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 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.
2 Configuring and installing GNU *lightning*

Here we will assume that your system already has the dependencies necessary to build GNU *lightning*. For more on dependencies, see GNU *lightning’s* README-hacking file.

The first thing to do to build 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
```

The configure accepts 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.

configure also accepts the `--enable-devel-disassembler`, option useful to check exactly what machine instructions were generated for a GNU *lightning* instruction. Basically mixing `jit_print` and `jit_disassembly`.

The `--enable-assertions` option, which enables several consistency checks in the runtime 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.

The `--enable-devel-strong-type-checking` option that does extra type checking using `assert`. This option also enables the `--enable-assertions` unless it is explicitly disabled.

The option `--enable-devel-get-jit-size` should only be used when doing updates or maintenance to lightning. It regenerates the `jit_`.\$ARCH\]-sz.c creating a table or maximum bytes usage when translating a GNU *lightning* instruction to machine code.

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.

```
addr   _f _d 01 = 02 + 03
addi   _f _d 01 = 02 + 03
addxr  01 = 02 + (03 + carry)
addxi  01 = 02 + (03 + carry)
addcr  01 = 02 + 03, set carry
addci  01 = 02 + 03, set carry
subr   _f _d 01 = 02 - 03
sbi    _f _d 01 = 02 - 03
subxr  01 = 02 - (03 + carry)
subxi  01 = 02 - (03 + carry)
subcr  01 = 02 - 03, set carry
subci  01 = 02 - 03, set carry
rsbr   _f _d 01 = 03 - 01
rsbi   _f _d 01 = 03 - 01
mulr   _f _d 01 = 02 * 03
muli   _f _d 01 = 02 * 03
divr   _u _f _d 01 = 02 / 03
divi   _u _f _d 01 = 02 / 03
remr   _u 01 = 02 % 03
remi   _u 01 = 02 % 03
andr   01 = 02 & 03
andi   01 = 02 & 03
orrr   01 = 02 | 03
orri   01 = 02 | 03
xorrr  01 = 02 ^ 03
xorri  01 = 02 ^ 03
lshr   01 = 02 >> 03^1
rshri  _u 01 = 02 >> 03^2
lrotr  01 = (02 << 03) | (03 >> (WORDSIZE - 03))
lroti  01 = (02 << 03) | (03 >> (WORDSIZE - 03))
```

^1 The sign bit is propagated unless using the _u modifier.

^2 The sign bit is propagated unless using the _u modifier.
\r
\r
\texttt{rrot\nobreakspace r}  \quad  \texttt{01} = (\texttt{02} \gg \texttt{03}) | (\texttt{03} \ll (\texttt{WORDSIZE} - \texttt{03}))
\texttt{rroti}  \quad  \texttt{01} = (\texttt{02} \gg \texttt{03}) | (\texttt{03} \ll (\texttt{WORDSIZE} - \texttt{03}))
\texttt{movzr}  \quad  \texttt{01} = \texttt{03} ? \texttt{01} : \texttt{02}
\texttt{movnr}  \quad  \texttt{01} = \texttt{03} ? \texttt{02} : \texttt{01}
\r
Note that \texttt{lrot\nobreakspace r}, \texttt{lroti}, \texttt{rrot\nobreakspace r} and \texttt{rroti} are described as the fallback operation. These are bit shift/rotation operation.

\textbf{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{01} and the high word in \texttt{02}. For unsigned multiplication, \texttt{02} zero means there was no overflow. For signed multiplication, no overflow check is based on sign, and can be detected if \texttt{02} is zero or minus one.

\texttt{qdiv} stores the quotient in \texttt{01} and the remainder in \texttt{02}. It can be used as a quick way to check if a division is exact, in which case the remainder is zero.

