The NEATCC C Compiler

Ali Gholami Rudi

NEATCC is a small C compiler that implements a large subset of ANSI C. Despite its size, NEATCC implements effective optimizations and generates code for different architectures. In this document, I shall briefly introduce NEATCC, its intermediate code, its final code generation interface, and some other details that seem helpful for inspecting its source code and extending it.

Overview

In NEATCC, compilation phases are implemented in different source files, which use the interfaces declared in ncc.h to interact. The main components of NEATCC are implemented in the following files.

cpp.c	Preprocessing.
tok.c	Tokenisation.
ncc.c	Parsing and directing compiler phases.
int.c	Intermediate code generation.
gen.c	Register allocation and sending intermediate code to backends.
reg.c	Global register allocation.
x64.c	Final code generation (x86.c and arm.c as well).

Most NEATCC optimizations are performed on the intermediate code (implemented in int.c), such as using instruction immediates, removing unused values, or constant folding; they are enabled when the optimization level is at least one. For global register allocation, NEATCC performs liveness analysis for local variables when the optimization level is two; level one enables a simpler register allocation algorithm and zero disables global register allocation altogether.

Intermediate Code

NEATCC's parser (ncc.c) calls some of the functions defined in int.c (prefixed with “o_”), to generate the intermediate code. The latter also performs optimizations on the generated intermediate code, such as constant folding, in functions prefixed with “io_”. The intermediate code is stored as an array of ic struct, which is defined as follows:

```
struct ic {
    long op;    /* instruction opcode */
    long a1;    /* first argument */
    long a2;    /* second argument */
    long a3;    /* third argument, jump target, argument count */
    long *args; /* call arguments */
};
```

The arguments of instructions can be compiler temporaries (or intermediate values), immediates, branch instruction targets, local identifiers, and symbol identifiers. A compiler temporary is specified as positive integer, indicating the instruction that defines them (thus, the value of compiler temporaries cannot be changed, once defined). For instance, temporary number 5 is the output in the 5th intermediate code instruction, which may define it to be the result of adding two other temporaries.

Instruction opcode (`ic->op`) can be one of the macros prefixed with “`O_`” in ncc.h; `ic->op` also specifies the type of the operands with `O_MK` macro.

`O_ADD`: Performs addition for temporaries `ic->a1` and `ic->a2`; the same applies to other binary instructions such as `O_SUB`.

`O_ADD|O_NUM`: Similar to `O_ADD` except that `ic->a2` is an immediate.

`O_NEG`: Negates `ic->a1`; the same applies to other unary instructions like `O_NOT`.

`O_CALL`: Calls a function, whose address is stored in `ic->arg1`. `ic->a3` specifies the number of arguments and `ic->args` is the list of arguments.

`O_CALL|O_SYM`: Similar to `O_CALL`, except that the function is specified as a symbol identifier (instead of a temporary containing the address of the function) in `ic->a1`.

`O_MOV`: Assigns the value of `ic->a1`, casting the value according to `O_T(ic->op)`, if necessary.

`O_MOV|O_NUM`: Like `O_MOV`, but loads `ic->a1` as an immediate.

O_MOV|O_SYM: Like **O_MOV**, but loads the address of the given symbol **ic->a1** with offset **ic->a2**.

O_MOV|O_LOC: Like **O_MOV**, but loads the address of the given local variable **ic->a1** with offset **ic->a2**.

O_MSET: Performs memset() with the given arguments.

O_MCPY: Performs memcpy() with the given arguments.

O_RET: Returns **ic->a1** from a function.

O_LD|O_NUM: Loads the value of the address specified as **ic->a1** with offset **ic->a2**; the same applies to **O_ST** for storing values, with the exception that the first argument is the destination and the second argument is the address.

O_LD|O_SYM: Like **O_LD|O_NUM**, except that **ic->a1** specifies a symbol.

O_LD|O_LOC: Like **O_LD|O_NUM**, except that **ic->a1** specifies a local.

O_JMP: Unconditional branch to instruction **ic->a3**.

O_JZ: Conditional branch to instruction **ic->a3**, if **ic->a1** is zero (**O_JNZ** for nonzero).

O_JCC: Conditional branch to instruction **ic->a3**, if the given relation (**ic->op & 0x0f**) holds for **ic->a1** and **ic->a2**.

Stack Frame Layout

NEATCC uses the following stack frame layout for function. Note that, some of these sections may be omitted for functions that do not require them.

```
[ STACK ARGUMENTS              ]
[ SAVED REGISTER ARGUMENTS     ]
[ THE PREVIOUS VALUE OF IP     ]
[ THE PREVIOUS VALUE OF FP     ]  <- FP points here
[ LOCAL VARIABLES              ]
[ COMPILER TEMPORARIES         ]
[ SAVED REGISTERS              ]
[ FUNCTION ARGUMENTS           ]
[ NEXT FRAME                   ]  <- SP points here
```

Final Code Generation

The functions whose names begin with "i_" are the low-level architecture-specific code generation entry points. For each output architecture, a header (e.g., x64.h) is included and these entry points are implemented in a C file (e.g., in x64.c).

The function **`i_reg(op, md, m1, m2, m3, mt)`** returns the mask of allowed registers for each operand of an instruction. The first argument op, specifies the instruction (O_* macros); i_reg() sets the value md, m1, m2, and m3 to indicate the mask of acceptable registers for the destination, first, second, and third operands of the instruction. For immediates, the corresponding argument indicates the bit width of the operand (e.g., 8 means the operand is encoded in 8 bits). The value of these masks may be changed to zero to indicate fewer than three operands. If md is zero and m1 nonzero, the destination register should be equal to the first register, as is common in some CISC instructions. mt denotes the mask of registers that may lose their contents after the instruction. The function i_ins() generates code for the given instruction. The arguments indicate the instruction and its operands.

Some macros should be defined in architecture-dependent headers and a few variables should be defined for each architecture, such as tmpregs, which is an array of register numbers that can be used for holding temporaries and argregs, which is an array of register numbers for holding the first N_ARGS arguments. Consult x64.h, as an example for the macros defined for each architecture.

Compiling NEATCC

The neatcc_make GIT repository, includes a makefile to obtain and build neatcc, neatld, and neatlibc (and a few other programs). To do so, use the following commands:

```
$ git clone git://repo.or.cz/neatcc_make.git
$ cd neatcc_make
$ make init          # fetches the required programs
$ make neat          # compiles the programs using the host compiler
$ make boot          # compiles neatcc using itself
$ cd demo && make    # to make sure it works
```

The output architecture is x86-64 by default. To compile for other architectures, the value of OUT Makefile variable can be changed. For instance, the following commands build and bootstrap neatcc for ARM32:

```
$ make OUT=arm neat
```

```
$ make OUT=arm boot
```

After compilation, the neatcc excutable in neatrun directory can be invoked as a C compiler. It executes the linker or the compiler based on the presence of -c option.