Discussion 4

CSE 131

what to do next

- fix project 1
  - ensure no errors generated for semantically valid RC code
- familiarize yourself with SPARC assembly
- plan out your project’s structure
- start on phase 1 of project 2

overview

- project 2 overview
- SPARC architecture and assembly
- code generation example

project 2 overview

- no type checking
- only testing syntactically and semantically valid code
- should utilize the structures you created in project 1
  - but make sure your code isn't printing out errors when there are none

project 2 overview

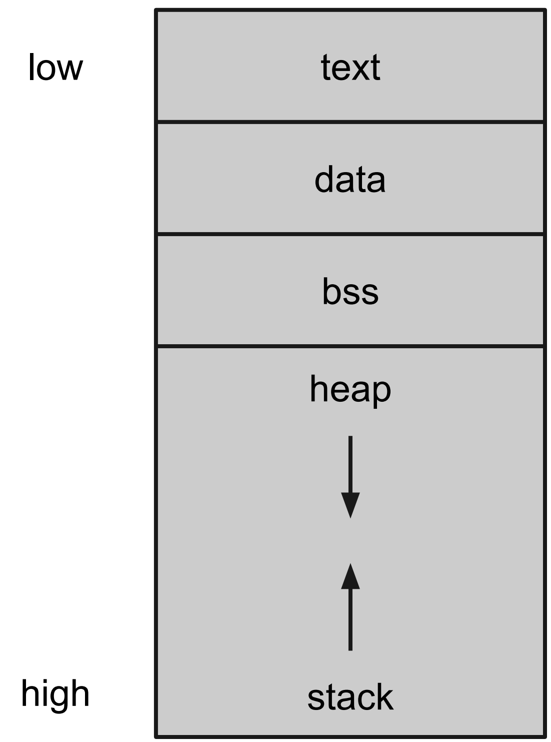
- what will I be doing?
  - you will be generating SPARC assembly in your parser
  - then you will feed the generated assembly into a C compiler to create an executable

SPARC architecture

- it’s a reduced instruction set computer (RISC)
  - small number of simple instructions (as opposed to CISC)
- load/store architecture
  - 32 32-bit integer registers (global, local, input, output)
  - 32 32-bit floating-point registers
  - sliding register window

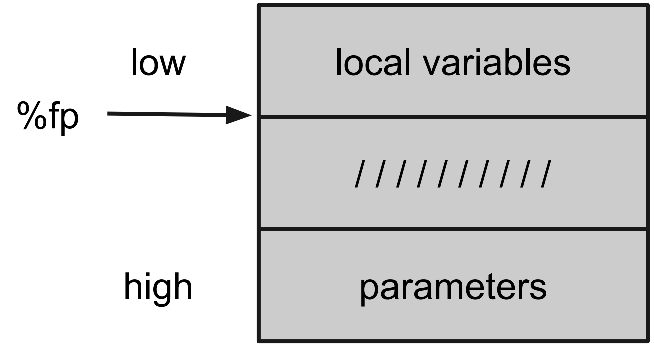
SPARC memory

- text - instructions
- data - initialized global and static variables
- bss - uninitialized global and static variables
- heap - dynamically allocated memory
- stack - stack frames
  - local variables and function parameters



stack frames

- local variables at negative offset
- parameters at positive offset



integer registers

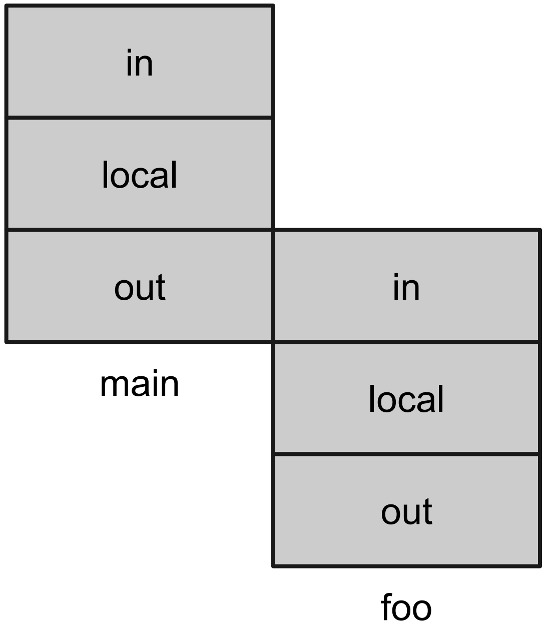
- local (%l0 - %l7)
  - values local to each function
- input (%i0 - %i5)
  - input parameters to a function
  - %i0 is also for the return value
  - %i6 and %i7 are reserved
- output (%o0 - %o5)
  - output arguments to a function (fill right before call)
  - return value in %o0 once call returns
  - %o6 and %o7 are reserved

