1 Introduction to GNU *lightning*

This document describes installing and using the GNU *lightning* library for dynamic code generation.

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.

*GNU lightning* provides a portable, fast and easily retargetable dynamic code generation system.

To be portable, *GNU lightning* abstracts over current architectures’ quirks and unorthogonality. 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.
2 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 the `--enable-disassembler` option, that enables linking to GNU binutils and optionally print human readable disassembly of the jit code. This option can be disabled by the `--disable-disassembler` option.

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*, run `make` as usual.

GNU *lightning* has an extensive set of tests to validate it is working correctly in the build host. To test it run:

```
make check
```

The next important step is:

```
make install
```

This ends the process of installing GNU *lightning*. 
3 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 synthesized if absent. Each instruction is composed of:

- an operation, like `sub` or `mul`
- most times, a register/immediate flag (`r` or `i`)
- an unsigned modifier (`u`), a type identifier or two, when applicable.

Examples of legal mnemonics are `addr` (integer add, with three register operands) and `muli` (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.

Most *GNU lightning* integer operations are signed wordsize operations, with the exception of operations that convert types, or load or store values to/from memory. When applicable, the types and C types are as follow:

- `_c` signed char
- `_uc` unsigned char
- `_s` short
- `_us` unsigned short
- `_i` int
- `_ui` unsigned int
- `_l` long
- `_f` float
- `_d` double

Most integer operations do not need a type modifier, and when loading or storing values to memory there is an alias to the proper operation using wordsize operands, that is, if omitted, the type is int on 32-bit architectures and long on 64-bit architectures. Note that *lightning* also expects `sizeof(void*)` to match the wordsize.

When an unsigned operation result differs from the equivalent signed operation, there is a the `_u` modifier.

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

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

There are at least six floating-point registers, named `F0` to `F5`. These are usually caller-save and are separate from the integer registers on the supported architectures; on Intel architectures, in 32 bit mode if SSE2 is not available or use of X87 is forced, the register stack is mapped to a flat register file. As for the integer registers, the macro `JIT_F_NUM` yields the number of floating-point registers.
The complete instruction set follows; as you can see, most non-memory operations only take integers (either signed or unsigned) 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. `addx` operations must directly follow `addc`, and `subx` must follow `subc`; otherwise, results are undefined. Most, if not all, architectures do not support float or double immediate operands; lightning emulates those operations by moving the immediate to a temporary register and emitting the call with only register operands.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>_f _d 01 = 02 + 03</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>_f _d 01 = 02 + 03</td>
<td></td>
</tr>
<tr>
<td>addxx</td>
<td>01 = 02 + (03 + carry)</td>
<td></td>
</tr>
<tr>
<td>addxi</td>
<td>01 = 02 + (03 + carry)</td>
<td></td>
</tr>
<tr>
<td>addcr</td>
<td>01 = 02 + 03, set carry</td>
<td></td>
</tr>
<tr>
<td>addci</td>
<td>01 = 02 + 03, set carry</td>
<td></td>
</tr>
<tr>
<td>subr</td>
<td>_f _d 01 = 02 - 03</td>
<td></td>
</tr>
<tr>
<td>subi</td>
<td>_f _d 01 = 02 - 03</td>
<td></td>
</tr>
<tr>
<td>subxr</td>
<td>01 = 02 - (03 + carry)</td>
<td></td>
</tr>
<tr>
<td>subxi</td>
<td>01 = 02 - (03 + carry)</td>
<td></td>
</tr>
<tr>
<td>subcr</td>
<td>01 = 02 - 03, set carry</td>
<td></td>
</tr>
<tr>
<td>subci</td>
<td>01 = 02 - 03, set carry</td>
<td></td>
</tr>
<tr>
<td>rsbr</td>
<td>_f _d 01 = 03 - 01</td>
<td></td>
</tr>
<tr>
<td>rsbi</td>
<td>_f _d 01 = 03 - 01</td>
<td></td>
</tr>
<tr>
<td>mulr</td>
<td>_f _d 01 = 02 * 03</td>
<td></td>
</tr>
<tr>
<td>muli</td>
<td>_f _d 01 = 02 * 03</td>
<td></td>
</tr>
<tr>
<td>divr</td>
<td>_u _f _d 01 = 02 / 03</td>
<td></td>
</tr>
<tr>
<td>divi</td>
<td>_u _f _d 01 = 02 / 03</td>
<td></td>
</tr>
<tr>
<td>remr</td>
<td>_u</td>
<td>01 = 02 % 03</td>
</tr>
<tr>
<td>remi</td>
<td>_u</td>
<td>01 = 02 % 03</td>
</tr>
<tr>
<td>andr</td>
<td>_u</td>
<td>01 = 02 &amp; 03</td>
</tr>
<tr>
<td>andi</td>
<td>_u</td>
<td>01 = 02 &amp; 03</td>
</tr>
<tr>
<td>orr</td>
<td>_u</td>
<td>01 = 02</td>
</tr>
<tr>
<td>ori</td>
<td>_u</td>
<td>01 = 02</td>
</tr>
<tr>
<td>xorrr</td>
<td>_u</td>
<td>01 = 02 ^ 03</td>
</tr>
<tr>
<td>xori</td>
<td>_u</td>
<td>01 = 02 ^ 03</td>
</tr>
<tr>
<td>lshr</td>
<td>_u</td>
<td>01 = 02 &lt;&lt; 03</td>
</tr>
<tr>
<td>lshi</td>
<td>_u</td>
<td>01 = 02 &lt;&lt; 03</td>
</tr>
<tr>
<td>rshr</td>
<td>_u</td>
<td>01 = 02 &gt;&gt; 03</td>
</tr>
<tr>
<td>rshi</td>
<td>_u</td>
<td>01 = 02 &gt;&gt; 03</td>
</tr>
</tbody>
</table>

1. The sign bit is propagated unless using the _u modifier.
2. The sign bit is propagated unless using the _u modifier.
Four operand binary ALU operations

These accept two result registers, and two operands; the last one can be an immediate. The first two arguments cannot be the same register.

