ALaRI

Software Compilers

CODE GENERATION FOR LLVM USING THE ACSE COMPILER

Professor:
S. Crespi Reghizzi

Tutors:

Giovanni Agosta

Andrea Di Biagio

Project Authors:

Antonino Battaglia

Nikolaos Christianos

A.A. 2008 /09

CONTENTS

1 Introduction

The Advanced Compiler System for Education (ACSE) is a compiler designed to translate source code written in LanCE language into an assembler code for the MACE architecture. Even though the ACSE is a simple compiler, it provides all the elements to perfectly understand how a compiler works, since it is able to perform actions such as create a control flow graph, execute a liveness analysis and make a register allocation.

Due to the fact that the ACSE is hardly restricted to generate code for the MACE architecture, this project aims to create a modified core for the ACSE compiler, which will allow to generate code for LLVM. Modifications mainly in the *back-end* have been made to reach the goal, but some other modifications in the *front-end* were used to extend the initial grammar supported by the parser, and allow the insertion of the **Switch** structure and the **Break** and **Continue** statements. This report briefly describes first some hints of the LLVM, then the changes made to the ACSE compiler's back-end and front-end, and finally the results obtained with the new compiler.

2 Low Level Virtual Machine (LLVM)

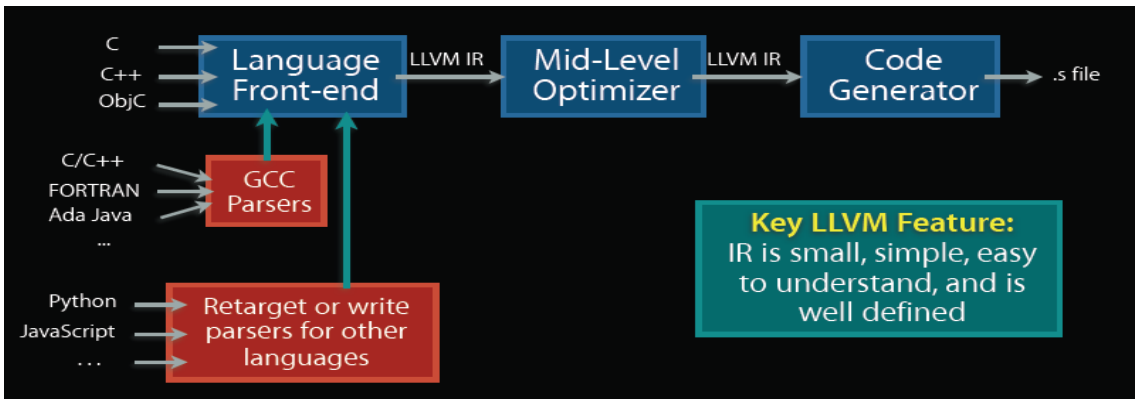
The **Low Level Virtual Machine**, generally known as **LLVM**, is a compiler infrastructure, written in C++ which is designed for compiler-time, link-time, run-time and "idle-time" optimization of programs written in arbitrary imperative programming language.

LLVM currently supports the compilation of C, C++, Objective-C, Ada, D and Fortran programs, using front-ends derived from GNU Compiler Collection (GCC).

Using LLVM, one can create compilers and code generators for specific machine architectures, and optimizers independent from particular platforms or languages. The LLVM intermediate representation (IR) is language and architecture independent; it lies between a language-specific module and a code generator for a specific machine. LLVM includes aggressive inter-procedural optimization support, static and JIT compilers.

LLVM supports a language-independent instruction set and type system. Most of the instructions have a form similar to three address code. Each instruction is in static assignment form (SSA), meaning that each variable (called a typed register) is assigned once and is frozen. This helps simplify the analysis of dependencies among variables.

LLVM has basic types, like integers of fixed sizes, and exactly five derived: pointers, arrays, vectors, structures, and functions. A type construct in a concrete language can be represented by combining these basic types in LLVM. For example, a class in C++ can be represented by a combination of structures, functions and arrays of function pointers.



3 Back-End

Due to the fact that LLVM instructions significantly differ from those in the MACE assembler, it was necessary to make a lot of changes in the components of the back-end, such as *axe_gencode, axe_constants, axe_expressions, axe_engine, axe_cflow_graph, axe_struct* and *Acse.y*.

Basically, the work involved to remove incompatible instructions, change the name to some other instructions, add some new ones, and modify the addressing modes in others.

