# Using and porting GNU *lightning*

Version 1.2
[No value for "UPDATE-MONTH"]

by Paolo Bonzini

# 1 Introduction to GNU *lightning*

This document describes installing, using and porting the GNU *lightning* library for dynamic code generation. Unlike other dynamic code generation systems, which are usually either inefficient or non-portable, GNU *lightning* is both retargetable and very fast.

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.

Yet other applications come to mind: for example, windowing *bitblt* operations, matrix manipulations, and network packet filters. Albeit very powerful and relatively well known within the compiler community, dynamic code generation techniques are rarely exploited to their full potential and, with the exception of the two applications described above, have remained curiosities because of their portability and functionality barriers: binary instructions are generated, so programs using dynamic code generation must be retargeted for each machine; in addition, coding a run-time code generator is a tedious and error-prone task more than a difficult one.

This manual describes the GNU *lightning* dynamic code generation library. 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 (see section "RTL representation" in *Using and porting GNU CC*). 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.

## 1.1 Drawbacks

GNU *lightning* has been useful in practice; however, it does have at least four drawbacks: it has limited registers, no peephole optimizer, no instruction scheduler and no symbolic debugger. Of these, the last is the most critical even though it does not affect the quality of generated code: the only way to debug code generated at run-time is to step through it at the level of host specific machine code. A decent knowledge of the underlying instruction set is thus needed to make sense of the debugger's output.

The low number of available registers (six) is also an important limitation. However, let's take the primary application of dynamic code generation, that is, bytecode translators. The underlying virtual machines tend to have very few general purpose registers (usually 0 to 2) and the translators seldom rely on sophisticated graph-coloring algorithms to allocate registers to temporary variables. Rather, these translators usually obtain performance increases because: a) they remove indirect jumps, which are usually poorly predicted, and thus often form a bottleneck, b) they parameterize the generated code and go through the process of decoding the bytecodes just once. So, their usage of registers is rather sparse—in fact, in practice, six registers were found to be enough for most purposes.

The lack of a peephole optimizer is most important on machines where a single instruction can map to multiple native instructions. For instance, Intel chips' division instruction hard-codes the dividend to be in EAX and the quotient and remainder to be output, respectively, in EAX and EDX: on such chips, GNU *lightning* does lots of pushing and popping of EAX and EDX to save those registers that are not used. Unnecessary stack operations could be removed by looking at whether preserved registers are destroyed soon. Unfortunately, the current implementation of GNU *lightning* is so fast because it only knows about the single instruction that is being generated; performing these optimizations would require a flow analysis pass that would probably hinder GNU *lightning*'s speed.

The lack of an instruction scheduler is not very important—pretty good instruction scheduling can actually be obtained by separating register writes from register reads. The only architectures on which a scheduler would be useful are those on which arithmetic instructions have two operands; an example is, again, the x86, on which the single instruction

```
subr_i  R0, R1, R2      !Compute R0 = R1 - R2
```

is translated to two instruction, of which the second depends on the result of the first:

```
movl    %ebx, %eax      ! Move R1 into R0

subl    %edx, %eax      ! Subtract R2 from R0
```

# 2 Using GNU *lightning*

This chapter describes installing and using GNU *lightning*.

## 2.1 Configuring and installing GNU *lightning*

The first thing to do to use GNU *lightning* is to configure the program, picking the set of macros to be used on the host architecture; this configuration is automatically performed by the '`configure`' shell script; to run it, merely type:

```
./configure
```

GNU *lightning* supports cross-compiling in that you can choose a different set of macros from the one needed on the computer that you are compiling GNU *lightning* on. For example,

```
./configure --host=sparc-sun-linux
```

will select the SPARC set of runtime assemblers. You can use configure's ability to make reasonable assumptions about the vendor and operating system and simply type

```
./configure --host=i386
./configure --host=ppc
./configure --host=sparc
```

Another option that '`configure`' accepts is `--enable-assertions`, which enables several consistency checks in the run-time assemblers. These are not usually needed, so you can decide to simply forget about it; also remember that these consistency checks tend to slow down your code generator.

After you've configured GNU *lightning*, you don't have to compile it because it is nothing more than a set of include files. If you want to compile the examples, run '`make`' as usual. The next important step is:

```
make install
```

This ends the process of installing GNU *lightning*.

## 2.2 GNU *lightning*'s instruction set

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, an 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.

GNU *lightning* supports a full range of integer types: operands can be 1, 2 or 4 bytes long (64-bit architectures might support 8 bytes long operands), either signed or unsigned. The types are listed in the following table together with the C types they represent:

| | |
|---|---|
| `c` | signed char |
| `uc` | unsigned char |
| `s` | short |
| `us` | unsigned short |
| `i` | int |
| `ui` | unsigned int |
| `l` | long |
| `ul` | unsigned long |
| `f` | float |
| `d` | double |
| `p` | void * |

Some of these types may not be distinct: for example, (e.g., `l` is equivalent to `i` on 32-bit machines, and `p` is substantially equivalent to `ul`).

There are at least seven integer registers, of which six are general-purpose, while the last is used to contain the stack pointer (`SP`). The stack pointer can be used to allocate and access local variables on the stack (which is supposed to grow downwards in memory on all architectures).

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 caller- and callee-save registers.

In addition, there is a special `RET` register which contains the return value. 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 separate from the integer registers on all the supported architectures; on Intel architectures, the register stack is mapped to a flat register file.

The complete instruction set follows; as you can see, most non-memory operations only take integers, long integers (either signed or unsigned) and pointers as operands; this was done in order to reduce the instruction set, and because most architectures only provide word and long word operations on registers. There are instructions that allow operands to be extended to fit a larger data type, both in a signed and in an unsigned way.

**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     i  ui  l  ul  p  f  d  O1 = O2 + O3
addi     i  ui  l  ul  p        O1 = O2 + O3
addxr    i  ui  l  ul           O1 = O2 + (O3 + carry)
addxi    i  ui  l  ul           O1 = O2 + (O3 + carry)
addcr    i  ui  l  ul           O1 = O2 + O3, set carry
addci    i  ui  l  ul           O1 = O2 + O3, set carry
subr     i  ui  l  ul  p  f  d  O1 = O2 - O3
subi     i  ui  l  ul  p        O1 = O2 - O3
```

```
subxr    i  ui  l  ul           O1 = O2 - (O3 + carry)
subxi    i  ui  l  ul           O1 = O2 - (O3 + carry)
subcr    i  ui  l  ul           O1 = O2 - O3, set carry
subci    i  ui  l  ul           O1 = O2 - O3, set carry
rsbr     i  ui  l  ul  p  f  d  O1 = O3 - O2
rsbi     i  ui  l  ul  p        O1 = O3 - O2
mulr     i  ui  l  ul     f  d  O1 = O2 * O3
muli     i  ui  l  ul           O1 = O2 * O3
hmulr    i  ui  l  ul           O1 = high bits of O2 * O3
hmuli    i  ui  l  ul           O1 = high bits of O2 * O3
divr     i  ui  l  ul     f  d  O1 = O2 / O3
divi     i  ui  l  ul           O1 = O2 / O3
modr     i  ui  l  ul           O1 = O2 % O3
modi     i  ui  l  ul           O1 = O2 % O3
andr     i  ui  l  ul           O1 = O2 & O3
andi     i  ui  l  ul           O1 = O2 & O3
orr      i  ui  l  ul           O1 = O2 | O3
ori      i  ui  l  ul           O1 = O2 | O3
xorr     i  ui  l  ul           O1 = O2 ^ O3
xori     i  ui  l  ul           O1 = O2 ^ O3
lshr     i  ui  l  ul           O1 = O2 << O3
lshi     i  ui  l  ul           O1 = O2 << O3
rshr     i  ui  l  ul           O1 = O2 >> O3[1]
rshi     i  ui  l  ul           O1 = O2 >> O3[2]
```

**Unary ALU operations**

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

```
negr     i     l         f  d  O1 = -O2
notr     i  ui l  ul          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.

The conditions given below are for the standard behavior of C, where the "unordered" comparison result is mapped to false.

```
ltr      i  ui  l  ul  p  f  d  O1 = (O2 <  O3)
lti      i  ui  l  ul  p        O1 = (O2 <  O3)
ler      i  ui  l  ul  p  f  d  O1 = (O2 <= O3)
lei      i  ui  l  ul  p        O1 = (O2 <= O3)
gtr      i  ui  l  ul  p  f  d  O1 = (O2 >  O3)
gti      i  ui  l  ul  p        O1 = (O2 >  O3)
ger      i  ui  l  ul  p  f  d  O1 = (O2 >= O3)
gei      i  ui  l  ul  p        O1 = (O2 >= O3)
eqr      i  ui  l  ul  p  f  d  O1 = (O2 == O3)
```

[1] The sign bit is propagated for signed types.

[2] The sign bit is propagated for signed types.

```
eqi      i  ui  l  ul  p        O1 = (O2 == O3)
ner      i  ui  l  ul  p  f  d  O1 = (O2 != O3)
nei      i  ui  l  ul  p        O1 = (O2 != O3)
unltr                   f  d  O1 = !(O2 >= O3)
unler                   f  d  O1 = !(O2 >  O3)
ungtr                   f  d  O1 = !(O2 <= O3)
unger                   f  d  O1 = !(O2 <  O3)
uneqr                   f  d  O1 = !(O2 <  O3) && !(O2 >  O3)
ltgtr                   f  d  O1 = !(O2 >= O3) || !(O2 <= O3)
ordr                    f  d  O1 =  (O2 == O2) &&  (O3 == O3)
unordr                  f  d  O1 =  (O2 != O2) ||  (O3 != O3)
```

**Transfer operations**

These accept two operands; for `ext` both of them must be registers, while `mov` accepts an immediate value as the second operand.

Unlike `movr` and `movi`, the other instructions are applied between operands of different data types, and they need **two** data type specifications. You can use `extr` to convert between integer data types, in which case the first must be smaller in size than the second; for example `extr_c_ui` is correct while `extr_ul_us` is not. You can also use `extr` to convert an integer to a floating point value: the only available possibilities are `extr_i_f` and `extr_i_d`. The other instructions convert a floating point value to an integer, so the possible suffixes are `_f_i` and `_d_i`.

```
movr                    i  ui  l  ul  p  f  d  O1 = O2
movi                    i  ui  l  ul  p  f  d  O1 = O2
extr        c  uc  s  us  i  ui  l  ul     f  d  O1 = O2
roundr                  i                 f  d  O1 = round(O2)
truncr                  i                 f  d  O1 = trunc(O2)
floorr                  i                 f  d  O1 = floor(O2)
ceilr                   i                 f  d  O1 = ceil(O2)
```

Note that the order of the arguments is *destination first, source second* as for all other GNU *lightning* instructions, but the order of the types is always reversed with respect to that of the arguments: *shorter*—source—*first, longer*—destination—*second*. This happens for historical reasons.

**Network extensions**

These accept two operands, both of which must be registers; these two instructions actually perform the same task, yet they are assigned to two mnemonics for the sake of convenience and completeness. As usual, the first operand is the destination and the second is the source.

| | | |
|---|---|---|
| `hton` | `us ui` | Host-to-network (big endian) order |
| `ntoh` | `us ui` | Network-to-host order |

**Load operations**

`ld` accepts two operands while `ldx` accepts 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     c  uc  s  us  i  ui  l  ul  p  f  d  O1 = *O2
```

```
ldi     c  uc  s  us  i  ui  l  ul  p  f  d  O1 = *O2
ldxr    c  uc  s  us  i  ui  l  ul  p  f  d  O1 = *(O2+O3)
ldxi    c  uc  s  us  i  ui  l  ul  p  f  d  O1 = *(O2+O3)
```

**Store operations**

`st` accepts two operands while `stx` accepts 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     c  uc  s  us  i  ui  l  ul  p  f  d  *O1 = O2
sti     c  uc  s  us  i  ui  l  ul  p  f  d  *O1 = O2
stxr    c  uc  s  us  i  ui  l  ul  p  f  d  *(O1+O2) = O3
stxi    c  uc  s  us  i  ui  l  ul  p  f  d  *(O1+O2) = O3
```

**Stack management**

These accept a single register parameter. These operations are not guaranteed to be efficient on all architectures.

```
pushr                    i  ui  l  ul  p    push O1 on the stack
popr                     i  ui  l  ul  p    pop O1 off the stack
```

**Argument management**

These are:

```
prepare                  i                 f  d
pusharg     c  uc  s  us  i  ui  l  ul  p  f  d
getarg      c  uc  s  us  i  ui  l  ul  p  f  d
arg         c  uc  s  us  i  ui  l  ul  p  f  d
```

Of these, the first two are used by the caller, while the last two are used by the callee. A code snippet that wants to call another procedure and has to pass registers must, in order: use the `prepare` instruction, giving the number of arguments to be passed to the procedure (once for each data type); use `pusharg` to push the arguments **in reverse order**; and use `calli` or `finish` (explained below) to perform the actual call.

