CSE 131 – Compiler Construction

Discussion 5: SPARC Assembly

(The Fun Stuff!)

2/8/2010

2/12/2010

Overview

- Project II Overview
- SPARC Architecture and Assembly
- An Example of Code Generation

Project II Overview

- For Project II, there will be no type checking
  - Only syntactically and semantically correct code will be given to your compiler.
  - You will most likely still need the structures you created in Project I, but need to make sure your Project I code isn't printing error messages when there are none (false positives).

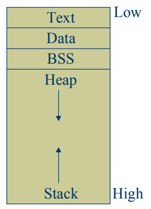
Project II Overview

- So, what is Project II supposed to do?
  - You will be generating SPARC assembly code for a given RC program (creating rc.s)
  - Once your compiler outputs an assembly program, it will be fed into a C compiler to create the resulting executable (a.out)

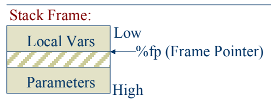
SPARC Architecture

- SPARC is a RISC (Reduced Instruction Set Computer) Architecture
  - This means there are a small number of simple instructions (as opposed to CISC).
  - Load/Store Architecture (load-load-compute-store)
  - Branching on Condition Codes
    - Z (zero), N (negative), C (carry), V (overflow)
  - 32 32-bit integer registers (global, local, input, output)
  - 32 32-bit floating-point registers
  - Sliding Register Window

SPARC Memory

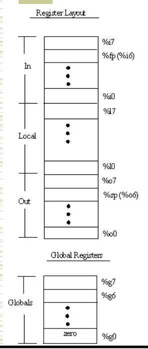


- Text – Instructions
- Data – Initialized global & static variables
- BSS – Uninitialized global & static variables
- Heap – Dynamically allocated memory
- Stack – Stack Frames (local variables and function parameters)



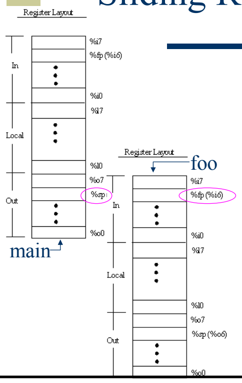
- Local variables at negative offset
- Parameters at positive offset

SPARC Integer Registers



- Global (%g1 - %g7)
  - Available throughout function calls (no sliding)
  - %g0 is always 0
  - %g1 - %g4 are volatile (temp use only)
- Local (%l0 - %l7)
  - Values local to each function (disappear after function returns)
- Input (%i0 - %i5)
  - Input parameters to a function (this is why there is a 6 parameter limit issue)
  - %i0 is where to store return value before returning
  - %i6 and %i7 are reserved
- Output (%o0 - %o5)
  - Output arguments to a function (do right before "call")
  - Once function returns, return result is in %o0
  - %o6 and %o7 are reserved

Sliding Register Window



- When main calls foo, main's output registers will become foo's input registers.
- Also, as you can see, foo's frame pointer is just main's stack pointer.

SPARC FloatingPoint Registers

- Floating-point Registers (%f0 - %f31)
  - These registers are *NOT* windowed.
  - These will remain intact across function calls within your code
  - These can be changed by other external functions you call, so don't leave things in them that you may need!

Common SPARC Instructions

Set (no 4K restriction):
```
    set 12345, %l0              ! %l0 = 12345
    set x, %l0                  ! %l0 = addr/mem loc. labeled by x
```
Move (constants between +/- 4K OK – but use set!):
```
    mov %l0, %o0                ! %o0 = %l0
```
Simple Arithmetic (add, sub):
```
    add %o0, %o1, %o2           ! %o2 = %o0 + %o1
```
Increment/Decrement (inc, dec):
```
    inc %l0                     ! %l0 = %l0 + 1
```
Shifting (sll, srl, sra):
```
    sll %o1, 5, %o0             ! %o0 = %o1 << 5
```

Common SPARC Instructions

Load:
```
    ld [%fp – 4], %i4           ! %i4 = *(%fp – 4)
```
Store:
```
    st %i3, [%fp – 8]           ! *(%fp – 8) = %i3
```
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 was <=
    nop
```
Call**:
```
    call foo                    ! Jumps to subroutine labeled "foo" (saves PC)
    nop
```
** Remember to have a "nop" after a branch or call statement!!!

The Save Instruction

```
set -(92 + ?) & -8, %g1     ! Where ? = # of bytes of local variables
save %sp, %g1, %sp
```

- The reason we have the "set" instruction is to avoid the 4K pitfall of just placing the number in the "save" instruction.
- The 92 comes from:
  - 64 bytes for 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)
- What if you had a value parameter to foo that needed to be sent as a reference parameter to baz?
```
  st  %i0, [%fp+68]         ! Put value of parameter in memory parameter location
  add %fp, 68, %o0          ! Put address in output argument to baz.
```

Assembly Sections

- .text – Instructions
- .data – Data (Initialized global/static vars)
- .rodata – Read-only Data
- .bss – BSS (will be automatically set to 0)

- You can switch between the different sections throughout the code (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       ! 0 to initialize to zero
y:      .word   0       ! 0 to initialize to zero
```