3.1 Modifications

Significative changes were made inside the *gencode* file, where we added our new instructions, so as to perform compare with and without immediate, respectively constructed as binary and ternary instructions. Below there are two examples of the format of these two type of instructions we added:

```
t_axe_instruction * gen_icmp_EQi_instruction (t_program_infos *program,
                                int r_dest, int r_source1, int immediate)
{
return gen_binary_instruction
          (program, CEQI, r_dest, r_source1, immediate);
}
t_axe_instruction * gen_icmp_EQ_instruction (t_program_infos *program,
                   int r_dest, int r_source1, int r_source2, int flags)
{
 return gen_ternary_instruction
         (program, CEQ, r_dest, r_source1, r_source2, flags);
}
```

In the file *axe_constants.h* we defined the identifiers for the new instructions we added for compare instructions.

More tasks had to be done for *do-while*, *while* and *if* statements which require particular compare instructions and, above all, only conditional jump instructions with two labels for true and false conditions. MACE assembler doesn't support conditional branches with two labels, so modifications to file *axe_expression.c*, *Acse.y*, *axe_struct.h* and *axe_struct.c* were needed to represent LLVM format.

In *axe_expressions.c* we modified the *perform_binary_comparison()* function in such a way that we replaced the *sub/subi* instructions with our appropriate compare instructions. Considering only the case with one immediate operand we have :

```
switch(condition)
      {
         case _LT_ : gen_icmp_LTi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
         case _GT_ : gen_icmp_GTi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
        case _EQ_  : gen_icmp_EQi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
       case _NOTEQ_ : gen_icmp_NEQi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
        case _LTEQ_ : gen_icmp_LEi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
        case _GTEQ_ : gen_icmp_GEi_instruction (program, output_register,
                        exp1.value, exp2.value); break;
          default :
                notifyError(AXE_INVALID_EXPRESSION);
    }
```

In the files *axe_struct.h* and *axe_struct.c* we added two more structs, the do_statement and the if_statement, so as to create the appropriate labels for do-while and if structures.

Some modifications were made in the parser *Acse.y*, so as to create labels and place them in the correct point of the code and also to acquire the needed labels for the LLVM format.

An example of if-else statement can be seen in the following code:

```
Value* R0 = builder.CreateICmpEQ(R1, R2, "R0");
builder.CreateCondBr(R0, L1, L2);
builder.SetInsertPoint("L1");
      .
      .
builder.CreateBr(L3);
builder.SetInsertPoint("L2");
      .
      .
builder.SetInsertPoint("L3");
```

Finally in *axe_engine.c* we changed almost the whole *translateCodeSegment()* function in order to print the output according to the LLVM format, depending of the instruction we had to print.

3.2 Test - Results

In order to complete the program structure we have to attach the following LLVM code, to include the necessary libraries and to perform some standard steps, which are fixed for every LLVM program.

```
#include <llvm/Module.h>
#include <llvm/Function.h>
#include <llvm/PassManager.h>
#include <llvm/CallingConv.h>
#include <llvm/Analysis/Verifier.h>
#include <llvm/Assembly/PrintModulePass.h>
#include <llvm/Support/IRBuilder.h>

using namespace llvm;

Module* makeLLVMModule();

int main(int argc, char**argv) {
  Module* Mod = makeLLVMModule();
  verifyModule(*Mod, PrintMessageAction);
  PassManager PM;
  PM.add(new PrintModulePass(&llvm::cout));
  PM.run(*Mod);
  delete Mod;
  return 0;
   }
```

In order to prove the correctness of code implemented, we provide an example that states some representative aspects of code generation.

Source Code : power.src

```
int value, result, power, negative;

read(value);
read(power);

if (power == 0)
{
       write(0);
       return;
}

if (power < 0)
{
       negative = 1;
       power = -power;
}
else
       negative = 0;
result = value;

while (power > 1)
{
       result = result * value;
       power = power - 1;
}
if (negative == 1)
       result = 1 / result;

write(result);
```

