Week 9, part C: Arrays and Structs

Arrays!

- A sequence of data elements of same size which is contiguous in memory (i.e. no spaces) .
- B is an array of 9 bytes starting at address 8:

Address:	8	9	10	11	12	13	14	15	16
	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]

- H is an array of 4 half-words starting at address 8:

<table>
<tr><td>Address:</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td></td><td colspan="2">H[0]</td><td colspan="2">H[1]</td><td colspan="2">H[2]</td><td colspan="2">H[3]</td></tr>
</table>

Arrays in Assembly

```
int A[100];
...
A[i] = 123;
...
```

- In assembly arrays are just a range of memory.
- Access arrays using address of the first element.
- To access element i in the array:
  - Start with the address of the first element
  - Add an offset (distance) in bytes from that address.
  - address of i = address of first element + i * (size of an element)
- Example: address of `H[3]` = address of `H[0] + 3*2`

Address:	8	9	10	11	12	13	14	15
	H[0]		H[1]		H[2]		H[3]	

6 bytes = 3 elements

Translate to Assembly

```
int A[100], B[100];
for (i=0; i<100; i++) {
   A[i] = B[i] + 1;
}
```

- Two arrays A, and B, of size 100.
  - Each element in the array is an integer (4 bytes)
- We set $A_i = B_i + 1$
  - i goes from 0 to 99

Translating arrays

```c
int A[100], B[100];
for (i=0; i<100; i++) {
   A[i] = B[i] + 1;
}
```

```
.data
A:      .space 400          # array of 100 integers
B:      .word  42:100       # array of 100 integers, all
                            # initialized to value of 42
.text
main:   la $t8, A                   # $t8 holds address of array A
        la $t9, B                   # $t9 holds address of array B
        add $t0, $zero, $zero       # $t0 holds i = 0
        addi $t1, $zero, 100        # $t1 holds 100

LOOP:   bge $t0, $t1, END   # exit loop when i>=100
        sll $t2, $t0, 2     # $t2 = $t0 * 4 = i * 4 = offset
        add $t3, $t8, $t2   # $t3 = addr(A) + i*4 = addr(A[i])
        add $t4, $t9, $t2   # $t4 = addr(B) + i*4 = addr(B[i])
        lw $t5, 0($t4)      # $t5 = B[i]
        addi $t5, $t5, 1    # $t5 = $t5 + 1 = B[i] + 1
        sw $t5, 0($t3)      # A[i] = $t5
UPDATE: addi $t0, $t0, 1    # i++
        j LOOP              # jump to loop condition check
END:    ...                 # continue remainder of program.
```

Optimizations!

- First, avoid left shift: `sll $t2, $t0, 2`
  - We can increase $t0 by 4 each time
  - Must update stopping condition to be 400 instead of 100
  - Instead of: 0, 1, 2, …, stop at 100
    Do: 0, 4, 8, …, stop at 400

Optimization!

```c
int A[100], B[100];
for (i=0; i<100; i++) {
   A[i] = B[i] + 1;
}
```

```
.data
A:        .space   400                # array of 100 integers
B:        .word    21:100             # array of 100 integers,
                               # all initialized to 21 decimal.
.text
.globl main
main:     la $t8, A                   # $t8 holds address of A
          la $t9, B                   # $t9 holds address of B
          add $t0, $zero, $zero # $t0 holds 4*i; initially 0
          addi $t1, $zero, 400  # $t1 holds 100 * sizeof(int)

LOOP:     bge $t0, $t1, END   # branch if $t0 >= 400
          add $t3, $t8, $t0   # $t3 holds addr(A[i])
          add $t4, $t9, $t0   # $t4 holds addr (B[i])
          lw $t5, 0($t4)      # $t5 = B[i]
          addi $t5, $t5, 1    # $t5 = B[i] + 1
          sw $t5, 0($t3)      # A[i] = $t5
          addi $t0, $t0, 4    # update offset in $t0 by 4
          j LOOP
END:
```

Optimizations!

- Second, avoid extra jump:
  - Move condition to the end, so there is only one branch/jump per iteration.
  - Only works if loop iteration will happen at least once!

