ALaRI
Software Compilers

# CODE GENERATION FOR GNU LIGHTNING INSTRUCTION SET USING THE ACSE COMPILER

Tutors:
Stefano Crespi Reghizzi
Giovanni Agosta
Andrea DiBiagio

Authors:
Ioannis Argyris
Konstantinos Padarnitsas

A.A. 2009

# Introduction

The Advanced Compiler System for Education (ACSE) is a compiler deisgned to efficiently generate assembly code for the MACE architecture, from source code written in LanCE language. There are two main goals associated with this particular project. The first one, is to implement the switch and break/continue constructs, so as to be supported from the compiler. It is obvious that in order to achieve this, moderations in the front-end of the compiler must be performed. The second objective of the project is to rewrite part of the back-end of the compiler, in order to generate code for the GNU Lightning library.

In this report we are going to see all the alterations that were performed both in the front-end and back-end of the compiler, as well as an introduction to GNU Lightning's “architecture” and its instruction set.

# Front-end Modifications

The front-end of the compiler had to be modified in order to support switch and break/continue constructs. As the front-end supports LanCE, which is a C based language, these constructs had to be supported in their C format. To implement these statements we needed to perform some changes in the Lexical Analyzer and the Parser of the comiler. Particularly, we had to introduce new tokens inside the Lexical Analyzer, whereas in the Parser, the grammatical rules and the actions to control these tokens had to be added.

### *Scanner Modifications*

So as for the compiler to support any new constructs, the first step was to write them in its Lexical Analyzer *Acse.lex*.

### *Parser Modifications*

The second step is to add support in the parser (*Acse.y*). To do that we needed first to define some variables that were to be used for the implementation of the desired statements. So, inside this module many changes were made. In particular, in order to implement the switch construct in ACSE we have to modify the grammar by adding in the *control statements* list the new *switch statement.* On the other hand, the other keywords, such as *break*, *default*, *case* and *continue* are defined and implemented in the *statement* list. For them to work properly they are treated by introducing labels and branching to them accordingly. An example of the implementation of *continue* is shown below:

```
| CONTINUE SEMI

 {

      /* We insert an uncoditional branch to start the new
         iteration */

      gen_bt_instruction (program, label_continue, 0);

 }
```

We have to notice that *break*, *default* and *case* are used inside a switch statement and for this reason they are included in its implementation. *Break* and *continue* were also added to be supported in the *while* and *do-while* clauses. For this reason we had to write for the *do-while* clause a new kind of statement, which was included in *axe_struct.c* and *axe_struct.h*, so as to be able to handle more than one labels, namely three. Finally please have in mind that the support for all the above statements was only added in the original ACSE and so the generated code is assembly for the MACE architecture and not for GNU Lightning, which are going to see in the next paragraphs. An example of the generated code from a switch statement follows:

```
//switchcase.src
int a,b;
a = 0;
read(b);
switch( b )
{
    case 3 :
       a = a + 1;
       b = b - 1;
    break;
    case 4:
      b = b + 1;
    break;
    case 5:
      b = b + 2;
    break;
    default:
      b = b + 3;
    break;
     }
write(a);
```

```
//switchcase.asm
      .data
L0 :  .WORD 0
L1 :  .WORD 0
      .text
      ADDI R1 R0 #0
      STORE R1 L0
      READ R2 0
      STORE R2 L1
L2 :  LOAD R2 L1
      ADD R3 R2 R0
      STORE R2 L1
      ADDI R4 R0 #3
      SUB R0 R4 R3
      SNE R0 0
      BEQ L4
      LOAD R1 L0
      ADDI R4 R1 #1
      ADD R1 R0 R4
      STORE R1 L0
      LOAD R2 L1
      SUBI R4 R2 #1
      ADD R2 R0 R4
      STORE R2 L1
      BT L7
L4 :  ADDI R4 R0 #4
      SUB R0 R4 R3
      SNE R0 0
      BEQ L5
```

```
//continue of switchcase.asm
      LOAD R2 L1
      ADDI R4 R2 #1
      ADD R2 R0 R4
      STORE R2 L1
      BT L7
L5 :  ADDI R4 R0 #5
      SUB R0 R4 R3
      SNE R0 0
      BEQ L6
      LOAD R2 L1
      ADDI R3 R2 #2
      ADD R2 R0 R3
      STORE R2 L1
      BT L7
L6 :  LOAD R2 L1
      ADDI R3 R2 #3
      ADD R2 R0 R3
      STORE R2 L1
      BT L7
L7 :  LOAD R1 L0
      WRITE R1 0
      STORE R1 L0
      HALT
```

# GNU Lightning

*GNU lightning* is a C library that generates assembly language code at run-time; it is very fast, making it ideal for *Just-In-Time* compilers, and it abstracts over the target CPU, as it exposes to the clients a standardized RISC instruction set inspired by the MIPS and SPARC chips. It is usable in complex code generation tasks. The available backends cover the x86, SPARC and PowerPC architectures.