\texttt{qlsh} shifts from 0 to \texttt{wordsize}, doing a normal left shift for the first result register and setting the second result register to the overflow bits. \texttt{qlsh} can be used as a quick way to multiply by powers of two.

\texttt{qrsh} shifts from 0 to \texttt{wordsize}, doing a normal right shift for the first result register and setting the second result register to the overflow bits. \texttt{qrsh} can be used as a quick way to divide by powers of two.

Note that \texttt{qlsh} and \texttt{qrsh} are basically implemented as two shifts. It is undefined behavior to pass a value not in the range 0 to \texttt{wordsize}. Most cpus will usually \texttt{and} the shift amount with \texttt{wordsize} - 1, or possible use the \texttt{remainder}. GNU \texttt{lightning} only generates code to specially handle 0 and \texttt{wordsize} shifts. Since in a code generator for a \textit{safe language} should usually check the shift amount, these instructions usually should be used as a fast path to check for division without remainder or multiplication that does not overflow.

\begin{align*}
\texttt{qmulr} \quad &\texttt{01} \texttt{02} = \texttt{03} \times \texttt{04} \\
\texttt{qmuli} \quad &\texttt{01} \texttt{02} = \texttt{03} \times \texttt{04} \\
\texttt{qdivr} \quad &\texttt{01} \texttt{02} = \texttt{03} \div \texttt{04} \\
\texttt{qdivi} \quad &\texttt{01} \texttt{02} = \texttt{03} \div \texttt{04} \\
\texttt{qlshr} \quad &\texttt{01} = \texttt{03} \ll \texttt{04}, \texttt{02} = \texttt{03} \gg (\texttt{WORDSIZE} - \texttt{04}) \\
\texttt{qlshi} \quad &\texttt{01} = \texttt{03} \ll \texttt{04}, \texttt{02} = \texttt{03} \gg (\texttt{WORDSIZE} - \texttt{04}) \\
\texttt{qrshr} \quad &\texttt{01} = \texttt{03} \gg \texttt{04}, \texttt{02} = \texttt{03} \ll (\texttt{WORDSIZE} - \texttt{04}) \\
\texttt{qrshi} \quad &\texttt{01} = \texttt{03} \gg \texttt{04}, \texttt{02} = \texttt{03} \ll (\texttt{WORDSIZE} - \texttt{04})
\end{align*}

These four operand ALU operations are only defined for float operands.

\begin{align*}
\texttt{fm\texttt{ar}} \quad &\texttt{01} = 02 \times 03 + 04 \\
\texttt{fma\texttt{i}} \quad &\texttt{01} = 02 \times 03 + 04 \\
\texttt{fms\texttt{r}} \quad &\texttt{01} = 02 \times 03 - 04 \\
\texttt{fms\texttt{i}} \quad &\texttt{01} = 02 \times 03 - 04 \\
\texttt{fnmar} \quad &\texttt{01} = -02 \times 03 - 04 \\
\texttt{fnmai} \quad &\texttt{01} = -02 \times 03 - 04 \\
\texttt{fnmsr} \quad &\texttt{01} = -02 \times 03 + 04 \\
\texttt{fnmsi} \quad &\texttt{01} = -02 \times 03 + 04
\end{align*}
These are a family of fused multiply-add instructions. Note that GNU lightning does not handle rounding modes nor math exceptions. Also note that not all backends provide a instruction for the equivalent GNU lightning instruction presented above. Some are completely implemented as fallbacks and some are composed of one or more instructions. For common input this should not cause major issues, but note that when implemented by the cpu, these are implemented as the multiplication calculated with infinite precision, and after the addition step rounding is done. Due to this, For specially crafted input different ports might show different output. When implemented by the CPU, it is also possible to have exceptions that do not happen if implemented as a fallback.

Unary ALU operations

These accept two operands, the first must be a register and the second is a register if the r modifier is used, otherwise, the i modifier is used and the second argument is a constant.