`arg` and `getarg` are used by the callee. `arg` is different from other instruction in that it does not actually generate any code: instead, it is a function which returns a value to be passed to `getarg`.[3] You should call `arg` as soon as possible, before any function call or, more easily, right after the `prolog` or `leaf` instructions (which are treated later).

`getarg` accepts a register argument and a value returned by `arg`, and will move that argument to the register, extending it (with or without sign, according to the data type specification) to fit a whole register. These instructions are more intimately related to the usage of the GNU *lightning* instruction set in code that generates other code, so they will be treated more specifically in Section 2.3 [Generating code at run-time], page 10.

You should observe a few rules when using these macros. First of all, it is not allowed to call functions with more than six arguments; this was done to

[3] "Return a value" means that GNU *lightning* macros that compile these instructions return a value when expanded.

simplify and speed up the implementation on architectures that use registers for parameter passing.

You should not nest calls to `prepare`, nor call zero-argument functions (which do not need a call to `prepare`) inside a `prepare/calli` or `prepare/finish` block. Doing this might corrupt already pushed arguments.

You **cannot** pass parameters between subroutines using the six general-purpose registers. This might work only when targeting particular architectures.

On the other hand, it is possible to assume that callee-saved registers (`R0` through `R2`) are not clobbered by another dynamically generated function which does not use them as operands in its code and which does not return a value.

**Branch instructions**

Like `arg`, these also return a value which, in this case, is to be used to compile forward branches as explained in Section 2.3.4 [Fibonacci numbers], page 17. 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. They are:

```
bltr      i  ui  l  ul  p  f  d  if (O2 <  O3) goto O1
blti      i  ui  l  ul  p        if (O2 <  O3) goto O1
bler      i  ui  l  ul  p  f  d  if (O2 <= O3) goto O1
blei      i  ui  l  ul  p        if (O2 <= O3) goto O1
bgtr      i  ui  l  ul  p  f  d  if (O2 >  O3) goto O1
bgti      i  ui  l  ul  p        if (O2 >  O3) goto O1
bger      i  ui  l  ul  p  f  d  if (O2 >= O3) goto O1
bgei      i  ui  l  ul  p        if (O2 >= O3) goto O1
beqr      i  ui  l  ul  p  f  d  if (O2 == O3) goto O1
beqi      i  ui  l  ul  p        if (O2 == O3) goto O1
bner      i  ui  l  ul  p  f  d  if (O2 != O3) goto O1
bnei      i  ui  l  ul  p        if (O2 != O3) goto O1

bunltr                     f  d  if !(O2 >= O3) goto O1
bunler                     f  d  if !(O2 >  O3) goto O1
bungtr                     f  d  if !(O2 <= O3) goto O1
bunger                     f  d  if !(O2 <  O3) goto O1
buneqr                     f  d  if !(O2 <  O3) && !(O2 >  O3) goto O1
bltgtr                     f  d  if !(O2 >= O3) || !(O2 <= O3) goto O1
bordr                      f  d  if  (O2 == O2) &&  (O3 == O3) goto O1
bunordr                    f  d  if !(O2 != O2) ||  (O3 != O3) goto O1

bmsr      i ui l  ul             if O2 &  O3 goto O1
bmsi      i ui l  ul             if O2 &  O3 goto O1
bmcr      i ui l  ul             if !(O2 & O3) goto O1
bmci      i ui l  ul             if !(O2 & O3) goto O1[4]
boaddr    i ui l  ul             O2 += O3, goto O1 on overflow
boaddi    i ui l  ul             O2 += O3, goto O1 on overflow
bosubr    i ui l  ul             O2 -= O3, goto O1 on overflow
```

[4] These mnemonics mean, respectively, *branch if mask set* and *branch if mask cleared*.

| | | |
|---|---|---|
| `bosubi` | `i ui l ul` | `O2 -= O3`, goto `O1` on overflow |

**Jump and return operations**

These accept one argument except `ret` which has none; the difference between `finish` and `calli` is that the latter does not clean the stack from pushed parameters (if any) and the former must **always** follow a `prepare` instruction. Results are undefined when using function calls in a leaf function.

| | | |
|---|---|---|
| `calli` | `(not specified)` | function call to O1 |
| `callr` | `(not specified)` | function call to a register |
| `finish` | `(not specified)` | function call to O1 |
| `finishr` | `(not specified)` | function call to a register |
| `jmpi/jmpr` | `(not specified)` | unconditional jump to O1 |
| `prolog` | `(not specified)` | function prolog for O1 args |
| `leaf` | `(not specified)` | the same for leaf functions |
| `ret` | `(not specified)` | return from subroutine |
| `retval` | `c uc s us i ui l ul p f d` | move return value to register |

Like branch instruction, `jmpi` also returns a value which is to be used to compile forward branches. See Section 2.3.4 [Fibonacci numbers], page 17.

As a small appetizer, here is a small function that adds 1 to the input parameter (an `int`). I'm using an assembly-like syntax here which is a bit different from the one used when writing real subroutines with GNU *lightning*; the real syntax will be introduced in See Section 2.3 [Generating code at run-time], page 10.

```
incr:
     leaf      1
in = arg_i                    ! We have an integer argument

     getarg_i  R0, in         ! Move it to R0

     addi_i    RET, R0, 1     ! Add 1\, put result in return value

     ret                      ! And return the result
```

And here is another function which uses the `printf` function from the standard C library to write a number in hexadecimal notation:

```
printhex:
     prolog    1
in = arg_i                     ! Same as above

     getarg_i  R0, in
     prepare   2              ! Begin call sequence for printf

     pusharg_i R0             ! Push second argument

     pusharg_p "%x"           ! Push format string
```

```
finish    printf          ! Call printf

ret                       ! Return to caller
```

## 2.3 Generating code at run-time

To use GNU *lightning*, you should include the '`lightning.h`' file that is put in your include directory by the '`make install`' command. That include files defines about four hundred public macros (plus others that are private to GNU *lightning*), one for each opcode listed above.

Each of the instructions above translates to a macro. All you have to do is prepend `jit_` (lowercase) to opcode names and `JIT_` (uppercase) to register names. Of course, parameters are to be put between parentheses, just like with every other CPP macro.

This small tutorial presents three examples:

- The `incr` function found in Section 2.2 [GNU *lightning*'s instruction set], page 3:
- A simple function call to `printf`
- An RPN calculator.
- Fibonacci numbers

### 2.3.1 A function which increments a number by one

Let's see how to create and use the sample `incr` function created in Section 2.2 [GNU *lightning*'s instruction set], page 3:

```
#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;
  }
```

Let's examine the code line by line (well, almost...):

`#include "lightning.h"`
: You already know about this. It defines all of GNU *lightning*'s macros.

`static jit_insn codeBuffer[1024];`
: You might wonder about what is `jit_insn`. It is just a type that is defined by GNU *lightning*. Its exact definition depends on the architecture; in general, defining an array of 1024 `jit_insn`s allows one to write 100 to 400 GNU *lightning* instructions (depending on the architecture and exact instructions).

`typedef int (*pifi)(int);`
: Just a handy typedef for a pointer to a function that takes an `int` and returns another.

`pifi incr = (pifi) (jit_set_ip(codeBuffer).iptr);`
: This is the first GNU *lightning* macro we encounter that does not map to an instruction. It is `jit_set_ip`, which takes a pointer to an area of memory where compiled code will be put and returns the same value, cast to a `union` type whose members are pointers to functions returning different C types. This union is called `jit_code` and is defined as follows:

```
typedef union jit_code {
  char               *ptr;
  void               (*vptr)();
  char               (*cptr)();
  unsigned char      (*ucptr)();
  short              (*sptr)();
  unsigned short     (*usptr)();
  int                (*iptr)();
  unsigned int       (*uiptr)();
  long               (*lptr)();
  unsigned long      (*ulptr)();
  void *             (*pptr)();
  float              (*fptr)();
  double             (*dptr)();
} jit_code;
```

Any of the members could have been used, since the result is soon casted to type `pifi` but, for the sake of clarity, the program uses `iptr`, a pointer to a function with no prototype and returning an `int`.

Analogous to `jit_set_ip` is `jit_get_ip`, which does not modify the instruction pointer—it is nothing more than a cast of the current IP to `jit_code`.

`int in;` A footnote in Section 2.2 [GNU *lightning*'s instruction set], page 3, under the description of `arg`, says that macros implementing `arg` return a value—we'll be using this variable to store the result of `arg`.

`jit_leaf(1);`
Ok, so we start generating code for our beloved function... it will accept one argument and won't call any other function.

```
in = jit_arg_i();
jit_getarg_i(JIT_R0, in);
```
We retrieve the first (and only) argument, an integer, and store it into the general-purpose register `R0`.

`jit_addi_i(JIT_RET, JIT_R0, 1);`
We add one to the content of the register and store the result in the return value.

`jit_ret();`
This instruction generates a standard function epilog that returns the contents of the `RET` register.

`jit_flush_code(codeBuffer, jit_get_ip().ptr);`
This instruction is very important. It flushes the generated code area out of the processor's instruction cache, avoiding the processor executes bogus data that it happens to find there. The `jit_flush_code` function accepts the first and the last address to flush; we use `jit_get_ip` to find out the latter.

`printf("%d + 1 = %d", 5, incr(5));`
Calling our function is this simple—it is not distinguishable from a normal C function call, the only difference being that `incr` is a variable.

GNU *lightning* abstracts two phases of dynamic code generation: selecting instructions that map the standard representation, and emitting binary code for these instructions. The client program has the responsibility of describing the code to be generated using the standard GNU *lightning* instruction set.

Let's examine the code generated for `incr` on the SPARC and x86 architectures (on the right is the code that an assembly-language programmer would write):

**SPARC**

```
save %sp, -96, %sp
mov  %i0, %l0                    retl
add  %l0, 1,  %i0                add %o0, 1, %o0
ret
restore
```

In this case, GNU *lightning* introduces overhead to create a register window (not knowing that the procedure is a leaf procedure) and to move the argument to the general purpose register `R0` (which maps to `%l0` on the SPARC). The former overhead could be avoided by teaching GNU *lightning* about leaf procedures (see Chapter 4 [Future], page 53); the latter could instead be avoided by rewriting the getarg instruction as `jit_getarg_i(JIT_RET, in)`, which was not done in this example.

**x86**

```
pushl %ebp
movl  %esp, %ebp
pushl %ebx
pushl %esi
pushl %edi
movl  8(%ebp), %eax        movl 4(%esp), %eax
addl  $1, %eax             incl %eax
popl  %edi
popl  %esi
popl  %ebx
popl  %ebp
ret                        ret
```

In this case, the main overhead is due to the function's prolog and epilog, which is nine instructions long on the x86; a hand-written routine would not save unused callee-preserved registers on the stack. It is to be said, however, that this is not a problem in more complicated uses, because more complex procedure would probably use the `V0` through `V2` registers (`%ebx`, `%esi`, `%edi`); in this case, a hand-written routine would have included the prolog too. Also, a ten byte prolog would probably be a small overhead in a more complex function.

In such a simple case, the macros that make up the back-end compile reasonably efficient code, with the notable exception of prolog/epilog code.

### 2.3.2 A simple function call to `printf`

Again, here is the code for the example:

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

static jit_insn codeBuffer[1024];

typedef void (*pvfi)(int);       /* Pointer to Void Function of Int */


int main()
{
  pvfi          myFunction;             /* ptr to generated code */

  char          *start, *end;           /* a couple of labels */

  int           in;                     /* to get the argument */


  myFunction = (pvfi) (jit_set_ip(codeBuffer).vptr);
  start = jit_get_ip().ptr;
  jit_prolog(1);
  in = jit_arg_i();
```

```
    jit_movi_p(JIT_R0, "generated %d bytes\n");
    jit_getarg_i(JIT_R1, in);
    jit_prepare(2);
      jit_pusharg_i(JIT_R1);              /* push in reverse order */

      jit_pusharg_p(JIT_R0);
    jit_finish(printf);
    jit_ret();
    end = jit_get_ip().ptr;

    /* call the generated code\, passing its size as argument */

    jit_flush_code(start, end);
    myFunction(end - start);
  }
```

The function shows how many bytes were generated. Most of the code is not very interesting, as it resembles very closely the program presented in Section 2.3.1 [A function which increments a number by one], page 10.

For this reason, we're going to concentrate on just a few statements.

```
start = jit_get_ip().ptr;
...
end = jit_get_ip().ptr;
```

These two instruction call the `jit_get_ip` macro which was mentioned in Section 2.3.1 [A function which increments a number by one], page 10 too. In this case we use the only field of `jit_code` that is not a function pointer: `ptr`, which is a simple `char *`.

```
jit_movi_p(JIT_R0, "generated %d bytes\n");
```

Note the use of the '`p`' type specifier, which automatically casts the second parameter to an `unsigned long` to make the code more clear and less cluttered by typecasts.

