CSE 131 -- Winter 2010
Compiler Project #2 -- Code Generation
Due: Friday night March 12, 2010 @ 11:59pm

Disclaimer

This handout is not perfect; corrections may be made. Updates and major clarifications will be incorporated in this document and noted in the Project Updates section of the Moodle message board as they are made. Please check for updates regularly.

Note about Turn-in

Please refer to the turn-in procedure document on the website for instructions on the turn-in procedure.

Note that all output from the generated assembly file must go to **stdout**. No debugging output should be generated from the assembly file. Debugging comments in the generated assembly source file is fine.

Description

In this project you will generate SPARC assembly code for a subset of the Reduced-C (RC) language that, when assembled and linked, results in an executable program. You should do all your compiling, linking, and testing on ieng9, which is a SUN SPARC architecture.

You will be adding code to the actions in your compiler to emit SPARC assembly language. In particular, your compiler should, given an RC program as input, write the corresponding SPARC assembly code to a file named `rc.s`. At that point, `rc.s` can be fed to the C compiler (to run the assembler and to link with Standard C Library routines you use and the `input.c` and `output.s` files we supply you in the class `public` directory for the implementation of the `inputInt()`, `inputFloat()`, and `printFloat()` functions) to generate an executable named `a.out`. We will run the `a.out` executable and check that it produces the expected output.

All output should go to **stdout** including any run time error messages produced (see Phase III.2).

We will also have test cases involving **separate compilation** (using the `extern` and `static` keywords). Some of these test cases should pass through the linker without any errors, and an executable `a.out` should be produced. However, some of these cases will intentionally contain unresolved `extern` references that should cause the compilation to fail at link-time.

The features are assigned in phases, each worth a percentage of the grade. The subdivisions within each phase do **not** reflect separately-graded units; they merely suggest an order in which the subparts of each phase can be implemented.

What We Aren't Doing

As a clarification, here are examples of things you **DO NOT** have to implement.

- Constructs omitted in Project I
- String expressions (String literals are OK, of course. i.e. `"Hello World\n"`).
- Structs and arrays passed by value (structs and arrays will be passed by reference only).
- Array identifier usage as a pointer to the first element of the array.

- Arrays of arrays.
- Pointers to arrays.
- Passing more than 6 arguments to a regular function. *Passing more than 6 arguments to a regular function is an Extra Credit option.*
- Passing more than 5 arguments to a struct-member function. *Passing more than 5 arguments to a struct-member function is an Extra Credit option.*
- Ambiguous statements (e.g. y = ++x + x, where it is unclear if the second operand to + gets bound to the incremented value of x or not).
- Address-of an array name or a function name. (Note: address-of array elements that are not arrays themselves are allowed).
- Comparison of function pointers to one another.
- Any other construct not listed below (unless we forgot something really important).

Makefile change for target language

Since the assembly code you emit/generate will be dependent on the conventions of the C compiler you are using, you must also **add the following rule in your Makefile**:

```
CC=cc
compile:
        $(CC) rc.s input.c output.s $(LINKOBJ)
```

where variable `CC` is bound to the **cc** or **gcc** compiler on ieng9. The variable `LINKOBJ` can either be left undefined, or, when invoking the `make` command, it can be set to hold the name of one or more object files that should be linked into the executable. Examples of invoking `make` this way are shown below.

```
make compile
make compile LINKOBJ=test1a.o
make compile LINKOBJ='test2a.o test2b.o'
```

The first command will permit us (and you!) to assemble your compiler's generated assembly code to create an executable `a.out` file for testing. Commands like the second and third one will also generate an executable `a.out` if all external references are resolved, or generate the appropriate linker errors otherwise.

You may want to consider emulating (and using) cc vs. gcc for testing purposes.

Notes/Hints/Etc.

- Only syntactically, semantically legal (according to the Project 1 spec) RC programs will be given to your compiler. This doesn't mean you can scrap all your code from last project -- some of it will be needed in this project. For example, the type aliasing system must be working, since semantically correct type aliases will be used.

- **The C compiler is your friend.** This cannot be overemphasized. A wealth of knowledge can be gained by simply seeing how the C compiler does things, and emulating it. (How do you think we learned SPARC assembly language in general and SPARC floating point instructions in particular?!) In most cases, the assembly code generated by `cc` will be similar to what you want your compiler to produce. You may also look at `gcc`, but it is much less straightforward. However, you should only emulate *one* compiler, since the techniques you use have to be internally consistent. In deciding which to use, think about which one produces the simpler code (in your opinion).

