The Design and Implementation of the Yasm Assembler

Peter Johnson

April 22, 2010

The Design and Implementation of the Yasm Assembler
by Peter Johnson

Published 2005

Contents

List of Figures

Preface

Yasm, according to the classifications given by [Saloman92], is a one-pass macro meta-assembler. In essence this jumble of words means that yasm reads the source file once, supports macros, and can target many different instruction sets. However, yasm also falls outside the set of definitions given by Saloman: while it is a one-pass assembler in that it only reads the source file once, it performs many in-memory passes over the source *contents* during assembly, and usually performs a number of out-of-order passes as well.

Nearly all large assemblers prior to yasm were forced to make multiple passes over the source code due to memory limitations. Yasm was designed with the modern system in mind, in which the amount of memory available in the system is vastly greater than the size of an entire executable, and certainly greater than the size of a single object file.

The design of yasm thus relies on the fundamental assumption that the parse of the source file is only performed once, and the entire source file contents are available in memory for use by later stages of assembly.

Material Covered in this Book

This book is about the *internal* structure of the yasm assembler: how the core libyasm library, modules, and frontend interoperate and the algorithms used in the process of turning a source file into an object file. While there will be some mention of yasm's user interface, that is not the primary focus of this book.

As yasm started out as an x86 architecture, NASM syntax assembler, most examples in this book will use x86 instructions and the NASM assembler syntax. The concepts will generally apply to all architectures and syntaxes supported by yasm, but these provide a convenient point of reference.

Chapter 1

Goals

The goal of the yasm project is to write an assembler that is a more easily extensible version of NASM, ultimately allowing for machine architectures other than x86 and multiple assembler syntaxes (such as TASM and GAS in addition to NASM). In general, yasm tries to work like NASM, except where there's a compelling reason to be different. To allow these features, the yasm assembler is structured in a very modular fashion.

1.1 Key Internal Features

- A NASM syntax parser written in Yacc. This simplifies the source code and increases performance: Yacc-generated parsers are almost always faster than hand-written ones. Also, Yacc (and its GNU implementation, bison) is an extremely well-tested and well-documented tool.
- Architecture-specific instruction parsers hand-written for simplicity and size, as well as to make it easy to add additional architectures while retaining the same front-end syntax. The blend of Yacc for syntax and a hand-written parser for instructions strikes a great balance between the strengths and weaknesses of each approach.
- A NASM syntax lexer written in re2c. A highly efficient scanner generator (almost always faster than lex/flex), it's also very embeddable due to its code generation methodology, allowing a number of re2c scanners to be used in various places in yasm without any worries about naming conflicts.
- Many of the modular interfaces at least superficially finished. This is still an area that needs a lot of work.
- A small set of portable equivalents of useful functions that are standard on some systems (detected via configure), such as the queue(3) set of functions, strdup, strcasecmp, and mergesort.
- A decent (and growing) set of assembler test input files to test the entire assembler as well as specific modules.

Chapter 2

Architecture

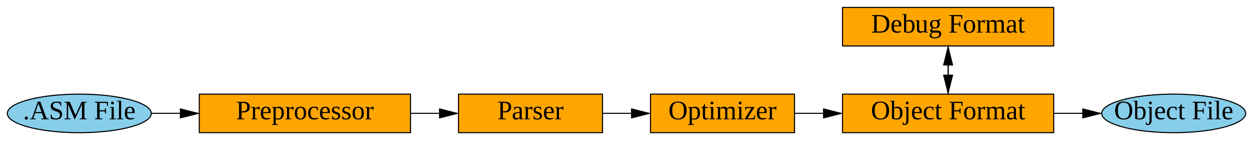
Yasm is conceptually divided into a number of separable modules.

Assuming that the programmatic interface of each module is well-defined, it is easy to customize the different parts of the assembler. Contributors can write new parsers, new preprocessors, new optimizers, and new object formats. Multiple types of each module may be simultaneously compiled in and are user-selectable at runtime via command-line or other configuration methods.

Figure 2.1 illustrates the 3-stage pipeline architecture of the assembler, and where each particular interface fits between the pipeline stages. The data passed between each stage is structured as a linked list of bytecodes (an internal representation of a machine instruction or assembler pseudo-instruction, see Section 3.1 for more information). Two additional stages are shown in Figure 2.1, the preprocessor and the debug format, which while being major components of the overall architecture, execute in parallel with a main pipeline stage.

