CSE 131 – Compiler Construction

Discussion 6: Operations, Branches and Functions

02/16/2010

02/19/2010

Overview

- Phase 1
- Some Phase 2

Phase 1 –Arithmetic Expressions

- Given:

```
int x; // Global, static, or extern variable
x = x + 7;

set          x, %l0
ld           [%l0], %l0
set          7, %l1
add          %l0, %l1, %l1
st           %l1, [%fp-4]     ! tmp allocated on stack
ld           [%fp-4], %l1
set          x, %l0
st           %l1, [%l0]
```

- **Note**: the assembly is the same regardless of whether `x` is declared global, static, or extern!

Arithmetic Expressions

- OK, that's easy for a simple statement.
- What if we had a statement like this:

$$z = ((a+b) + (c+d)) + ((e+f) + (x+y))$$

- Which registers do we use???

Method 1 – Clumsy

- We can take advantage of Java Strings and encapsulate chunks of assembly code within each ExprSTO.
- This will require a fixed register approach – you will need to have registers that serve a specific purpose (ie, a specific register is the result of an operation, etc).
- Very confusing to keep track of!

Method 1 – Clumsy

- Always put operands into %o0 and %o1
  - Useful for calling .mul, .div, .rem
- Compute operation, and place result in %g1
- Store the resulting assembly code into the ExprSTO, without outputting to the file
- When that ExprSTO is used in another Expression, dump the stored code and make some small register moves.

Method 1 – Example

(a + b) + (x + y)

```
ld    a, %o0
ld    b, %o1
add   %o0, %o1, %g1
```

```
ld    x, %o0
ld    y, %o1
add   %o0, %o1, %g1
```

```
{place code (a+b) here}
mov   %g1, %l0
{place code (x+y) here}
mov   %l0, %o0
mov   %g1, %o1
add   %o0, %o1, %g1
```

Method 2 – Register Allocation

- Have some data structure that lets you know what registers are free and which are currently in use.
- Every time you need a register, request one from the data structure, which will remove that register from the available list
- When you are done with a register, let the data structure know it is available for use again.
- This is a much better method!

Method 2 – Register Allocation

- Make some class (RegClass) with:
  - GetFreeReg() – returns an available register
  - FreeReg(String r) – marks that "r" is available
- You will need to store the allotted register in your ExprSTO so you can reference it later.

Method 3 – Ld/Ld/Ex/St

- The load-load-compute-store method is by far the easiest way to get through this project.
- The drawback is that it is highly inefficient.
- The benefit is that you don't need to remember very much stuff, nor keep track of resources!
- Highly recommend if you are not very familiar with SPARC and just want to get something working!

Methods are your friend!

- Consider adding methods to your VarSTO's that make generating assembly for certain cases easier:
  - GetAddress() – returns base/offset (ie, %fp – 4)
  - GetValue() – will combine GetAddress with an appropriate load instruction
  - Etc.

Conditions – Branching

- Given this:

```
if( b1 ) {
        // statements
}

set     b1, %l0
ld      [%l0], %l0
cmp     %l0, %g0
be      IfL1    ! Opposite logic
nop
// statements here
IfL1:
```

Branching – Where to?

- You will need to generate labels for your branch statements.
  - These labels must be **unique**
- A simple solution would be to use some prefix string (i.e., IfL), and append some counter at the end:
  - IfL1, IfL2, IfL3, …

Branching – Label Stack

- Consider if you had:

```
if( b1 ) {
  if( b2 ) { /*…*/ }
}
```

- You will eventually need some sort of label **stack** to alleviate issues that arise from nested conditions.

Branching – Label Stack

```
if( b1 ) {            – load b1, compare, branch to
L1,        push L1 onto stack
    if( b2 ) {            – load b2, compare,
branch to                     L2, push L2 onto
stack
          /*…*/
    }              - Pop L2 from stack and output
label
}                  - Pop L1 from stack and output
```

Functions

- How to call a function?
  - Ex: call foo
    nop
- How to return from a function?
  - Ex: ret
    restore
- How to return a value from a function?
  - Ex: mov %l0, %i0
    ret
    restore

Functions – Example

```
function : int foo () {
     int x;
     x = 2;
     return x;
}
```

Functions – Example

The following can be generated just by parsing "function : int foo":

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

Functions – Example

Now, the body of the function:

```
        set     2, %l0              ! Put “2” in a reg.
        st      %l0, [%fp-8]        ! tmp1
        ld      [%fp-8], %l0
        st      %l0, [%fp-4]        ! “x” is at %fp-4
        ld      [%fp-4], %i0        ! Put “x” in return
```

Functions – Example

Lastly, now that we got to “}” (end of the function):

```
foo.SIZE = -(92 + 4 + 4) & -8
! Bytes of local vars and tmp vars
```

- By leaving this to the end, you can also allocate extra stack space for intermediate expression storage if needed during the body of the function, like shown in this example.

Functions – What about float?

```
function : float foo () {
        int x;
        x = 2;
        return x;           /* must promote to float */
}
```

Functions – What about float?

```
        .section  ".text"
        .align    4
        .global   foo
foo:
        set       foo.SIZE, %g1
        save      %sp, %g1, %sp
        set       2, %l0            ! Put “2” in a reg.
        st        %l0, [%fp-8]      ! tmp1
        ld        [%fp-8], %l0
        st        %l0, [%fp-4]      ! “x” is at %fp-4
        ld        [%fp-4], %f0      ! Load x into an FP register
        fitos     %f0, %f0          ! Convert bit pattern to FP
                                    ! Now, return value is in %f0 after return
        ret                         ! Return statement
        restore
foo.SIZE = -(92 + 4 + 4) & -8
```

Float Arithmetic

```
float x, y;
function : int main() {
        x = 94.25;
        y = (x + 1) / x;
   cout << y;
   return 0;
}
```

Float Arithmetic

(slightly simplified)

```
        .section   ".bss"
        .align     4
y:      .skip      4
x:      .skip      4
        .global    x, y

        .section   ".text"
        .align     4
        .global    main

main:
        set        SAVE.main, %g1
        save       %sp, %g1, %sp

! switch to "data" to put FP constant
        .section   ".data"
        .align     4
t1:     .single    0r94.25

! switch back to "text"
        .section   ".text"
        .align     4

! x = 94.25
        set        t1, %l0
        ld         [%l0], %f1
        set        x, %l1
        st         %f1, [%l1]

! y = (x + 1) / x;
        set        x, %l0
        ld         [%l0], %f1
        set        1, %l0
        st         %l0, [%fp-4]
        ld         [%fp-4], %f2      ! Promote 1
        fitos      %f2, %f2          !  to a float
        fadds      %f1, %f2, %f1     ! x + 1
        set        x, %l0
        ld         [%l0], %f2
        fdivs      %f1, %f2, %f1
        set        y, %l1
        st         %f1, [%l1]

! cout << y;
        set        y, %l0
        ld         [%l0], %f0
        call       printFloat
        nop

        mov        %g0, %i0
        ret
        restore
SAVE.main = -(92 + 4) & -8
! 4 bytes needed for temporary location
```

What to do Next!

1. Continue planning out how you want to structure your project – good planning leads to an easier design in the long run.
2. Finish Phase 1.
3. Start of Phase 2.
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?