\texttt{qmul} stores the low word of the result in \texttt{O1} and the high word in \texttt{O2}. For unsigned multiplication, \texttt{O2} zero means there was no overflow. For signed multiplication, no overflow check is based on sign, and can be detected if \texttt{O2} is zero or minus one.

\texttt{qdiv} stores the quotient in \texttt{O1} and the remainder in \texttt{O2}. It can be used as quick way to check if a division is exact, in which case the remainder is zero.

\begin{verbatim}
qmulr  _u  01 02 = 03 * 04
qmuli  _u  01 02 = 03 * 04
qdivr  _u  01 02 = 03 / 04
qdivi  _u  01 02 = 03 / 04
\end{verbatim}

Unary ALU operations

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

\begin{verbatim}
negr  _f _d  01 = -02
comr  01 = ~02
\end{verbatim}

These unary ALU operations are only defined for float operands.

\begin{verbatim}
abser  _f _d  01 = fabs(02)
sqrtr  01 = sqrt(02)
\end{verbatim}

Besides requiring the \texttt{r} modifier, there are no unary operations with an immediate operand.

Compare instructions

These accept three operands; again, the last can be an immediate. The last two operands are compared, and the first operand, that must be an integer register, 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.

\begin{verbatim}
ltr  _u _f _d  01 = (02 < 03)
lti  _u _f _d  01 = (02 < 03)
ler  _u _f _d  01 = (02 <= 03)
lei  _u _f _d  01 = (02 <= 03)
gtr  _u _f _d  01 = (02 > 03)
gti  _u _f _d  01 = (02 > 03)
gei  _u _f _d  01 = (02 >= 03)
eqr  _f _d  01 = (02 == 03)
eqi  _f _d  01 = (02 == 03)
ner  _f _d  01 = (02 != 03)
nei  _f _d  01 = (02 != 03)
unltr  _f _d  01 = !(02 > 03)
unler  _f _d  01 = !(02 > 03)
unltr  _f _d  01 = !(02 <= 03)
unger  _f _d  01 = !(02 < 03)
\end{verbatim}
uneqr  _f _d  01 = !(O2 < 03) && !(O2 > 03)
ltgtr  _f _d  01 = !(O2 >= 03) || !(O2 <= 03)
ordr  _f _d  01 = (O2 == O2) && (O3 == O3)
unordr _f _d  01 = (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 used to truncate a wordsize operand to a smaller integer data type or to convert float data types. You can also use extr to convert an integer to a floating point value: the usual options are extr_f and extr_d.

movr _f _d  01 = O2
movi _f _d  01 = O2
extr _c _uc _s _us _i _ui _f _d  01 = O2
truncr _f _d  01 = trunc(02)

In 64-bit architectures it may be required to use truncr_f_i, truncr_f_l, truncr_d_i and truncr_d_l to match the equivalent C code. Only the _i modifier is available in 32-bit architectures.

truncr_f_i = <int> 01 = <float> O2
truncr_f_l = <long>01 = <float> O2
truncr_d_i = <int> 01 = <double>02
truncr_d_l = <long>01 = <double>02

The float conversion operations are destination first, source second, but the order of the types is reversed. This happens for historical reasons.
extr_f_d = <double>01 = <float> O2
extr_d_f = <float> 01 = <double>02

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. The _ul variant is only available in 64-bit architectures.

htonr _us _ui _ul Host-to-network (big endian) order
ntohr _us _ui _ul 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. The _ui and _l types are only available in 64-bit architectures. For convenience, there is a version without a type modifier for integer or pointer operands that uses the appropriate wordsize call.

ldr _c _uc _s _us _i _ui _l _f _d  01 = *(02)
lxi _c _uc _s _us _i _ui _l _f _d  01 = *(02+03)
ldxi  _c  _uc  _s  _us  _i  _ui  _l  _f  _d  01 = *(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  _f  _d  *01 = 02
sti  _c  _uc  _s  _us  _i  _ui  _l  _f  _d  *01 = 02
stxr _c  _uc  _s  _us  _i  _ui  _l  _f  _d  *(O1+O2) = 03
stxi _c  _uc  _s  _us  _i  _ui  _l  _f  _d  *(O1+O2) = 03

As for the load operations, the _ui and _l types are only available in 64-bit architectures, and for convenience, there is a version without a type modifier for integer or pointer operands that uses the appropriate wordsize call.

