Discussion 5

CSE 131
overview

- operations
- branches
- functions
int x;
x = x + 7;

set x, %l0
add %g0, %l0, %l0
ld [%l0], %l0
set 7, %l1
add %l0, %l1, %l1
set %l0, %l0
add %fp, %l0, %l0
st %l1, [%l0]               ! tmp on stack

set -4, %l0
add %fp, %l0, %l0
ld [%l0], %l1
set x, %l0
add %g0, %l0, %l0
st %l1, [%l0]
arithmetic expressions

- ok, that's easy for a simple statement
- but what if we had something like this?
  - \( z = ((a+b) + (c+d)) + ((e+f) + (x+y)) \)
  - which registers do we use?
The load-load-compute-store method is by far the easiest way to get through this project:
- even though it is highly inefficient

The benefit is that you don't need to remember much stuff, or keep track of resources.
methods are your friend!
consider adding methods to your STOs to make generating assembly easier
  - getAddress() - return or output base/offset assembly code
  - getValue() - combine getAddress() with an appropriate load instruction
  - etc
if (b1) {
    // statements
}

set b1, %l0
ld [%l0], %l0
cmp %l0, %g0
be endif1  ! opposite logic
nop

// statements here

dendif1:

branching, where?

- you will need to generate **unique** labels for your branch statements
- a simple solution is to use some prefix string (e.g. `_ifL`) and append some counter at the end
  - `_ifL1`, `_ifL2`, `_ifL3`, ...
branching, where?

- what if you have something like this?
  - if (b1) {
    if (b2) {/* … */ }
  }

- you will eventually need some sort of label stack to deal with nested conditions
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 label
functions

- how to call?
  - call foo
  - nop

- how to return?
  - ret
  - restore

- how to return value?
  - mov {value, register}, %i0
  - ret
  - restore
function example

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

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

.section " .text"
.align 4
.global foo

foo:
    set SAVE.foo, %g1
    save %sp, %g1, %sp
function example

// now the body of the function

    set  2, %l0          ! put "2" in register
    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

    ret
    restore
function example

- now we're at the end of the function
  - SAVE.foo = -(92 + 4 + 4) & -8
    ! bytes of local vars and tmp vars
- need to do this at the end, since the stack size of the function isn't known until now
what about float?

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

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

foo:  
    set SAVE.foo, %g1
    save %sp, %g1, %sp
    set 2, %l0               ! put "2" in register
    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
    ! leave float returns in %f0

    ret
    restore

SAVE.foo = -(92 + 4 + 4) & -8
```
what about float?

float x, y;
function : int main() {
   x = 9000.01;
   y = (x + 1) / x;
   cout << y;
   return 0;
}


float example (simplified)

```
.float example (simplified)

x: .single 0r0
y: .single 0r0

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

! switch to data to put FP constant
    .section "".data"
    .align 4
_t1: .single 0r9000.01

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

! x = 9000.01
    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 float
    fadds %f1, %f2, %f1
    fdivs %f1, %f2, %f1
    set y, %l1
    st %f1, [%l1]
```
float example (simplified)

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

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