Discussion 6

CSE 131

overview

- phase 2
- some 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

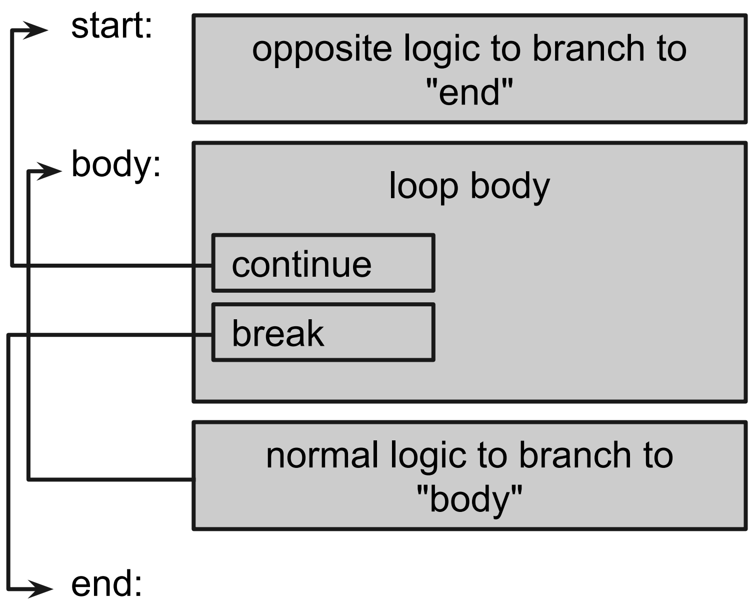
- think of how you handle an if-else statement
- short circuiting follows the same principle
    - in the A && B case
        - if not A then false, else B
    - in the A || B case
        - if A then true, else B

short circuiting

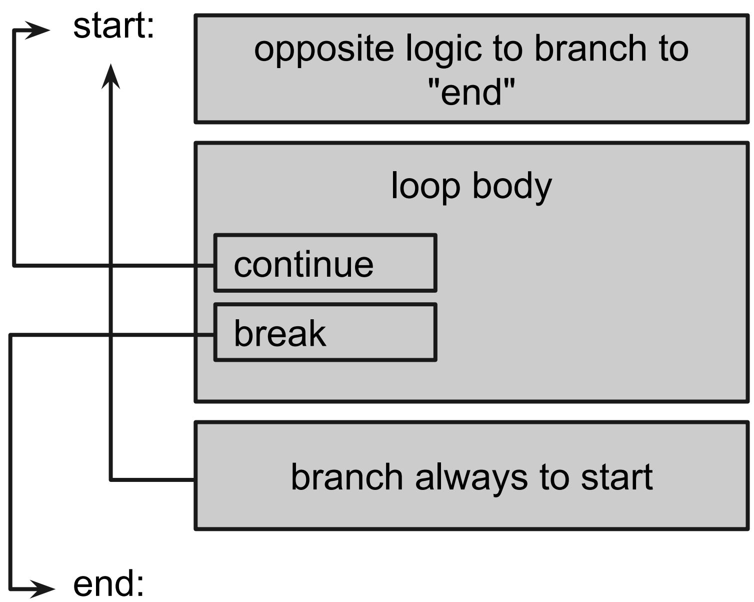
```
! RC: bool c = a && b
! load a and check if false
    set  a, %l0
    ld   [%l0], %l0
    cmp  %l0, %g0
    be   flabel
    nop

! 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

endlabel:
    set  c, %l0
    st   %l5, [%l0]
```

while loops

the ideal way



the easier way



while loops

- similar to if-else statements, you'll need a label stack of some kind to handle nested while-loops