Argument management

These are:

prepare  (not specified)
va_start  (not specified)
pushargr  _f  _d
pushargi  _f  _d
va_push  (not specified)
arg  _f  _d
getarg  _c  _uc  _s  _us  _i  _ui  _l  _f  _d
va_arg  _d
putargr  _f  _d
putargi  _f  _d
ret  (not specified)
retr  _f  _d
reti  _f  _d
va_end  (not specified)
retval  _c  _uc  _s  _us  _i  _ui  _l  _f  _d
epilog  (not specified)

As with other operations that use a type modifier, the _ui and _l types are only available in 64-bit architectures, but there are operations without a type modifier that alias to the appropriate integer operation with wordsize operands.

prepare, pusharg, and retval are used by the caller, while arg, getarg and ret are used by the callee. A code snippet that wants to call another procedure and has to pass arguments must, in order: use the prepare instruction and use the pushargr or pushargi to push the arguments in left to right order; and use finish or call (explained below) to perform the actual call.

va_start returns a C compatible va_list. To fetch arguments, use va_arg for integers and va_arg_d for doubles. va_push is required when passing a va_list to another function, because not all architectures expect it as a single pointer. Known case is DEC Alpha, that requires it as a structure passed by value.

arg, getarg and putarg 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` or `putarg`. You should call `arg` as soon as possible, before any function call or, more easily, right after the `prolog` instructions (which is 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 Chapter 4 [Generating code at run-time], page 15.

`putarg` is a mix of `getarg` and `pusharg` in that it accepts as first argument a register or immediate, and as second argument a value returned by `arg`. It allows changing, or restoring an argument to the current function, and is a construct required to implement tail call optimization. Note that arguments in registers are very cheap, but will be overwritten at any moment, including on some operations, for example division, that on several ports is implemented as a function call.

Finally, the `retval` instruction fetches the return value of a called function in a register. The `retval` instruction takes a register argument and copies the return value of the previously called function in that register. A function with a return value should use `retr` or `reti` to put the return value in the return register before returning. See Section 4.4 [Fibonacci], page 22, for an example.

`epilog` is an optional call, that marks the end of a function body. It is automatically generated by GNU `lightning` if starting a new function (what should be done after a `ret` call) or finishing generating jit. It is very important to note that the fact that `epilog` being optional may cause a common mistake. Consider this:

```assembly
fun1:
    prolog
    ...
    ret
fun2:
    prolog
```

Because `epilog` is added when finding a new `prolog`, this will cause the `fun2` label to actually be before the return from `fun1`. Because GNU `lightning` will actually understand it as:

```assembly
fun1:
    prolog
    ...
    ret
fun2:
    epilog
    prolog
```

---

3 "Return a value" means that GNU `lightning` code that compile these instructions return a value when expanded.
You should observe a few rules when using these macros. First of all, if calling a varargs function, you should use the `ellipsis` call to mark the position of the ellipsis in the C prototype.

You should not nest calls to `prepare` inside a `prepare/finish` block. Doing this will result in undefined behavior. Note that for functions with zero arguments you can use just `call`.

**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 4.4 [Fibonacci numbers], page 22. They accept two operands to be compared; of these, the last can be either a register or an immediate. They are:

- `bltr _u _f _d` if `(O2 < 03)` goto `O1`
- `blti _u _f _d` if `(O2 < 03)` goto `O1`
- `bler _u _f _d` if `(O2 <= 03)` goto `O1`
- `blei _u _f _d` if `(O2 <= 03)` goto `O1`
- `bgtr _u _f _d` if `(O2 > 03)` goto `O1`
- `bgti _u _f _d` if `(O2 > 03)` goto `O1`
- `bger _u _f _d` if `(O2 >= 03)` goto `O1`
- `bgei _u _f _d` if `(O2 >= 03)` goto `O1`
- `beqr _f _d` if `(O2 == 03)` goto `O1`
- `beqi _f _d` if `(O2 == 03)` goto `O1`
- `bner _f _d` if `(O2 != 03)` goto `O1`
- `bnei _f _d` if `(O2 != 03)` goto `O1`
- `bunltr _f _d` if `!(O2 >= 03)` goto `O1`
- `bunler _f _d` if `!(O2 > 03)` goto `O1`
- `bungtr _f _d` if `!(O2 <= 03)` goto `O1`
- `bunger _f _d` if `!(O2 < 03)` goto `O1`
- `buneqr _f _d` if `!(O2 < 03) && !(O2 > 03)` goto `O1`
- `bltgtr _f _d` if `!(O2 >= 03) || !(O2 <= 03)` goto `O1`
- `bordr _f _d` if `(O2 == O2) && (O3 == O3)` goto `O1`
- `bunordr _f _d` if `!(O2 != O2) || (O3 != O3)` goto `O1`
- `bmsr` if `O2 & O3` goto `O1`
- `bmsi` if `O2 & O3` goto `O1`
- `bmcr` if `!(O2 & O3)` goto `O1`
- `bmci` if `!(O2 & O3)` goto `O1`
- `boaddr _u` `O2 += 03`, goto `O1` if overflow
- `boaddi _u` `O2 += 03`, goto `O1` if overflow
- `bxaddr _u` `O2 += 03`, goto `O1` if no overflow
- `bxaddi _u` `O2 += 03`, goto `O1` if no overflow
- `bosubr _u` `O2 -= 03`, goto `O1` if overflow
- `bosubi _u` `O2 -= 03`, goto `O1` if overflow
- `bxsubr _u` `O2 -= 03`, goto `O1` if no overflow

