CS 434 Meeting 8 — 2/28/06

Anouncements

1. Phase 1.2 and assignment on grammars are due on Thursday.

Operand Descriptors (cont.)

1. In addition to providing a mechanism for communication between high-level code generation routines, the operand descriptor type is an essential component of the interface between the high and low level code-generation modules.

2. The type used for operand descriptors should be flexible enough to handle several possibilities:

   registers Since access to values in registers is generally faster than access to memory, we would like the code we generate to keep intermediate results in registers whenever possible.

   memory locations Since we will eventually run out of registers, our compiler may have to store some intermediate results in memory. Even if this were not the case, other factors would make it necessary/desirable to have operand descriptors for memory locations.

   constants When processing an expression like `x+1` we would like to produce the code:

   ```
   move   x,D1
   add    #1,D1
   ```

3. There are several details we do have to consider at the high level if we want to produce decent code. For example, a more realistic version of the routine shown above might be:

```
operand_desc *expr_gen(node * expr)
{ operand_desc *left_loc, *right_loc, *new_loc;

  switch (expr->internal.type) {
     .  .   .
  case Nplus:
  left_loc = expr_gen(expr->internal.child[0]);
  right_loc = expr_gen(expr->internal.child[1]);

  if is_temporary(left_loc) {
       output ‘‘add right_loc,left_loc’’;
       freeOp(right_loc);
       return left_loc;
  } else if is_temporary(right_loc) {
       output ‘‘add left_loc,right_loc’’;
       freeOp(left_loc)
       return right_loc;
  } else {
       new_loc = get_temporary();
       output ‘‘move left_loc,new_loc’’;
       output ‘‘add right_loc,new_loc’’;
       freeOp(left_loc);
       freeOp(right_loc);
       return new_loc;
  }
  break;
  .
  .
  .
```

   As a result, the low-level code generator's interface must provide the means to determine various properties of the locations described by operand descriptors:

   - is it a temporary?
   - is it an address register?

4. This code illustrates the distinction between a high-level code generation routine and the low-level utility functions that allocate temporaries and actually output instructions.

5. In this code we are assuming that the values returned by expr_gen,

operand descriptors, provide both a mechanism for communication between high-level code generation routines, and an important component of the interface between the low-level and high-level code generation routines.

Managing data temporaries

1. Dealing with the allocation of data temporaries (as opposed to address temporaries) will be fairly simple in your compiler for several reasons:
   - All data register are "equivalent", and
   - You can't run out of data temporaries because a memory temporary can always be used if all the registers are used up (with merely a performance penalty).
2. For data temporaries you should use a fairly simple allocation scheme.
   - The low-level code generator will provide "get" and "free" routines that return/expect operand descriptors for data temporaries.
   - Your low-level code generator should maintain a table with one entry for each data register indicating whether it is used of free.
   - When memory temporaries are used and freed, the allocator should reuse freed slots rather than always "bumping up the stack pointer".
     - To do this, maintain a list of freed memory temporaries.
     - When "get" is called:
       * First look for a free data register,
       * Next, look for a freed memory temporary,
       * If all else failed, bump up the procedure's local variable space size to make room for an additional memory temporary.
     - When "free" is called:
       * If operand is a constant just free descriptor memory,
       * If operand is a register, update free register table, or
       * If operand is a memory temporary, add it to the list of free temporary locations.
   - Later in the semester, we will want to know which operand is in which register (as opposed to which register each operand is in). In preparation for this, you should keep a pointer to the operand descriptor using each data register within your data register status table.

Handling Address Temporaries

1. When allocating memory to allocate temporary values (i.e. results of evaluating sub-expressions), we will want to distinguish situations where it is better to use an address register than a data register or memory temporary.
   - A reference such as

     a[3*i+j]

     is best handled by computing the address of a[3*i+j] and putting it in an address register and then returning an operand descriptor indicating that the result can be found at some displacement relative to the address register.
     - The generation of code for the address of a[3*i+j] will be accomplished by applying genExpr to a subtree describing the address arithmetic required.
     - We need a way to get genExpr to try to leave the result of this computation in an address register even though it would normally leave the result of an expression like 3*i+j in a data temporary.
     - To do this, we include an extra parameter in the definition of genExpr indicating the preferred destination type (address or data register). We will call this *temporary targeting*.
     - The setting of this parameter must be determined by context (it is an inherited attribute). In Woolite we will basically want to target the sub-expresions referred to by refvar nodes to address registers and target all other expressions for data registers or memory locations.