example (simplified)

```
! RC: while (x < 5) {
!   cout << x;
!   x = x + 1;
! }

.l1start:
     set  x, %l0
     ld   [%l0], %l0
     set  5, %l1
     cmp  %l0, %l1
     bge  .l1end    ! opposite logic
     nop

     set  x, %l0
     ld   [%l0], %l0
     mov  %l0, %o1
     set  _intFmt, %o0
     call printf
     nop

     set  x, %l0
     ld   [%l0], %l1
     add  %l1, 1, %l1
     st   %l1, [%l0]
     ba   .l1start
     nop

.l1end:
```

array/struct allocation

- when you declare a global array, allocate an entire chunk in the BSS and have a variable label at the beginning

```
! int [7]x;
        .section ".bss"
        .align   4
        .global  x
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

- a useful attribute to have for arrays and structs is "size", so you know how much space to allocate
  - should have this from project 1 already
- offsets are also useful
  - for arrays, offsets are simply multiples of element size
  - for structs, offsets are the collective size of the preceding fields

array usage (simplified)

```
! RC: a = x[b] + 7;
! x is array of int

set  b, %l0
ld   [%l0], %l0
sll  %l0, 2, %l0   ! b * 4 -> scaled 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
- start at the base address of the struct
- move some offset to a specific field
- then, load or store depending on what you wanted to do

pass/return by ref

- think of them as pointers

```
function : void foo(int &x) {
     x = 10;
}

function : void foo(int *x) {
     *x = 10;
}
```

passing arrays

- arrays must be passed by reference
- internally, pass the base address of the array like you would any other argument

passing structs

- structs must be passed by reference
- internally, pass the address of the struct like you would any other argument

value vs reference

- further reading
  - http://www.cse.ucsd.edu/~ricko/CSE131/RefVsValue.pdf

pointers

- consider p = q
  - this is just copying the address 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 where q is pointing and making where p points that value

```
set  q, %l0
ld   [%l0], %l0    ! get address in q
ld   [%l0], %l0    ! additional load to get value
set  p, %l1
ld   [%l1], %l1    ! get address in p
st   %l0, [%l1]    ! store into place p points
```

pointers

- new
    - just a call to calloc() to allocate memory on the heap that is zero-initialized
- delete
    - just a call to free() with the address
    - remember to set the pointer to nullptr afterwards

pointer return types

- don't forget functions can return pointer types
  - in that case, you want to place the address (value of the pointer) in the %i0 register
- that address can then be assigned into another pointer like so:
  - ptr = foo(...)

example (simplified)

```
typedef int* PTRTYPE;
PTRTYPE myGlobal;

function : PTRTYPE foo() {
     PTRTYPE myLocal;
     new myLocal;
     *myLocal = 42;
     return myLocal;
}

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

example (simplified)

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

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

! new myLocal
     set  1, %o0    ! numelem
     set  4, %o1    ! sizeof(int)
     call calloc
     nop
     st   %o0, [%fp-4]
```

```
! *myLocal = 42
     set  42, %l0
     ld   [%fp-4], %l1
     st   %l0, [%l1]

! return myLocal
     ld   [%fp-4], %i0
     ret
     restore
SAVE.foo = -(92 + 4) & -8

     .global   main
main:
     save %sp, 96, %sp

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

example (simplified)

```
! cout << *myGlobal
    set  intfmt, %o0
    set  myGlobal, %l7
    ld   [%l7], %l0
    ld   [%l0], %o1
    call printf
    nop

    mov  %g0, %i0
    ret
    restore
```

tip

- use gdb!
- why?
  - helps locate bugs easily
  - print statements are too high level for debugging code generators
  - provides an inside view of the processor state
    - memory, registers, etc
  - you can apply breakpoints at specific machine instructions and step through them
  - saves a lot of time

using gdb

- make sure to compile with debug symbols

```
CC=gcc
compile:
	$(CC) -g rc.s input.c output.s $(LINKOBJ)
```

- run "gdb a.out" from the terminal
- gdb quick reference card
  - http://www.digilife.be/quickreferences/QRC/GDB%20Quick%20Reference.pdf