---

4 These mnemonics mean, respectively, `branch if mask set` and `branch if mask cleared.`
Jump and return operations

These accept one argument except ret and jmp which have none; the difference between finishi 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callr (not specified)</td>
<td>function call to register O1</td>
</tr>
<tr>
<td>calli (not specified)</td>
<td>function call to immediate O1</td>
</tr>
<tr>
<td>finishr (not specified)</td>
<td>function call to register O1</td>
</tr>
<tr>
<td>finishi (not specified)</td>
<td>function call to immediate O1</td>
</tr>
<tr>
<td>jmp (not specified)</td>
<td>unconditional jump</td>
</tr>
<tr>
<td>retr _c _uc _s _us _i _ui _l _f _d</td>
<td>return from subroutine</td>
</tr>
<tr>
<td>reti _c _uc _s _us _i _ui _l _f _d</td>
<td>return from subroutine</td>
</tr>
<tr>
<td>retval _c _uc _s _us _i _ui _l _f _d</td>
<td>move return value to register</td>
</tr>
</tbody>
</table>

Like branch instruction, jmp also returns a value which is to be used to compile forward branches. See Section 4.4 [Fibonacci numbers], page 22.

Labels

There are 3 GNU lightning instructions to create labels:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label (not specified)</td>
<td>simple label</td>
</tr>
<tr>
<td>forward (not specified)</td>
<td>forward label</td>
</tr>
<tr>
<td>indirect (not specified)</td>
<td>special simple label</td>
</tr>
</tbody>
</table>

label is normally used as patch_at argument for backward jumps.

```c
jit_node_t *jump, *label;
label = jit_label();
...
jump = jit_bebr(JIT_R0, JIT_R1);
jit_patch_at(jump, label);
```

forward is used to patch code generation before the actual position of the label is known.

```c
jit_node_t *jump, *label;
label = jit_forward();
jump = jit_bebr(JIT_R0, JIT_R1);
jit_patch_at(jump, label);
...
jit_link(label);
```

indirect is useful when creating jump tables, and tells GNU lightning to not optimize out a label that is not the target of any jump, because an indirect jump may land where it is defined.

```c
jit_node_t *jump, *label;
...
```
jmpr(JIT_R0); /* may jump to label */
...
label = jit_indirect();

indirect is a special case of note and name because it is a valid argument to address.

Note that the usual idiom to write the previous example is

\[
\begin{align*}
\text{jit_node_t}\ *\ addr, \ *\ jump; \\
\text{addr} & = \text{jit_movi}(\text{JIT}_R0, 0); \quad \text{/* immediate is ignored */} \\
\text{...} \\
\text{jmpr(}\text{JIT}_R0); \\
\text{...} \\
\text{jit_patch(addr);} \quad \text{/* implicit label added */}
\end{align*}
\]

that automatically binds the implicit label added by patch with the movi, but on some special conditions it is required to create an "unbound" label.

**Function prolog**

These macros are used to set up a function prolog. The allocai call accepts a single integer argument and returns an offset value for stack storage access. The allocar accepts two registers arguments, the first is set to the offset for stack access, and the second is the size in bytes argument.

\[
\begin{align*}
\text{prolog} & \quad (\text{not specified}) \qquad \text{function prolog} \\
\text{allocai} & \quad (\text{not specified}) \quad \text{reserve space on the stack} \\
\text{allocar} & \quad (\text{not specified}) \quad \text{allocate space on the stack}
\end{align*}
\]

allocai receives the number of bytes to allocate and returns the offset from the frame pointer register FP to the base of the area.

allocar receives two register arguments. The first is where to store the offset from the frame pointer register FP to the base of the area. The second argument is the size in bytes. Note that allocar is dynamic allocation, and special attention should be taken when using it. If called in a loop, every iteration will allocate stack space. Stack space is aligned from 8 to 64 bytes depending on backend requirements, even if allocating only one byte. It is advisable to not use it with frame and tramp; it should work with frame with special care to call only once, but is not supported if used in tramp, even if called only once.

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 Chapter 4 [Generating code at run-time], page 15.

```
in = arg
getarg R0, in  ! We have an integer argument
addi R0, R0, 1  ! Add 1
retr R0  ! 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:
```
Trampolines, continuations and tail call optimization

Frequently it is required to generate jit code that must jump to code generated later, possibly from another `jit_context_t`. These require compatible stack frames.

**GNU lightning** provides two primitives from where trampolines, continuations and tail call optimization can be implemented.

- `frame` (not specified) — create stack frame
- `tramp` (not specified) — assume stack frame

`frame` receives an integer argument\(^5\) that defines the size in bytes for the stack frame of the current, C callable, jit function. To calculate this value, a good formula is maximum number of arguments to any called native function times eight\(^6\), plus the sum of the arguments to any call to `jit_allocai`. GNU lightning automatically adjusts this value for any backend specific stack memory it may need, or any alignment constraint.

`frame` also instructs GNU lightning to save all callee save registers in the prolog and reload in the epilog.

```plaintext
main:
    ! jit entry point
    prolog
    ! function prolog
    frame 256
    ! save all callee save registers and
    ! reserve at least 256 bytes in stack

main_loop:
    ...
    jmpi handler
    ! jumps to external code
    ...
    ret
    ! return to the caller
```

`tramp` differs from `frame` only that a prolog and epilog will not be generated. Note that `prolog` must still be used. The code under `tramp` must be ready to be entered with a jump at the prolog position, and instead of a return, it must end with a non conditional jump. `tramp` exists solely for the fact that it allows optimizing out prolog and epilog code that would never be executed.

```plaintext
handler:
    ! handler entry point
    prolog  ! function prolog
```

---

\(^5\) It is not automatically computed because it does not know about the requirement of later generated code.

