # for (i=0;i<5;i++) {
li $v0,1  # cout << i << " " << *ptr_s;
syscall
la $a0,spaces
li $v0,4
syscall
lw $a0,($s1)
li $v0,1
syscall
la $a0,newline
li $v0,4
syscall
add     $s1,$s1,4       #    ptr_s++;
add     $s0,$s0,1       #}
blt     $s0,5,wr_loop
li      $v0,10
syscall

We’ll look at the first loop that reads in x[]. $s1 (ptr_x) is initialized with &x[0]. On each iteration, we read an integer, store it in *ptr_x with sw $v0,($s1), and increment $s1 (ptr_x) so it points to the next integer array element. In iteration 0, $s1 contains &x[0] (0x10010000), in iteration 1, $s1 contains &x[1] (0x10010004), in iteration 2 $s1 contains &x[2] (0x10010008) and so on.

li      $s0,0
la      $s1,x           # ptr_x = &x[0];

rd_loop:                        # for (i=0; i<5;i++) {
    li      $v0,5           #    cin >> *ptr_x;
syscall
    sw      $v0,($s1)
    add     $s1,$s1,4       #    ptr_x++;
    add     $s0,$s0,1
    blt     $s0,5,rd_loop   # }

The read loop for y[] is (of course) very similar. You will need to fill in the code for the main compute loop, in particular the statement *ptr_s = *ptr_x + *ptr_y, and the pointer increments. Before the loop begins, ptr_x ($s1) is initialized to &x[0], ptr_y ($s2) to &y[0], and ptr_s ($s3) to &sum[0]. In each iteration, you’ll need to 1) dereference ptr_x to get x[i], 2) dereference ptr_y to get y[i], 3) add x[i] and y[i], 4) dereference ptr_s to store sum[i], and finally 5) increment ptr_x, ptr_y and ptr_s:

li      $s0,0
la      $s1,x
la      $s2,y
la      $s3,sum
    # for (i=0; i<5;i++) {
    loop:                           #    *ptr_s = *ptr_x + *ptr_y
        #    ptr_x++;
        #    ptr_y++;
        #    ptr_s++;
    add     $s0,$s0,1
    blt     $s0,5,loop   # }
If you’re having problems with this part, check the solution in

~wusu/csc256/LABS/PROGS/arr1_ptr.soln

You can also access the same file via a web browser:

http://unixlab.sfsu.edu/~wusu/csc256/LABS/PROGS/arr1_ptr.soln