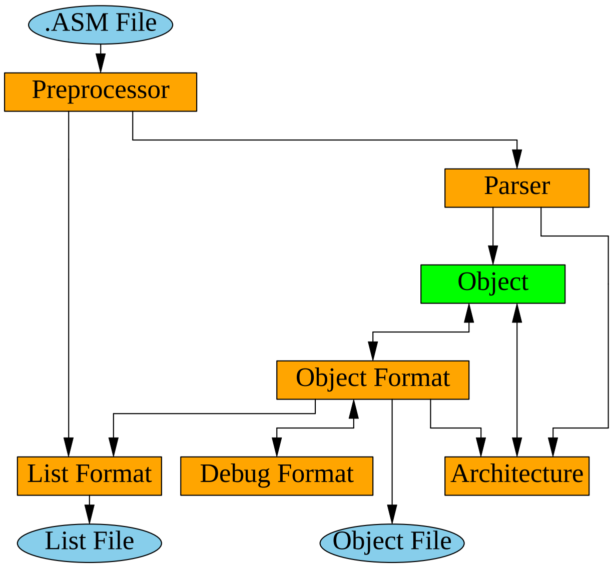
However, yasm isn't a pipeline in the traditional computer architecture sense of the word, as none of the stages execute in parallel.

Figure 2.1 Pipeline Architecture of Yasm



In addition to the main pipeline shown in Figure 2.1, a number of additional modules are necessary in order to have a fully functional assembler. Some of these modules are called from multiple stages and thus must be able to function under many different conditions. Figure 2.2 shows how the pipeline interfaces with these additional modules to create a complete assembler.

Figure 2.2 Modular Architecture of Yasm



A variety of support modules provide basic data structures and I/O functions. Many of these modules are called or used in some fashion by nearly every module shown in Figure 2.2. Figure 2.3 shows the relationships between these support modules.

Figure 2.3 Yasm Support Modules

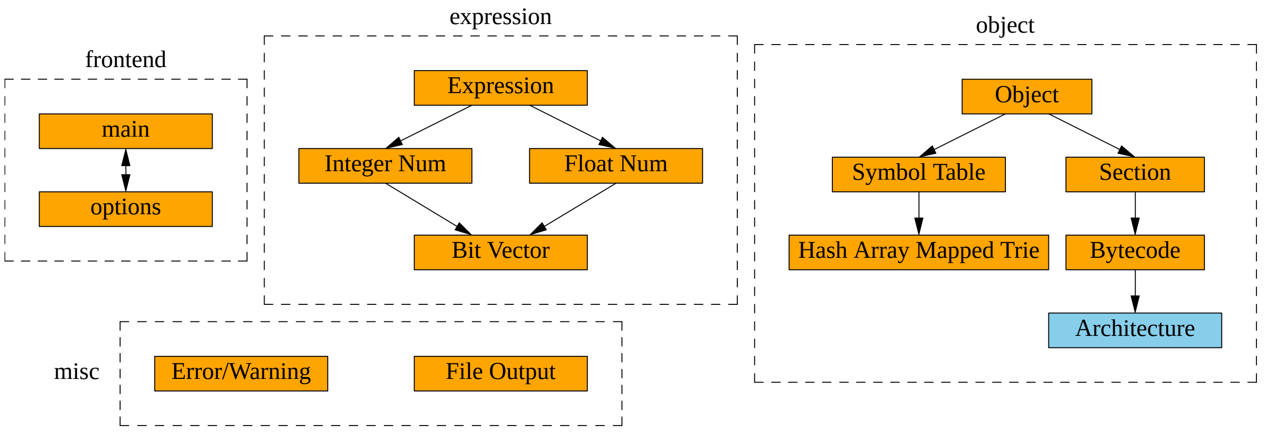
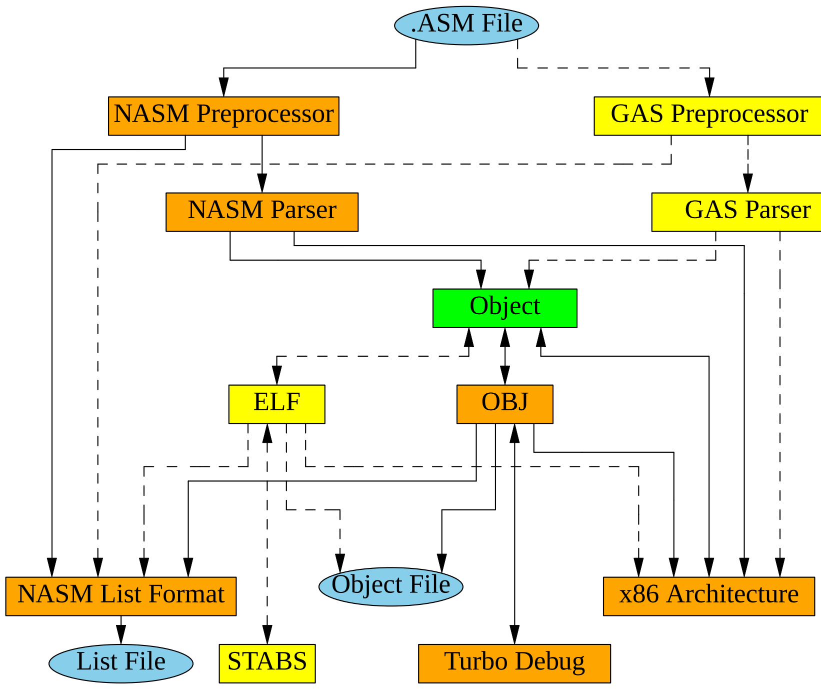