Global Variables – Method 2

- RC code:

```
int x, y;
```

- Assembly code:

```
        .global x, y
        .section        ".bss"   ! Will auto initialize to zero
        .align  4
x:      .space  4       ! Same as .skip
y:      .skip   4       ! Same as .space
```

Global Variables

- Note that for all these methods shown, we assumed initializing to 0.
- If you wanted to initialize a global variable with another value, you need to either use the ".data" method (Method 1), or use the ".bss" methods (Method 2) with initialization code at the beginning of main() to set the initial value into those locations in the BSS (similar to what will need to be done with initializing local variables). If you use the bss-code initialization method, make sure you have guards to only initialize once, in case you have recursion.

Local Variables

- One important thing we will need to know about variables will be their base and offset in memory.
- For example, if X was at %fp – 8, you would want to store the following in X's VarSTO:

  X.base = "%fp"
  X.offset = "-8"
  X.isLocal = true
- Now, when you see X used in the code, you know to load from base+offset (i.e., [%fp-8])

GlobalVariables

- The base of a global variable is always its name. To get this name into a register, we need to use the "set" command.
- It may be beneficial to include a flag in your STO class to designate whether an STO is a global or not. If the STO is not local, you'll use the set instruction instead of calculating the offset based on the frame pointer (%fp). (Note that we will be using another flag, isGlobal, later).

  X.base = "x"
  X.offset = "0"
  X.isLocal = false
- Now when you see X used in the code, you'll know that you have to set it's label into a register to get it's address.

Global & Local Variable Examples

```
int x;
function : void main() {
    int y;
}
--------------------------------
    .section ".data"
    .align 4
    .global x, main
x:  .word 0
    .section ".text"
    .align 4
```

```
main:
        set SAVE.main, %g1
        save %sp, %g1, %sp
        set x, %l0              ! Loads the address of x into %l0
        ld [%l0], %l1           ! Put the value of x into %l1
        st %l2, [%l0]           ! Put the value in %l2 into x

        add, %fp, -4, %l3       ! Put the address of y into %l3
        ld [%l3], %l4           ! Put the value of y into %l4
        st %l5, [%l3]           ! Put the value in %l5 into y
        ret
        restore
SAVE.main = -(92 + 4) & -8
```

GlobalVariables

- Note that all of our global variables have been marked global by the ".global" directive. This makes the symbol visible to other files.
- All global variables and functions that are defined in the file and are not static must be declared as global. "main" must also be declared as global.
- You should add a flag to your STOs, isGlobal, to mark whether or not a symbol should be put in the global directive.

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 constants:

```
         .section ".rodata"
endl:    .asciz "\n"
intFmt:  .asciz "%d"
boolT:   .asciz "true"
boolF:   .asciz "false"
```

Note: 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         ! Assuming you have defined this
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
```

A Large Example

- Given this RC Code, what is the assembly?

```
int x = 4;
int y;
int z = x;
const int c = 5;
function : int main() {
        y = 11;
        z = c - y;
        z = z + x;
        cout << z << endl;
        return -2;
}
```

A Large Example

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

!----Default String Formatters----
                .section        ".rodata"
intFmt:         .asciz          "%d"
endl:           .asciz          "\n"

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

A Large Example

```
main:
        set SAVE.main, %g1
        save %sp, %g1, %sp
! init z, modify to support recursion
        set x, %l0
        ld [%l0], %l1
        set z, %l0
        st %l1, [%l0]
! y = 11
        set y, %l0
        set 11, %l1
        st %l1, [%l0]
! z = c – y
        set c, %l0
        ld [%l0], %l1
        set y, %l0
        ld [%l0], %l2
        sub %l1, %l2, %l1
        st %l1, [%fp-4]         ! tmp1
        ld [%fp-4], %l1
        set z, %l0
        st %l1, [%l0]
```

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

```
! cout
        set intFmt, %o0
        set z, %l0
        ld [%l0], %o1
        call printf
        nop
        set endl, %o0
        call printf
        nop
        set -2, %i0             ! return -2
        ret
        restore
! 8 bytes tmp
SAVE.main = -(92 + 8) & -8
```

Where do you 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
    - Evan (a previous tutor) provides an excellent idea for a possible class: http://www.cse.ucsd.edu/users/ricko/CSE131/AssemblyCodeGenerator.html
    - Make sure to use ample formatting (tabs, blank lines, comments), as this will help you greatly with debugging.

What to do Next!

1. Fix Project I – ensure there are no errors generated.
2. Familiarize yourself with SPARC assembly.
3. Plan out how you want to structure your project – good planning leads to an easier design in the long run.
4. Start on Phase 1.
5. 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?