```
jit_prepare(2);
jit_pusharg_i(JIT_R1);
jit_pusharg_p(JIT_R0);
jit_finish(printf);
```

Once the arguments to `printf` have been put in general-purpose registers, we can start a prepare/pusharg/finish sequence that moves the argument to either the stack or registers, then calls `printf`, then cleans up the stack. Note how GNU *lightning* abstracts the differences between different architectures and ABI's – the client program does not know how parameter passing works on the host architecture.

### 2.3.3 A more complex example, an RPN calculator

We create a small stack-based RPN calculator which applies a series of operators to a given parameter and to other numeric operands. Unlike previous examples, the code generator is

fully parameterized and is able to compile different formulas to different functions. Here is the code for the expression compiler; a sample usage will follow.

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

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


pifi compile_rpn(char *expr)
{
  pifi fn;
  int in;
  fn = (pifi) (jit_get_ip().iptr);
  jit_leaf(1);
  in = jit_arg_i();
  jit_getarg_i(JIT_R0, in);

  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) {
      expr += n - 1;
      jit_push_i(JIT_R0);
      jit_movi_i(JIT_R0, atoi(buf));
    } else if (*expr == '+') {
      jit_pop_i(JIT_R1);
      jit_addr_i(JIT_R0, JIT_R1, JIT_R0);
    } else if (*expr == '-') {
      jit_pop_i(JIT_R1);
      jit_subr_i(JIT_R0, JIT_R1, JIT_R0);
    } else if (*expr == '*') {
      jit_pop_i(JIT_R1);
      jit_mulr_i(JIT_R0, JIT_R1, JIT_R0);
    } else if (*expr == '/') {
      jit_pop_i(JIT_R1);
      jit_divr_i(JIT_R0, JIT_R1, JIT_R0);
    } else {
      fprintf(stderr, "cannot compile: %s\n", expr);
      abort();
    }
    ++expr;
  }
  jit_movr_i(JIT_RET, JIT_R0);
  jit_ret();
  return fn;
}
```

The principle on which the calculator is based is easy: the stack top is held in R0, while the remaining items of the stack are held on the hardware stack. Compiling an operand pushes the old stack top onto the stack and moves the operand into R0; compiling an operator pops the second operand off the stack into R1, and compiles the operation so that the result goes into R0, thus becoming the new stack top.

Try to locate a call to `jit_set_ip` in the source code. You will not find one; this means that the client has to manually set the instruction pointer. This technique has one advantage and one drawback. The advantage is that the client can simply set the instruction pointer once and then generate code for multiple functions, one after another, without caring about passing a different instruction pointer each time; see Section 2.4 [Re-entrant usage of GNU *lightning*], page 21 for the disadvantage.

Source code for the client (which lies in the same source file) follows:

```
static jit_insn codeBuffer[1024];

int main()
{
  pifi c2f, f2c;
  int i;

  jit_set_ip(codeBuffer);
  c2f = compile_rpn("9*5/32+");
  f2c = compile_rpn("32-5*9/");
  jit_flush_code(codeBuffer, jit_get_ip().ptr);

  printf("\nC:");
  for (i = 0; i <= 100; i += 10) printf("%3d ", i);
  printf("\nF:");
  for (i = 0; i <= 100; i += 10) printf("%3d ", c2f(i));
  printf("\n");

  printf("\nF:");
  for (i = 32; i <= 212; i += 10) printf("%3d ", i);
  printf("\nC:");
  for (i = 32; i <= 212; i += 10) printf("%3d ", f2c(i));
  printf("\n");
  return 0;
}
```

The client displays a conversion table between Celsius and Fahrenheit degrees (both Celsius-to-Fahrenheit and Fahrenheit-to-Celsius). The formulas are, $F(c) = c * 9/5 + 32$ and $C(f) = (f - 32) * 5/9$, respectively.

Providing the formula as an argument to `compile_rpn` effectively parameterizes code generation, making it possible to use the same code to compile different functions; this is what makes dynamic code generation so powerful.

The '`rpn.c`' file in the GNU *lightning* distribution includes a more complete (and more complex) implementation of `compile_rpn`, which does constant folding, allows the argument

to the functions to be used more than once, and is able to assemble instructions with an immediate parameter.

### 2.3.4 Fibonacci numbers

The code in this section calculates a variant of the Fibonacci sequence. While the traditional Fibonacci sequence is modeled by the recurrence relation:

f(0) = f(1) = 1
f(n) = f(n-1) + f(n-2)

the functions in this section calculates the following sequence, which is more interesting as a benchmark[5]:

nfibs(0) = nfibs(1) = 1
nfibs(n) = nfibs(n-1) + nfibs(n-2) + 1

The purpose of this example is to introduce branches. There are two kind of branches: backward branches and forward branches. We'll present the calculation in a recursive and iterative form; the former only uses forward branches, while the latter uses both.

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

static jit_insn codeBuffer[1024];

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


int main()
{
  pifi      nfibs = (pifi) (jit_set_ip(codeBuffer).iptr);
  int       in;                 /* offset of the argument */

  jit_insn  *ref;               /* to patch the forward reference */


        jit_prolog   (1);
  in =  jit_arg_ui   ();
        jit_getarg_ui(JIT_V0, in);              /* V0 = n */

  ref = jit_blti_ui  (jit_forward(), JIT_V0, 2);
        jit_subi_ui  (JIT_V1, JIT_V0, 1);       /* V1 = n-1 */

        jit_subi_ui  (JIT_V2, JIT_V0, 2);       /* V2 = n-2 */

        jit_prepare(1);
          jit_pusharg_ui(JIT_V1);
        jit_finish(nfibs);
```

[5] That's because, as is easily seen, the sequence represents the number of activations of the `nfibs` procedure that are needed to compute its value through recursion.

```
        jit_retval(JIT_V1);                         /* V1 = nfibs(n-1) */

        jit_prepare(1);
          jit_pusharg_ui(JIT_V2);
        jit_finish(nfibs);
        jit_retval(JIT_V2);                         /* V2 = nfibs(n-2) */

        jit_addi_ui(JIT_V1,  JIT_V1,  1);
        jit_addr_ui(JIT_RET, JIT_V1, JIT_V2);       /* RET = V1 + V2 + 1 */

        jit_ret();

    jit_patch(ref);                                 /* patch jump */

        jit_movi_i(JIT_RET, 1);                     /* RET = 1 */

        jit_ret();

    /* call the generated code\, passing 32 as an argument */

    jit_flush_code(codeBuffer, jit_get_ip().ptr);
    printf("nfibs(%d) = %d", 32, nfibs(32));
    return 0;
}
```

As said above, this is the first example of dynamically compiling branches. Branch instructions have three operands: two contains the values to be compared, while the first is a *label*; GNU *lightning* label's are represented as `jit_insn *` values. Unlike other instructions (apart from `arg`, which is actually a directive rather than an instruction), branch instructions also return a value which, as we see in the example above, can be used to compile forward references.

Compiling a forward reference is a two-step operation. First, a branch is compiled with a dummy label, since the actual destination of the jump is not yet known; the dummy label is returned by the `jit_forward()` macro. The value returned by the branch instruction is saved to be used later.

Then, when the destination of the jump is reached, another macro is used, `jit_patch()`. This macro must be called once for **every** point in which the code had a forward branch to the instruction following `jit_patch` (in this case a `movi_i` instruction).

Now, here is the iterative version:

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

static jit_insn codeBuffer[1024];

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

```
int main()
{
  pifi     nfibs = (pifi) (jit_set_ip(codeBuffer).iptr);
  int      in;                  /* offset of the argument */

  jit_insn *ref;                /* to patch the forward reference */

  jit_insn *loop;               /* start of the loop */


        jit_leaf     (1);
  in =  jit_arg_ui   ();
        jit_getarg_ui(JIT_R2, in);              /* R2 = n */

        jit_movi_ui  (JIT_R1, 1);
  ref = jit_blti_ui  (jit_forward(), JIT_R2, 2);
        jit_subi_ui  (JIT_R2, JIT_R2, 1);
        jit_movi_ui  (JIT_R0, 1);

  loop= jit_get_label();
        jit_subi_ui  (JIT_R2, JIT_R2, 1);       /* decr. counter */

        jit_addr_ui  (JIT_V0, JIT_R0, JIT_R1);  /* V0 = R0 + R1 */

        jit_movr_ui  (JIT_R0, JIT_R1);          /* R0 = R1 */

        jit_addi_ui  (JIT_R1, JIT_V0, 1);       /* R1 = V0 + 1 */

        jit_bnei_ui  (loop, JIT_R2, 0);         /* if (R2) goto loop; */


  jit_patch(ref);                               /* patch forward jump */

        jit_movr_ui  (JIT_RET, JIT_R1);         /* RET = R1 */

        jit_ret      ();

  /* call the generated code\, passing 36 as an argument */

  jit_flush_code(codeBuffer, jit_get_ip().ptr);
  printf("nfibs(%d) = %d", 36, nfibs(36));
  return 0;
}
```

This code calculates the recurrence relation using iteration (a `for` loop in high-level languages). There is still a forward reference (indicated by the `jit_forward`/`jit_patch`

pair); there are no function calls anymore: instead, there is a backward jump (the `bnei` at the end of the loop).

In this case, the destination address should be known, because the jumps lands on an instruction that has already been compiled. However the program must make a provision and remember the address where the jump will land. This is achieved with `jit_get_label`, yet another macro that is much similar to `jit_get_ip` but, instead of a `jit_code` union, it answers an `jit_insn *` that the branch macros accept.

Now, let's make one more change: let's rewrite the loop like this:

```
...

jit_delay(
      jit_movi_ui  (JIT_R1, 1),
ref = jit_blti_ui  (jit_forward(), JIT_R2, 2));
      jit_subi_ui  (JIT_R2, JIT_R2, 1);

loop= jit_get_label();
      jit_subi_ui  (JIT_R2, JIT_R2, 1);       /* decr. counter */

      jit_addr_ui  (JIT_V0, JIT_R0, JIT_R1);  /* V0 = R0 + R1 */

      jit_movr_ui  (JIT_R0, JIT_R1);          /* R0 = R1 */

jit_delay(
      jit_addi_ui  (JIT_R1, JIT_V0, 1),       /* R1 = V0 + 1 */

      jit_bnei_ui  (loop, JIT_R2, 0));        /* if (R2) goto loop; */


...
```

The `jit_delay` macro is used to schedule delay slots in jumps and branches. This is optional, but might lead to performance improvements in tight inner loops (of course not in a loop that is executed 35 times, but this is just an example).

`jit_delay` takes two GNU *lightning* instructions, a *delay instruction* and a *branch instruction*. Note that the two instructions must be written in execution order (first the delay instruction, then the branch instruction), **not** with the branch first. If the current machine has a delay slot, the delay instruction (or part of it) is placed in the delay slot after the branch instruction; otherwise, it emits the delay instruction before the branch instruction. The delay instruction must not depend on being executed before or after the branch.

Instead of `jit_patch`, you can use `jit_patch_at`, which takes two arguments: the first is the same as for `jit_patch`, and the second is the valued to be patched in. In other words, these two invocations have the same effect:

```
jit_patch (jump_pc);
jit_patch_at (jump_pc, jit_get_ip ());
```

Dual to branches and `jit_patch_at` are `jit_movi_p` and `jit_patch_movi`, which can also be used to implement forward references. `jit_movi_p` is carefully implemented to use

an encoding that is as long as possible, so that it can always be patched; in addition, like branches, it will return an address which is then passed to `jit_patch_movi`. The usage of `jit_patch_movi` is similar to `jit_patch_at`.

## 2.4 Re-entrant usage of GNU *lightning*

By default, GNU *lightning* is able to compile different functions at the same time as long as it happens in different object files, and on the other hand constrains code generation tasks to reside in a single object file.

The reason for this is not apparent, but is easily explained: the '`lightning.h`' header file defines its state as a `static` variable, so calls to `jit_set_ip` and `jit_get_ip` residing in different files access different instruction pointers. This was not done without reason: it makes the usage of GNU *lightning* much simpler, as it limits the initialization tasks to the bare minimum and removes the need to link the program with a separate library.

On the other hand, multi-threaded or otherwise concurrent programs require reentrancy in the code generator, so this approach cannot be the only one. In fact, it is possible to define your own copy of GNU *lightning*'s instruction state by defining a variable of type `jit_state` and `#define`-ing `_jit` to it:

```
struct jit_state lightning;
#define _jit lightning
```

You are free to define the `jit_state` variable as you like: `extern`, `static` to a function, `auto`, or global.

This feature takes advantage of an aspect of macros (*cascaded macros*), which is documented thus in CPP's reference manual:

> A cascade of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,
>
> ```
> #define BUFSIZE 1020
> #define TABLESIZE BUFSIZE
> ```
>
> This is not at all the same as defining `TABLESIZE` to be '`1020`'. The `#define` for `TABLESIZE` uses exactly the body you specify—in this case, `BUFSIZE`—and does not check to see whether it too is the name of a macro; it's only when you use `TABLESIZE` that the result of its expansion is checked for more macro names.
>
> This makes a difference if you change the definition of `BUFSIZE` at some point in the source file. `TABLESIZE`, defined as shown, will always expand using the definition of `BUFSIZE` that is currently in effect: #define BUFSIZE 1020 #define TABLESIZE BUFSIZE #undef BUFSIZE #define BUFSIZE 37
>
> Now `TABLESIZE` expands (in two stages) to '37'. (The `#undef` is to prevent any warning about the nontrivial redefinition of `BUFSIZE`.)

In the same way, `jit_get_label` will adopt whatever definition of `_jit` is in effect:

```
#define jit_get_label() (_jit.pc)
```

Special care must be taken when functions residing in separate files must access the same state. This could be the case, for example, if a special library contained function for strength reduction of multiplications to adds & shifts, or maybe of divisions to multiplications and

shifts. The function would be compiled using a single definition of `_jit` and that definition would be used whenever the function would be called.

Since GNU *lightning* uses a feature of the preprocessor to obtain re-entrancy, it makes sense to rely on the preprocessor in this case too.

The idea is to pass the current `struct jit_state` to the function:

```
static void
_opt_muli_i(jit, dest, source, n)
     register struct jit_state *jit;
     register int dest, source, n;
{
#define _jit          jit
...
#undef _jit
}
```

doing this unbeknownst to the client, using a macro in the header file:

```
extern void _opt_muli_i(struct jit_state *, int, int, int);

#define opt_muli_i(rd, rs, n) _opt_muli_i(&_jit, (rd), (rs), (n))
```

### 2.4.1 Registers

## 2.5 Accessing the whole register file

As mentioned earlier in this chapter, all GNU *lightning* back-ends are guaranteed to have at least six integer registers and six floating-point registers, but many back-ends will have more.

To access the entire register files, you can use the `JIT_R`, `JIT_V` and `JIT_FPR` macros. They accept a parameter that identifies the register number, which must be strictly less than `JIT_R_NUM`, `JIT_V_NUM` and `JIT_FPR_NUM` respectively; the number need not be constant. Of course, expressions like `JIT_R0` and `JIT_R(0)` denote the same register, and likewise for integer callee-saved, or floating-point, registers.

## 2.6 Using `autoconf` with GNU *lightning*

It is very easy to include GNU *lightning*'s source code (without the documentation and examples) into your program's distribution so that people don't need to have it installed in order to use it.

Here is a step by step explanation of what to do:

1. Run `lightningize` from your package's main distribution directory.

   ```
   lightningize
   ```

   If you're using Automake, you might be pleased to know that '`Makefile.am`' files will be already there.

2. If you're not using Automake and `aclocal`, instead, you should delete the '`Makefile.am`' files (they are of no use to you) and copy the contents of the '`lightning.m4`' file, found in `aclocal`'s macro repository (usually

'`/usr/share/aclocal`', to your '`configure.in`' or '`acinclude.m4`' or '`aclocal.m4`' file.

3. Include a call to the `LIGHTNING_CONFIGURE_IF_NOT_FOUND` macro in your '`configure.in`' file.

`LIGHTNING_CONFIGURE_IF_NOT_FOUND` will first look for a pre-installed copy of GNU *lightning* and, if it can be found, it will use it; otherwise, it will do exactly the same things that GNU *lightning*'s own configure script does. If GNU *lightning* is already installed, or if the configuration process succeeds, it will define the `HAVE_LIGHTNING` symbol.

In addtion, an Automake conditional named `HAVE_INSTALLED_LIGHTNING` will be set if GNU *lightning* is already installed, which can be used to set up include paths appropriately.

Finally, `LIGHTNING_CONFIGURE_IF_NOT_FOUND` accepts two optional parameters: respectively, an action to be taken if GNU *lightning* is available, and an action to be taken if it is not.

# 3 Porting GNU *lightning*

This chapter describes the process of porting GNU *lightning*. It assumes that you are pretty comfortable with the usage of GNU *lightning* for dynamic code generation, as described in Chapter 2 [Using GNU lightning], page 3.

## 3.1 An overview of the porting process

A particular port of GNU *lightning* is composed of four files. These have a common suffix which identifies the port (for example, `i386` or `ppc`), and a prefix that identifies their function; they are:

- '`asm-`*`suffix`*`.h`', which contains the description of the target machine's instruction format. The creation of this file is discussed in Section 3.3 [Creating the run-time assembler], page 26.
- '`core-`*`suffix`*`.h`', which contains the mappings from GNU *lightning*'s instruction set to the target machine's assembly language format. The creation of this file is discussed in Section 3.4 [Creating the platform-independent layer], page 29.
- '`funcs-`*`suffix`*`.h`', for now, only contains the definition of `jit_flush_code`. The creation of this file is briefly discussed in Section 3.5 [More complex tasks in the platform-independent layer], page 50.
- '`fp-`*`suffix`*`.h`', which contains the description of the target machine's instruction format and the internal macros for doing floating point computation. The creation of this file is discussed in Section 3.6 [Implementing macros for floating point], page 51.

Before doing anything, you have to add the ability to recognize the new port during the configuration process. This is explained in Section 3.2 [Automatically recognizing the new platform], page 25.

## 3.2 Automatically recognizing the new platform

Before starting your port, you have to add the ability to recognize the new port during the configure process. You only have to run '`config.guess`', which you'll find in the main distribution directory, and note down the first part of the output (up to the first dash).

Then, in the two files '`configure.in`' and '`lightning.m4`', lookup the line

```
case "$host_cpu" in
```

and, right after it, add the line:

```
cpu-name)  cpu=file-suffix          ;;
```

where *cpu-name* is the cpu as output by '`config.guess`', and *file-suffix* is the suffix that you are going to use for your files (see Section 3.1 [An overview of the porting process], page 25).

Now create empty files for your new port:

```
touch lightning/asm-xxx.h
touch lightning/fp-xxx.h
touch lightning/core-xxx.h
touch lightning/funcs-xxx.h
```

and run '`configure`', which should create the symlinks that are needed by `lightning.h`. This is important because it will allow you to use GNU *lightning* (albeit in a limited way) for testing even before the port is completed.

## 3.3 Creating the run-time assembler

The run-time assembler is a set of macros whose purpose is to assemble instructions for the target machine's assembly language, translating mnemonics to machine language together with their operands. While a run-time assembler is not, strictly speaking, part of GNU *lightning* (it is a private layer to be used while implementing the standard macros that are ultimately used by clients), designing a run-time assembler first allows you to think in terms of assembly language rather than binary code (ouch!...), making it considerably easier to write the standard macros.

Creating a run-time assembler is a tedious process rather than a difficult one, because most of the time will be spent collecting and copying information from the architecture's manual.

Macros defined by a run-time assembler are conventionally named after the mnemonic and the type of its operands. Examples took from the SPARC's run-time assembler are `ADDrrr`, a macro that assembles an `ADD` instruction with three register operands, and `SUBCCrir`, which assembles a `SUBCC` instruction whose second operand is an immediate and the remaining two are registers.

The first step in creating the assembler is to pick a convention for operand specifiers (`r` and `i` in the example above) and for register names. On the SPARC, this convention is as follows

**r** A register name. For every `r` in the macro name, a numeric parameter `RR` is passed to the macro, and the operand is assembled as `%r`*RR*.

**i** An immediate, usually a 13-bit signed integer (with exception for instructions such as `SETHI` and branches). The macros check the size of the passed parameter if GNU *lightning* is configured with `--enable-assertions`.

**x** A combination of two `r` parameters, which are summed to determine the effective address in a memory load/store operation.

**m** A combination of an `r` and `i` parameter, which are summed to determine the effective address in a memory load/store operation.

Additional macros can be defined that provide easier access to register names. For example, on the SPARC, `_Ro(3)` and `_Rg(5)` map respectively to `%o3` and `%g5`; on the x86, instead, symbolic representations of the register names are provided (for example, `_EAX` and `_EBX`).

CISC architectures sometimes have registers of different sizes–this is the case on the x86 where `%ax` is a 16-bit register while `%esp` is a 32-bit one. In this case, it can be useful to embed information on the size in the definition of register names. The x86 machine language, for example, represents all three of `%bh`, `%di` and `%edi` as 7; but the x86 run-time assemblers defines them with different numbers, putting the register's size in the upper nybble (for example, '`17h`' for `%bh` and '`27h`' for `%di`) so that consistency checks can be made on the operands' sizes when `--enable-assertions` is used.

The next important part defines the native architecture's instruction formats. These can be as few as ten on RISC architectures, and as many as fifty on CISC architectures. In the latter case it can be useful to define more macros for sub-formats (such as macros for different addressing modes) or even for sub-fields in an instruction. Let's see an example of these macros.

```
#define _2i( OP, RD, OP2, IMM)
        _I((_u2 (OP )<<30)  |  (_u5(RD)<<25)  |  (_u3(OP2)<<22)  |
            _u22(IMM)                                             )
```

The name of the macro, `_2i`, indicates a two-operand instruction comprising an immediate operand. The instruction format is:

```
.------.---------.------.-----------------------------------------.
|  OP  |   RD    | OP2  |                IMM                      |
|------+---------+------+-----------------------------------------|
|2 bits|  5 bits |3 bits|             22 bits                     |
|31-30 |  29-25  | 22-24|              0-21                       |
'------'---------'------'-----------------------------------------'
```

GNU *lightning* provides macros named `_sXX(OP)` and `_uXX(OP)`, where XX is a number between 1 and 31, which test[1] whether `OP` can be represented as (respectively) a signed or unsigned integer of the given size. What the macro above does, then, is to shift and OR together the different fields, ensuring that each of them fits the field.

Here is another definition, this time for the PowerPC architecture.

```
#define _X(OP,RD,RA,RB,XO,RC)
        _I((_u6 (OP)<<26)  |  (_u5(RD)<<21)  |  (_u5(RA)<<16)  |
           ( _u5(RB)<<11)  |  (_u10(XO)<<1)  |   _u1(RC)       )
```

Here is the bit layout corresponding to this instruction format:

```
.--------.--------.--------.--------.---------------------.-------.
|    OP  |   RD   |   RA   |   RB   |          XO         |   RC  |
|--------+--------+--------+--------+---------------------------|
| 6 bits | 5 bits | 5 bits | 5 bits |        10 bits      | 1 bit |
| 31-26  | 25-21  | 16-20  | 11-15  |        1-10         |   0   |
'--------'---------'-------'--------'---------------------'-------'
```

How do these macros actually generate code? The secret lies in the `_I` macro, which is one of four predefined macros which actually store machine language instructions in memory. They are `_B`, `_W`, `_I` and `_L`, respectively for 8-bit, 16-bit, 32-bit, and `long` (either 32-bit or 64-bit, depending on the architecture) values.

Next comes another set of macros (usually the biggest) which represents the actual mnemonics—macros such as `ADDrrr` and `SUBCCrir`, which were cited earlier in this chapter, belong to this set. Most of the times, all these macros will do is to use the "instruction format" macros, specifying the values of the fields in the different instruction formats. Let's see a few of these definitions, again taken from the SPARC assembler:

```
#define BAi(DISP)                       _2   (0, 0,  8, 2, DISP)
#define BA_Ai(DISP)                     _2   (0, 1,  8, 2, DISP)
```

[1] Only when `--enable-assertions` is used.

