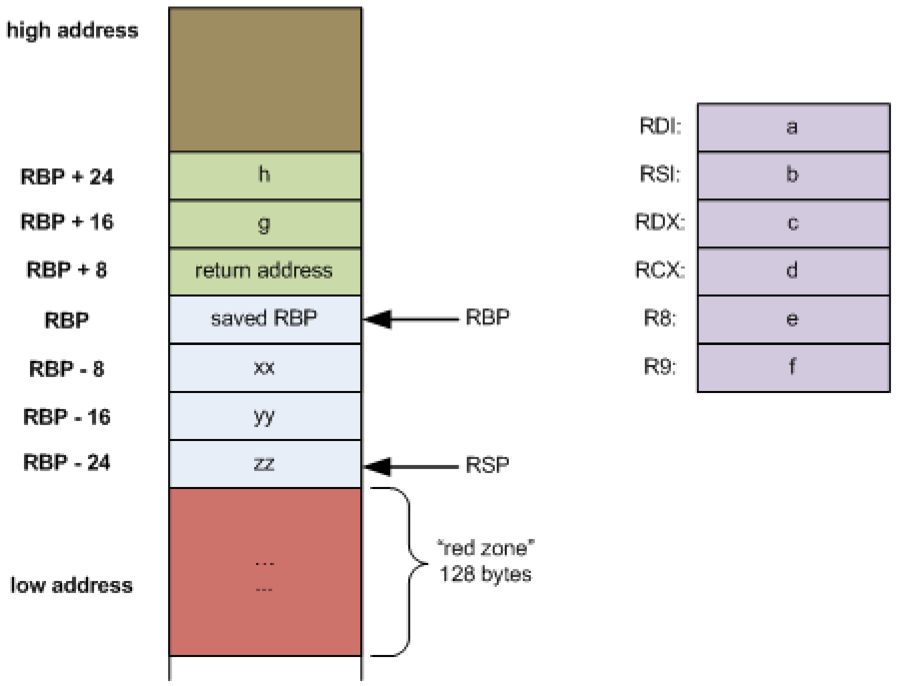
Part 25: Function Calls and Arguments

In this part of our compiler writing journey, I'm going to add the ability to call functions with an arbitrary number of arguments; the argument's values will be copied into the function's parameters and appear as local variables.

I haven't done this yet, because there is a bit of design thinking to be done before the coding can begin. Once more, let's review the image from Eli Bendersky's article on the stack frame layout on x86-64.



Up to six "call by value" arguments to a function are passed in via the registers `%rdi` to `%r9`. For more than six arguments, the remaining arguments are pushed on the stack.

Look closely at the argument values on the stack. Even though `h` is the last argument, it is pushed first on the stack (which grows downwards), and the `g` argument is pushed *after* the `h` argument.

One of the *Bad Things* about C is that there is no defined order of expression evaluation. As noted here:

> [The] order of evaluation of the operands of any C operator, including the order of evaluation of function arguments in a function-call expression ... is unspecified ... . The compiler will evaluate them in any order ...

This makes the language potentially unportable: the behaviour of code on one platform with one compiler may have different behaviour on a different platform or when compiled with a different compiler.

For us, though, this lack of defined evaluation order is a *Good Thing*, only because we can generate our argument values in the order that makes it easier to write our compiler. I'm being flippant here: this is really not much of a good thing.

Because the x86-64 platform expects the last argument's value to be pushed on the stack first, I'll need to write the code to process arguments from the last to the first. I should make sure that the code could be easily altered to allow processing in the other direction: perhaps a `genXXX()` query function could be written to tell our code which direction to process the arguments. I'll leave that to be written later.

Generating an AST of Expressions

We already have the A_GLUE AST node type, so it should be easy to write a function to parse the argument expressions and build an AST tree. For a function call `function(expr1, expr2, expr3, expr4)`, I've decided to build the tree like this:

```
                 A_FUNCCALL
                  /
              A_GLUE
               /   \
          A_GLUE   expr4
           /   \
      A_GLUE   expr3
       /   \
  A_GLUE   expr2
```

```
     /    \
   NULL  expr1
```

Each expression is on the right, and previous expressions are on the left. I will have to traverse the sub-tree of expressions right to left, to ensure that I process `expr4` before `expr3` in case the former has to be pushed on the x86-64 stack before the latter.

We already have a `funccall()` function to parse a simple function call with always one argument. I'll modify this to call an `expression_list()` function to parse the expression list and build the A_GLUE sub-tree. It will return a count of the number of expressions by storing this count in the top A_GLUE AST node. Then, in `funccall()`, we can check the type of all the expressions against the function's prototype which should be stored in the global symbol table.