\(^6\) Times eight so that it works for double arguments. And would not need conditionals for ports that pass arguments in the stack.
tramp 256
! assumes all callee save registers
! are saved and there is at least
! 256 bytes in stack

... jmpi main_loop ! return to the main loop

GNU lightning only supports Tail Call Optimization using the tramp construct. Any other way is not guaranteed to work on all ports.

An example of a simple (recursive) tail call optimization:

factorial:
! Entry point of the factorial function
prolog
in = arg
! Receive an integer argument
getarg R0, in
! Move argument to RO
prepare
pushargi 1
pushargr R0
finishi fact
retval R0
retr R0
epilog
! Epilog *before* label before prolog

fact:
! Entry point of the helper function
prolog
frame 16
! Reserve 16 bytes in the stack
fact_entry:
! This is the tail call entry point
ac = arg
! The accumulator is the first argument
in = arg
! The factorial argument
getarg R0, ac
! Move the accumulator to R0
getarg R1, in
! Move the argument to R1
blei fact_out, R1, 1
! Done if argument is one or less
mulr R0, R0, R1
! accumulator *= argument
putargr R0, ac
! Update the accumulator
subi R1, R1, 1
! argument -= 1
putargr R1, in
! Update the argument
jmpi fact_entry
! Tail Call Optimize it!

fact_out:
retr R0
! Return the accumulator

Predicates
forward_p (not specified)  forward label predicate
indirect_p (not specified)  indirect label predicate
target_p (not specified)    used label predicate
arg_register_p (not specified)  argument kind predicate
callee_save_p (not specified)  callee save predicate
pointer_p (not specified)      pointer predicate
forward_p expects a jit_node_t* argument, and returns non zero if it is a forward label reference, that is, a label returned by forward, that still needs a link call.

indirect_p expects a jit_node_t* argument, and returns non zero if it is an indirect label reference, that is, a label that was returned by indirect.

target_p expects a jit_node_t* argument, that is any kind of label, and will return non zero if there is at least one jump or move referencing it.

arg_register_p expects a jit_node_t* argument, that must have been returned by arg, arg_f or arg_d, and will return non zero if the argument lives in a register. This call is useful to know the live range of register arguments, as those are very fast to read and write, but have volatile values.

callee_save_p expects a valid JIT_Rn, JIT_Vn, or JIT_Fn, and will return non zero if the register is callee save. This call is useful because on several ports, the JIT_Rn and JIT_Fn registers are actually callee save; no need to save and load the values when making function calls.

pointer_p expects a pointer argument, and will return non zero if the pointer is inside the generated jit code. Must be called after jit_emit and before jit_destroy_state.
4 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.

Each of the instructions above translates to a macro or function call. 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.

This small tutorial presents three examples:

• The incr function found in Chapter 3 [GNU lightning’s instruction set], page 3:
• A simple function call to printf
• An RPN calculator.
• Fibonacci numbers

4.1 A function which increments a number by one

Let’s see how to create and use the sample incr function created in Chapter 3 [GNU lightning’s instruction set], page 3:

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

static jit_state_t *_jit;

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

int main(int argc, char *argv[])
{
    jit_node_t *in;
    pifi incr;

    init_jit(argv[0]);
    _jit = jit_new_state();

    jit_prolog();    /* prolog */
    in = jit_arg();   /* in = arg */
    jit_getarg(JIT_R0, in);  /* getarg R0 */
    jit_addi(JIT_R0, JIT_R0, 1); /* addi R0, R0, 1 */
    jit_retr(JIT_R0);   /* retr R0 */

    incr = jit_emit();
    jit_clear_state();

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

    jit_destroy_state();
    finish_jit();
```

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_state_t *_jit;

You might wonder about what is jit_state_t. It is a structure that stores jit code generation information. The name _jit is special, because since multiple jit generators can run at the same time, you must either #define _jit my_jit_state or name it _jit.

typedef int (*pifi)(int);

Just a handy typedef for a pointer to a function that takes an int and returns another.

jit_node_t *in;

Declares a variable to hold an identifier for a function argument. It is an opaque pointer, that will hold the return of a call to arg and be used as argument to getarg.

pifi incr;

Declares a function pointer variable to a function that receives an int and returns an int.

init_jit(argv[0]);

You must call this function before creating a jit_state_t object. This function does global state initialization, and may need to detect CPU or Operating System features. It receives a string argument that is later used to read symbols from a shared object using GNU binutils if disassembly was enabled at configure time. If no disassembly will be performed a NULL pointer can be used as argument.

_jit = jit_new_state();

This call initializes a GNU lightning jit state.

jit_prolog();

Ok, so we start generating code for our beloved function...

in = jit_arg();

jit_getarg(JIT_R0, in);

We retrieve the first (and only) argument, an integer, and store it into the general-purpose register R0.

jit_addi(JIT_R0, JIT_R0, 1);

We add one to the content of the register.

jit_retr(JIT_R0);

This instruction generates a standard function epilog that returns the contents of the R0 register.
incr = jit_emit();
This instruction is very important. It actually translates the GNU lightning macros used before to machine code, flushes the generated code area out of the processor’s instruction cache and return a pointer to the start of the code.

jit_clear_state();
This call cleans up any data not required for jit execution. Note that it must be called after any call to jit_print or jit_address, as this call destroy the GNU lightning intermediate representation.

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.

jit_destroy_state();
Releases all memory associated with the jit context. It should be called after known the jit will no longer be called.

finish_jit();
This call cleans up any global state hold by GNU lightning, and is advisable to call it once jit code will no longer be generated.

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_64 architecture (on the right is the code that an assembly-language programmer would write):

**SPARC**

