XR/station Project
TOWER Compiler Design
Revision 1.0, July 24, 2023
Revision 1.1, June 4, 2024

TOWER Compiler Design

1. Introduction

This document contains an overview of the design for the self-hosting Tower compiler. This compiler will be used by the XR/station project for many years to come, and thus its design should be flexible enough to allow the iterative implementation of any new requirements that arise.

The fact that Tower will be the implementation language for the MINTIA operating system (from version 0.2 onwards), as well as that for the entire XR/station toolchain, highlights the importance of this compiler's design.

2. Requirements

The design and implementation of the Tower compiler must accommodate the following requirements, in no particular order:

- Self-hosting. The Tower compiler will be written in the Tower language. To this end, a bootstrap transpiler to C has already been written.
- Efficiency. Since the Tower compiler is intended to be self-hosting on RISC systems of 16-25MHz and 4-16MB of memory, it will use time and space-efficient data structures and algorithms for its tasks.
- Portability. The Tower compiler will be easy to run on new platforms.
- Retargetability. The Tower compiler will be easy to add new target architectures to. The first supported architecture will be XR/17032. A single build of the compiler will be able to generate code for any supported architecture.
- Code quality. The Tower compiler will output code of reasonable quality with the aid of a simple optimizer, however a "sophisticated" optimizer is a non-goal (due to the limited time and manpower).
- Robustness. The Tower compiler will reliably generate diagnostic messages due to source text errors, and its implementation will avoid falling into strange states caused by unexpected user input.

An implementation of the Dhrystone 2.1 benchmark will be made in Tower to test the performance improvements of this compiler over the old Dragonfruit infrastructure. The Tower compiler will have the goal of reaching a 75% ratio of DMIPS to MHz (i.e. 18.75 DMIPS @ 25MHz), a great improvement over Dragonfruit which only reaches a ratio of 57% (14.25 DMIPS @ 25MHz). For reference, MIPS Computer Company's C compiler for the R3000, a chip comparable to the 17032, reaches a ratio of 88.8% (22.2 DMIPS @ 25MHz).[1]

[1] https://netlib.org/performance/html/dhrystone.data.col0.html

3. Organization

The overall organization of the Tower compiler will be traditional, comprising a completely target-independent "frontend" consisting of the lexer and parser, [what else?]

Additionally, there will be a "library" component consisting of host-independent runtime utility functions.

4. Lexer

The lexer's fundamental "engine" is the function LexGetCharacter, which returns the next character in the source stream.

For performance and flexibility reasons, preprocessing is done inline with lexing.

Additionally, two slashes in a row "//" are interpreted as a comment, and the rest of the line is skipped. This is repeated until it finds the next relevant source character, which is returned to the caller.

Regardless of the case, each time a newline character is encountered, the line number of the current text stream is incremented. When a source text stream's end is reached, which could either be an EOF in the case of a file stream or simply the end of the text in the case of macro contents, it is popped from the stack. In the case of a function macro, the old macro argument scope must be restored (so, that should probably be stored in the scope struct itself).

There must be some way for the preprocessor directive parser to skip ahead in the source stream to the next directive, in order to implement conditional compilation.

LexGetToken is the next layer of the lexer, which assembles and returns tokens, along with handling macro expansion. It must **NOT** have strange cases such as sometimes returning empty tokens in odd whitespace cases, as has plagued other lexers of the same descent (i.e. descent of the MCL lexer). This is the *good* lexer that will be used for a long time, not a half-baked goofy one. This can probably be avoided by treating all whitespace identically, which can be done here since whitespace literally does not matter above the LexGetCharacter layer.

As an aside, since this is important in a second, a function LexPutBackToken will be provided which places the given token on the top of a putback stack, with enough room for four full token structures. When the stack contains items, they are popped from the stack upon LexGetToken calls. This allows an arbitrary amount of lookahead, with the stack depth to be tweaked as

needed. There is also a LexMatchToken which helps with a more efficient one-token lookahead, by matching the token on the top of the putback stack with a token type ID and only consuming it if it matches, by decrementing the stack index. If the putback stack is empty when LexMatchToken is called, a token is read directly into the first entry and the stack index is incremented. Note that LexGetToken cannot recursively call LexMatchToken, because that would be a layering violation and would corrupt the putback stack.

When LexGetToken completes a token, it checks if it is a string. If so, it is returned immediately. If not, it checks if it is a number. If so, it is parsed and returned as a numerical token. If not, it sees if its raw text can be identified as a macro. If not, it checks if it can be identified as a keyword. If so, it is returned as a keyword token with the matching type number. If not, it is returned as an identifier token.

Macros are expanded by placing their text contents atop the stack of source streams.

5. Parser

Hand-written recursive descent …

6. Code Generator

The abstract syntax tree (AST) produced by the parser will be converted into a linear intermediate representation (IR). This will be what the optimizer operates on, and in fact will be directly used by the rest of the compiler for efficiency reasons; namely, it avoids converting back and forth between several representations.

During conversion to the IR, some basic constant folding will be done, blocks of code that obviously cannot be executed (due to a constant conditional that evaluates to zero) will be omitted, and blocks of code that will obviously be executed (due to a constant conditional that evaluates to non-zero) will be simplified for WHILE loops to infinite loops, and for IF statements to the inlining of the block contents into the “containing” basic block.

The basic blocks of the IR will contain relatively low level statements; i.e. the operands of the statements are always virtual registers or symbol names, and never expressions themselves; the basic blocks will be simple lists rather than forests of expression trees. Some statements, such as function calls, may have arbitrarily many operands.

The optimizer will support basic degrees of the following optimizations, some as a side-effect of other stages:

- Dead variable elimination.
- Constant propagation.
- Arithmetic simplification.
- …

7. Instruction Selection

Instruction selection will be performed by assembling the IR instructions into basic block DAGs with local value numbering. Local constant propagation will be opportunistically done at this time. These DAGs will then be matched against instruction patterns.

The output of instruction selection is low-level intermediate representation (LIR), intermediate representation but with the symbolic instructions replaced with a generic format for machine instructions. This generic format must support the semantics of usual RISC architectures, as well as fox32, i486, and m68k (basically, it should support anything you can think of).

8. Register Allocator

The design of the register allocator is a simple linear scan allocator.

After register allocation, prologues and epilogues will be added.

8. Code Emission

Code emission in the standard backends will be to xrsdk assembler code for the given architecture.