```
#define SETHIir(IMM, RD)                _2i  (0, RD, 4, IMM)

#define ADDrrr(RS1, RS2, RD)            _3   (2, RD,  0, RS1, 0, 0, RS2)
#define ADDrir(RS1, IMM, RD)            _3i  (2, RD,  0, RS1, 1,    IMM)
#define ADDCCrrr(RS1, RS2, RD)          _3   (2, RD, 16, RS1, 0, 0, RS2)
#define ADDCCrir(RS1, IMM, RD)          _3i  (2, RD, 16, RS1, 1,    IMM)
#define ANDrrr(RS1, RS2, RD)            _3   (2, RD,  1, RS1, 0, 0, RS2)
#define ANDrir(RS1, IMM, RD)            _3i  (2, RD,  1, RS1, 1,    IMM)
#define ANDCCrrr(RS1, RS2, RD)          _3   (2, RD, 17, RS1, 0, 0, RS2)
#define ANDCCrir(RS1, IMM, RD)          _3i  (2, RD, 17, RS1, 1,    IMM)
```

A few things have to be noted. For example:

- The SPARC assembly language sometimes uses a comma inside a mnemonic (for example, `ba,a`). This symbol is not allowed inside a CPP macro name, so it is replaced with an underscore; the same is done with the dots found in the PowerPC assembly language (for example, `andi.` is defined as `ANDI_rri`).
- It can be useful to group together instructions with the same instruction format, as doing this tends to make the source code more readable (numbers are put in the same columns).
- Using an editor without automatic wrap at end of line can be useful, since run-time assemblers tend to have very long lines.

A final touch is to define the synthetic instructions, which are usually found on RISC machines. For example, on the SPARC, the `LD` instruction has two synonyms (`LDUW` and `LDSW`) which are defined thus:

```
#define LDUWxr(RS1, RS2, RD)            LDxr(RS1, RS2, RD)
#define LDUWmr(RS1, IMM, RD)            LDmr(RS1, IMM, RD)
#define LDSWxr(RS1, RS2, RD)            LDxr(RS1, RS2, RD)
#define LDSWmr(RS1, IMM, RD)            LDmr(RS1, IMM, RD)
```

Other common case are instructions which take advantage of registers whose value is hard-wired to zero, and short-cut instructions which hard-code some or all of the operands:

```
/* Destination is %g0\, which the processor never overwrites. */

#define CMPrr(R1, R2)   SUBCCrrr(R1, R2, 0) /* subcc %r1\, %r2\, %g0 */


/* One of the source registers is hard-coded to be %g0. */

#define NEGrr(R,S)      SUBrrr(0, R, S)     /* sub %g0\, %rR\, %rS */


/* All of the operands are hard-coded. */

#define RET()           JMPLmr(31,8 ,0)     /* jmpl [%r31+8]\, %g0 */


/* One of the operands acts as both source and destination */
```

```
#define BSETrr(R,S)     ORrrr(R, S, S)      /* or %rR\, %rS\, %rS */
```

Specific to RISC computers, finally, is the instruction to load an arbitrarily sized immediate into a register. This instruction is usually implemented as one or two basic instructions:

1. If the number is small enough, an instruction is sufficient (`LI` or `ORI` on the PowerPC, `MOV` on the SPARC).
2. If the lowest bits are all zeroed, an instruction is sufficient (`LIS` on the PowerPC, `SETHI` on the SPARC).
3. Otherwise, the high bits are set first (with `LIS` or `SETHI`), and the result is then ORed with the low bits

Here is the definition of such an instruction for the PowerPC:

```
#define MOVEIri(R,I)      (_siP(16,I) ? LIri(R,I) :     \ /* case 1 */

                          (_uiP(16,I) ? ORIrri(R,0,I) : \ /* case 1 */

                          _MOVEIri(R, _HI(I), _LO(I)) ))  /* case 2/3 */


#define _MOVEIri(H,L,R)  (LISri(R,H), (L ? ORIrri(R,R,L) : 0))
```

and for the SPARC:

```
#define SETir(I,R)      (_siP(13,I) ? MOVir(I,R) : \
 _SETir(_HI(I), _LO(I), R))

#define _SETir(H,L,R)   (SETHIir(H,R), (L ? ORrir(R,L,R) : 0))
```

In both cases, `_HI` and `_LO` are macros for internal use that extract different parts of the immediate operand.

You should take a look at the run-time assemblers distributed with GNU *lightning* before trying to craft your own. In particular, make sure you understand the RISC run-time assemblers (the SPARC's is the simplest) before trying to decypher the x86 run-time assembler, which is significantly more complex.

## 3.4 Creating the platform-independent layer

The platform-independent layer is the one that is ultimately used by GNU *lightning* clients. Creating this layer is a matter of creating a hundred or so macros that comprise part of the interface used by the clients, as described in Section 2.2 [GNU *lightning*'s instruction set], page 3.

Fortunately, a number of these definitions are common to the different platforms and are defined just once in one of the header files that make up GNU *lightning*, that is, '`core-common.h`'.

Most of the macros are relatively straight-forward to implement (with a few caveats for architectures whose assembly language only offers two-operand arithmetic instructions). This section will cover the tricky points, before presenting the complete listing of the macros that make up the platform-independent interface provided by GNU *lightning*.

### 3.4.1 Implementing forward references

Implementation of forward references takes place in:

- The branch macros
- The `jit_patch_at` macros

Roughly speaking, the branch macros, as seen in Section 2.3 [Generating code at runtime], page 10, return a value that later calls to `jit_patch` or `jit_patch_at` use to complete the assembly of the forward reference. This value is usually the contents of the program counter after the branch instruction is compiled (which is accessible in the `_jit.pc` variable). Let's see an example from the x86 back-end:

```
#define jit_bmsr_i(label, s1, s2)                               \
   (TESTLrr((s1), (s2)), JNZm(label,0,0,0), _jit.pc)
```

The `bms` (*branch if mask set*) instruction is assembled as the combination of a `TEST` instruction (bit-wise AND between the two operands) and a `JNZ` instruction (jump if non-zero). The macro then returns the final value of the program counter.

`jit_patch_at` is one of the few macros that need to possess a knowledge of the machine's instruction formats. Its purpose is to patch a branch instruction (identified by the value returned at the moment the branch was compiled) to jump to the current position (that is, to the address identified by `_jit.pc`).

On the x86, the displacement between the jump and the landing point is expressed as a 32-bit signed integer lying in the last four bytes of the jump instruction. The definition of `_jit_patch_at` is:

```
#define jit_patch(jump_pc, pv)     (*_PSL((jump_pc) - 4) = \
   (pv) - (jump_pc))
```

The `_PSL` macro is nothing more than a cast to `long *`, and is used here to shorten the definition and avoid cluttering it with excessive parentheses. These type-cast macros are:

- `_PUC(X)` to cast to a `unsigned char *`.
- `_PUS(X)` to cast to a `unsigned short *`.
- `_PUI(X)` to cast to a `unsigned int *`.
- `_PSL(X)` to cast to a `long *`.
- `_PUL(X)` to cast to a `unsigned long *`.

On other platforms, notably RISC ones, the displacement is embedded into the instruction itself. In this case, `jit_patch_at` must first zero out the field, and then OR in the correct displacement. The SPARC, for example, encodes the displacement in the bottom 22 bits; in addition the right-most two bits are suppressed, which are always zero because instruction have to be word-aligned.

```
#define jit_patch_at(delay_pc, pv)   jit_patch_ (((delay_pc) - 1), (pv))

/* branch instructions return the address of the delay
* instruction—this is just a helper macro that makes the code more
* readable.
*/
```

```
#define jit_patch_(jump_pc, pv)   (*jump_pc =     \
 (*jump_pc & ~_MASK(22)) |     \
        ((_UL(pv) - _UL(jump_pc)) >> 2) & _MASK(22))
```

This introduces more predefined shortcut macros:

- `_UC(X)` to cast to a `unsigned char`.
- `_US(X)` to cast to a `unsigned short`.
- `_UI(X)` to cast to a `unsigned int`.
- `_SL(X)` to cast to a `long`.
- `_UL(X)` to cast to a `unsigned long`.
- `_MASK(N)` gives a binary number made of N ones.

Dual to branches and `jit_patch_at` are `jit_movi_p` and `jit_patch_movi`, since they can also be used to implement forward references. `jit_movi_p` should be carefully implemented to use an encoding that is as long as possible, and it should return an address which is then passed to `jit_patch_movi`. The implementation of `jit_patch_movi` is similar to `jit_patch_at`.

### 3.4.2 Common features supported by 'core-common.h'

The '`core-common.h`' file contains hundreds of macro definitions which will spare you defining a lot of things in the files the are specific to your port. Here is a list of the features that '`core-common.h`' provides.

**Support for common synthetic instructions**

These are instructions that can be represented as a simple operation, for example a bit-wise AND or a subtraction. '`core-common.h`' recognizes when the port-specific header file defines these macros and avoids compiler warnings about redefined macros, but there should be no need to define them. They are:

```
#define jit_extr_c_ui(d, rs)
#define jit_extr_s_ui(d, rs)
#define jit_extr_c_ul(d, rs)
#define jit_extr_s_ul(d, rs)
#define jit_extr_i_ul(d, rs)
#define jit_negr_i(d, rs)
#define jit_negr_l(d, rs)
```

**Support for the ABI**

All of `jit_prolog`, `jit_leaf` and `jit_finish` are not mandatory. If not defined, they will be defined respectively as an empty macro, as a synonym for `jit_prolog`, and as a synonym for `jit_calli`. Whether to define them or not in the port-specific header file, it depends on the underlying architecture's ABI—in general, however, you'll need to define at least `jit_prolog`.

**Support for uncommon instructions**

These are instructions that many widespread architectures lack. '`core-common.h`' is able to provide default definitions, but they are usually inefficient if the hardware provides a way to do these operations with a single instruction. They are extension with sign and "reverse subtraction" (that is, REG2=IMM−REG1):

```
#define jit_extr_c_i(d, rs)
#define jit_extr_s_i(d, rs)
#define jit_extr_c_l(d, rs)
#define jit_extr_s_l(d, rs)
#define jit_extr_i_l(d, rs)
#define jit_rsbi_i(d, rs, is)
#define jit_rsbi_l(d, rs, is)
#define jit_rsbi_p(d, rs, is)
```

**Conversion between network and host byte ordering**

These macros are no-ops on big endian systems. Don't define them on such systems; on the other hand, they are mandatory on little endian systems. They are:

```
#define jit_ntoh_ui(d, rs)
#define jit_ntoh_us(d, rs)
```

**Support for a "zero" register**

Many RISC architectures provide a read-only register whose value is hard-coded to be zero; this register is then used implicitly when referring to a memory location using a single register. For example, on the SPARC, an operand like `[%l6]` is actually assembled as `[%l6+%g0]`. If this is the case, you should define `JIT_RZERO` to be the number of this register; '`core-common.h`' will use it to implement all variations of the `ld` and `st` instructions. For example:

```
#define jit_ldi_c(d, is)        jit_ldxi_c(d, JIT_RZERO, is)
#define jit_ldr_i(d, rs)        jit_ldxr_c(d, JIT_RZERO, rs)
```

If available, JIT_RZERO is also used to provide more efficient definitions of the `neg` instruction (see "Support for common synthetic instructions", above).

**Synonyms** '`core-common.h`' provides a lot of trivial definitions which make the instruction set as orthogonal as possible. For example, adding two unsigned integers is exactly the same as adding two signed integers (assuming a two's complement representation of negative numbers); yet, GNU *lightning* provides both `jit_addr_i` and `jit_addr_ui` macros. Similarly, pointers and unsigned long integers behave in the same way, but GNU *lightning* has separate instruction for the two data types—those that operate on pointers usually include a typecast that makes programs clearer.

**Shortcuts** These define "synthetic" instructions whose definition is not as trivial as in the case of synonyms, but is anyway standard. This is the case for bitwise NOT (which is implemented by XORing a string of ones), "reverse subtraction" between registers (which is converted to a normal subtraction with the two source operands inverted), and subtraction of an immediate from a register (which is converted to an addition). Unlike `neg` and `ext` (see "Support for common synthetic instructions", above), which are simply non-mandatory, you must not define these functions.

**Support for `longs`**

On most systems, `longs` and `unsigned longs` are the same as, respectively, `ints` and `unsigned ints`. In this case, '`core-common.h`' defines operations on these types to be synonyms.

`jit_state`

Last but not least, '`core-common.h`' defines the `jit_state` type. Part of this `struct` is machine-dependent and includes all kinds of state needed by the back-end; this part is always accessible in a re-entrant way as `_jitl`. `_jitl` will be of type `struct jit_local_state`; this struct must be defined even if no state is required.

### 3.4.3 Supporting scheduling of delay slots

Delay slot scheduling is obtained by clients through the `jit_delay` macro. However this macro is not to be defined in the platform-independent layer, because GNU *lightning* provides a common definition in '`core-common.h`'.

Instead, the platform-independent layer must define another macro, called `jit_fill_delay_after`, which has to exchange the instruction to be scheduled in the delay slot with the branch instruction. The only parameter accepted by the macro is a call to a branch macro, which must be expanded **exactly once** by `jit_fill_delay_after`. The client must be able to pass the return value of `jit_fill_delay_after` to `jit_patch_at`.

There are two possible approaches that can be used in `jit_fill_delay_after`. They are summarized in the following pictures:

- The branch instructions assemble a NOP instruction which is then removed by `jit_fill_delay_after`.