```
save %sp, -112, %sp
mov %i0, %g2    retl
inc %g2    inc %o0
mov %g2, %i0
restore
retl
nop
```

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 %g2 on the SPARC).

**x86_64**

```
sub $0x30,%rsp
mov %rbp,(%rsp)
mov %rsp,%rbp
sub $0x18,%rsp
mov %rdi,%rax    mov %rdi, %rax
add $0x1,%rax    inc %rax
mov %rbp,%rsp
mov (%rsp),%rbp
```
add $0x30,%rsp
retq retq
retq

In this case, the main overhead is due to the function’s prolog and epilog, and stack alignment after reserving stack space for word to/from float conversions or moving data from/to x87 to/from SSE. Note that besides allocating space to save callee saved registers, no registers are saved/restored because GNU lightning notices those registers are not modified. There is currently no logic to detect if it needs to allocate stack space for type conversions neither proper leaf function detection, but these are subject to change (FIXME).

4.2 A simple function call to printf

Again, here is the code for the example:

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

static jit_state_t *_jit;

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

int main(int argc, char *argv[])
{
    pvfi myFunction;             /* ptr to generated code */
    jit_node_t *start, *end;     /* a couple of labels */
    jit_node_t *in;              /* to get the argument */

    init_jit(argv[0]);
    _jit = jit_new_state();

    start = jit_note(__FILE__, __LINE__);
    jit_prolog();
    in = jit_arg();
    jit_getarg(JIT_R1, in);
    jit_prepare();
    jit_pushargi((jit_word_t)"generated %d bytes\n");
    jit_ellipsis();
    jit_pushargr(JIT_R1);
    jit_finishi(printf);
    jit_ret();
    jit_epilog();
    end = jit_note(__FILE__, __LINE__);

    myFunction = jit_emit();

    /* call the generated code, passing its size as argument */
    myFunction((char*)jit_address(end) - (char*)jit_address(start));
    jit_clear_state();
```
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 4.1 [A function which increments a number by one], page 15.

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

```c
start = jit_note(__FILE__, __LINE__);
...  
end = jit_note(__FILE__, __LINE__);
These two instruction call the jit_note macro, which creates a note in the jit code; arguments to jit_note usually are a filename string and line number integer, but using NULL for the string argument is perfectly valid if only need to create a simple marker in the code.

jit_ellipsis();
ellipse is usually is only required if calling varargs functions with double arguments, but it is a good practice to properly describe the ... in the call sequence.

jit_pushargi((jit_word_t)'generated %d bytes\n');
Note the use of the (jit_word_t) cast, that is used only to avoid a compiler warning, due to using a pointer where a wordsize integer type was expected.

jit_prepare();
...  
jit_finishi(printf);
Once the arguments to printf have been pushed, what means moving them to stack or register arguments, the printf function is called and the stack cleaned. Note how GNU lightning abstracts the differences between different architectures and ABIs – the client program does not know how parameter passing works on the host architecture.

jit_epilog();
Usually it is not required to call epilog, but because it is implicitly called when noticing the end of a function, if the end variable was set with a note call after the ret, it would not consider the function epilog.

myFunction((char*)jit_address(end) - (char*)jit_address(start));
This calls the generate jit function passing as argument the offset difference from the start and end notes. The address call must be done after the emit call or either a fatal error will happen (if GNU lightning is built with assertions enable) or an undefined value will be returned.
jit_clear_state();
    Note that jit_clear_state was called after executing jit in this example. It was done because it must be called after any call to jit_address or jit_print.

jit_disassemble();
    disassemble will dump the generated code to standard output, unless GNU lightning was built with the disassembler disabled, in which case no output will be shown.

4.3 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.

Since GNU lightning does not provide push/pop instruction, this example uses a stack-allocated area to store the data. Such an area can be allocated using the macro alloca1, which receives the number of bytes to allocate and returns the offset from the frame pointer register FP to the base of the area.

Usually, you will use the ldxi and stxi instruction to access stack-allocated variables. However, it is possible to use operations such as add to compute the address of the variables, and pass the address around.

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

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

static jit_state_t *_jit;

void stack_push(int reg, int *sp)
{
    jit_stxi_i (*sp, JIT_FP, reg);
    *sp += sizeof (int);
}

void stack_pop(int reg, int *sp)
{
    *sp -= sizeof (int);
    jit_ldxi_i (reg, JIT_FP, *sp);
}

jit_node_t *compile_rpn(char *expr)
{
    jit_node_t *in, *fn;
    int stack_base, stack_ptr;

    fn = jit_note(NULL, 0);
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 in the memory area that we allocate with \texttt{allocai}.

Compiling a numeric operand or the argument \texttt{x} 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.

This example allocates a fixed area for 32 \texttt{ints}. This is not a problem when the function is a leaf like in this case; in a full-blown compiler you will want to analyze the input and
determine the number of needed stack slots—a very simple example of register allocation. The area is then managed like a stack using stack_push and stack_pop.

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