Figure 2.4 shows an example of a built-out assembler with multiple preprocessors, parsers, object formats, etc, and how all the pieces fit together.

Figure 2.4 Built-out Yasm (example)



Chapter 3

Data Structures

Yasm, like other assemblers and compilers, is at its heart just a data processing application. It transforms data from one form (ASCII source code) to another (binary object code). Thus, the data structures used to keep track of the internal state of the assembler are the most important things for a coder working on the assembler to understand. This chapter attempts to present reasoned explanations for the many decisions made while designing the most important data structures in the yasm assembler.

3.1 Bytecodes

The use of 'bytecodes' as the basic building block of the assembler was a fundamental requirement of both the goals (see Chapter 1) and the architecture (see Chapter 2) of yasm. A bytecode is essentially nothing more than a single machine instruction or assembler pseudo-instruction stored in an expanded format that keeps track of all the internal state information needed by the assembler to:

- Optimize the instruction size by resolving circular dependencies between instructions.
- Resolve labels that are used before they are defined.
- Detect error conditions such as undefined labels.
- Output symbolic debugging information and a list file.
- Calculate (and re-calculate) sizes and relative positions of instructions and data.

Most, if not all, other assemblers accomplish the above goals by re-parsing the source code in multiple passes. As yasm only parses the code once, bytecodes are needed to store all the information for every parsed instruction and pseudo-instruction. This fundamental difference is a trade-off choice between processing time and required memory space. The bytecode method requires that the entire source file's content must be stored in memory at one time (its content in terms of assembler state, not the actual ASCII source). To minimize the memory space that must be used, the yasm implementation tries to make the bytecode size as small as possible.

3.1.1 Bytecode Design Goals

- Instruction set independence: architecture-specific data is associated with each bytecode, but is not a part of the main bytecode data structure.
- Pseudo-instructions and instructions treated essentially the same. Data declarations and space reserving pseudo-instructions should be treated as a special case of architecture-specific data.
- Source code syntax independence. Commonalities between syntaxes should be utilized as to make non-parser stages as independent as possible of the original source.

3.1.2 Bytecode Data

To satisfy the above requirements, a good deal of data must be kept in the `bytecode` data structure. The main bytecode data structure contains such information as:

- The number of multiples of the bytecode. In some cases, it's desirable to have an instruction or set of data values repeated a number of times. Rather than storing multiple copies of the bytecode in memory, it saves memory to store the numerical multiple. In addition, the number of multiples may be a more complex expression, depending on the sizes and locations of other bytecodes.
- The virtual line number (TODO: add reference) describing the source location of the bytecode. Used for symbolic debugging output, list files, and error messages.
- The total calculated length of the bytecode's contents. While it may seem that this is a waste of space or a simple space/speed trade-off (instead of recalculating, we save the calculated value), this actually has a more complex reason for being present. See the discussion on optimization for details (TODO: add reference).
- The starting offset of the bytecode. Used to speed up relative offset lookups from expressions.

Additional data describing the actual contents of the bytecode is associated with the above data. There are two broad categories of this data: assembler pseudo-instructions, which are available on all architectures, and architecture-specific instructions. Data values are treated as pseudo-instructions.

3.1.2.1 Assembler Pseudo-Instructions

To be written.

3.2 Sections

Object files (and entire programs) are commonly divided into multiple segments or sections. Often these divisions are made based on what the data will be used for during execution of the program: code and various types of data. To save disk space, many object formats offer a special section that reserves memory space for data but does not actually store any data in the object file: UNIX systems usually refer to this section as the '`.bss`' section. An object file typically has many sections of various types, and may include special sections for debugging and symbol information.