I think that's enough on the design side of things. Let's now get on to the implementation.

Expression Parsing Changes

Well, I got the code done in an hour or so and I'm pleasantly surprised. To borrow a quote that floats around on Twitter:

> Weeks of programming can save you hours of planning.

Conversely, a bit of time spent on design always helps with the efficiency of coding. Let's have a look at the changes. We'll start with the parsing.

We now have to parse a comma-separated list of expressions, and build that A_GLUE AST tree with child expressions on the right, and previous expression trees on the left. Here is the code in `expr.c`:

```
// expression_list: <null>
//        | expression
//        | expression ',' expression_list
//        ;

// Parse a list of zero or more comma-separated expressions and
// return an AST composed of A_GLUE nodes with the left-hand child
// being the sub-tree of previous expressions (or NULL) and the right-hand
// child being the next expression. Each A_GLUE node will have size field
// set to the number of expressions in the tree at this point. If no
// expressions are parsed, NULL is returned
static struct ASTnode *expression_list(void) {
  struct ASTnode *tree = NULL;
  struct ASTnode *child = NULL;
  int exprcount = 0;

  // Loop until the final right parentheses
  while (Token.token != T_RPAREN) {

    // Parse the next expression and increment the expression count
    child = binexpr(0);
    exprcount++;

    // Build an A_GLUE AST node with the previous tree as the left child
    // and the new expression as the right child. Store the expression count.
    tree = mkastnode(A_GLUE, P_NONE, tree, NULL, child, exprcount);

    // Must have a ',' or ')' at this point
    switch (Token.token) {
      case T_COMMA:
        scan(&Token);
        break;
      case T_RPAREN:
        break;
      default:
        fatald("Unexpected token in expression list", Token.token);
    }
  }

  // Return the tree of expressions
  return (tree);
}
```

That turned out to be much easier to code than I was expecting. Now, we need to interface this with the existing function call parser:

```
// Parse a function call and return its AST
static struct ASTnode *funccall(void) {
  struct ASTnode *tree;
  int id;

  // Check that the identifier has been defined as a function,
  // then make a leaf node for it.
  if ((id = findsymbol(Text)) == -1 || Symtable[id].stype != S_FUNCTION) {
    fatals("Undeclared function", Text);
  }
  // Get the '('
  lparen();

  // Parse the argument expression list
  tree = expression_list();
```

```
    // XXX Check type of each argument against the function's prototype

    // Build the function call AST node. Store the
    // function's return type as this node's type.
    // Also record the function's symbol-id
    tree = mkastunary(A_FUNCCALL, Symtable[id].type, tree, id);

    // Get the ')'
    rparen();
    return (tree);
  }
```

Note the `XXX` which is my reminder that I still have work to perform. The parser does check that the function has previously been declared, but as yet it doesn't compare the argument types against the function's prototype. I'll do that soon.

The AST tree that is returned now has the shape that I drew up near the beginning of this article. Now it's time to walk it and generate assembly code.

Changes to the Generic Code Generator

The way the compiler is written, the code that walks the AST is architecture-neutral is in `gen.c`, and the actual platform-dependent back-end is in `cg.c`. So we start with the changes to `gen.c`.

There is a non-trivial amount of code needed to walk this new AST structure, so I now have a function to deal with function calls. In `genAST()` we now have:

```
    // n is the AST node being processed
    switch (n->op) {
      ...
      case A_FUNCCALL:
        return (gen_funccall(n));
    }
```

The code to walk the new AST structure is here:

```
  // Generate the code to copy the arguments of a
  // function call to its parameters, then call the
  // function itself. Return the register that holds
  // the function's return value.
  static int gen_funccall(struct ASTnode *n) {
    struct ASTnode *gluetree = n->left;
    int reg;
    int numargs=0;

    // If there is a list of arguments, walk this list
    // from the last argument (right-hand child) to the
    // first
    while (gluetree) {
      // Calculate the expression's value
      reg = genAST(gluetree->right, NOLABEL, gluetree->op);
      // Copy this into the n'th function parameter: size is 1, 2, 3, ...
      cgcopyarg(reg, gluetree->v.size);
      // Keep the first (highest) number of arguments
      if (numargs==0) numargs= gluetree->v.size;
      genfreeregs();
      gluetree = gluetree->left;
    }

    // Call the function, clean up the stack (based on numargs),
    // and return its result
    return (cgcall(n->v.id, numargs));
  }
```

There are a few things to note. We generate the expression code by calling `genAST()` on the right child. Also, we set `numargs` to the first `size` value, which is the number of arguments (one-based not zero-based). Then we call `cgcopyarg()` to copy this value into the function's *n'th* parameter. Once the copy is done, we can free all our registers in preparation for the next expression, and walk down the left child for the previous expression.

