CSE 131 – Compiler Construction

Discussion 8: Miscellaneous Stuff

03/01/2010

03/05/2010

Address-Of

- ptr = &x;
  - Since "x" would be addressable, take the address of "x" and store it in ptr (as opposed to the value of "x").

Address-Of

- ptr = &x;

```
set     x, %l0
set     ptr, %l1
st      %l0, [%l1]
```

Function Pointers

- Remember that all we need to call a function is the unique name to give the "call" instruction.
- For function pointers, you should have the name readily available at compile-time, so just output what the current function name is for the function pointer. Additionally, you can "call" on a register holding an address.
- Struct member functions can be also be assigned to function pointers: 8-byte function pointers

Function Pointers

```
function : void foo() {  /*  */ }
function : void main() {
  funcptr : void() x;
    x = foo;
    x();
}
```

```
                .section        ".text"
                .align          4
                .global         foo
foo:
                …

                .global         main
main:
                set             main.SAVE, %g1
                save            %sp, %g1, %sp
                set             foo, %l0
                st              %l0, [%fp-12] ! tmp1
                ld              [%fp-12], %l0
                st              %l0, [%fp-4]   ! x in fp-4
                st              %g0, [%fp-8]  ! not a struct function

                ld              [%fp-4], %l0   ! Load x
                call            %l0
                nop

                ret
                restore

main.SAVE = -(92 + 4 + 4) & -8
```

Type Casts

- Depends heavily on work from Project I. In addition to the compile-time conversions, you now have to handle run-time conversions.
- Remember a pointer is essentially an integer, so there is no conversion between pointers to/from int.

Static Variables

- External (in global scope):
  - Just like regular global variables (both initialized and uninitialized)
  - Need to handle runtime initialization (like regular global variables)
- Internal (inside a function):
  - Need to handle initialization.
    - If uninitialized or constant initialized, you can just use BSS or Data.
    - If runtime initialized, need to have special flag memory to only initialize the variable when the function is called for the first time

Static Variables

```
int w;
function : void foo() {
   static int x = w;
   // …
}
```

```
        .section    ".bss"
        .align      4
w:      .skip       4
foo_x:  .skip       4
foo_x_flag: .skip   4
```

```
        .section    ".text"
        .align      4
        .global     foo
foo:
        set         foo.SAVE, %g1
        save        %sp, %g1, %sp
        set         foo_x_flag, %l0
        ld          [%l0], %l0
        cmp         %l0, %g0
        bne         skipInit
        nop
        set         w, %l0
        ld          [%l0], %l0
        set         foo_x, %l1
        st          %l0, [%l1]
        set         1, %l0
        set         foo_x_flag, %l1
        st          %l0, [%l1]
skipInit:
        …
foo.SAVE = -(92 + 0) & -8
```

Extern variables

- What to do for a variable declared "extern":
  - Put it on the symbol table
  - In assembly, access the variable through its name by setting the corresponding label to a register
- So the *only* assembly you generate for extern variables are the instructions you need to read and write to the variable (and of course possibly take address-of)
  - No space for "extern" vars is allocated by the module in which they are declared

Extern vs. static vs. global

Module a	Module b	Link-time error?
`int x;`	`extern int x;`	NO
`static int y;`	`extern int y;`	YES
`extern int z;`	`extern int z;`	YES
`extern int i;`	`int i;`	NO

- Above, it does not really matter which module is your rc.s and which module is the separately linked object module – either case should produce (or not produce) the specified link-time errors

Extern vs. static vs. global

```
int  x;
static int y;
extern int z;

function : void main(){
   cout << x << y << z;
}
```

```
        .section    ".rodata"
intFmt: .asciiz     "%d"
endl:   .asciiz     "\n"
        .section    ".bss"
        .align      4
x:      .skip       4
        .global     x
y:      .skip       4

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

! print global var x
        set         intFmt, %o0
        set         x, %l0
        ld          [%l0], %o1
        call        printf
        nop

!print static var y
        set         intFmt, %o0
        set         y, %l0
        ld          [%l0], %o1
        call        printf
        nop

!print extern var z
        set         intFmt, %o0
        set         z, %l0
        ld          [%l0], %o1
        call        printf
        nop

        ret
        restore
main.SAVE = -(92 + 0) & -8
```

Runtime Checks

- We are providing syntactically and semantically correct code, but there are a few runtime errors you need to check:
  - Array out-of-bounds for array indexes that are unknown at compile time
  - Dereferencing or deleting NULL pointers and "calling" NULL function pointers

Runtime Array Checks

```
int[3] myArr;
int x;
function : int main() {
   x = 47;
   x = myArr[x];
  return 0;
}
```

Runtime Array Checks

- x = myArr[x];

```
   set    x, %l0
   ld     [%l0], %l0        ! Value of x at runtime
   set    3, %l1            ! Dimension of array (from declaration)
   cmp  %l0, %l1            ! Checking upper-bound
   bl     arrayLabel22
   nop

   … Do error msg here and exit program
arrayLabel22:
   … continue checking of lower-bound (>= 0), then get value at myArr[x]
```

Extra Credit

- EC 1: *Deallocated stack space* is defined as the entire region of the RTS that:
  - *Starts* at the **lowest** value of %sp that is reached during program execution
  - *Ends* at the **current** value of %sp+92
  - So, you will need to make memory address comparisons: use **unsigned** branches after the "cmp" instruction: bgu, bgeu, blu, bleu

Extra Credit

- EC 2: Passing more than 6 arguments to functions (more than 5 arguments to struct member functions)
  - See the example in the Project II Specs
  - Basically, need to allocate additional stack space in the caller, copy arguments into that stack space, call the function, deallocate stack space

Extra Credit

- EC 3: Extend function overloading for code generation
  - If you did the EC from Project I, this should be easy.
  - We will not test assigning an overloaded functions to function pointers

What to do Next!

1. Finish Phase 3.
2. Thoroughly test and re-test Phase 1, 2, and 3.
3. Come to lab hours and ask questions.
4. Work on Extra Credit (9%)!!

Topics/Questions you may have

- Anything else you would like me to go over now?