Dynamic code generation is the generation of machine code at runtime. It is typically used to strip a layer of interpretation by allowing compilation to occur at runtime. One of the most well-known applications of dynamic code generation is perhaps that of interpreters that compile source code to an intermediate bytecode form, which is then recompiled to machine code at run-time: this approach effectively combines the portability of bytecode representations with the speed of machine code. Another common application of dynamic code generation is in the field of hardware simulators and binary emulators, which can use the same techniques to translate simulated instructions to the instructions of the underlying machine. *GNU Lightning* provides a portable, fast and easily retargetable dynamic code generation system.

To be fast, *GNU lightning* emits machine code without first creating intermediate data structures such as RTL representations traditionally used by optimizing compilers. *GNU lightning* translates code directly from a machine independent interface to that of the underlying architecture. This makes code generation more efficient, since no intermediate data structures have to be constructed and consumed. A collateral benefit it that *GNU lightning* consumes little space: other than the memory needed to store generated instructions and data structures such as parse trees, the only data structure that client will usually need is an array of pointers to labels and unresolved jumps, which you can often allocate directly on the system stack.

To be portable, gnu *lightning* abstracts over current architectures' quirks and unorthogonalities. The interface that it exposes to is that of a standardized RISC architecture loosely based on the SPARC and MIPS chips. There are a few general-purpose registers (six, not including those used to receive and pass

parameters between subroutines), and arithmetic operations involve three operands—either three registers or two registers and an arbitrarily sized immediate value.

On one hand, this architecture is general enough that it is possible to generate pretty efficient code even on CISC architectures such as the Intel x86 or the Motorola 68k families. On the other hand, it matches real architectures closely enough that, most of the time, the compiler's constant folding pass ends up generating code which assembles machine instructions without further tests.

## *Gnu Lightning registers*

There are at least seven integer registers, of which six are general-purpose, while the last is used to contain the frame pointer (`FP`). The frame pointer can be used to allocate and access local variables on the stack, using the `allocai` instruction.

Of the general-purpose registers, at least three are guaranteed to be preserved across function calls (`V0`, `V1` and `V2`) and at least three are not (`R0`, `R1` and `R2`). Six registers are not very much, but this restriction was forced by the need to target CISC architectures which, like the x86, are poor of registers; anyway, backends can specify the actual number of available registers with the macros `JIT_R_NUM` (for caller-save registers) and `JIT_V_NUM` (for callee-save registers).

In addition, there is a special `RET` register which contains the return value of the current function (*not* the return value of callees—use the `retval` instruction for this). You should always remember, however, that writing this register could overwrite either a general-purpose register or an incoming parameter, depending on the architecture.

There are at least six floating-point registers, named FPR0 to FPR5. These are caller-save and are separate from the integer registers on all the supported architectures; on Intel architectures, the register stack is mapped to a flat register file. As for the integer registers, the macro JIT_FPR_NUM yields the number of floating-point registers, and the special FPRET register contains the return value of the current function. These registers were not implemented in our back-end in ACSE, since ACSE operates only with integers. For this same reason we had to implement only the integer instructions.

## *Gnu Lightning instructions*

Gnu *lightning*'s instruction set was designed by deriving instructions that closely match those of most existing RISC architectures, or that can be easily syntesized if absent. Each instruction is composed of:

- an operation like sub or mul
- sometimes, a register/immediate flag (r or I)
- a type identifier or, occasionally, two

The second and third field are separated by an underscore; thus, examples of legal mnemonics are `addr_i` (integer add, with three register operands) and `muli_l` (long integer multiply, with two

register operands and an immediate operand). Each instruction takes two or three operands; in most cases, one of them can be an immediate value instead of a register. Finally we have to notice that GNU Lightning supports many different data types, such as integer, float, double, etc. As ACSE supports only integer type, we were forced to implement only the integer operations. We consider that making ACSE support all the Lighning data types was out of the scope of this particular project.

### *GNU Lightning instruction set*

The instruction set of GNU Lightning follows. We are going to show all the instructions, along with their operation. It is very important though to understand, that this is not the way these instructions are written, as macro operations are used. We have to remember that GNU Lightning is a C library and not an assembly language.