Finally, we run `cgcall()` to generate the actual call to the function. Because we may have pushed argument values on the stack, we provide this with the number of arguments in total so it can work out how many to pop back off the stack.

There is no hardware-specific code here but, as I mentioned at the top, we are walking the expression tree from the last expression to the first. Not all architectures will want this, so there is room to make the code more flexible in terms of the order of evaluation.

Changes to `cg.c`

Now we get to the functions that generate actual x86-64 assembly code output. We have created a new one, `cgcopyarg()`, and modified an existing one, `cgcall()`.

But first, a reminder that we have these lists of registers:

```
#define FIRSTPARAMREG 9         // Position of first parameter register
static char *reglist[] =
  { "%r10", "%r11", "%r12", "%r13", "%r9", "%r8", "%rcx", "%rdx", "%rsi", "%rd

static char *breglist[] =
  { "%r10b", "%r11b", "%r12b", "%r13b", "%r9b", "%r8b", "%cl", "%dl", "%sil",

static char *dreglist[] =
  { "%r10d", "%r11d", "%r12d", "%r13d", "%r9d", "%r8d", "%ecx", "%edx", "%esi"
```

with FIRSTPARAMREG set to the last index position: we will walk backwards down this list.

Also, remember that the argument position numbers we will get are one-based (i.e 1, 2, 3, 4, ...) not zero-based (0, 1, 2, 3, ...), but the array above is zero-based. You will see a few `+1` or `-1` adjustments in the code below.

Here is `cgcopyarg()`:

```
// Given a register with an argument value,
// copy this argument into the argposn'th
// parameter in preparation for a future function
// call. Note that argposn is 1, 2, 3, 4, ..., never zero.
void cgcopyarg(int r, int argposn) {

  // If this is above the sixth argument, simply push the
  // register on the stack. We rely on being called with
  // successive arguments in the correct order for x86-64
  if (argposn > 6) {
    fprintf(Outfile, "\tpushq\t%s\n", reglist[r]);
  } else {
    // Otherwise, copy the value into one of the six registers
    // used to hold parameter values
    fprintf(Outfile, "\tmovq\t%s, %s\n", reglist[r],
            reglist[FIRSTPARAMREG - argposn + 1]);
  }
}
```

Nice and simple except for the `+1`. Now the code for `cgcall()`:

```
// Call a function with the given symbol id
// Pop off any arguments pushed on the stack
// Return the register with the result
int cgcall(int id, int numargs) {
  // Get a new register
```

```
    int outr = alloc_register();
    // Call the function
    fprintf(Outfile, "\tcall\t%s\n", Symtable[id].name);
    // Remove any arguments pushed on the stack
    if (numargs>6)
      fprintf(Outfile, "\taddq\t$%d, %%rsp\n", 8*(numargs-6));
    // and copy the return value into our register
    fprintf(Outfile, "\tmovq\t%%rax, %s\n", reglist[outr]);
    return (outr);
  }
```

Again, nice and simple.

Testing the Changes

In the last part of our compiler writing journey, we had two separate test programs `input27a.c` and `input27b.c` : we had to compile one of them with `gcc` . Now, we can combine them together and compile it all with our compiler. There is a second test program `input28.c` with some more examples of function calling. As always:

```
$ make test
cc -o comp1 -g -Wall cg.c decl.c expr.c gen.c main.c
    misc.c scan.c stmt.c sym.c tree.c types.c
(cd tests; chmod +x runtests; ./runtests)
  ...
input25.c: OK
input26.c: OK
input27.c: OK
input28.c: OK
```

Conclusion and What's Next

Right now, I feel that our compiler has just gone from being a "toy" compiler to one which is nearly useful: we can now write multi-function programs and call between the functions. It took a few steps to get there, but I think each step was not a giant one.

There is obviously still a big journey left. We need to add structs, unions, external identifiers and a pre-processor. Then we have to make the compiler more robust, provide better error detections, possibly add warnings etc. So, perhaps we are about half-way at this point.

In the next part of our compiler writing journey, I think I'm going to add the ability to write function prototypes. This will allow us to link in outside functions. I'm thinking of those original Unix functions and system calls which are `int` and `char *` based such as `open()`, `read()`, `write()`, `strcpy()` etc. It will be nice to compile some useful programs with our compiler. Next step