Output Code : power.asm

```
builder.CreateAlloca(int, *L0);
builder.CreateAlloca(int, *L1);
builder.CreateAlloca(int, *L2);
builder.CreateAlloca(int, *L3);

R1 = builder.CreateCall(func_scanf, Attrs.begin(), Attrs.end(),
                        "",label_entry);
Value* R1 = builder.CreateStore(L0);
R2 = builder.CreateCall(func_scanf, Attrs.begin(), Attrs.end(),
                        "",label_entry);
Value* R0 = builder.CreateICmpEQ(R2, #0, "R0");
Value* R2 = builder.CreateStore(L2);
builder.CreateCondBr(R0, L5, L4);

builder.SetInsertPoint("L5");
Value* R3 = builder.CreateAdd(R0, #0, "R3");
R3 = builder.CreateCall(func_printf, Attrs.begin(), Attrs.end(),
                        "",label_entry);
builder.CreateRet();

builder.SetInsertPoint("L4");
Value* R2 = builder.CreateLoad(L2);
Value* R0 = builder.CreateICmpULE(R2, #0, "R0");
Value* R2 = builder.CreateStore(L2);
builder.CreateCondBr(R0, L7, L6);

builder.SetInsertPoint("L7");
Value* R3 = builder.CreateAdd(R0, #1, "R3");
Value* R3 = builder.CreateStore(L3);
Value* R2 = builder.CreateLoad(L2);
Value* R4 = builder.CreateSub(R0, R2, "R4");
Value* R2 = builder.CreateAdd(R0, R4, "R2");
Value* R2 = builder.CreateStore(L2);
builder.CreateBr(L8);

builder.SetInsertPoint("L6");
Value* R3 = builder.CreateAdd(R0, #0, "R3");
Value* R3 = builder.CreateStore(L3);

builder.SetInsertPoint("L8");
Value* R1 = builder.CreateLoad(L0);
Value* R4 = builder.CreateAdd(R0, R1, "R4");
Value* R4 = builder.CreateStore(L1);
Value* R1 = builder.CreateStore(L0);

builder.SetInsertPoint("L9");
Value* R2 = builder.CreateLoad(L2);
Value* R0 = builder.CreateICmpUGE(R2, #1, "R0");
Value* R2 = builder.CreateStore(L2);
builder.CreateCondBr(R0, L11, L10);

builder.SetInsertPoint("L11");
Value* R4 = builder.CreateLoad(L1);
Value* R1 = builder.CreateLoad(L0);
Value* R5 = builder.CreateMul(R4, R1, "R5");
Value* R1 = builder.CreateStore(L0);
Value* R4 = builder.CreateAdd(R0, R5, "R4");
Value* R4 = builder.CreateStore(L1);
Value* R2 = builder.CreateLoad(L2);
Value* R5 = builder.CreateSub(R2, #1, "R5");
Value* R2 = builder.CreateAdd(R0, R5, "R2");
Value* R2 = builder.CreateStore(L2);
builder.CreateBr(L9);

builder.SetInsertPoint("L10");
Value* R3 = builder.CreateLoad(L3);
```

```
Value* R0 = builder.CreateICmpEQ(R3, #1, "R0");
Value* R3 = builder.CreateStore(L3);
builder.CreateCondBr(R0, L13, L12);

builder.SetInsertPoint("L13");
Value* R3 = builder.CreateAdd(R0, #1, "R3");
Value* R4 = builder.CreateLoad(L1);
Value* R2 = builder.CreateSdiv(R3, R4, "R2");
Value* R4 = builder.CreateAdd(R0, R2, "R4");
Value* R4 = builder.CreateStore(L1);

builder.SetInsertPoint("L12");
Value* R4 = builder.CreateLoad(L1);
R4 = builder.CreateCall(func_printf, Attrs.begin(), Attrs.end(),
                        "",label_entry);
Value* R4 = builder.CreateStore(L1);
builder.CreateRet();
```

3.3 Notes

Due to the few documentations found about C++ API it's not so clear how arrays are represented and, above all, how operations involving array elements must be indicated.

To perform this task we have to modify *axe_array* and to implement some new instructions to extract all the parameters we need, such as the label related to the array, the index with which it's possible to access to the particular element of the array, and so on.

4 Front-end Modifications

In that part of the project he had to make some modifications at the front-end part of the ACSE compiler, so as to support the SWITCH statement and moreover the BREAK/ CONTINUE constructs. In order to implement the points mentioned above we had to modify the scanner (Acse.lex) and the parser (Acse.y) and two more files from the ACSE compiler (axe_struct.h and axe_struct.c).

4.1 Scanner Modifications

The only modifications we made in the Acse.lex file was to add in the tokens section the following tokens: SWITCH, CASE, BREAK, DEFAULT and CONTINUE, so as the scanner to be able to identify these tokens.