```
negr _f _d 01 = -02
negi _f _d 01 = -02
comr 01 = ~02
comi 01 = ~02
clor 01 = number of leading one bits in 02
cloi 01 = number of leading one bits in 02
clzr 01 = number of leading zero bits in 02
clzi 01 = number of leading zero bits in 02
ctor 01 = number of trailing one bits in 02
ctoi 01 = number of trailing one bits in 02
ctzr 01 = number of trailing zero bits in 02
ctzi 01 = number of trailing zero bits in 02
rbitr 01 = bits of 02 reversed
rbiti 01 = bits of 02 reversed
popcntr 01 = number of bits set in 02
popcnti 01 = number of bits set in 02
```

Note that ctzr is basically equivalent of a C call ffs but indexed at bit zero, not one.

Contrary to __builtin_ctz and __builtin_clz, an input value of zero is not an error, it just returns the number of bits in a word, 64 if GNU lightning generates 64 bit instructions, otherwise it returns 32.

The clor and ctor are just counterparts of the versions that search for zero bits.

These unary ALU operations are only defined for float operands.

```
absr _f _d 01 = fabs(02)
absi _f _d 01 = fabs(02)
sqrtr _f _d 01 = sqrt(02)
sqrti _f _d 01 = sqrt(02)
```

Note that for float and double unary operations, GNU lightning will generate code to actually execute the operation at runtime.
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.

\[
\begin{align*}
\text{ltr} & \quad _u \_f \_d \ 01 = (02 < 03) \\
\text{lti} & \quad _u \_f \_d \ 01 = (02 < 03) \\
\text{ler} & \quad _u \_f \_d \ 01 = (02 <= 03) \\
\text{lei} & \quad _u \_f \_d \ 01 = (02 <= 03) \\
\text{gtr} & \quad _u \_f \_d \ 01 = (02 > 03) \\
\text{gti} & \quad _u \_f \_d \ 01 = (02 > 03) \\
\text{ger} & \quad _u \_f \_d \ 01 = (02 >= 03) \\
\text{gei} & \quad _u \_f \_d \ 01 = (02 >= 03) \\
\text{eqr} & \quad _f \_d \ 01 = (02 == 03) \\
\text{eqi} & \quad _f \_d \ 01 = (02 == 03) \\
\text{ner} & \quad _f \_d \ 01 = (02 != 03) \\
\text{nei} & \quad _f \_d \ 01 = (02 != 03) \\
\text{unltr} & \quad _f \_d \ 01 = !(02 >= 03) \\
\text{unler} & \quad _f \_d \ 01 = !(02 > 03) \\
\text{ungtr} & \quad _f \_d \ 01 = !(02 <= 03) \\
\text{unger} & \quad _f \_d \ 01 = !(02 < 03) \\
\text{uneqr} & \quad _f \_d \ 01 = !(02 < 03) \land !(02 > 03) \\
\text{ltgtr} & \quad _f \_d \ 01 = !(02 >= 03) \lor !(02 <= 03) \\
\text{ordr} & \quad _f \_d \ 01 = (02 == 02) \land (03 == 03) \\
\text{unordr} & \quad _f \_d \ 01 = (02 != 02) \lor (03 != 03)
\end{align*}
\]

Transfer operations

These accept two operands; for \text{extr} both of them must be registers, while \text{mov} accepts an immediate value as the second operand.

Unlike \text{movr} and \text{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 \text{extr} to convert an integer to a floating point value: the usual options are \text{extr}_f and \text{extr}_d.

\[
\begin{align*}
\text{movr} & \quad _f \_d \ 01 = 02 \\
\text{movi} & \quad _f \_d \ 01 = 02 \\
\text{extr} & \quad _c \_uc \_s \_us \_i \_ui \_f \_d \ 01 = 02 \\
\text{truncr} & \quad _f \_d \ 01 = \text{trunc}(02) \\
\text{extr} & \quad 01 = \text{sign}_{\text{extend}}(02[03:03+04]) \\
\text{extr}_u & \quad 01 = 02[03:03+04] \\
\text{depr} & \quad 01[03:03+04] = 02
\end{align*}
\]

\text{extr}, \text{extr}_u and \text{depr} are useful to access C compatible bit fields, provided that these are contained in a machine word. \text{extr} is used to \text{extract} and signed extend a value from a bit field. \text{extr}_u is used to \text{extract} and zero extend a value from a bit field. \text{depr} is used to \text{deposit} a value into a bit field.

\[
\text{extr}(\text{result}, \text{source}, \text{offset}, \text{length})
\]
A common way to declare C and GNU lightning compatible bit fields is:

```c
union {
    struct {
        jit_word_t signed_bits: length;
        jit_uword_t unsigned_bits: length;
        ...
    } s;
    jit_word_t signed_value;
    jit_uword_t unsigned_value;
} u;
```

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.

