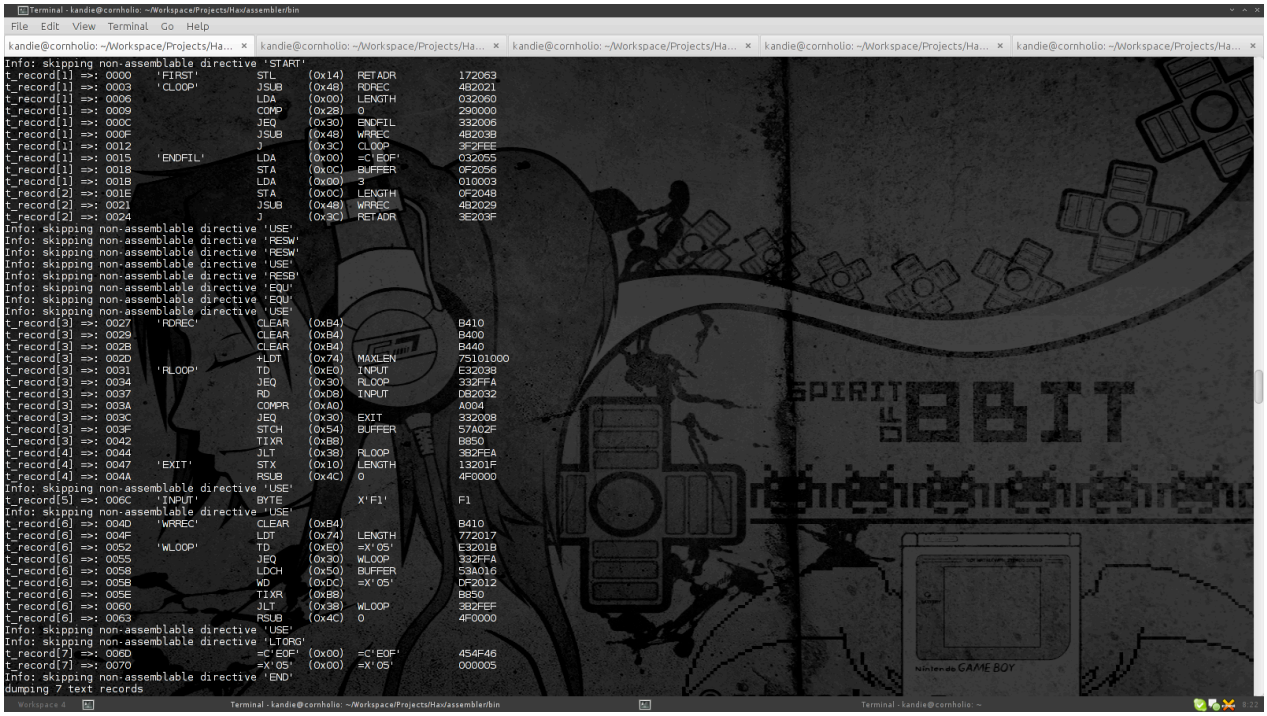
SYSTEMS PROGRAMMING

Hax ASM - a SIC/XE x86_64 assembler



Author: **Ahmad Amireh**

Supervisor: **Dr. Sami Al-Sarhan**, *Ph.D*

Computer Science dept., Fall 2011

University of Jordan

Preface

Material in this document is part of a series of assignments required by a Computer Science course, Systems Programming. The objective of said assignments is to design and program a primitive structural programming language, a compiler, an assembler, a loader, and finally a virtual machine to emulate it on the Intel x86_64 architecture.

To fulfill this objective, I will be utilizing the knowledge gained in Discrete Mathematics, Compiler Design & Construction, and Systems Programming courses.

This part is concerned with the assembler of the above-noted pipeline, the Hax assembler.

HAX

The Hax Language

For the purposes of this document, the following assumptions are made:

1. Hax is a strongly-typed language that has built-in support for two basic datatypes: integers and characters
2. The Hax compiler produces SIC/XE compatible assembly programs
3. Target machine architecture will be emulated by a VM that represents the hypothetical SIC/XE architecture

The Hax Assembler

Overview

The Hax Assembler (HASM) translates SIC/XE assembly input (.hasm) file into a SIC/XE compatible object program (.obj) which can be loaded by the Hax loader.

Specification

HASM is a **two-pass** assembler. The assembler accepts and produce only relocatable programs, and as such, any starting addresses defined will be silently ignored. The assembler is expected to provide support for the following:

- SIC Format 1 and Format 2 instructions
- SIC/XE Format 3 and Format 4
- Literals and literal pools
- Program Blocks and Control Sections
- SIC/XE Register-to-Register instructions
- SIC/XE addressing modes:
    - Immediate addressing (in the extended Format 4 version)
    - Indirect addressing (Format 3-only)
    - Base-Relative indexed addressing
    - PC-relative addressing
- Expressions (and nested expressions)
- User-defined symbols

Technical Overview

The assembler was written from the ground-up using C++. The decision of following an object-oriented approach to design the assembler as opposed to the classic structural one is an arbitrary one; while it does increase the complexity and has a slight impact on performance, it improves the readability and maintainability of the code.