integer registers

- global (%g1 - %g7)
  - available throughout function calls (no sliding)
  - %g0 is always 0
  - %g1 - %g4 are volatile (temp use only)

sliding register window

- when main calls foo, main’s output registers become foo’s input registers



floating point registers

- %f0 - %f31
  - not windowed
  - can be changed by functions you call, so don’t leave things in them you may need

common instructions

- set (no 4k restriction)
  - `set 12345, %l0        ! %l0 = 12345`
  - `set x, %l0            ! %l0 = address labeled by x`
- move (constants +-4K ok, but prefer set)
  - `mov %l0, %o0          ! %o0 = %l0`
- simple arithmetic (add, sub)
  - `add %o0, %o1, %o2     ! %o2 = %o0 + %o1`

common instructions

- increment/decrement (inc, dec)
  - `inc %l0                ! %l0 = %l0 + 1`
- shifting (sll, srl, sra)
  - `sll %o1, 5, %o0        ! %o0 = %o1 << 5`
- load
  - `ld [%fp-4], %i4        ! %i4 = *(%fp-4)`
- store
  - `st %i3, [%fp-8]        ! *(%fp-8) = %i3`

common instructions

- compare
    - `cmp %o0, %o1    ! sets condition codes based on %o0-%o1`
- branch (bg, bge, bl, ble, be, bne, ba)
    - `ble loop2       ! go to label loop2 if prior cmp <=`
    - `nop`
- call
    - `call foo  ! jumps to subroutine labeled foo (saves PC)`
    - `nop`

nops

important: remember to have a nop after a branch or call instruction

the save instruction

```
set -(92 + x) & -8, %g1    ! x = number of bytes of local vars
save %sp, %g1, %sp
```

- The reason we have the set instruction is to avoid the 4K pitfall of putting the number in the save instruction
- 92 comes from
  - 64 bytes for saving in/local registers
  - 4 bytes for returning a struct by value (64 + 4 = 68; start of params)
  - 24 bytes for first 6 parameters (%i0-%i5)

saving parameters

```
st %i0, [%fp+68]  ! put value of parameter in memory
```

- why would you want to do this?
  - registers don’t have addresses
  - needs to have a memory location to have an address

assembly sections

- .text - instructions
- .data - data (initialized global/static vars)
- .rodata - read-only data
- .bss - bss (will be automatically set to 0)

Can switch between the different sections whenever you want (just don’t forget to align)

global variables (method 1)

- rc code

```
int x, y;
```

- assembly code

```
    .global x,y
    .section “.data”
    .align 4
x:  .word 0
y:  .word 0
```

global variables (method 2)

- rc code

```
int x, y;
```

- assembly code

```
     .global x,y
     .section “.bss”   ! will auto init to zero
     .align 4
x:   .space 4          ! same as .skip
y:   .skip 4           ! same as .space
```

global variables

- both methods assumed init to 0
- to init to another value
  - use the .data section method

local variables

- important thing is their base and offset
  - if x is at %fp-8
    - x.base = “%fp”
    - x.offset = “-8”
  - then, when you see x used in the code, you know to load from base+offset

global variables

- similar to local variables except
  - base is “%g0”
  - offset is the name, e.g. “x”

why base+offset

- your assembly is the same regardless of global or local

```
set <offset>, %l0
add <base>, %l0, %l0
```