```c
truncr_f_i <int> O1 = <float> O2
truncr_f_l <long>O1 = <float> O2
truncr_d_i <int> O1 = <double>02
truncr_d_l <long>O1 = <double>02
```

The float conversion operations are *destination first, source second*, but the order of the types is reversed. This happens for historical reasons.

```c
extr_f_d <double>O1 = <float> O2
extr_d_f <float> O1 = <double>02
```

The float to/from integer transfer operations are also *destination first, source second*. These were added later, but follow the pattern of historic patterns.

```c
movr_w_f <float>O1 = <int>O2
movi_w_f <float>O1 = <int>O2
movr_f_w <int>O1 = <float>O2
movi_f_w <int>O1 = <float>O2
movr_w_d <double>O1 = <long>O2
movi_w_d <double>O1 = <long>O2
movr_d_w <long>O1 = <double>O2
movi_d_w <long>O1 = <double>O2
movr_ww_d <double>O1 = [<int>O2:<int>O3]
movi_ww_d <double>O1 = [<int>O2:<int>O3]
movr_d_ww [<int>O1:<int>O2] = <double>O3
movi_d_ww [<int>O1:<int>O2] = <double>O3
```

These are used to transfer bits to/from floats to/from integers, and are useful to access bits of floating point values.

`movr_w_d`, `movi_w_d`, `movr_d_w` and `movi_d_w` are only available in 64-bit. Conversely, `movr_ww_d`, `movi_ww_d`, `movr_d_ww` and `movi_d_ww` are only available in 32-bit. For the int pair to/from double transfers, integer arguments must respect endianess, to match how the cpu handles the verbatim byte values.
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 \ul: Host-to-network (big endian) order
- \ntohr \ul: Network-to-host order

\bswapr can be used to unconditionally byte-swap an operand. On little-endian architectures, \htonr and \ntohr resolve to this. The \ul variant is only available in 64-bit architectures.

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 \ul 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: Load word
- \ldi: Load immediate
- \ldxr: Load word from register
- \ldxi: Load immediate from register

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: Store word
- \sti: Store immediate
- \stxr: Store word from register
- \stxi: Store immediate from register

Note that the unsigned type modifier is not available, as the store only writes to the 1, 2, 4 or 8 sized memory address. The \ul type is 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.

Unaligned memory access

These allow access to integers of size 3, in 32-bit, and extra sizes 5, 6 and 7 in 64-bit. For floating point values only support for size 4 and 8 is provided.

- \unldr: Unaligned Load word
- \unldi: Unaligned Load immediate
- \unldr_u: Unaligned Load word (unsigned)
- \unldi_u: Unaligned Load immediate (unsigned)
- \unldr_x: Unaligned Load word (float)
- \unldi_x: Unaligned Load immediate (float)
unstr *(O3 byte integer)01 = 02
unsti *(O3 byte integer)01 = 02
unstr_x *(O3 byte float)01 = 02
unsti_x *(O3 byte float)01 = 02

With the exception of non-standard sized integers, these might be implemented as normal loads and stores, if the processor supports unaligned memory access, or, mode can be chosen at jit initialization time, to generate or not generate, code that does trap on unaligned memory access. Letting the kernel trap means smaller code generation as it is required to check alignment at runtime\(^3\).

**Argument management**

These are:

- `prepare` (not specified)
- `va_start` (not specified)
- `pushargr`  
  _c _uc _s _us _i _ui _l _f _d
- `pushargi`  
  _c _uc _s _us _i _ui _l _f _d
- `va_push` (not specified)
- `arg`  
  _c _uc _s _us _i _ui _l _f _d
- `getarg`  
  _c _uc _s _us _i _ui _l _f _d
- `va_arg`  
  _d
- `putargr`  
  _c _uc _s _us _i _ui _l _f _d
- `putargi`  
  _c _uc _s _us _i _ui _l _f _d
- `ret` (not specified)
- `retr`  
  _c _uc _s _us _i _ui _l _f _d
- `reti`  
  _c _uc _s _us _i _ui _l _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.

Note that `arg`, `pusharg`, `putarg` and `ret` when handling integer types can be used without a type modifier. It is suggested to use matching type modifiers to `arg`, `putarg` and `getarg` otherwise problems will happen if generating jit for environments that require arguments to be truncated and zero or sign extended by the caller and/or excess arguments might be passed packed in the stack. Currently only Apple systems with `aarch64` cpus are known to have this restriction.

---

\(^3\) This requires changing `jit.cpu.unaligned` to 0 to disable or 1 to enable unaligned code generation. Not all ports have the C `jit.cpu.unaligned` value.
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 20.

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 27, 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:
```