Yet Another Optimization

```
.data
A:        .space   400       # array of 100 integers
B:        .space   400       # array of 100 integers
.text
.globl main
main:     add $t0, $zero, $zero      # load "0" into $t0
          addi $t1, $zero, 400       # load "400" into $t1
          addi $t9, $zero, B         # store address of B
          addi $t8, $zero, A         # store address of A

loop:     add $t4, $t8, $t0   # $t4 = addr(A) + i
          add $t3, $t9, $t0   # $t3 = addr(B) + i
          lw $s4, 0($t3)      # $s4 = B[i]
          addi $t6, $s4, 1    # $t6 = B[i] + 1
          sw $t6, 0($t4)      # A[i] = $t6
          addi $t0, $t0, 4    # $t0 = $t0++
          bne $t0, $t1, loop # branch back if $t0<400
end:
```

Optimizations!

- First, avoid left shift: `sll $t2, $t0, 2`
  - We can increase `$t0` by 4 each time
  - Must update stopping condition to be 400 instead of 100
- Second, avoid extra jump:
  - Move condition to the end, so there is only one branch/jump per iteration.
  - Only works if loop iteration will happen at least once!
- Compilers do this and more for us all the time!

Strings

- What is a C string?
  - Array of `chars` (bytes).
  - Each `char` is ASCII code of one character
  - The value `0` (a.k.a NUL character) at the end of the string indicates this is the end.

`“Hi there"`

'H'	'i'		't'	'h'	'e'	'r'	'e'	0

- Other names: ASCIIZ or null-terminated string
  - Because it ends with Zero…

String in Assembly

- Use `.asciiz` storage directive
- Use system call 4 with the address in $a0
  - Add newline `'\n'` manually to move to next line.

```
.data
str1:  .asciiz "My hovercraft is full of eels\n"

.text
.globl main
main:  li $v0, 4
       la $a0, str1
       syscall

       # End program
       li $v0, 10
       syscall
```

Structs

- Structs are simply a collection of fields one after another in memory
- Assembly does not understand structs
  - But load/store instructions allow fixed offset!

```
struct {
      int a;
      int b;
      int c;
} s;

s.a = 5;
s.b = 13;
s.c = -7;
```

Example: A struct program

```
struct {
        int a;
        int b;
        int c;
} s;
```

address
s+0
s+4
s+8

```
.data
s:      .space   12
.text
.globl main
main:   la      $t0, s
        addi    $t1, $zero, 5
        sw      $t1, 0($t0)
        addi    $t1, $zero, 13
        sw      $t1, 4($t0)
        addi    $t1, $zero, -7
        sw      $t1, 8($t0)
```

- s.a is at the beginning of s
- s.b is after s.a, at address (s.a) + 4
  - Since s.a is int
- s.c is after s.b, so it is at address(s.a) + 8
  - Since s.a and s.b are ints

Alignment + Struct

- Remember we have alignment constraints.
- In this example, cannot store **c** immediately after **a**
  - If `a` is in address `0x1000`, `c` will be in address `0x1001`
  - `0x1001` is not word-aligned.
  - Will cause exception!
- What to do?

```
struct {
        char a;
        int c;
} s;

s.a = 5;
s.c = -7;
```

Padding

- Add padding: empty (unused) bytes between `a` and `c`.
- Add just enough padding after `a` until `c` is correctly aligned for its type.
- Size of struct `s` is therefore 8 bytes.
  - We also make sure the struct initial address is word-aligned.

```
struct {
        char a;
        int c;
} s;

s.a = 5;
s.c = -7;
```

Address	Contents
0x1000	a
0x1001	padding
0x1002	padding
0x1003	padding
0x1004	c
0x1005	c
0x1006	c
0x1007	c

Functions vs Code

- Up to this point, we've been looking at how to create pieces of code in isolation.
- A function is an interface to this code by defining the input and output parameters.
- How do we write one in assembly?
- And how do we call it?
- Move to next part!

```
int sign (int n) {
    if (n > 0)
        return 1;
    else if (n == 0)
        return 0;
    else
        return -1;
}

int x, y, r1, r2;
x = -42;
y = x*x;
r1 = sign(x);
r2 = sign(y);
r = r + r;
…
```