- To see how the C compiler generates assembly, write a C program (make it *small*!) and compile it with the `-S` option:

  ```
  cc -S program.c
  ```

  This produces `program.s`, which contains the generated assembly code. Also, some of the constructs in RC (e.g. global and static variables with run-time initialization values) are based on C++. In these instances, you may want to use `CC` or `g++` to see how the C++ compiler does this.

- To create object files for testing the inter-functionality of `static` and `extern`, typically you will write C code that uses extern and/or static and compile it using:

  ```
  gcc -c module.c
  ```

  This will create a `module.o` object file that you can then pass to the `make compile` command.

- Outputting assembly language comments has been found to be very helpful in debugging. But **do not** turn in a project which outputs any debugging statements to stdout or stderr from either your compiler or from the assembly code generated by your compiler!

Phase I Features: (2 weeks)

I.1

- int literals, constants and variables (including initialization and assignment). All uninitialized global and static variables should be initialized to zero. Local variables are not automatically initialized.
- float literals, constants and variables (including initialization and assignment). All uninitialized global and static variables should be initialized to zero. Local variables are not automatically initialized.
- bool literals, constants and variables (including initialization and assignment). All uninitialized global and static variables should be initialized to `false`. Local variables are not automatically initialized.
- String literals (only included for output with `cout`).
- `cout` for
  - int expressions (no newline is automatically appended on output). Use `printf()` with the `"%d"` format specifier.
  - float expressions (no newline is automatically appended on output). Use the `printFloat()` function provided in `output.s` (`public/output.s`)
    Note: Pass single precision floating point float values in register `%f0`.
  - bool expressions. Bools should be output as `"true"` or `"false"` (no newline is automatically appended on output). Use `printf()` with the `"%s"` format specifier.
  - String literals (no newline is automatically appended on output). Use `printf()` with the `"%s"` format specifier.
    Do not use **puts()** to output a string literal. puts() appends a newline character to the output.
  - The symbol `endl` represents the newline character and can be used interchangeably with `"\n"`

I.2

- Constant folding for arithmetic and logical expressions involving ints, floats, and bools (and no other types), as well as the *sizeof* construct. Note: These should already be working from Project I.

```
const int c1 = 210;
const int c2 = sizeof(c1) + c1;
cout << c1 + 210 << endl;      // outputs 420
cout << c2 << endl;            // outputs 214
const float r1 = 420.25;
cout << r1 + 662.50 << "\n";   // outputs 1082.75
cout << sizeof(r1) << endl;    // outputs 4
```

- Compile-time bounds checks for constant array accesses. This includes indices that are constant expressions, e.g.

```
const int c1 = 2;
const int c2 = 4;
int[10] x1;
cout << (x1[c1 * 23 - (c1 + c2)]);
```

  Note: This should already be working from Project I.

- int arithmetic expressions containing `+`, `-` and UnarySign.
  - See Garo's SPARC Instructions Summary
- int arithmetic expressions containing `*`, `/`, and `%`.
- int bit-wise expressions containing &, |, and ^.
- `if` statements of the form: `if` (*expression*) `{` *statements* `}`. *expression* will be limited to the form $x > y$ (we will extend these expressions in II.1), where x and y are int expressions. For Phase I, the only bool operator you need to deal with is greater-than (>).

I.3

- int arithmetic expressions using pre/post increment and decrement (++/--).
- Functions with no parameters (including recursion). This includes functions with int, float, bool, and void return types.
- `return` *expr* for int, float, and bool expressions and `return` for void functions.
- `exit` statements.