---

4 "Return a value" means that GNU lightning code that compile these instructions return a value when expanded.
prolog
... 
ret
fun2:
   epilog
prolog

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 27. 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 < O3) goto 01
blti _u _f _d if (O2 < O3) goto 01
bler _u _f _d if (O2 <= O3) goto 01
blei _u _f _d if (O2 <= O3) goto 01
bgtr _u _f _d if (O2 > O3) goto 01
bgti _u _f _d if (O2 > O3) goto 01
bger _u _f _d if (O2 >= O3) goto 01
bgei _u _f _d if (O2 >= O3) goto 01
beqr _f _d if (O2 == O3) goto 01
beqi _f _d if (O2 == O3) goto 01
bner _f _d if (O2 != O3) goto 01
bnei _f _d if (O2 != O3) goto 01
bunltr _f _d if !(O2 >= O3) goto 01
bunler _f _d if !(O2 > O3) goto 01
bungtr _f _d if !(O2 <= O3) goto 01
bungr _f _d if !(O2 < O3) goto 01
buneqr _f _d if !(O2 < O3) && !(O2 > O3) goto 01
bltgtr _f _d if !(O2 >= O3) || !(O2 <= O3) goto 01
bordr _f _d if (O2 == O2) && (O3 == O3) goto 01
bunordr _f _d if !(O2 != O2) || (O3 != O3) goto 01
```