global and local example

```
int x = 5;
function : void main() {
    int y = x;
}
```

```
        .section ".data"
        .align 4
        .global x
x:      .word 5

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

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

        ret
        restore
```

globals

- all global variables must be marked global
  - use the .global directive
  - this makes it visible to other files
- all functions must be marked global
  - including main()
- unless marked static, then don't make them global
- consider adding a flag to your STOs to mark whether the symbol is global or not

function calls

- in rc

```
foo(5,9)
```

- in assembly

```
set  5, %o0
set  9, %o1
call foo
nop
```

useful constants

- consider always defining the following useful internal constants

```
        .section ".rodata"
_endl:    .asciz "\n"
_intFmt:  .asciz "%d"
_boolT:   .asciz "true"
_boolF:   .asciz "false"
```

```
! always use .asciz instead of .ascii, since the former will
! automatically null-terminate your ASCII string
```

outputting stuff

- in rc

```
cout << 5;
```

- in assembly

```
set  _intFmt, %o0
set  5, %o1
call printf
nop
```

outputting stuff

- in rc

```
cout << 5.75;
```

- in assembly

```
.section ".data"
.align   4
tmp1:    .single 0r5.75

.section ".text"
.align   4
set  tmp1, %l0
ld   [%l0], %f0
call printFloat
nop
```

large example (rc)

```
int x = 4;
int y;
const int c = 5;
int z = c;

function : int main() {
     y = 11;
     z = c - y;
     z = z + x;
     cout << z << endl;
     return -2;
}
```

large example (sparc)

```
! --globals--
     .section ".data"
     .align 4
     .global x, y, c, z
x:   .word 4
y:   .word 0
c:   .word 5
z:   .word 5

     .section ".rodata"
_endl:    .asciz "\n"
_intFmt:  .asciz "%d"

! --main--
     .section ".text"
     .align 4
     .global main
```

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

! y = 11
     set  11, %l1
     set  y, %l0
     add  %g0, %l0, %l0
     st   %l1, [%l0]

! z = c - y
     set  c, %l0
     add  %g0, %l0, %l0
     ld   [%l0], %l1
     set  y, %l0
     add  %g0, %l0, %l0
     ld   [$l0], %l2
     sub  %l1, %l2, %l1
     set  -4, %l0
     add  %fp, %l0, %l0
     st   %l1, [%l0]    ! tmp1
```

large example (sparc)

```
    set  -4, %l0
    add  %fp, %l0, %l0
    ld   [%l0], %l1
    set  z, %l0
    add  %g0, %l0, %l0
    st   %l1, [%l0]

! z = z + x
    set  z, %l0
    add  %g0, %l0, %l0
    ld   [%l0], %l1
    set  x, %l0
    add  %g0, %l0, %l0
    ld   [$l0], %l2
    add  %l1, %l2, %l1
    set  -8, %l0
    add  %fp, %l0, %l0
    st   %l1, [%l0]    ! tmp2
    set  -8, %l0
    add  %fp, %l0, %l0
    ld   [%l0], %l1
    set  z, %l0
    add  %g0, %l0, %l0
    st   %l1, [%l0]

! cout << z << endl
    set  _intFmt, %o0
    set  z, %l0
    add  %g0, %l0, %l0
    ld   [%l0], %o1
    call printf
    nop
    set  _endl, %o0
    call printf
    nop

! return -2
    set  -2, %i0
    ret
    restore

! 8 bytes of tmp
    SAVE.main = -(92 + 8) & -8
```

where do I do this

- there are many ways to output your assembly code
  - try to be as organized as possible - don't just throw some println statements all over your cup and MyParser files
  - consider making a separate class that just deals with outputting code
  - make sure to use ample formatting (tabs, blank lines, comments) as this will help you greatly with debugging

important

- use load-load-compute-store
  - the previous large example uses this
- DO NOT try to implement register allocation
  - we're not going for efficiency
  - with a one-pass parser it's really hard to do register allocation right
  - save yourself time and weird bugs by not doing this to yourself
  - there are plenty of other places in this project to shoot yourself in the foot, no need to add another one