In yasm, each section (or segment) described in the input file is stored into a `section` structure. The most important data kept in the `section` structure is a linked list of bytecodes describing the section's contents. Other per-section data is also stored, including:

- The starting address of the section contents. This determines the virtual starting offset of data within the section.
- The (unique) section name.
- Object format specific data. Object formats may define extensions to the standard section options. The gathered data is kept with the section it pertains to, in an object format defined data structure.

Chapter 4

Parsing

The assembly process begins with yasm calling the `do_parse` routine implemented by the parser module. The primary parameters given `do_parse` include the object to parse into, the preprocessor to use, and the initial input file and filename. All of the output of the parser goes into the object; this consists primarily of a list of sections.

To build the sections and their constituent bytecodes, the parser must call functions provided by the architecture module (to identify instructions and registers) and the object format module (to create sections, implement the global, extern, common directives as well as any other object format specific directives such as ORG). The parser is expected to obtain its text input from the preprocessor moudle.

Usually the parser is implemented in two distinct portions: a tokenizer that breaks the inputs into discrete chunks (tokens) such as identifiers, separators (such as comma and semicolon), and numbers, and a parser that looks for certain sequences of tokens to generate the output. See a compiler book such as [AhoSethiUllman96] for more details.

All parsers need to use the `parse_check_id` function provided by the architecture module to determine if a particular text identifier is an instruction or register. Some assembler syntaxes may be able to infer this by usage, but it is still necessary for them to call `parse_check_id` in order to validate this assumption and to obtain the information required by the architecture to later recognize the instruction or register. The architecture is required to pass all information needed to uniquely identify an instruction or register through the 4-word data parameter passed to `parse_check_id`. The parser in return is expected to save this information in the standard insn bytecode or its operands.

In order to separate the parser module from in-depth parsing of an instruction and its operands, the parser is expected to form insn bytecodes that contain the arch data for the parsed instruction (from `parse_check_id`) and a list of yasm_insn_operand structures. Each operand may be designated as a register (in which case the arch data from the register's `parse_check_id` call is required), an immediate value or expression (as a yasm_expr), or a memory location (as a yasm_effaddr).

Chapter 5

Coding Style

5.1 Portability

For maximum portability, all code should conform to ANSI/ISO C89 (C99 is not yet well-supported enough to be portable). Prototypes should always be provided for all functions (we don't attempt to be friendly to pre-ANSI C compilers; there are just too many unknowns).

Adding small platform-specific features (such as using `abort` instead of `exit` on fatal errors) is allowed, but the functionality should be checked for using **autoconf**. Major platform-specific features should be avoided unless the functionality can be emulated on all platforms. Path and directory searches for header files fall into this latter category.

Functions which are standard in some environments but not strictly ANSI C (such as `strdup` and `strcasecmp`) should be checked for using **autoconf** and functionally identical replacements included in the package.

A small set of allocation replacements are provided that do internal allocation failure (NULL return) checks. These are named identically to the functions they replace, with a prepended 'x'. E.g. `xstrdup`, `xmalloc`, etc. These functions should be used instead of the corresponding standard C library functions in order to maintain standardized error checking and messages for all memory allocation.

5.2 Formatting and Indentation

An indentation level of 4 spaces and 8 spaces to a tab should be maintained. However, at the present time no other specific style rules are required. While any formatting style can be used, it is highly recommended that new code match existing code, especially when modifying existing files.

Chapter 6

Bibliography

[AhoSethiUllman96] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools*, Copyright © 1996 Bell Telephone Laboratories, Inc., 0-201-10088-6, Addison-Wesley Publishing Company.

[Levine00] John R. Levine, *Linkers and Loaders*, Copyright © 2000 Academic Press, 1-55860-496-0, Morgan Kaufmann Publishers.

[Saloman92] David Saloman, *Assemblers and Loaders*, Copyright © 1992 Ellis Horwood Limited, 0-13-052564-2, Ellis Horwood Limited.

Chapter 7

Glossary

Bytecode

The fundamental unit of yasm assembly. Usually represents a single machine instruction or assembler psuedo-instruction, stored in an expanded format that includes assembler state information such as length, source line number, etc. It can store duplicates in a compressed format by storing only the number of multiples.

Section

A contiguous memory area. Represented in yasm as a single list of bytecodes, a list of relocations, and a number of section-level flags. Generally a yasm section maps directly into a object file section.

Segment

See "Section".