Phase II Features: (2 weeks)

II.1

- float arithmetic expressions containing `+`, `-` and UnarySign.
  - See Garo's SPARC Instructions Summary
- float arithmetic expressions containing `*` and `/`.
- `if` statements of the form: `if` (*expression*) `{` *statements* `}`. *expression* will be limited to the form $x > y$ (we will extend these expressions in the next bullet), where x and y are float expressions.
- bool expressions containing >=, <=, >, <, ==, `!=`, `&&`, `||`, and `!`.
  Note: `&&` and `||` are **short-circuiting** operators. In other words, in `(a && b)`, if `a` evaluates to `false`, `b` is not evaluated. Similarly, in `(a || b)`, if `a` evaluates to `true`, `b` is not evaluated.
- `cin` with modifiable L-value ints. `cin` should use the `inputInt()` function provided in `input.c` for ints. (`public/input.c`)
  Note: The int return value from inputInt() is available to the caller in register `%o0`.

```
cin >> intVar;
```

- `cin` with modifiable L-value floats. `cin` should use the `inputFloat()` function provided in `input.c` for floats. (`public/input.c`)
  Note: The float return value from inputFloat() is available to the caller in register `%f0`.

```
cin >> floatVar;
```

- Handle mixed int and float expressions for all of the above.

```
floatVar = intVar + floatVar * 420;
```

- Handle assignment *expressions*. For example,

```
int i;
bool b1;
function : void foo(int j){
        if(b1 = j > 0){ foo(i = i - 1); }
}
```

II.2

- float arithmetic expressions using pre/post increment and decrement (++/--).
- `if` statements with (optional) `else` (for all valid bool expressions).
- `while` statements.
- `break` statements.
- `continue` statements.

II.3

- Functions with pass-by-value and pass-by-reference (e.g. &) int, float, and bool parameters (including recursion).
- Arrays and array indexing with one dimension with array run time layout in the style of C/C++ arrays. No bounds checking is required on run-time array access for Phase II. Each element of a global or static array should be initialized to the appropriate value for its data type (e.g. ints and floats should be initialized to zero, bools to false, and pointers to NULL). Local arrays are not automatically initialized.
- Structs. Global and static struct fields should be initialized in a way appropriate to their types (see arrays, above). Local structs are not automatically initialized. Struct assignment is supported. Support expressions with struct member variables and struct-member function calls (you should pass the "this" address of the struct in register %o0 for the struct-member function call). Support fields of all valid types.
- Arrays of structs.

Phase III Features: (1 week)

III.1

- Pointers to all types except arrays (this does not refer to function pointers). Uninitialized global and static pointers should be initialized to `NULL`. Local pointers are not automatically initialized. Support of the arrow (`->`) operator.
- Pointer arithmetic expressions using pre/post increment and decrement (++/--).

- Address-of operator (&) on valid objects of all types. Note that the address-of operator *can* be done on *elements* of an array (e.g. &myArray[3]), if the element type is not an array itself.
- Pass-by-reference array argument to array parameters.
- Pass-by-value and pass-by-reference pointer parameters.
- Pointer return values.
- Pass-by-reference struct parameters.
- `new` and `delete`. `new` should return **initialized** memory. `delete` should free the memory and set its argument to `NULL`. `new` and `delete` can be used on pointers of any type.

III.2

- `static` variable declarations (both internal and external). Support **initialization** (both constant and run-time values). All **uninitialized** static variables should be initialized appropriately for their given type.

```
function : void foo(int w) {
  static int x = w;  // Internal static (need conditional init)
}
static bool* x;      // External static (init to NULL)
```

- `extern` variable declarations. We will compile and link your `rc.s` together with a pre-compiled `*.o` object file in various ways, some of which should fail at link-time. All combinations of non-constant variables declared extern, static, or global are valid for testing and should be handled. Remember that a variable declared `static` is visible *only* to the module where it is defined.
- Function pointers. Function pointers can be to regular functions as well as struct member functions. You do not need to implement function pointer comparisons using == and `!=`.
- Type casts for all valid types (as described in the Project I spec). This includes the constant folding portion that should have been done in Project I.
- Run time array bounds checks. Out-of-bounds accesses should cause the program to print the following message to **stdout**

  ```
  "Index value of %D is outside legal range [0,%D).\n"
  ```

  and exit (call exit(1)). NOTE: The notation '[' means up to and including, ')' means up to and NOT including.

- Run time `NULL` pointer checks. Attempts to dereference or `delete` a `NULL` pointer should cause the program to print the following message to **stdout**

  ```
  "Attempt to dereference NULL pointer.\n"
  ```

  and exit (call exit(1)). This includes function pointers with regard to calling (e.g. dereference) a NULL function pointer.