4.2 Parser Modifications

In this part we made the most important modifications for the front-end part of the ACSE compiler. The first thing was to declare two variables (var1, var2) that they are going to keep the values of the register is used to keep the variable indicated inside the switch block. We also declare one variable (switch_end) so as to keep track of the label at the end of the switch block, another variable (case_begin) to keep track of the label at the beginning of the case and finally one more variable (next_iter) to keep track of the label at the beginning of the next iteration.

```
int var1, var2;
t_axe_label *switch_end;
t_axe_label *case_begin;
t_axe_label *next_iter;
```

The next that we had to do was to change the semantic records, in order to add the switch and the do statements.

```
t_switch_statement switch_stmt;
t_do_statement do_stmt;
```

After that we insert at the tokens section our new token and we also change the token do from <label> to <do_stmt> so as to add the appropriate instructions in order to support the BREAK and the CONTINUE constructs.

```
%token <do_stmt> DO

/*New tokens*/
%token <switch_stmt> SWITCH
%token <label> CASE
%token <label> BREAK
%token <label> DEFAULT
%token <label> CONTINUE
```

After all the previous declarations we had to modify the syntactic rules section. First we declare the CASE, BREAK, DEFAULT and CONTINUE constructs.

```
| CASE exp COLON
{
        int index,location;

        $1 = reserveLabel(program);
        case_begin = $1;

        /*Generation of 2 registers*/
        index = getNewRegister(program);
        location = getNewRegister(program);
        /*Comparison*/
  gen_addi_instruction(program, location, REG_0, $2.value);
   gen_sub_instruction(program, index, location, var2,CG_DIRECT_ALL);
  gen_sne_instruction(program, index);
  gen_beq_instruction(program, $1, 0);
}
| DEFAULT COLON
   {/* We reserve the label when the first instrunction occurs */
        $1 = reserveLabel(program);
   /* We load the label so as to pass it
   to the fix level in the break definition. */
        case_begin = $1;
   }
| BREAK SEMI
{  /* We insert a BRANCH at the end of the switch block */
   gen_bt_instruction (program, switch_end, 0);
   fixLabel(program, case_begin);
}
| CONTINUE SEMI
   {
   /* We insert a proper BRANCH so as to start the new iteration */
        gen_bt_instruction (program, next_iter, 0);
   }
```

Then we have declared the switch_statement in the control_statements list.

```
control_statement : if_statement         { /* does nothing */ }
            | do_while_statement SEMI    { /* does nothing */ }
            | while_statement            { /* does nothing */ }
            | switch_statement           { /* does nothing */ }
            | return_statement SEMI      { /* does nothing */ };
```

The next thing that we had to was to introduce the BREAK and the CONTINUE to the while_statement.

```
                    next_iter = $1.label_condition;
                    switch_end = $1.label_end;
```

At this point we had to implement the SWITCH statement, the implementation is shown below:

```
switch_statement  : SWITCH LPAR IDENTIFIER RPAR
            {
                    /* initialize the value of the non-terminal */
                    $1 = create_switch_statement();

                   /* this label points at the beggining of the switch  statement*/
                    $1.label_condition = assignNewLabel(program);
                    fixLabel(program, $1.label_condition);

    /* this label points at the first instruction after the switch statement*/
                  $1.label_end = reserveLabel(program);

                 /* this label is used for the identification of the break */
                    switch_end = $1.label_end;

                  /* we genarate 1 register, so as to keep the value inside
    the switch condition, and we use a second one, REG_0,    which is 0 */
                    var2 = getNewRegister(program);
                    var1 = get_symbol_location(program, $3, 0);
    gen_add_instruction(program, var2, var1, REG_0, CG_DIRECT_ALL);

            }

             code_block
                  {
                        fixLabel(program, $1.label_end);
                  }
;
```

In order to finish with the modifications in the Acse.y we also had to make some changes in the do_while_statement part so as to support the BREAK and CONTINUE constructs, as we did in the while_statement, and one last modification at the if_stmt so as to support the BREAK construct, with the same procedure as before.

4.3 Declaration of new structs

The last thing that we had to change in the front-end was the files axe_struct.h and axe_struct.c. In the first file we have declared the structs for the switch, do and if statements.

```
typedef struct t_switch_statement
{
   t_axe_label *label_condition;   /* this label points to the expression
                                    * that is used as loop condition   */
   t_axe_label *label_end;         /* this label points to the instruction
                                    * that follows the while construct */
} t_switch_statement;

/*  We declare the struct of the do statement */

typedef struct t_do_statement
{
   t_axe_label *label_condition;   /* this label points to the expression
                                    * that is used as loop condition   */
   t_axe_label *label_end;         /* this label points to the instruction
                                    * that follows the while construct */
   t_axe_label *label_begin;

} t_do_statement;

/*  We declare the struct of the if statement */

typedef struct t_if_statement
{
   t_axe_label *label_end;         /* this label points to the instruction
                                    * that follows the while construct */
   t_axe_label *label_begin;

} t_if_statement;
```

In the axe_struct.c file we have created and initialized the structs they declared above.