The code makes heavy use of the STL, exceptions, and of the latest standards, C++0x.

Pass 1, parsing the input file

During the first pass, HASM scans the input file for entries, and begins preparing the objects required for processing by Pass 2. The assembler expects the input to begin with a definition of a control section, either in the form of a START operation, or a CSECT one. Once encountered, a control section with the given name (or a default-assigned one) is constructed with a default program block.

From that point on, every entry parsed is pre-processed to determine its type; from a top-view perspective, instructions could be either assembler directives, which are usually not assembled, or one of the supported formats. The type of the instruction is deduced from the opcode specified in the entry; as per the SIC/XE specification of operations, each one supports a particular subset of formats.

When an instruction is constructed, the parser proceeds to scanning for operands. Operands are created in a similar fashion to instructions; the type of operand is deduced from the content of the operand field in the entry. An operand could be a constant, a literal, an expression, or a symbol.

Once an instruction is built, attached to the current program block, its operand scanned and assigned, it is given the chance to "preprocess" - a routine in which they can usually calculate their length. Right after that, it is assigned an address based on the location counter of that block and the counter is stepped forward.

Reaching the end of the input file, the parser hands the control over to each control section, requesting them to *assemble* in turn, and the second pass is then engaged.

Pass 2, writing object program

The current control section begins its assembly routine by dumping the literals declared in the literal pool into the back of its instruction list. Afterwards, each instruction in the set is asked to assemble itself by calculating its object code.

An instruction's object code depends mainly on its operand's value, which could vary greatly between the types of operands. However, thanks to the power of abstraction, all the instruction knows is that it has an operand which has a value, so it asks the operand to evaluate and then stores the result. That value is then modified based on the type of the instruction; for example, an instruction of format 3 with the simple addressing mode might have to calculate whether it should use PC or base-relative mode.

Instructions are responsible for determining whether they require relocation, based on their operands. If they do, they track every symbol that requires relocation.

With the instructions assembled, their object code produced and ready, the control section calls the *serializer* to perform the actual object program writing. The serializer constructs

H,E,R,D,T, and M records based on the instructions contained in every program block of the current control section and writes them to the output file.

Building

The assembler was developed under a Linux environment using GCC 4.6.2 as a compiler. Since the code uses the C++0x standard, a compatible compiler is required to build the code. Any version of GCC later than 4.5 is compatible, however, as of the time of writing neither MSVC 2011 nor Apple's LLVM 4.2 yet fully support the latest standards and thus might not be able to compile the software.

CMake 2.8 is used to build the source, and the source is maintained in a git SCM repository. It is recommended to check out the code from the online git repository instead of the version found on the media provided with this document as it will be updated. To clone the repository, install git and run the following in a terminal:

```
git clone git@amireh.net:~/hax/hasm.git
```

Then from the same terminal session, we have to navigate to the source directory, run CMake, and build the project:

```
cd /path/to/hasm/
mkdir build && cd build/
cmake ..
make -j3
```

If you would like to configure the build paramters, run ccmake instead of cmake in line 3 above. Hopefully, the above will be sufficient to build the code and you should receive no warnings or errors.

The Hax assembler was written and tested under Arch Linux on an Intel x86_64 architecture.

Usage

A few test programs are provided with the source code, found under hasm/text/fixture/. These fixtures can be used for testing. The general usage format is:

```
./hasm input_file
```

This will process a file named input_file and write the resulting object program to a.obj if it succeeded. To test using a provided program, run in a terminal:

```
cd /path/to/hasm/bin
./hasm -d -o out_program.obj ../test/fixture/program_87.asm
```

The last command will run the assembler with program_87.asm as an input file, and it will write to out_program.obj using the delimited output format. To view the arguments the assembler provides, run:

```
./hasm --help
```

Limitations

1. There is no support as of yet for the ORG assembler directive
2. Expressions are not evaluated for being relative or absolute based on their terms, and thus the assembler will not recognize any such erroneous expressions and will instead ignorantly process them
3. Since HASM is a two-pass assembler, EQU operands need to consist of previously defined symbols. The same rule applies to expressions; all symbolic terms must be evaluated by the time pass 2 is engaged

Source Code

You can view the source code online on Github using the following link:

https://github.com/amireh/hasm

To clone the repository, please refer to the Building section above. The code can also be found on the CD provided with the soft-copy of the document.

References

1. System Software, An Introduction to Systems Programming 3rd Ed. (L. L. Beck)
2. SIC/XE Addressing Modes, link: http://cis.csuohio.edu/~jackie/cis335/sicxe_address.txt
3. SIC Simulator, link: http://www.unf.edu/~cwinton/html/cop3601/supplements/sicsim.doc.html

Credits

Dr. Sami Al-Sarhan for being such an inspiration for his students, and his time spent in carrying us through core computer science theories.