```
  before                          |   after
----------------------------------+-----------------------------
  ...                             |
  <would-be delay instruction>    |    <branch instruction>
  <branch instruction>            |    <delay instruction>
  NOP                             |           <--- _jit.pc
           <--- _jit.pc           |
```

- The branch instruction assembles the branch so that the delay slot is annulled, `jit_fill_delay_after` toggles the bit:

```
  before                          |   after
----------------------------------+-----------------------------
  ...                             |
  <would-be delay instruction>    |    <branch instruction>
  <branch with annulled delay>    |    <delay instruction>
           <--- _jit.pc           |           <--- _jit.pc
```

Don't forget that you can take advantage of delay slots in the implementation of boolean instructions such as `le` or `gt`.

### 3.4.4 Supporting arbitrarily sized immediate values

This is a problem that is endemic to RISC machines. The basic idea is to reserve one or two register to represent large immediate values. Let's see an example from the SPARC:

```
 addi_i R0, V2, 45          |  addi_i R0, V2, 10000
----------------------------+----------------------------
 add    %l5, 45, %l0        |  set    10000, %l6
                            |  add    %l5, %l6, %l0
```

In this case, `%l6` is reserved to be used for large immediates. An elegant solution is to use an internal macro which automatically decides which version is to be compiled.

Beware of register conflicts on machines with delay slots. This is the case for the SPARC, where `%l7` is used instead for large immediates in compare-and-branch instructions. So the sequence

```
jit_delay(
   jit_addi_i(JIT_R0, JIT_V2, 10000),
   jit_blei_i(label, JIT_R1, 20000)
);
```

is assembled this way:

```
set 10000, %l6        ! prepare immediate for add

set 20000, %l7        ! prepare immediate for cmp

cmp %l1, %l7
ble label
add %l5, %l6, %l0     ! delay slot
```

Note that using `%l6` in the branch instruction would have given an incorrect result—`R0` would have been filled with the value of `V2+`*`20000`* rather than `V2+`*`10000`*.

### 3.4.5 Implementing the ABI

Implementing the underlying architecture's ABI is done in the macros that handle function prologs and epilogs and argument passing.

Let's look at the prologs and epilogs first. These are usually pretty simple and, what's more important, with constant content—that is, they always generate exactly the same instruction sequence. Here is an example:

```
SPARC                          x86
save %sp, -96, %sp             push %ebp
                               push %ebx
                               push %esi
                               push %edi
                               movl %esp, %ebp
...                            ...
ret                            popl %edi
restore                        popl %esi
                               popl %ebx
                               popl %ebp
                               ret
```

The registers that are saved (`%ebx`, `%esi`, `%edi`) are mapped to the `V0` through `V2` registers in the GNU *lightning* instruction set.

Argument passing is more tricky. There are basically three cases[2]:

[2] For speed and ease of implementation, GNU *lightning* does not currently support passing some of the parameters on the stack and some in registers.

**Register windows**
: Output registers are different from input registers—the prolog takes care of moving the caller's output registers to the callee's input registers. This is the case with the SPARC.

**Passing parameters via registers**
: In this case, output registers are the same as input registers. The program must take care of saving input parameters somewhere (on the stack, or in non-argument registers). This is the case with the PowerPC.

**All the parameters are passed on the stack**
: This case is by far the simplest and is the most common in CISC architectures, like the x86 and Motorola 68000.

In all cases, the port-specific header file will define two variable for private use—one to be used by the caller during the `prepare/pusharg/finish` sequence, one to be used by the callee, specifically in the `jit_prolog` and `jit_arg` macros.

Let's look again, this time with more detail, at each of the cases.

**Register windows**
: `jit_finish` is the same as `jit_calli`, and is defined in '`core-common.h`' (see Section 3.4.2 [Common features supported by '`core-common.h`'], page 31).

```
#define jit_prepare_i(numargs)  (_jitl.pusharg = _Ro(numargs))
#define jit_pusharg_i(rs)       (--_jitl.pusharg,          \
                                 MOVrr((rs), _jitl.pusharg))
```

: Remember that arguments pushing takes place in reverse order, thus giving a pre-decrement (rather than post-increment) in `jit_pusharg_i`.

: Here is what happens on the callee's side:

```
#define jit_arg_c()           (_jitl.getarg++)
#define jit_getarg_c(rd, ofs) jit_extr_c_i  ((rd), (ofs))
#define jit_prolog(numargs)   (SAVErir(JIT_SP, -96, JIT_SP), \
                               _jitl.getarg = _Ri(0))
```

: The `jit_arg` macros return nothing more than a register index, which is then used by the `jit_getarg` macros. `jit_prolog` resets the counter used by `jit_arg` to zero; the `numargs` parameter is not used. It is sufficient for `jit_leaf` to be a synonym for `jit_prolog`.

**Passing parameter via registers**
: The code is almost the same as that for the register windows case, but with an additional complexity—`jit_arg` will transfer the argument from the input register to a non-argument register so that function calls will not clobber it. The prolog and epilog code can then become unbearably long, up to 20 instructions on the PPC; a common solution in this case is that of *trampolines*.

: The prolog does nothing more than put the function's actual address in a caller-preserved register and then call the trampoline:

```
mflr    r0                  ! grab return address

movei   r10, trampo_2args   ! jump to trampoline
```

```
        mtlr    r10
        blrl
here:   mflr    r31             ! r31 = address of epilog

        ...actual code...

        mtlr    r31             ! return to the trampoline

        blr
```

In this case, `jit_prolog` does use its argument containing the number of parameters to pick the appropriate trampoline. Here, `trampo_2args` is the address of a trampoline designed for 2-argument functions.

The trampoline executes the prolog code, jumps to the contents of `r10`, and upon return from the subroutine it executes the epilog code.

**All the parameters are passed on the stack**

`jit_pusharg` uses a hardware push operation, which is commonly available on CISC machines (where this approach is most likely followed). Since the stack has to be cleaned up after the call, `jit_prepare_i` remembers how many parameters have been put there, and `jit_finish` adjusts the stack pointer after the call.

```
#define jit_prepare_i(numargs) (_jitl.args += (numargs))
#define jit_pusharg_i(rs)      PUSHLr(rs)
#define jit_finish(sub)        (jit_calli((sub)),                \
                                ADDLir(4 * _jitl.args, JIT_SP), \
                                _jitl.numargs = 0)
```

Note the usage of `+=` in `jit_prepare_i`. This is done so that one can defer the popping of the arguments that were saved on the stack (*stack pollution*). To do so, it is sufficient to use `jit_calli` instead of `jit_finish` in all but the last call.

On the caller's side, `arg` returns an offset relative to the frame pointer, and `getarg` loads the argument from the stack:

```
#define jit_getarg_c(rd, ofs) jit_ldxi_c((rd), _EBP, (ofs));
#define jit_arg_c()           ((_jitl.frame += sizeof(int) \
                                             -  sizeof(int))
```

The `_jitl.frame` variable is initialized by `jit_prolog` with the displacement between the value of the frame pointer (`%ebp`) and the address of the first parameter.

These schemes are the most used, so '`core-common.h`' provides a way to employ them automatically. If you do not define the `jit_getarg_c` macro and its companions, '`core-common.h`' will presume that you intend to pass parameters through either the registers or the stack.

If you define `JIT_FP`, stack-based parameter passing will be employed and the `jit_getarg` macros will be defined like this:

```
#define jit_getarg_c(reg, ofs)  jit_ldxi_c((reg), JIT_FP, (ofs));
```

In other words, the `jit_arg` macros (which are still to be defined by the platform-specific back-end) shall return an offset into the stack frame. On the other hand, if you don't define `JIT_FP`, register-based parameter passing will be employed and the `jit_arg` macros shall return a register number; in this case, `jit_getarg` will be implemented in terms of `jit_extr` and `jit_movr` operations:

```
#define jit_getarg_c(reg, ofs) jit_extr_c_i  ((reg), (ofs))
#define jit_getarg_i(reg, ofs) jit_movr_i    ((reg), (ofs))
```

### 3.4.6 Macros composing the platform-independent layer

**Register names (all mandatory but the last two)**

```
#define JIT_R
#define JIT_R_NUM
#define JIT_V
#define JIT_V_NUM
#define JIT_FPR
#define JIT_FPR_NUM
#define JIT_SP
#define JIT_FP
#define JIT_RZERO
```

**Helper macros (non-mandatory):**

```
#define jit_fill_delay_after(branch)
```

**Mandatory:**

```
#define jit_arg_c()
#define jit_arg_i()
#define jit_arg_l()
#define jit_arg_p()
#define jit_arg_s()
#define jit_arg_uc()
#define jit_arg_ui()
#define jit_arg_ul()
#define jit_arg_us()
#define jit_abs_d(rd,rs)
#define jit_addi_i(d, rs, is)
#define jit_addr_d(rd,s1,s2)
#define jit_addr_i(d, s1, s2)
#define jit_addxi_i(d, rs, is)
#define jit_addxr_i(d, s1, s2)
#define jit_andi_i(d, rs, is)
#define jit_andr_i(d, s1, s2)
#define jit_beqi_i(label, rs, is)
#define jit_beqr_d(label, s1, s2)
#define jit_beqr_i(label, s1, s2)
#define jit_bgei_i(label, rs, is)
#define jit_bgei_ui(label, rs, is)
```

```
#define jit_bger_d(label, s1, s2)
#define jit_bger_i(label, s1, s2)
#define jit_bger_ui(label, s1, s2)
#define jit_bgti_i(label, rs, is)
#define jit_bgti_ui(label, rs, is)
#define jit_bgtr_d(label, s1, s2)
#define jit_bgtr_i(label, s1, s2)
#define jit_bgtr_ui(label, s1, s2)
#define jit_blei_i(label, rs, is)
#define jit_blei_ui(label, rs, is)
#define jit_bler_d(label, s1, s2)
#define jit_bler_i(label, s1, s2)
#define jit_bler_ui(label, s1, s2)
#define jit_bltgtr_d(label, s1, s2)
#define jit_blti_i(label, rs, is)
#define jit_blti_ui(label, rs, is)
#define jit_bltr_d(label, s1, s2)
#define jit_bltr_i(label, s1, s2)
#define jit_bltr_ui(label, s1, s2)
#define jit_bmci_i(label, rs, is)
#define jit_bmcr_i(label, s1, s2)
#define jit_bmsi_i(label, rs, is)
#define jit_bmsr_i(label, s1, s2)
#define jit_bnei_i(label, rs, is)
#define jit_bner_d(label, s1, s2)
#define jit_bner_i(label, s1, s2)
#define jit_boaddi_i(label, rs, is)
#define jit_boaddi_ui(label, rs, is)
#define jit_boaddr_i(label, s1, s2)
#define jit_boaddr_ui(label, s1, s2)
#define jit_bordr_d(label, s1, s2)
#define jit_bosubi_i(label, rs, is)
#define jit_bosubi_ui(label, rs, is)
#define jit_bosubr_i(label, s1, s2)
#define jit_bosubr_ui(label, s1, s2)
#define jit_buneqr_d(label, s1, s2)
#define jit_bunger_d(label, s1, s2)
#define jit_bungtr_d(label, s1, s2)
#define jit_bunler_d(label, s1, s2)
#define jit_bunltr_d(label, s1, s2)
#define jit_bunordr_d(label, s1, s2)
#define jit_calli(label)
#define jit_callr(label)
#define jit_ceilr_d_i(rd, rs)
#define jit_divi_i(d, rs, is)
#define jit_divi_ui(d, rs, is)
#define jit_divr_d(rd,s1,s2)
```

```
#define jit_divr_i(d, s1, s2)
#define jit_divr_ui(d, s1, s2)
#define jit_eqi_i(d, rs, is)
#define jit_eqr_d(d, s1, s2)
#define jit_eqr_i(d, s1, s2)
#define jit_extr_i_d(rd, rs)
#define jit_floorr_d_i(rd, rs)
#define jit_gei_i(d, rs, is)
#define jit_gei_ui(d, s1, s2)
#define jit_ger_d(d, s1, s2)
#define jit_ger_i(d, s1, s2)
#define jit_ger_ui(d, s1, s2)
#define jit_gti_i(d, rs, is)
#define jit_gti_ui(d, s1, s2)
#define jit_gtr_d(d, s1, s2)
#define jit_gtr_i(d, s1, s2)
#define jit_gtr_ui(d, s1, s2)
#define jit_hmuli_i(d, rs, is)
#define jit_hmuli_ui(d, rs, is)
#define jit_hmulr_i(d, s1, s2)
#define jit_hmulr_ui(d, s1, s2)
#define jit_jmpi(label)
#define jit_jmpr(reg)
#define jit_ldxi_f(rd, rs, is)
#define jit_ldxr_f(rd, s1, s2)
#define jit_ldxi_c(d, rs, is)
#define jit_ldxi_d(rd, rs, is)
#define jit_ldxi_i(d, rs, is)
#define jit_ldxi_s(d, rs, is)
#define jit_ldxi_uc(d, rs, is)
#define jit_ldxi_us(d, rs, is)
#define jit_ldxr_c(d, s1, s2)
#define jit_ldxr_d(rd, s1, s2)
#define jit_ldxr_i(d, s1, s2)
#define jit_ldxr_s(d, s1, s2)
#define jit_ldxr_uc(d, s1, s2)
#define jit_ldxr_us(d, s1, s2)
#define jit_lei_i(d, rs, is)
#define jit_lei_ui(d, s1, s2)
#define jit_ler_d(d, s1, s2)
#define jit_ler_i(d, s1, s2)
#define jit_ler_ui(d, s1, s2)
#define jit_lshi_i(d, rs, is)
#define jit_lshr_i(d, r1, r2)
#define jit_ltgtr_d(d, s1, s2)
#define jit_lti_i(d, rs, is)
#define jit_lti_ui(d, s1, s2)
```