Extra Credit (total of 9%)

Note: We reserve the right to add more extra credit options and adjust the percentage each extra credit option is worth so they total 9%.

1) Detect and report an attempt to dereference a pointer into deallocated stack space. (4%)

For the unary dereference operator *, detect when the pointer operand points into deallocated stack space. Here are some examples:

```
function : int * good(bool alloc) {
    static int y = 5;
    int * yy;

    if (alloc) {
        new yy;
        return yy;
    }

    return &y;
}

function : int * bad() {
    int z;

    return &z;   // This is bad
}

function : int * goodBad(int *** zz) {
    *zz;    // Okay - when goodBad is called by main below, zz contains
            // the address of ipp (in main), which points to
            // an address in an existing stack frame

    return &zz;   // But this is bad
}

int * ip;         // global var allocated in BSS
function : void main() {
    int ** ipp = &ip;

    *ipp;         //Okay - ipp has the value &ip, which points into the BSS
    *&ipp;        //Okay - &ipp points into the current stack frame

    *good(true);  //Okay - good returns a pointer into the Heap
    *good(false); //Okay - good returns a pointer into the Data segment

    *bad();         // Error - bad returns a pointer to a local variable
                    // in deallocated stack space

    *goodBad(&ipp); // Error - goodBad returns a pointer to a
                    // formal parameter in deallocated stack space
}
```

When this error is detected, the program should print the following message to **stdout**

```
"Attempt to dereference a pointer into deallocated stack space.\n"
```

and then exit (call exit(1)). Note that you need to detect dangling pointers *only* for deallocated stack space and *not* deallocated heap space.

2) Support passing more than 6 arguments to regular functions and passing more than 5 arguments to struct-member functions. (4%)

Here is an example of passing 8 arguments to a function. First 6 args passed in regs %o0-%o5 as usual. Args 7 and 8 have to be passed on the stack.

```
/*
 * main.s
 */

main:
        /* Normal save sequence here */

        /* Move args 1-6 into %o0-%o5 per normal arg passing */

        add     %sp, -(2*4) & -8, %sp   ! allocate stack space for args 7 & 8

        mov     7, %l0                  ! value of arg #7 (7 in this example)
        st      %l0, [%sp + 92]         ! pass arg #7 on stack

        mov     8, %l0                  ! value of arg #8 (8 in this example)
        st      %l0, [%sp + 96]         ! pass arg #8 on stack

        call    foo, 8                  ! %pc -> %o7 (addr of call instr)
        nop                             ! foo -> %pc (addr of target -> %pc)

        sub     %sp, -(2*4) & -8, %sp   ! deallocate stack space for args 7 & 8

        /* Retrieve return value in %o0 */

        /* Rest of main ... */


/*
 * foo.s
 */

foo:
        /* Normal save sequence here */

        /*
         * args #1-6 are in %i0-%i5
         * st  %i0, [%fp + 68]          ! store arg #1 on stack
         *       ...
         * arg #7 at %fp + 92           ! arg #7 already on stack
         * arg #8 at %fp + 96           ! arg #8 already on stack
         */

        /*
         * Body of function
         * Access formal params from memory locations %fp + 68 thru %fp + 96
         */

        /* Put return value -> %i0 */

        ret                             ! %i7 + 8 -> %pc
        restore                         ! slide register window up 16 regs
```

The main thing you have to do is allocate stack space for the additional args and copy the actual arguments into those stack locations before the call -- all done via %sp. Then in the function that was called, access those additional params at the same offsets but via %fp. Then back in the caller after the function call return you have to deallocate the stack space you allocated for those additional args. This is essentially what you have to do in most CISC architectures for every arg passed.

3) Support calls to overloaded functions. (1%)

Any valid calls to overloaded functions, as described in the Project I extra credit check, should be made to work for codegen.

Useful SPARC References

- Garo's SPARC Instructions Summary
- CSE 30 reference: Richard Paul, *SPARC Architecture, Assembly Language Programming, and C*, 2nd Edition. Prentice Hall 2000 (available at S&E Library Reserves). And CSE 30 Class Notes. Useful Links on CSE 30 webpage)
- University of New Mexico: http://www.cs.unm.edu/~maccabe/classes/341/labman/labman.html
- Sun: http://docs.sun.com/app/docs/doc/806-3774 (ignore V9 Instruction Set)