```
bmsr if 02 & 03 goto 01
bmsi if 02 & 03 goto 01
bmcr if !(02 & 03) goto 01
bmci if !(02 & 03) goto 01
5
boaddr _u 02 += 03, goto 01 if overflow
```

5 These mnemonics mean, respectively, branch if mask set and branch if mask cleared.
boaddi _u O2 += O3, goto O1 if overflow
bxaddr _u O2 += O3, goto O1 if no overflow
bxaddi _u O2 += O3, goto O1 if no overflow
bosubr _u O2 -= O3, goto O1 if overflow
bosubi _u O2 -= O3, goto O1 if overflow
bxsurb _u O2 -= O3, goto O1 if no overflow
bxsubi _u O2 -= O3, goto O1 if no overflow

Note that the C code does not have an O1 argument. It is required to always use the return value as an argument to patch, patch_at or patch_abs.

Jump and return operations

These accept one argument except ret and jmpi 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.

callr (not specified) function call to register O1
calli (not specified) function call to immediate O1
finishr (not specified) function call to register O1
finishi (not specified) function call to immediate O1
jmp r (not specified) unconditional jump to register
jmpi (not specified) unconditional jump
ret (not specified) return from subroutine
retr _c _uc _s _us _i _ui _l _f _d
re ti _c _uc _s _us _i _ui _l _f _d
retval _c _uc _s _us _i _ui _l _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 4.4 [Fibonacci numbers], page 27.

Labels

There are 3 GNU lightning instructions to create labels:

label (not specified) simple label
forward (not specified) forward label
indirect (not specified) special simple label

The following instruction is used to specify a minimal alignment for the next instruction, usually with a label:

align (not specified) align code

Similar to align is the next instruction, also usually used with a label:

skip (not specified) skip code

It is used to specify a minimal number of bytes of nops to be inserted before the next instruction.

label is normally used as patch_at argument for backward jumps.

    jit_node_t *jump, *label;
    label = jit_label();
    ...

jump = jit_beqr(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.

    jit_node_t *jump, *label;
    label = jit_forward();
    jump = jit_beqr(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.

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

indirect is an 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

    jit_node_t *addr, *jump;
    addr = jit_movi(JIT_R0, 0);  /* immediate is ignored */
    ...
    jmpr(JIT_R0);
    ...
    jit_patch(addr);  /* implicit label added */

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.

align is useful for creating multiple entry points to a (trampoline) function that are all accessible through a single function pointer. align receives an integer argument that defines the minimal alignment of the address of a label directly following the align instruction. The integer argument must be a power of two and the effective alignment will be a power of two no less than the argument to align. If the argument to align is 16 or more, the effective alignment will match the specified minimal alignment exactly.

    jit_node_t *forward, *label1, *label2, *jump;
    unsigned char *addr1, *addr2;
    forward = jit_forward();
    jit_align(16);
    label1 = jit_indirect();  /* first entry point */
    jump = jit_jmpi();  /* jump to first handler */
    jit_patch_at(jump, forward);
    jit_align(16);
    label2 = jit_indirect();  /* second entry point */
/* second handler */
jit_jmpr(...);
jit_link(forward);
...  /* first handler */
jit_jmpr(...);
...
jit_emit();
addr1 = jit_address(label1);
addr2 = jit_address(label2);
assert(addr2 - addr1 == 16);  /* only one of the ad-
dresses needs to be remembered */
skip is useful for reserving space in the code buffer that can later be filled (possibly with the help of the pair of functions jit_unprotect and jit_protect).

Function prolog
These macros are used to set up a function prolog. The allocai call accept 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.

    prolog    (not specified)    function prolog
allocai  (not specified)    reserve space on the stack
allocar  (not specified)    allocate space on the stack

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 20.

    incr:
        prolog
        in = arg    ! We have an integer argument
        getarg R0, in  ! Move it to R0
        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:
        prolog
Register liveness

During code generation, GNU lightning occasionally needs scratch registers or needs to use architecture-defined registers. For that, GNU lightning internally maintains register liveness information.

In the following example, qdivr will need special registers like R0 on some architectures. As GNU lightning understands that R0 is used in the subsequent instruction, it will create save/restore code for R0 in case.

```
...  
qdivr V0, V1, V2, V3  
movr V3, R0  
...  
```

The same is not true in the example that follows. Here, R0 is not alive after the division operation because R0 is neither an argument register nor a callee-save register. Thus, no save/restore code for R0 will be created in case.

```
...  
qdivr V0, V1, V2, V3  
jmpr R1  
...  
```

The `live` instruction can be used to mark a register as live after it as in the following example. Here, R0 will be preserved across the division.

```
...  
qdivr V0, V1, V2, V3  
live R0  
jmpr R1  
...  
```

The `live` instruction is useful at code entry and exit points, like after and before a `callr` instruction.

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\(^6\) 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\(^7\), 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
    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
    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:

```plaintext
factorial:
    ! Entry point of the factorial function
    prolog
    in = arg
    ! Receive an integer argument
    getarg R0, in
    ! Move argument to RO
    prepare
    pushargi 1
    ! This is the accumulator
    pushargr R0
    ! This is the argument
    finishi fact
    ! Call the tail call optimized function
    retval R0
    ! Fetch the result
```

---

\(^6\) It is not automatically computed because it does not know about the requirement of later generated code.

\(^7\) Times eight so that it works for double arguments. And would not need conditionals for ports that pass arguments in the stack.
retr R0 ! Return it
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.
Atomic operations

Only compare-and-swap is implemented. It accepts four operands; the second can be an immediate.

The first argument is set with a boolean value telling if the operation did succeed.

Arguments must be different, cannot use the result register to also pass an argument.

The second argument is the address of a machine word.

The third argument is the old value.

The fourth argument is the new value.

\[
\text{casr} \quad 01 = (*O_2 == O_3) \ ? \ (*O_2 = O_4, 1) : 0 \\
\text{casi} \quad 01 = (*O_2 == O_3) \ ? \ (*O_2 = O_4, 1) : 0
\]

If value at the address in the second argument is equal to the third argument, the address value is atomically modified to the value of the fourth argument and the first argument is set to a non zero value.

If the value at the address in the second argument is not equal to the third argument nothing is done and the first argument is set to zero.
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 mul-
tiple 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 destroys 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 held 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**