**Binary ALU operations**

These accept three operands; the last one can be an immediate value for integer operands, or a register for all operand types. `addx` operations must directly follow `addc`, and `subx` must follow `subc`; otherwise, results are undefined.

```
addr     O1 = O2 + O3
addi     O1 = O2 + O3
addxr    O1 = O2 + (O3 + carry)
addxi    O1 = O2 + (O3 + carry)
addcr    O1 = O2 + O3, set carry
addci    O1 = O2 + O3, set carry
subr     O1 = O2 - O3
subi     O1 = O2 - O3
subxr    O1 = O2 - (O3 + carry)
subxi    O1 = O2 - (O3 + carry)
subcr    O1 = O2 - O3, set carry
subci    O1 = O2 - O3, set carry
rsbr     O1 = O3 - O2
rsbi     O1 = O3 - O2
mulr     O1 = O2 * O3
muli     O1 = O2 * O3
hmulr    O1 = high bits of O2 * O3
hmuli    O1 = high bits of O2 * O3
divr     O1 = O2 / O3
divi     O1 = O2 / O3
modr     O1 = O2 % O3
modi     O1 = O2 % O3
andr     O1 = O2 & O3
andi     O1 = O2 & O3
orr      O1 = O2 | O3
ori      O1 = O2 | O3
xorr     O1 = O2 ^ O3
xori     O1 = O2 ^ O3
lshr     O1 = O2 << O3
lshi     O1 = O2 << O3
rshr     O1 = O2 >> O3
```

```
rshi     O1 = O2 >> O3
```

**Unary ALU operations**

These accept two operands, both of which must be registers.

```
negr     O1 = -O2
notr     O1 = ~O2
```

**Compare instructions**

These accept three operands; again, the last can be an immediate value for integer data types. The last two operands are compared, and the first operand is set to either 0 or 1, according to whether the given condition was met or not.

```
ltr      O1 = (O2 <  O3)
lti      O1 = (O2 <  O3)
ler      O1 = (O2 <= O3)
lei      O1 = (O2 <= O3)
gtr      O1 = (O2 >  O3)
gti      O1 = (O2 >  O3)
ger      O1 = (O2 >= O3)
gei      O1 = (O2 >= O3)
eqr      O1 = (O2 == O3)
eqi      O1 = (O2 == O3)
ner      O1 = (O2 != O3)
nei      O1 = (O2 != O3)
```

**Transfer operations**

These accept two operands.

```
movr      O1 = O2
movi      O1 = O2
```

**Load operations**

`ld` accept two operands while `ldx` accept three; in both cases, the last can be either a register or an immediate value. Values are extended (with or without sign, according to the data type specification) to fit a whole register.

```
ldr     O1 = *O2
ldi     O1 = *O2
```

**Store operations**

`st` accept two operands while `stx` accept three; in both cases, the first can be either a register or an immediate value. Values are sign-extended to fit a whole register.

```
str     *O1 = O2
sti     *O1 = O2
```

**Branch instructions**

These return a value which, in this case, is to be used to compile forward branches as explained in They accept a pointer to the destination of the branch and two operands to be compared; of these, the

last can be either a register or an immediate.

```
bltr      if (O2 <  O3) goto O1
blti      if (O2 <  O3) goto O1
bler      if (O2 <= O3) goto O1
blei      if (O2 <= O3) goto O1
bgtr      if (O2 >  O3) goto O1
bgti      if (O2 >  O3) goto O1
bger      if (O2 >= O3) goto O1
bgei      if (O2 >= O3) goto O1
beqr      if (O2 == O3) goto O1
beqi      if (O2 == O3) goto O1
bner      if (O2 != O3) goto O1
bnei      if (O2 != O3) goto O1
bmsr      if O2 &  O3 goto O1
bmsi      if O2 &  O3 goto O1
bmcr      if !(O2 & O3) goto O1
bmci      if !(O2 & O3) goto O1
boaddr    O2 += O3, goto O1 on overflow
boaddi    O2 += O3, goto O1 on overflow
bosubr    O2 -= O3, goto O1 on overflow
bosubi    O2 -= O3, goto O1 on overflow
```

**Uncoditional jump and return**

```
jmpi      unconditional jump to O1
ret       return from subroutine
```

## *A simple GNU Lightning example*

Now that we have seen all the instructions, we are going to show a small example of GNU Lightning code, to understand its syntax and its use of macros. Below we can see a small program, which increments one variable.

```
#include <stdio.h>
#include "lightning.h"

static jit_insn codeBuffer[1024];

typedef int (*pifi)(int);    /* Pointer to Int Function of Int */

int main()
{
  pifi  incr = (pifi) (jit_set_ip(codeBuffer).iptr);
  int   in;

  jit_leaf(1);                    /*      leaf  1             */
  in = jit_arg_i();               /* in = arg_i               */
  jit_getarg_i(JIT_R0, in);       /*      getarg_i R0         */
  jit_addi_i(JIT_RET, JIT_R0, 1); /*      addi_i   RET, R0, 1 */
  jit_ret();                      /*      ret                 */
```

```
    jit_flush_code(codeBuffer, jit_get_ip().ptr);

    /* call the generated code, passing 5 as an argument */
    printf("%d + 1 = %d\n", 5, incr(5));
    return 0;
}
```

