CSE 131 – Compiler Construction

Discussion 7: Short-Circuiting, Loops, Pointers, and Arrays/Structs

2/22/2010

2/26/2010

Overview

- Phase 2
- Some of Phase 3

Short-Circuiting

- && and || are "short-circuiting" operators.
  - In A && B, if A evaluates to false, B is not evaluated.
  - In A || B, if A evaluates to true, B is not evaluated.

Short-Circuiting

- Think of how you handle an if-else statement.
- Short-circuiting follows the same principle:
  - In the A && B case:
    - if not A then false, else B
  - In the A || B case:
    - if A then true, else B

Short-Circuiting

- RC: bool c = a && b:

```
! Load a and check if false
set   a, %l0
ld    [%l0], %l0
cmp   %l0, %g0
be    flabel
nop
! a is true, so check b
set   b, %l0
ld    [%l0], %l0
cmp   %l0, %g0
be    flabel
nop
    ! b is true, so result is true
    mov  1, %l5
    ba    endlabel
    nop

flabel:
    mov  0, %l5    ! Result is now in %l5
endlabel:
    set    c, %l0
    st     %l5, [%l0]
```

While Loops

- Also similar to an If-Else statement
- You need to:
  - Branch to the end of the loop
  - Check the loop condition
  - If true, branch back into the loop body
- Also, to handle break/continue statements, you will most likely need a Loop Label Stack just for the loops.

While Loops

- Consider the following in RC:

  while (x < 5) { x = x + 1; }

```
    ba    test
    nop

loop:
    set   x, %l0
    ld    [%l0], %l1
    inc   %l1
    st    %l1, [%l0]

test:
    set   x, %l0
    ld    [%l0], %l0
    set   5, %l1
    cmp   %l0, %l1
    bl    loop
    nop

end:      ! Useful label for break
```

Break/Continue Statements

- If you always place a unique "end" label after each of your loop chunks of code, you can simply branch to that label when you encounter an break statement. Continue just goes to the "test" label where the condition is checked:
  - break → ba endlabel w/ nop
  - continue → ba testlabel w/ nop
- This is where that Loop Label Stack comes in handy, since you just grab the top string and that's the label you want to branch to

Array/Struct Allocation Method

- When you declare an array, the way you would allocate space for it is to allocate an entire chunk in the BSS and have the variable label at the beginning of it:

  int[7] x;

```
    .section  ".bss"
    .align    4
x:  .skip     28    ! 7 * sizeof(int)
```

Now x[0] is at x+0, x[1] is at x+4, and so on.

Array/Struct Allocation Method

- A useful attribute to have for Arrays and Structs is "size", so you know how much space to allocate for the entire object (should already have this from Project I anyway!).
  - Offsets would also be useful. For arrays, the offsets are simply multiples of the element's size. For structs, the offsets are the collective sizes of the preceding fields.

Array Usage (Simplified)

- a = x[b] + 7;   ! x is array of int

```
set   b, %l0          ! b
ld    [%l0], %l0
sll   %l0, 2, %l0     ! b * 4 → offset
set   x, %l1          ! x → base address
add   %l1, %l0, %l0   ! Base + offset
ld    [%l0], %l0      ! x[b]'s value
add   %l0, 7, %l0     ! x[b] + 7
set   a, %l1
st    %l0, [%l1]      ! a = x[b] + 7
```

Struct Usage

- Very similar to array usage.
- You need to start at the base address of where the struct is located.
- Then, you have to move some offset to get to a specific field you are interested in.
- Once at that location, you either load or store, depending on what you wanted to do.

Passing Arrays

- Arrays must be passed by reference (&), where you pass the address to the beginning of the array.

```
function : void baz (ARRTYPE &a)
baz(myArr);
```

- In the above case, you would store the base address of the first element in %o0. Once in baz, %i0 will have the address of the first element. All other elements will be accessed by some offset from that first element address.

Passing Structs

- Structs must be passed by reference (&), with the address to the beginning of the struct being passed.

```
function : void bar (STRUCTTYPE &r)
bar(myStruct);
```

- In this case, you would store the base address of myStruct in %o0, and in function bar, use the given base address that is in %i0.
  - As you can see, Struct and Array passing are the same.

Value versus Reference

- If you have any doubts about value vs. reference parameters or parameter passing, please look at the following URL:

- http://www.cse.ucsd.edu/users/ricko/CSE131/RefVsValue.pdf

Pointers

- Consider p = q;
  - This is just copying the address that is in q into p.

```
set     q, %l0
ld      [%l0], %l0   ! Get address in q
set     p, %l1
st      %l0, [%l1]   ! Store into p
```

Pointers

- Consider *p = *q;
  - This is getting the actual value at where q is pointing and making where p points that value.

```
set     q, %l0
ld      [%l0], %l0
ld      [%l0], %l0   ! Double load to get value
set     p, %l1
ld      [%l1], %l1
st      %l0, [%l1]   ! Store value
```

Pointers

- New
  - Basically just boils down to a call to calloc to allocate memory on the heap that is zero-initialized
  - Prototype of calloc:
    void *calloc(size_t nelem, size_t elsize);
  - For simplicity, you can say nelem is 1 and that elsize is the size of the object you are allocating
- Delete
  - Basically just a call to “free” with the address:
    void free(void *ptr);
  - Remember to also set the argument to NULL afterwards.

Pointer Return Types

- Don’t forget that functions can return pointer types.
- In this case, you want to place the address (value of the pointer) in the %i0 register.
- That address can then be assigned into another pointer as so:
  - ptr = foo(…);

An Example

```
typedef int* PTRTYPE;
PTRTYPE myGlobal;
function : PTRTYPE foo() {
    PTRTYPE myLocal;
    new myLocal;
    *myLocal = 420;
    return myLocal;
}

function : int main() {
    myGlobal = foo();
    cout << *myGlobal;
    return 0;
}
```

The Result

```
                .section    ".bss"
                 .align     4
myGlobal:.skip              4

        .section            ".rodata"
        .align              4
ifmt: .asciz                "%d"

        .section            ".text"
        .align              4
        .global             foo
foo:
        set     foo.SAVE, %g1
        save    %sp, %g1, %sp

        ! new myLocal
        set     1, %o0      ! numelem
        set     4, %o1      ! sizeof(int)
```

```
        call    calloc      ! new
        nop
        st      %o0, [%fp-4]

        ! *myLocal = 420
        set     420, %l0
        ld      [%fp-4], %l1
        st      %l0, [%l1]

        ! return myLocal
        ld      [%fp-4], %i0
        ret
        restore

foo.SAVE = -(92 + 4) & -8

        .global             main
main:
```

```
        save    %sp, -96, %sp

        ! myGlobal = foo()
        call    foo
        nop
        set     myGlobal, %l7
        st      %o0, [%l7]

        ! cout << *myGlobal
        set     ifmt, %o0
        set     myGlobal, %l7
        ld      [%l7], %l0
        ld      [%l0], %o1
        call    printf
        nop
        mov     %g0, %i0
        ret
        restore
```

What to do Next!

1. Finish Phase 2.
2. Start of Phase 3.
3. Thoroughly test and re-test Phase 1, 2, and 3.
4. Come to lab hours and ask questions.

Topics/Questions you may have

- Anything else you would like me to go over now?
- Anything in particular you would like to see next week?