```
mov %rdi,%rax
add $0x1,%rax
ret
```

In this case, for the x86 port, GNU lightning has simple optimizations to understand it is a leaf function, and that it is not required to create a stack frame nor update the stack pointer.
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();

    jit_disassemble();

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

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 20.
For this reason, we're going to concentrate on just a few statements.

\[
\text{start} = \text{jit\_note}(_-_\text{FILE}_-, _-_\text{LINE}_-);
\]

\[
\text{...}
\]

\[
\text{end} = \text{jit\_note}(_-_\text{FILE}_-, _-_\text{LINE}_-);
\]

These two instructions call the \text{jit\_note} macro, which creates a note in the jit code; arguments to \text{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.

\text{jit\_ellipsis();}

\text{ellipsis} 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.

\[
\text{jit\_pushargi}((\text{jit\_word\_t})\text{"generated }%d\text{ bytes\n"});
\]

Note the use of the \text{(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.

\text{jit\_prepare();}

\[
\text{...}
\]

\[
\text{jit\_finishi(printf);}\]

Once the arguments to \text{printf} have been pushed, what means moving them to stack or register arguments, the \text{printf} function is called and the stack cleaned. 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.

\text{jit\_epilog();}

Usually it is not required to call \text{epilog}, but because it is implicitly called when noticing the end of a function, if the \text{end} variable was set with a \text{note} call after the \text{ret}, it would not consider the function epilog.

\text{myFunction((char*)jit\_address(end) - (char*)jit\_address(start));}

This calls the generate jit function passing as argument the offset difference from the \text{start} and \text{end} notes. The \text{address} call must be done after the \text{emit} call or either a fatal error will happen (if GNU lightning is built with assertions enable) or an undefined value will be returned.

\text{jit\_clear\_state();}

Note that \text{jit\_clear\_state} was called after executing jit in this example. It was done because it must be called after any call to \text{jit\_address} or \text{jit\_print}.

\text{jit\_disassemble();}

\text{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 alloca,
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.

```
#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);
    jit_prolog();
    in = jit_arg();
    stack_ptr = stack_base = jit_alloca (32 * sizeof (int));

    jit_getarg(JIT_R2, in);

    while (*expr) {
        char buf[32];
        int n;
        if (sscanf(expr, "%[0-9]%n", buf, &n)) {
            expr += n - 1;
            stack_push(JIT_R0, &stack_ptr);
```
jit_mov(JIT_R0, atoi(buf));
} else if (*expr == 'x') {
    stack_push(JIT_R0, &stack_ptr);
    jit_movr(JIT_R0, JIT_R2);
} else if (*expr == '+') {
    stack_pop(JIT_R1, &stack_ptr);
    jit_addr(JIT_R0, JIT_R1, JIT_R0);
} else if (*expr == '-') {
    stack_pop(JIT_R1, &stack_ptr);
    jit_subr(JIT_R0, JIT_R1, JIT_R0);
} else if (*expr == '*') {
    stack_pop(JIT_R1, &stack_ptr);
    jit_mulr(JIT_R0, JIT_R1, JIT_R0);
} else if (*expr == '/') {
    stack_pop(JIT_R1, &stack_ptr);
    jit_divr(JIT_R0, JIT_R1, JIT_R0);
} else {
    fprintf(stderr, "cannot compile: %s\n", expr);
    abort();
    ++expr;
}
jit_retr(JIT_R0);
jit_epilog();
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 in the memory area that we allocate with alloca.
Compiling a numeric operand or the argument 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 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();
```
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

\[
C(f) = (f - 32) \times \frac{5}{9},
\]