As we can see all the macros are written in the *jit_inst_i (arg);* format. The *i* is used to determine that this macro uses integer as its data type. Also, all the registers are written in the *JIT_REG* format. This is the format that we have to use for the generation of GNU Lightning code via ACSE. The final code that is going to be generated is just the true GNU Lightning code, which includes only the macros, without the C Language part.

# Back-end Modifications

ACSE has been designed for the generation of assembly code for the MACE architecture. In order to make ACSE able to generate code for GNU Lightning, a lot of changes had to be done in the back-end of the compiler. Specifically the components that were changed are:

- axe_gencode
- axe_cflow_graph
- axe_engine
- axe_array
- axe_costants
- axe_expression

Most of the changes were made in *axe_gencode.c* and *axe_gencode.h*, where, despite the fact that we kept many instructions, we had to add a lot of instructions that are supported by GNU Lightning and not by MACE. By keeping the instructions, which have the same function in MACE and Lightning, we had to adjust them all, so as to be generated in the proper format. For this reason massive changes had to be performed in the *axe_engine.c* file. Moreover, the *axe_array.c*, was almost re-written, since for the load and store functions we do not need to run so many instructions in GNU Lightning, as in MACE. Finally, we have to notice that for the proper generation of branches, some changes in the front-end (*Acse.y*) of the compiler had to be done, as to translate correctly the control statements of LanCE.

As far as the registers are concerned, as we have said before there are only six general purpose registers in GNU Lightning. For this reason we had to reduce the number of registers of MACE. However, as ACSE uses three registers for spill operations, we decided to keep nine registers instead of six, which were also renamed to have the proper format. A final thing we have to notice is the use of labels in GNU Lightning. Since there is no thing exact thing as labels, GNU Lighning exploits the integer data type of C to generate them. For this reason with the use `jit_get_label()`, we can use a variable as a label for the bracnhes. Variables and arrays are also used to store values in the same way as C, and with the macro `jit_allocai()` we can allocate as memory as we wish to store an array.

Finally, we are going to see an example of the generated GNU Lightning code in comparison to a source file and to the assembly code generated for the MACE architecture. We want to remind that the generated GNU Lightning code doesn't contain the total C structures, such as a *main()* or the include of the libraries:

```
//dowhile.src
int a,b;

a = 0;
read(b);

do
{
      a = a + 1;
      b = b - 1;
}while(b > 0);

write(a);
```

```
//dowhile.asm (MACE)
      .data
L0 :  .WORD 0
L1 :  .WORD 0
      .text
      ADDI R1 R0 #0
      STORE R1 L0
      READ R2 0
      STORE R2 L1
L2 :  LOAD R1 L0
      ADDI R3 R1 #1
      ADD R1 R0 R3
      STORE R1 L0
      LOAD R2 L1
      SUBI R3 R2 #1
      ADD R2 R0 R3
      SUBI R0 R2 #0
      STORE R2 L1
      SGT R0 0
      BNE L2
      LOAD R1 L0
      WRITE R1 0
      STORE R1 L0
      HALT
```

```
//dowhile.asm(GNU Lightning)
int *L0 = 0;
int *L1 = 0;

      jit_addi_i  (JIT_R1, JIT_R0, 0);
      jit_str_i   (*L0, JIT_R1);
      scanf       ("%d",&JIT_V0);
      jit_str_i   (*L1, JIT_V0);
jit_insn *L2;
L2 =  jit_get_label();
      jit_ldi_i   (JIT_R1, *L0);
      jit_addi_i  (JIT_V1, JIT_R1, 1);
      jit_addr_i  (JIT_R1, JIT_R0, JIT_V1);
      jit_ldi_i   (JIT_V0, *L1);
      jit_subi_i  (JIT_V1, JIT_V0, 1);
      jit_addr_i  (JIT_V0, JIT_R0, JIT_V1);
      jit_gti_i   (JIT_R0, JIT_V0, 0);
      jit_str_i   (*L1, JIT_V0);
      jit_movi_i  (JIT_R2, 0);
      jit_bner_i  (L2, JIT_R0, JIT_R2);
      printf      ("%d",JIT_R1);
      jit_str_i   (*L0, JIT_R1);
      jit_ret();
```