Discussion Week 5: SPARC Assembly
(The Fun Stuff!)
04/29/2013
05/03/2013
AND THEN I SAID

LET'S GIVE THEM MORE TIME ON PROJECT 2
WHAT TO DO NEXT!

1. Fix Project I – ensure there are no errors generated for semantically valid RC code.
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.
5. Come to lab hours and ask questions.
OVERVIEW

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

Stack Frame:
- Local Vars
- 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
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 Floating Point 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!!!
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:
  ```c
  int x, y;
  ```

- Assembly code:
  ```assembly
  .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:
  ```c
  int x, y;
  ```

- Assembly code:
  ```assembly
  .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.

- To initialize a global variable with another value, you need to use one of the following:
  - The "data" section (Method 1)
  - The "bss" section (Method 2)

- For this, you need initialization code at the beginning of `main()` to set the initial value into those locations in the BSS. Make sure you have guards to only initialize once.
 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"
- Now, when you see X used in the code, you know to load from base+offset (i.e., [%fp-8])
- For global variables, the base should be "%g0" and the offset will be the variable name (e.g. "x")
**VARIABLES**

- **Global Variables:**
  
  ```
  X.base = "%g0" // always %g0, which is always 0
  X.offset = "x" // Name of global variable
  ```

- **Local Variables:**
  
  ```
  Y.base = "%fp" // always %fp
  Y.offset = "-8" // Offset of local variable
  ```

- **Then, your assembly can **consistently** be the following to get the address of the variable into %l0:**

  ```
  set <offset>, %l0
  add <base>, %l0, %l0
  ```
GLOBAL & LOCAL VARIABLE EXAMPLES

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

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

    ! Global initialization guard (in case main is called recursively!)
    set _init, %l0
    ld [%l0], %l1
    cmp %l1, %g0
    bne _init_done
    mov 1, %l1
    ! Branch delay slot
    st %l1, [%l0]
    ! Mark _init = 1
    ! Global initializations
    set 5, %l1
    ! Initialization for x
    set x, %l0
    ! Put the name of x into %l0
    add %g0, %l0, %l0
    ! Add offset/name to base (0 in this case)
    st %l1, [%l0]
    ! Store the value in %l1 into x

_init_done:
    set x, %l0
    ! Put the name of x into %l0
    add %g0, %l0, %l0
    ! Add offset/name to base (0 in this case)
    ld [%l0], %l1
    ! Load the value of x into %l1
    set -4, %l0
    ! Put the offset/name of y into %l0
    add %fp, %l0, %l0
    ! Add offset/name to base
    st %l1, [%l0]
    ! Store the value in %l1 into y

    ret
    restore

SAVE.main = -(92 + 4) & -8
! 4 is for the one local var ‘y’
```

GLOBAL VARIABLES

- Note that all of our global variables have to be 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
Consider always defining the following useful internal constants:

```
.section "\rodada"
_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:
  ```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?

```cpp
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    ".bss"
.align 4
_init:
.global x, y, z, c
x:
.y:
.z:
c:

!----Default String Formatters----

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

!----Main----

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

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

    set _init, %l0
    ld [%l0], %l1
    cmp %l1, %g0
    bne _init_done  ! Global initialization guard
    mov 1, %l1      ! Branch delay slot
    st %l1, [%l0]   ! Mark _init = 1
    ! x = 4
    set 4, %l1
    set x, %l0
    add %g0, %l0, %l0
    st %l1, [%l0]
    ! z = x
    set x, %l0
    add %g0, %l0, %l0
    ld [%l0], %l1
    set z, %l0
    add %g0, %l0, %l0
    st %l1, [%l0]
    ! c = 5
    set 5, %l1
    set c, %l0
    add %g0, %l0, %l0
    st %l1, [%l0]
_init_done:
A LARGE EXAMPLE

! 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
    set -4, %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], %l1
    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 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
  - Make sure to use ample formatting (tabs, blank lines, comments), as this will help you greatly with debugging.
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?