```c
int main(int argc, char *argv[]) {
    jit_node_t *nc, *nf;
pifi c2f, f2c;
int i;

    init_jit(argv[0]);
    _jit = jit_new_state();

    nc = compile_rpn("32x9*5/+);
    nf = compile_rpn("x32-5*9/");
    (void)jit_emit();
c2f = (pifi)jit_address(nc);
f2c = (pifi)jit_address(nf);
jit_clear_state();

    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 += 18) printf("%3d ", i);
    printf("\nC:");
    for (i = 32; i <= 212; i += 18) printf("%3d ", f2c(i));
    printf("\n");

    jit_destroy_state();
    finish_jit();
    return 0;
}
```

The client displays a conversion table between Celsius and Fahrenheit degrees (both Celsius-to-Fahrenheit and Fahrenheit-to-Celsius). The formulas are, \( F(c) = c \times \frac{9}{5} + 32 \) and \( C(f) = \frac{(f - 32) \times 5}{9} \), 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.

### 4.4 Fibonacci numbers

The code in this section calculates the Fibonacci sequence. That is modeled by the recurrence relation:
f(0) = 0
f(1) = f(2) = 1
f(n) = f(n-1) + f(n-2)

The purpose of this example is to introduce branches. There are two kinds 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.

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

static jit_state_t *jit;

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

int main(int argc, char *argv[]) {
    pifi fib;
    jit_node_t *label;
    jit_node_t *call;
    jit_node_t *in; /* offset of the argument */
    jit_node_t *ref; /* to patch the forward reference */
    jit_node_t *zero; /* to patch the forward reference */

    init_jit(argv[0]);
    jit = jit_new_state();

    label = jit_label();
    jit_prolog ();
    in = jit_arg ();
    jit_getarg (JIT_V0, in); /* R0 = n */
    zero = jit_beqi (JIT_R0, 0);
    jit_movr (JIT_V0, JIT_R0); /* V0 = R0 */
    jit_movi (JIT_R0, 1);
    ref = jit_blei (JIT_V0, 2);
    jit_subi (JIT_V1, JIT_V0, 1); /* V1 = n-1 */
    jit_subi (JIT_V2, JIT_V0, 2); /* V2 = n-2 */
    jit_prepare();
    jit_pushargr(JIT_V1);
    call = jit_finishi(NULL);
    jit_patch_at(call, label);
    jit_retval(JIT_V1); /* V1 = fib(n-1) */
    jit_prepare();
    jit_pushargr(JIT_V2);
    call = jit_finishi(NULL);
    jit_patch_at(call, label);
    jit_retval(JIT_R0); /* R0 = fib(n-2) */
    jit_addr(JIT_R0, JIT_R0, JIT_V1); /* R0 = R0 + V1 */
    }
As said above, this is the first example of dynamically compiling branches. Branch instructions have two operands containing the values to be compared, and return a jit_note_t * object to be patched.

Because labels final address are only known after calling emit, it is required to call patch or patch_at, what does tell GNU lightning that the target to patch is actually a pointer to a jit_node_t * object, otherwise, it would assume that is a pointer to a C function. Note that conditional branches do not receive a label argument, so they must be patched.

You need to call patch_at on the return of value calli, finishi, and calli if it is actually referencing a label in the jit code. All branch instructions do not receive a label argument. Note that movi is an special case, and patching it is usually done to get the final address of a label, usually to later call jmp_r.

Now, here is the iterative version:

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

static jit_state_t *_jit;

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

int main(int argc, char *argv[]) {
    pifi fib;
    jit_node_t *in; /* offset of the argument */
    jit_node_t *ref; /* to patch the forward reference */
    jit_node_t *zero; /* to patch the forward reference */
    jit_node_t *jump; /* jump to start of loop */
    jit_node_t *loop; /* start of the loop */

    init_jit(argv[0]);
    _jit = jit_new_state();

    jit_prolog ();
    in = jit_arg ();
```
This code calculates the recurrence relation using iteration (a `for` loop in high-level languages). There are no function calls anymore: instead, there is a backward jump (the `bnei` at the end of the loop).

Note that the program must remember the address for backward jumps; for forward jumps it is only required to remember the jump code, and call `patch` for the implicit label.
5 Re-entrant usage of GNU *lightning*

GNU *lightning* uses the special `_jit` identifier. To be able to use multiple jit generation states at the same time, it is required to used code similar to:

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

This will cause the symbol defined to `_jit` to be passed as the first argument to the underlying GNU *lightning* implementation, that is usually a function with an _ (underscode) prefix and with an argument named `_jit`, in the pattern:

```c
static void _jit_mnemonic(jit_state_t *, jit_gpr_t, jit_gpr_t);
#define jit_mnemonic(u, v) _jit_mnemonic(_jit, u, v);
```

The reason for this is to use the same syntax as the initial lightning implementation and to avoid needing the user to keep adding an extra argument to every call, as multiple jit states generating code in parallel should be very uncommon.

5.1 Registers
6 Accessing the whole register file

As mentioned earlier in this chapter, all GNU lightning back-ends are guaranteed to have at least six general-purpose 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_F macros. They accept a parameter that identifies the register number, which must be strictly less than JIT_R_NUM, JIT_V_NUM and JIT_F_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.
7 Customizations

Frequently it is desirable to have more control over how code is generated or how memory is used during jit generation or execution.