```
#define jit_ltr_d(d, s1, s2)
#define jit_ltr_i(d, s1, s2)
#define jit_ltr_ui(d, s1, s2)
#define jit_modi_i(d, rs, is)
#define jit_modi_ui(d, rs, is)
#define jit_modr_i(d, s1, s2)
#define jit_modr_ui(d, s1, s2)
#define jit_movi_d(rd,immd)
#define jit_movi_f(rd,immf)
#define jit_movi_i(d, is)
#define jit_movi_p(d, is)
#define jit_movr_d(rd,rs)
#define jit_movr_i(d, rs)
#define jit_muli_i(d, rs, is)
#define jit_muli_ui(d, rs, is)
#define jit_mulr_d(rd,s1,s2)
#define jit_mulr_i(d, s1, s2)
#define jit_mulr_ui(d, s1, s2)
#define jit_negr_d(rd,rs)
#define jit_nei_i(d, rs, is)
#define jit_ner_d(d, s1, s2)
#define jit_ner_i(d, s1, s2)
#define jit_nop()
#define jit_ordr_d(d, s1, s2)
#define jit_ori_i(d, rs, is)
#define jit_orr_i(d, s1, s2)
#define jit_patch_at(jump_pc, value)
#define jit_patch_movi(jump_pc, value)
#define jit_pop_i(rs)
#define jit_prepare_d(numargs)
#define jit_prepare_f(numargs)
#define jit_prepare_i(numargs)
#define jit_push_i(rs)
#define jit_pusharg_i(rs)
#define jit_ret()
#define jit_retval_i(rd)
#define jit_roundr_d_i(rd, rs)
#define jit_rshi_i(d, rs, is)
#define jit_rshi_ui(d, rs, is)
#define jit_rshr_i(d, r1, r2)
#define jit_rshr_ui(d, r1, r2)
#define jit_sqrt_d(rd,rs)
#define jit_stxi_c(rd, id, rs)
#define jit_stxi_d(id, rd, rs)
#define jit_stxi_f(id, rd, rs)
#define jit_stxi_i(rd, id, rs)
#define jit_stxi_s(rd, id, rs)
```

```
#define jit_stxr_c(d1, d2, rs)
#define jit_stxr_d(d1, d2, rs)
#define jit_stxr_f(d1, d2, rs)
#define jit_stxr_i(d1, d2, rs)
#define jit_stxr_s(d1, d2, rs)
#define jit_subr_d(rd,s1,s2)
#define jit_subr_i(d, s1, s2)
#define jit_subxi_i(d, rs, is)
#define jit_subxr_i(d, s1, s2)
#define jit_truncr_d_i(rd, rs)
#define jit_uneqr_d(d, s1, s2)
#define jit_unger_d(d, s1, s2)
#define jit_ungtr_d(d, s1, s2)
#define jit_unler_d(d, s1, s2)
#define jit_unltr_d(d, s1, s2)
#define jit_unordr_d(d, s1, s2)
#define jit_xori_i(d, rs, is)
#define jit_xorr_i(d, s1, s2)
```

**Non mandatory—there should be no need to define them:**

```
#define jit_extr_c_ui(d, rs)
#define jit_extr_s_ui(d, rs)
#define jit_extr_c_ul(d, rs)
#define jit_extr_s_ul(d, rs)
#define jit_extr_i_ul(d, rs)
#define jit_negr_i(d, rs)
#define jit_negr_l(d, rs)
```

**Non mandatory—whether to define them depends on the ABI:**

```
#define jit_prolog(n)
#define jit_finish(sub)
#define jit_finishr(reg)
#define jit_leaf(n)
#define jit_getarg_c(reg, ofs)
#define jit_getarg_i(reg, ofs)
#define jit_getarg_l(reg, ofs)
#define jit_getarg_p(reg, ofs)
#define jit_getarg_s(reg, ofs)
#define jit_getarg_uc(reg, ofs)
#define jit_getarg_ui(reg, ofs)
#define jit_getarg_ul(reg, ofs)
#define jit_getarg_us(reg, ofs)
#define jit_getarg_f(reg, ofs)
#define jit_getarg_d(reg, ofs)
```

**Non mandatory—define them if instructions that do this exist:**

```
#define jit_extr_c_i(d, rs)
#define jit_extr_s_i(d, rs)
#define jit_extr_c_l(d, rs)
```

```
#define jit_extr_s_l(d, rs)
#define jit_extr_i_l(d, rs)
#define jit_rsbi_i(d, rs, is)
#define jit_rsbi_l(d, rs, is)
```

**Non mandatory if condition code are always set by add/sub, needed on other systems:**

```
#define jit_addci_i(d, rs, is)
#define jit_addci_l(d, rs, is)
#define jit_subci_i(d, rs, is)
#define jit_subci_l(d, rs, is)
```

**Mandatory on little endian systems—don't define them on other systems:**

```
#define jit_ntoh_ui(d, rs)
#define jit_ntoh_us(d, rs)
```

**Mandatory if JIT_RZERO not defined—don't define them if it is defined:**

```
#define jit_ldi_c(d, is)
#define jit_ldi_i(d, is)
#define jit_ldi_s(d, is)
#define jit_ldr_c(d, rs)
#define jit_ldr_i(d, rs)
#define jit_ldr_s(d, rs)
#define jit_ldi_uc(d, is)
#define jit_ldi_ui(d, is)
#define jit_ldi_ul(d, is)
#define jit_ldi_us(d, is)
#define jit_ldr_uc(d, rs)
#define jit_ldr_ui(d, rs)
#define jit_ldr_ul(d, rs)
#define jit_ldr_us(d, rs)
#define jit_sti_c(id, rs)
#define jit_sti_i(id, rs)
#define jit_sti_s(id, rs)
#define jit_str_c(rd, rs)
#define jit_str_i(rd, rs)
#define jit_str_s(rd, rs)
#define jit_ldi_f(rd, is)
#define jit_sti_f(id, rs)
#define jit_ldi_d(rd, is)
#define jit_sti_d(id, rs)
#define jit_ldr_f(rd, rs)
#define jit_str_f(rd, rs)
#define jit_ldr_d(rd, rs)
#define jit_str_d(rd, rs)
```

**Synonyms—don't define them:**

```
#define jit_addi_p(d, rs, is)
#define jit_addi_ui(d, rs, is)
#define jit_addi_ul(d, rs, is)
```

```
#define jit_addr_p(d, s1, s2)
#define jit_addr_ui(d, s1, s2)
#define jit_addr_ul(d, s1, s2)
#define jit_andi_ui(d, rs, is)
#define jit_andi_ul(d, rs, is)
#define jit_andr_ui(d, s1, s2)
#define jit_andr_ul(d, s1, s2)
#define jit_beqi_p(label, rs, is)
#define jit_beqi_ui(label, rs, is)
#define jit_beqi_ul(label, rs, is)
#define jit_beqr_p(label, s1, s2)
#define jit_beqr_ui(label, s1, s2)
#define jit_beqr_ul(label, s1, s2)
#define jit_bmci_ui(label, rs, is)
#define jit_bmci_ul(label, rs, is)
#define jit_bmcr_ui(label, s1, s2)
#define jit_bmcr_ul(label, s1, s2)
#define jit_bmsi_ui(label, rs, is)
#define jit_bmsi_ul(label, rs, is)
#define jit_bmsr_ui(label, s1, s2)
#define jit_bmsr_ul(label, s1, s2)
#define jit_bgei_p(label, rs, is)
#define jit_bger_p(label, s1, s2)
#define jit_bgti_p(label, rs, is)
#define jit_bgtr_p(label, s1, s2)
#define jit_blei_p(label, rs, is)
#define jit_bler_p(label, s1, s2)
#define jit_blti_p(label, rs, is)
#define jit_bltr_p(label, s1, s2)
#define jit_bnei_p(label, rs, is)
#define jit_bnei_ui(label, rs, is)
#define jit_bnei_ul(label, rs, is)
#define jit_bner_p(label, s1, s2)
#define jit_bner_ui(label, s1, s2)
#define jit_bner_ul(label, s1, s2)
#define jit_eqi_p(d, rs, is)
#define jit_eqi_ui(d, rs, is)
#define jit_eqi_ul(d, rs, is)
#define jit_eqr_p(d, s1, s2)
#define jit_eqr_ui(d, s1, s2)
#define jit_eqr_ul(d, s1, s2)
#define jit_extr_c_s(d, rs)
#define jit_extr_c_us(d, rs)
#define jit_extr_uc_s(d, rs)
#define jit_extr_uc_us(d, rs)
#define jit_extr_uc_i(d, rs)
#define jit_extr_uc_ui(d, rs)
```

```
#define jit_extr_us_i(d, rs)
#define jit_extr_us_ui(d, rs)
#define jit_extr_uc_l(d, rs)
#define jit_extr_uc_ul(d, rs)
#define jit_extr_us_l(d, rs)
#define jit_extr_us_ul(d, rs)
#define jit_extr_ui_l(d, rs)
#define jit_extr_ui_ul(d, rs)
#define jit_gei_p(d, rs, is)
#define jit_ger_p(d, s1, s2)
#define jit_gti_p(d, rs, is)
#define jit_gtr_p(d, s1, s2)
#define jit_ldr_p(d, rs)
#define jit_ldi_p(d, is)
#define jit_ldxi_p(d, rs, is)
#define jit_ldxr_p(d, s1, s2)
#define jit_lei_p(d, rs, is)
#define jit_ler_p(d, s1, s2)
#define jit_lshi_ui(d, rs, is)
#define jit_lshi_ul(d, rs, is)
#define jit_lshr_ui(d, s1, s2)
#define jit_lshr_ul(d, s1, s2)
#define jit_lti_p(d, rs, is)
#define jit_ltr_p(d, s1, s2)
#define jit_movi_p(d, is)
#define jit_movi_ui(d, rs)
#define jit_movi_ul(d, rs)
#define jit_movr_p(d, rs)
#define jit_movr_ui(d, rs)
#define jit_movr_ul(d, rs)
#define jit_nei_p(d, rs, is)
#define jit_nei_ui(d, rs, is)
#define jit_nei_ul(d, rs, is)
#define jit_ner_p(d, s1, s2)
#define jit_ner_ui(d, s1, s2)
#define jit_ner_ul(d, s1, s2)
#define jit_hton_ui(d, rs)
#define jit_hton_us(d, rs)
#define jit_ori_ui(d, rs, is)
#define jit_ori_ul(d, rs, is)
#define jit_orr_ui(d, s1, s2)
#define jit_orr_ul(d, s1, s2)
#define jit_pop_ui(rs)
#define jit_pop_ul(rs)
#define jit_push_ui(rs)
#define jit_push_ul(rs)
#define jit_pusharg_c(rs)
```

```
#define jit_pusharg_p(rs)
#define jit_pusharg_s(rs)
#define jit_pusharg_uc(rs)
#define jit_pusharg_ui(rs)
#define jit_pusharg_ul(rs)
#define jit_pusharg_us(rs)
#define jit_retval_c(rd)
#define jit_retval_p(rd)
#define jit_retval_s(rd)
#define jit_retval_uc(rd)
#define jit_retval_ui(rd)
#define jit_retval_ul(rd)
#define jit_retval_us(rd)
#define jit_rsbi_p(d, rs, is)
#define jit_rsbi_ui(d, rs, is)
#define jit_rsbi_ul(d, rs, is)
#define jit_rsbr_p(d, rs, is)
#define jit_rsbr_ui(d, s1, s2)
#define jit_rsbr_ul(d, s1, s2)
#define jit_sti_p(d, is)
#define jit_sti_uc(d, is)
#define jit_sti_ui(d, is)
#define jit_sti_ul(d, is)
#define jit_sti_us(d, is)
#define jit_str_p(d, rs)
#define jit_str_uc(d, rs)
#define jit_str_ui(d, rs)
#define jit_str_ul(d, rs)
#define jit_str_us(d, rs)
#define jit_stxi_p(d, rs, is)
#define jit_stxi_uc(d, rs, is)
#define jit_stxi_ui(d, rs, is)
#define jit_stxi_ul(d, rs, is)
#define jit_stxi_us(d, rs, is)
#define jit_stxr_p(d, s1, s2)
#define jit_stxr_uc(d, s1, s2)
#define jit_stxr_ui(d, s1, s2)
#define jit_stxr_ul(d, s1, s2)
#define jit_stxr_us(d, s1, s2)
#define jit_subi_p(d, rs, is)
#define jit_subi_ui(d, rs, is)
#define jit_subi_ul(d, rs, is)
#define jit_subr_p(d, s1, s2)
#define jit_subr_ui(d, s1, s2)
#define jit_subr_ul(d, s1, s2)
#define jit_subxi_p(d, rs, is)
#define jit_subxi_ui(d, rs, is)
```

```
#define jit_subxi_ul(d, rs, is)
#define jit_subxr_p(d, s1, s2)
#define jit_subxr_ui(d, s1, s2)
#define jit_subxr_ul(d, s1, s2)
#define jit_xori_ui(d, rs, is)
#define jit_xori_ul(d, rs, is)
#define jit_xorr_ui(d, s1, s2)
#define jit_xorr_ul(d, s1, s2)
```

**Shortcuts—don't define them:**