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 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.

```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 */

    jit_patch(ref); /* patch jump */
    jit_patch(zero); /* patch jump */
    jit_retr(JIT_R0);

    /* call the generated code, passing 32 as an argument */
    fib = jit_emit();
    jit_clear_state();
    printf("fib(%d) = %d\n", 32, fib(32));
    jit_destroy_state();
    finish_jit();
    return 0;
}
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.

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 ();
    jit_getarg (JIT_R0, in); /* R0 = n */
    zero = jit_beqi (JIT_R0, 0);
    jit_movr (JIT_R1, JIT_R0);
    jit_movi (JIT_R0, 1);
    ref = jit_blti (JIT_R1, 2);
    jit_subi (JIT_R2, JIT_R2, 2);
    jit_movr (JIT_R1, JIT_R0);

    loop= jit_label();
    jit_subi (JIT_R2, JIT_R2, 1); /* decr. counter */
    jit_movr (JIT_V0, JIT_R0);    /* V0 = R0 */
    jit_addr (JIT_R0, JIT_R0, JIT_R1); /* R0 = R0 + R1 */
    jit_movr (JIT_R1, JIT_V0);    /* R1 = V0 */
```
jump= jit_bnei (JIT_R2, 0); /* if (R2) goto loop; */
jump= jit_patch_at(jump, loop);

jump= jit_patch(ref); /* patch forward jump */
jump= jit_patch(zero); /* patch forward jump */
jump= jit_retr (JIT_R0);

/* call the generated code, passing 36 as an argument */
fib = jit_emit();
jit_clear_state();
printf("fib(%d) = %d\n", 36, fib(36));
jit_destroy_state();
finish_jit();
return 0;
}

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 use 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 `_` (underscore) 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.
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.

6.1 Scratch registers

For operations, GNU *lightning* does not support directly, like storing a literal in memory, *jit_get_reg* and *jit_unget_reg* can be used to acquire and release a scratch register as in the following pattern:

```c
    jit_int32_t reg = jit_get_reg (jit_class_gpr);
    jit_movi (reg, immediate);
    jit_stxi (offsetof (some_struct, some_field), JIT_V0, reg);
    jit_unget_reg (reg);
```

As *jit_get_reg* and *jit_unget_reg* may generate spills and reloads but don’t follow branches, the code between both must be in the same basic block and must not contain any branches as in the following (bad) example.

```c
    jit_int32_t reg = jit_get_reg (jit_class_gpr);
    jit_ldxi (reg, JIT_V0, offset);
    jump = jit_bnei (reg, V0);
    jit_movr (JIT_V1, reg);
    jit_patch (jump);
    jit_unget_reg (reg);
```
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 Protection

Unless an alternate code buffer is used (see below), jit_emit set the access protections that the code buffer’s memory can be read and executed, but not modified. One can use the following functions after jit_emit but before jit_clear to temporarily lift the protection:

```c
void jit_unprotect ()
```

Changes the access protection that the code buffer’s memory can be read and modified. Before the emitted code can be invoked, jit_protect has to be called to reset the change.

This procedure has no effect when an alternate code buffer (see below) is used.

```c
void jit_protect ()
```

Changes the access protection that the code buffer’s memory can be read and executed.

This procedure has no effect when an alternate code buffer (see below) is used.
7.3 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 ()
    [Function]
    Must be called once, before `jit_emit`, to instruct GNU *lightning* that no other `jit_xyz` call will be made.

jit_pointer_t jit_get_code (jit_word_t *code_size)
    [Function]
    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:

```c
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.
7.4 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.

```c
jit_pointer_t jit_get_data (jit_word_t *data_size, jit_word_t *note_size) [Function]
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.
```

```c
void jit_set_data (jit_pointer_t data, jit_word_t size, jit_word_t flags) [Function]
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:

```
...  
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.