7.1 Memory functions

To aid in complete control of memory allocation and deallocation GNU lightning provides wrappers that default to standard malloc, realloc and free. These are loosely based on the GNU GMP counterparts, with the difference that they use the same prototype of the system allocation functions, that is, no size for free or old_size for realloc.

```c
void jit_set_memory_functions (    
    void (*)(alloc_func_ptr)(size_t),  
    void (*)(realloc_func_ptr)(void *, size_t),  
    void (*free_func_ptr)(void *)
)
```

GNU lightning guarantees that memory is only allocated or released using these wrapped functions, but you must note that if lightning was linked to GNU binutils, malloc is probably will be called multiple times from there when initializing the disassembler.

Because init_jit may call memory functions, if you need to call `jit_set_memory_functions`, it must be called before init_jit, otherwise, when calling finish_jit, a pointer allocated with the previous or default wrappers will be passed.

```c
void jit_get_memory_functions (    
    void (**alloc_func_ptr)(size_t),  
    void (**realloc_func_ptr)(void *, size_t),  
    void (**free_func_ptr)(void *)
)
```

Get the current memory allocation function. Also, unlike the GNU GMP counterpart, it is an error to pass NULL pointers as arguments.

7.2 Alternate code buffer

To instruct GNU lightning to use an alternate code buffer it is required to call `jit_realize` before `jit_emit`, and then query states and customize as appropriate.

```c
void jit_realize ()
```

Must be called once, before `jit_emit`, to instruct GNU lightning that no other jit_xyz call will be made.

```c
jit_pointer_t jit_get_code (jit_word_t *code_size)
```

Returns NULL or the previous value set with `jit_set_code`, and sets the `code_size` argument to an appropriate value. If `jit_get_code` is called before `jit_emit`, the `code_size` argument is set to the expected amount of bytes required to generate code. If `jit_get_code` is called after `jit_emit`, the `code_size` argument is set to the exact amount of bytes used by the code.
void jit_set_code (jit_pointer_t code, jit_word_t size)  [Function]
Instructs GNU lightning to output to the code argument and use size as a guard to
not write to invalid memory. If during jit_emit GNU lightning finds out that the
code would not fit in size bytes, it halts code emit and returns NULL.

A simple example of a loop using an alternate buffer is:

    jit_uint8_t *code;
    int *(func)(int);      /* function pointer */
    jit_word_t code_size;
    jit_word_t real_code_size;
    ...
    jit_realize();         /* ready to generate code */
    jit_get_code(&code_size);  /* get expected code size */
    code_size = (code_size + 4095) & -4096;
    do (;;) {
        code = mmap(NULL, code_size, PROT_EXEC | PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANON, -1, 0);
        jit_set_code(code, code_size);
        if ((func = jit_emit()) == NULL) {
            munmap(code, code_size);
            code_size += 4096;
        }
    } while (func == NULL);
    jit_get_code(&real_code_size);  /* query exact size of the code */

The first call to jit_get_code should return NULL and set the code_size argument to
the expected amount of bytes required to emit code. The second call to jit_get_code is
after a successful call to jit_emit, and will return the value previously set with jit_set_code and set the real_code_size argument to the exact amount of bytes used to emit the code.

### 7.3 Alternate data buffer

Sometimes it may be desirable to customize how, or to prevent GNU lightning from using
an extra buffer for constants or debug annotation. Usually when also using an alternate
code buffer.

jit_pointer_t jit_get_data (jit_word_t *data_size, jit_word_t  [Function]
  *note_size)
Returns NULL or the previous value set with jit_set_data, and sets the data_size
argument to how many bytes are required for the constants data buffer, and note_size
to how many bytes are required to store the debug note information. Note that it
always preallocate one debug note entry even if jit_name or jit_note are never
called, but will return zero in the data_size argument if no constant is required;
constants are only used for the float and double operations that have an immediate
argument, and not in all GNU lightning ports.
void jit_set_data (jit_pointer_t data, jit_word_t size, jit_word_t flags)

data can be NULL if disabling constants and annotations, otherwise, a valid pointer must be passed. An assertion is done that the data will fit in size bytes (but that is a noop if GNU lightning was built with -DNDEBUG).

size tells the space in bytes available in data.

flags can be zero to tell to just use the alternate data buffer, or a composition of JIT_DISABLE_DATA and JIT_DISABLE_NOTE

JIT_DISABLE_DATA

Instructs GNU lightning to not use a constant table, but to use an alternate method to synthesize those, usually with a larger code sequence using stack space to transfer the value from a GPR to a FPR register.

JIT_DISABLE_NOTE

Instructs GNU lightning to not store file or function name, and line numbers in the constant buffer.

A simple example of a preventing usage of a data buffer is:

```c
...  
jit_realize();           /* ready to generate code */
jit_get_data(NULL, NULL);
jit_set_data(NULL, 0, JIT_DISABLE_DATA | JIT_DISABLE_NOTE);
...  
Or to only use a data buffer, if required:

jit_uint8_t *data;
jit_word_t data_size;
...  
jit_realize();           /* ready to generate code */
jit_get_data(&data_size, NULL);
if (data_size)
    data = malloc(data_size);
else
    data = NULL;
jit_set_data(data, data_size, JIT_DISABLE_NOTE);
...  
if (data)
    free(data);
...  
```
8 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.