```
#define JIT_R0
#define JIT_R1
#define JIT_R2
#define JIT_V0
#define JIT_V1
#define JIT_V2
#define JIT_FPR0
#define JIT_FPR1
#define JIT_FPR2
#define JIT_FPR3
#define JIT_FPR4
#define JIT_FPR5
#define jit_patch(jump_pc)
#define jit_notr_c(d, rs)
#define jit_notr_i(d, rs)
#define jit_notr_l(d, rs)
#define jit_notr_s(d, rs)
#define jit_notr_uc(d, rs)
#define jit_notr_ui(d, rs)
#define jit_notr_ul(d, rs)
#define jit_notr_us(d, rs)
#define jit_rsbr_d(d, s1, s2)
#define jit_rsbr_i(d, s1, s2)
#define jit_rsbr_l(d, s1, s2)
#define jit_subi_i(d, rs, is)
#define jit_subi_l(d, rs, is)
```

**Mandatory unless target arithmetic is always done in the same precision:**

```
#define jit_abs_f(rd,rs)
#define jit_addr_f(rd,s1,s2)
#define jit_beqr_f(label, s1, s2)
#define jit_bger_f(label, s1, s2)
#define jit_bgtr_f(label, s1, s2)
#define jit_bler_f(label, s1, s2)
#define jit_bltgtr_f(label, s1, s2)
#define jit_bltr_f(label, s1, s2)
#define jit_bner_f(label, s1, s2)
#define jit_bordr_f(label, s1, s2)
```

```
#define jit_buneqr_f(label, s1, s2)
#define jit_bunger_f(label, s1, s2)
#define jit_bungtr_f(label, s1, s2)
#define jit_bunler_f(label, s1, s2)
#define jit_bunltr_f(label, s1, s2)
#define jit_bunordr_f(label, s1, s2)
#define jit_ceilr_f_i(rd, rs)
#define jit_divr_f(rd,s1,s2)
#define jit_eqr_f(d, s1, s2)
#define jit_extr_d_f(rs, rd)
#define jit_extr_f_d(rs, rd)
#define jit_extr_i_f(rd, rs)
#define jit_floorr_f_i(rd, rs)
#define jit_ger_f(d, s1, s2)
#define jit_gtr_f(d, s1, s2)
#define jit_ler_f(d, s1, s2)
#define jit_ltgtr_f(d, s1, s2)
#define jit_ltr_f(d, s1, s2)
#define jit_movr_f(rd,rs)
#define jit_mulr_f(rd,s1,s2)
#define jit_negr_f(rd,rs)
#define jit_ner_f(d, s1, s2)
#define jit_ordr_f(d, s1, s2)
#define jit_roundr_f_i(rd, rs)
#define jit_rsbr_f(d, s1, s2)
#define jit_sqrt_f(rd,rs)
#define jit_subr_f(rd,s1,s2)
#define jit_truncr_f_i(rd, rs)
#define jit_uneqr_f(d, s1, s2)
#define jit_unger_f(d, s1, s2)
#define jit_ungtr_f(d, s1, s2)
#define jit_unler_f(d, s1, s2)
#define jit_unltr_f(d, s1, s2)
#define jit_unordr_f(d, s1, s2)
```

**Mandatory if sizeof(long) != sizeof(int)—don't define them on other systems:**

```
#define jit_addi_l(d, rs, is)
#define jit_addr_l(d, s1, s2)
#define jit_andi_l(d, rs, is)
#define jit_andr_l(d, s1, s2)
#define jit_beqi_l(label, rs, is)
#define jit_beqr_l(label, s1, s2)
#define jit_bgei_l(label, rs, is)
#define jit_bgei_ul(label, rs, is)
#define jit_bger_l(label, s1, s2)
#define jit_bger_ul(label, s1, s2)
#define jit_bgti_l(label, rs, is)
```

```
#define jit_bgti_ul(label, rs, is)
#define jit_bgtr_l(label, s1, s2)
#define jit_bgtr_ul(label, s1, s2)
#define jit_blei_l(label, rs, is)
#define jit_blei_ul(label, rs, is)
#define jit_bler_l(label, s1, s2)
#define jit_bler_ul(label, s1, s2)
#define jit_blti_l(label, rs, is)
#define jit_blti_ul(label, rs, is)
#define jit_bltr_l(label, s1, s2)
#define jit_bltr_ul(label, s1, s2)
#define jit_bosubi_l(label, rs, is)
#define jit_bosubi_ul(label, rs, is)
#define jit_bosubr_l(label, s1, s2)
#define jit_bosubr_ul(label, s1, s2)
#define jit_boaddi_l(label, rs, is)
#define jit_boaddi_ul(label, rs, is)
#define jit_boaddr_l(label, s1, s2)
#define jit_boaddr_ul(label, s1, s2)
#define jit_bmci_l(label, rs, is)
#define jit_bmcr_l(label, s1, s2)
#define jit_bmsi_l(label, rs, is)
#define jit_bmsr_l(label, s1, s2)
#define jit_bnei_l(label, rs, is)
#define jit_bner_l(label, s1, s2)
#define jit_divi_l(d, rs, is)
#define jit_divi_ul(d, rs, is)
#define jit_divr_l(d, s1, s2)
#define jit_divr_ul(d, s1, s2)
#define jit_eqi_l(d, rs, is)
#define jit_eqr_l(d, s1, s2)
#define jit_extr_c_l(d, rs)
#define jit_extr_c_ul(d, rs)
#define jit_extr_s_l(d, rs)
#define jit_extr_s_ul(d, rs)
#define jit_extr_i_l(d, rs)
#define jit_extr_i_ul(d, rs)
#define jit_gei_l(d, rs, is)
#define jit_gei_ul(d, rs, is)
#define jit_ger_l(d, s1, s2)
#define jit_ger_ul(d, s1, s2)
#define jit_gti_l(d, rs, is)
#define jit_gti_ul(d, rs, is)
#define jit_gtr_l(d, s1, s2)
#define jit_gtr_ul(d, s1, s2)
#define jit_hmuli_l(d, rs, is)
#define jit_hmuli_ul(d, rs, is)
```

```
#define jit_hmulr_l(d, s1, s2)
#define jit_hmulr_ul(d, s1, s2)
#define jit_ldi_l(d, is)
#define jit_ldi_ui(d, is)
#define jit_ldr_l(d, rs)
#define jit_ldr_ui(d, rs)
#define jit_ldxi_l(d, rs, is)
#define jit_ldxi_ui(d, rs, is)
#define jit_ldxi_ul(d, rs, is)
#define jit_ldxr_l(d, s1, s2)
#define jit_ldxr_ui(d, s1, s2)
#define jit_ldxr_ul(d, s1, s2)
#define jit_lei_l(d, rs, is)
#define jit_lei_ul(d, rs, is)
#define jit_ler_l(d, s1, s2)
#define jit_ler_ul(d, s1, s2)
#define jit_lshi_l(d, rs, is)
#define jit_lshr_l(d, s1, s2)
#define jit_lti_l(d, rs, is)
#define jit_lti_ul(d, rs, is)
#define jit_ltr_l(d, s1, s2)
#define jit_ltr_ul(d, s1, s2)
#define jit_modi_l(d, rs, is)
#define jit_modi_ul(d, rs, is)
#define jit_modr_l(d, s1, s2)
#define jit_modr_ul(d, s1, s2)
#define jit_movi_l(d, rs)
#define jit_movr_l(d, rs)
#define jit_muli_l(d, rs, is)
#define jit_muli_ul(d, rs, is)
#define jit_mulr_l(d, s1, s2)
#define jit_mulr_ul(d, s1, s2)
#define jit_nei_l(d, rs, is)
#define jit_ner_l(d, s1, s2)
#define jit_ori_l(d, rs, is)
#define jit_orr_l(d, s1, s2)
#define jit_pop_l(rs)
#define jit_push_l(rs)
#define jit_pusharg_l(rs)
#define jit_retval_l(rd)
#define jit_rshi_l(d, rs, is)
#define jit_rshi_ul(d, rs, is)
#define jit_rshr_l(d, s1, s2)
#define jit_rshr_ul(d, s1, s2)
#define jit_sti_l(d, is)
#define jit_str_l(d, rs)
#define jit_stxi_l(d, rs, is)
```

```
#define jit_stxr_l(d, s1, s2)
#define jit_subr_l(d, s1, s2)
#define jit_xori_l(d, rs, is)
#define jit_xorr_l(d, s1, s2)
```

## 3.5 More complex tasks in the platform-independent layer

There is actually a single function that you **must** define in the '`funcs-`*`suffix`*`.h`' file, that is, `jit_flush_code`.

As explained in Section 2.3 [Generating code at run-time], page 10, its purpose is to flush part of the processor's instruction cache (usually the part of memory that contains the generated code), avoiding the processor executing bogus data that it happens to find in the cache. The `jit_flush_code` function takes the first and the last address to flush.

On many processors (for example, the x86 and the all the processors in the 68k family up to the 68030), it is not even necessary to flush the cache. In this case, the contents of the file will simply be

```
#ifndef __lightning_funcs_h
#define __lightning_funcs_h

#define jit_flush_code(dest, end)

#endif /* __lightning_core_h */
```

On other processors, flushing the cache is necessary for proper behavior of the program; in this case, the file will contain a proper definition of the function. However, we must make yet another distinction.

On some processors, flushing the cache is obtained through a call to the operating system or to the C run-time library. In this case, the definition of `jit_flush_code` will be very simple: two examples are the Alpha and the 68040. For the Alpha the code will be:

```
#define jit_flush_code(dest, end) \
        __asm__ __volatile__("call_pal 0x86");
```

and, for the Motorola

```
#define jit_flush_code(start, end) \
        __clear_cache((start), (end))
```

As you can see, the Alpha does not even need to pass the start and end address to the function. It is good practice to protect usage of the GNU CC-specific `__asm__` directive by relying on the preprocessor. For example:

```
#if !defined(__GNUC__) && !defined(__GNUG__)
#error Go get GNU C, I do not know how to flush the cache
#error with this compiler.
#else
#define jit_flush_code(dest, end) \
        __asm__ __volatile__("call_pal 0x86");
#endif
```

GNU *lightning*'s configuration process tries to compile a dummy file that includes `lightning.h`, and gives a warning if there are problem with the compiler that is installed on the system.

In more complex cases, you'll need to write a full-fledged function. Don't forget to make it `static`, otherwise you'll have problems linking programs that include `lightning.h` multiple times. An example, taken from the '`funcs-ppc.h`' file, is:

```
#ifndef __lightning_funcs_h
#define __lightning_funcs_h

#if !defined(__GNUC__) && !defined(__GNUG__)
#error Go get GNU C, I do not know how to flush the cache
#error with this compiler.
#else
static void
jit_flush_code(start, end)
     void       *start;
     void       *end;
{
  register char *dest = start;

  for (; dest <= end; dest += SIZEOF_CHAR_P)
    __asm__ __volatile__
      ("dcbst 0,%0; sync; icbi 0,%0; isync"::"r"(dest));
}
#endif

#endif /* __lightning_funcs_h */
```

The '`funcs-`*`suffix`*`.h`' file is also the right place to put helper functions that do complex tasks for the '`core-`*`suffix`*`.h`' file. For example, the PowerPC assembler defines `jit_prolog` as a function and puts it in that file (for more information, see Section 3.4.5 [Implementing the ABI], page 34). Take special care when defining such a function, as explained in Section 2.4 [Reentrant usage of GNU *lightning*], page 21.

## 3.6 Implementing macros for floating point

# 4 The future of GNU *lightning*

Presented below is the set of tasks that I feel need to be performed to make GNU *lightning* a more fully functional, viable system. They are presented in no particular order. I would *very much* welcome any volunteers who would like to help with the implementation of one or more of these tasks. Please write to me, Paolo Bonzini, at bonzini@gnu.org if you are interested in adding your efforts to the GNU *lightning* project.

Tasks:

- The most important task to make GNU *lightning* more widely usable is to retarget it. Although currently supported architectures (x86, SPARC, PowerPC) are certainly some of the most widely used, GNU *lightning* could be ported to others—namely, the Alpha and MIPS architectures.
- Another interesting task is to allow the instruction stream to grow dynamically. This is a problem because not all architectures allow to write position independent code.[1]
- Optimize leaf procedures on the SPARC. This involves using the output registers (`%oX`) instead of the local registers (`%lX`) when writing leaf procedures; the problem is, leaf procedures also receive parameters in the output registers, so they would be overwritten by write accesses to general-purpose registers.

[1] The x86's absolute jumps, for example, are actually slow indirect jumps, and need a register.

# 5 Acknowledgements

As far as I know, the first general-purpose portable dynamic code generator is DCG, by Dawson R. Engler and T. A. Proebsting. Further work by Dawson R. Engler resulted in the VCODE system; unlike DCG, VCODE used no intermediate representation and directly inspired GNU *lightning*.

Thanks go to Ian Piumarta, who kindly accepted to release his own program CCG under the GNU General Public License, thereby allowing GNU *lightning* to use the run-time assemblers he had wrote for CCG. CCG provides a way of dynamically assemble programs written in the underlying architecture's assembly language. So it is not portable, yet very interesting.

I also thank Steve Byrne for writing GNU Smalltalk, since GNU *lightning* was first developed as a tool to be used in GNU Smalltalk's dynamic translator from bytecodes to native code.

# Table of Contents