– The getTemporary function should also expect a parameter telling it whether to try to get an address register or a data temporary.
– So, an even more realistic code generation routine might be:

```
operand_desc *expr_gen(node * expr, int useAddr)
{
  operand_desc *left_loc, *right_loc, *new_loc;

  switch (expr->internal.type) {
     .
     .
     .
  case Nplus:
    left_loc = expr_gen(expr->internal.child[0],
                        useAddr);
    right_loc = expr_gen(expr->internal.child[1],
                         useAddr && ! isAddr(left_loc));

    if is_temporary(left_loc) {
         output ‘‘add right_loc,left_loc’’;
         freeOp(right_loc);
         return left_loc;
    } else if is_temporary(right_loc) {
         output ‘‘add left_loc,right_loc’’;
         freeOp(left_loc);
         return right_loc;
    } else {
         new_loc = get_temporary( useAddr );
         output ‘‘move left_loc,new_loc’’;
         output ‘‘add right_loc,new_loc’’;
         freeOp(left_loc);
         freeOp(right_loc);
         return new_loc;
    }
    break;
    .
    .
```

2. Temporarly location for values computed for use as addresses pose more challenges for your code generator because:

   - Some of them are reserved (frame pointer, stack pointer, etc.), and much more importantly,
   - You can run out (memory temporaries can not substitute for address register when you want to reference a location relative to a base address).

3. We will employ three “strategies” to address these issues when allocating address temporaries:

   Procrastination We will try to avoid committing an address register to hold a value for as long as possible.

   Indirection We will let an operand descriptor that refers to a memory location use another operand to describe the base address.

   Reclamation We will make sure that, when necessary, we can steal a busy address register by moving its value into a data register or into memory (even more) temporarily.

4. Indirection — Because operand descriptors can refer to address registers indirectly, it makes sense to define the operand descriptor type recursively:

   - Operand descriptors will be a union type with four possible subcomponents:
     (a) data register descriptor — just need a register number
     (b) address register descriptor — just need a register number
     (c) constant descriptor — just need constant value
     (d) memory operand descriptor — need a constant displacement and a base address specification.

- Ideally, the base addresses for memory operands would be stored in address registers, but to be more general we can represent the base address in a memory operand descriptor with a pointer to an operand descriptor for the base address computation's result.

5. Procrastination — We can take advantage of the idea of using an operand descriptor within an operand descriptor to reduce the total demand for address registers.

6. While complex variable such as a[3*i+j] require the use of a temporary for a base address, a simple reference (to a non-local and non-global variable) such as "x" can be handled by either:

   (a) immediately loading the needed static link pointer into a register and then returning an operand descriptor identifying the register and giving x's displacement from it, or

   (b) simply returning a descriptor identifying the static link needed to reference x and giving x's displacement from this pointer.

   - A reference to a static link pointer can be described by an operand descriptor specifying a displacement relative to the frame pointer, object pointer or another static link.

   In this case, the routine that is eventually asked to generate an instruction referencing the operand may first have to generate instructions to load the needed static link pointer.

7. The second approach avoids reserving a register unnecessarily. Consider the forms that code for an expression like $x+bigexpression$ might take using the two approaches.

```
        move    static-link-pointer,a1
(a)         code for big expression
        add     disp(a1),result-of-big-expr

            code for big expression
(b)     move    static-link-pointer,a1
        add     disp(a1),result-of-big-expr
```

   The second approach reduces the total demand for registers by reducing the range of instruction during which a register must be reserved for a base address.

8. This means that the low-level code generation routine to output an instruction may actually output several instructions.

   - the output-instruction routine will first need to ensure that all of operands the instruction should reference are addressable.
   - If any of the operand descriptors passed to output-instruction refer to a base address not currently in an address register, output-instruction will first have to output instructions to place the base address in an address register.

9. Reclamation — Even using this technique to avoid reserving address registers to hold static link pointers, you might still run out of address registers:

   a[i]+ (a[j] + (a[k] + (a[l] + (a[m] + a[n]))))

10. Luckily, when you do run out of address registers, you can always free one by moving its contents to a data temporary.

    - In order to do this, you must keep a table with an entry for each address register pointing to the operand descriptor currently using it. (I also want you to keep such pointer for data registers, but you won't use them for a while.)
    - When you move the contents of the address register to another location, you must update the operand descriptor that referred to it to describe the location of the data temporary.
    - It the operand was referenced as a base address by some other operand descriptor, it will have to be moved back into an address register when the code generator attempts to use the memory operand.
    - You have to be careful not to force-free an address register that you just allocated for a base address of another operand of the current instruction. You can do this by alway force-freeing the least recently allocated register.

11. With Woolite, it is possible to make a slight improvement on this scheme.

    - If an address register is being use indirectly by an operand descriptor that refers to a memory location, then you can load the operand from memory into a data temporary to free the address register holding the base address.
    - The advantage of this approach is that you won't later have to move the operand back into an address register.
    - For the course project, this little improvement is strictly optional. (It is unlikely you will run out of address register on a "real" program anyway.)