COMPILER FOR TINY REGISTER MACHINE

An Undergraduate Research Scholars Thesis

by

SIDIAN WU

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Riccardo Bettati

May 2015

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Design Embedded Operating System Based on Tiny Register Machine. (May 2015)

Sidian Wu
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Riccardo Bettati
Department of Computer Science and Engineering

The remarkable development of embedded computing devices in the past few decades has greatly improved the computing ability of our embedded and mobile devices. Mobile phones, for example, have developed from simple telephony and messaging devices into integrated computation, visualization, and communication devices that interact with a variety of external sensors to respond to health signals, location information, video input, and others. This is typically achieved through a combination of a small number of general-purpose computing cores with support from a highly-parallel graphics unit. This thesis is part of an investigation to study the applicability for embedded applications of extremely simple configurable computing components particularly designed for mapping on field-programmable gate arrays (FPGAs). This research is focusing on investigating of the low-level software requirements of the so-called Tiny Register Machine (TRM). This thesis describes, step by step, the construction of an adaptable C compiler, to program the TRM. This will provide the foundation for developing an experimental operating system, test the performance of this operating system and explore the possibility of TRM to run complicated Operating Systems or applications later on.

CHAPTER I

INTRODUCTION

Increasingly, high-performance embedded computers – be they in smartphones, wearable devices, in health and entertainment applications, or others – play important roles in in any of field in people's daily life. The need for and the ability to flexibly design the processor units of such computers have grown significantly in the recent past. Therefore, it is important to further investigate and develop and understanding for the design space for embedded device architectures. One approach to the design of computation nodes for embedded systems is to combine traditional microprocessor design together with flexible deployment on field-programmable gate arrays (FPGAs). This approach would allow the embedded-system designer to leverage both well-proven software development techniques (e.g., use of high-level programming languages, toolchains, and development environments) and the ability to tune the architecture parameters (number of cores, cache size, interconnection network, etc.) to match the application at hand. The research in this thesis is an investigation of the low-level software requirements of the so-called Tiny Register machine, which is a processor architecture that, we predict, will lend itself for deployment in low-power embedded environments, in particular on FPGAs. The advent in the recent years of inexpensive development boards with sizeable and fast FPGA chips has made it possible for us to design and deploy embedded system architectures with a significant number (32 or more) of cores. Some research has gone into the design of minimal architectures that are suitable for deployment on FPGAs. One such architecture, the Tiny Register Machine [1], is a simple processor architecture optimized for implementation in FPGAs. TRM not only has all the characters and components a modern CPU has but has also been well-designed to be both simple and efficient enough for use in high-performance and low-power embedded applications. The final goal of our research is to build an event-driven operating system to build TRM based low-power embedded systems. The objective of this thesis is to develop a C compiler, which translates programs represented in the high-level language C into machine operations. The availability of such a compiler allows the system designer to develop code for the TRM in a familiar high-level programming language on a desktop and then use the compiler to generate machine code, which can finally be downloaded to and executed on the TRM.

Follow-up projects can then make use of this compiler to develop a software framework to host embedded applications on the TRM. For example, it becomes possible to port a minimal operating system, such as FreeRTOS [2] to the TRM. Based on the experiences gained with FreeRTOS, one could proceed to port the L4 micro-kernel (NEED A REFERENCE) to the TRM as well. This will enable people to later run a POSIX compliant OS on top of the TRM.

This thesis is organized as follows: In Chapter one, we lay out importance and the purpose we are researching on TRM. In Chapter two, we proceed to describe background knowledge and a blueprint of we will build up. In Chapter three, we introduce the specific methods we used during the experiment. Finally, we conclude in Chapter four and provide an outlook to future work.

CHAPTER II
BACKGROUND

Tiny Register Machine

Tiny Register Machine (TRM) is a straightforward RISC architecture implemented of a Field Programmable Gate Array (FPGA) [1] by Dr. Niklaus Wirth. TRM was initially designed for educational purpose. With the relentless effort and rigorous attitude, TRM has been developed as a perfect pattern for students to study microchips and the software on the top of it. The design of TRM is extremely neat and efficient aiming to provide a lightweight platform for experiment and education.

In the design of CPU, register is the storage unit that enables the CPU to perform calculation. The number and type of register is usually regarded as the most important specifications in the CPU architecture. Tiny Register Machine owns 16 32-bit registers. The local data memory consists of 2048 words of 32 bits. The local program memory consists of 4096 instructions with 18 bits. [3]

Its so-called RISC architecture stands for *Reduced Instruction Set Computing,* which is in comparison to Complex Instruction Set Computing. The latter approach is widely used in industrial level CPU. RISC architectures usually have extremely small and very uniform instruction sets, which allows for relatively simple implementations of the CPU. TRM contains only 16 instructions in total. Each instruction can be divided into 4 fields, which in turn represent operation, destination register address, initial register address, and auxiliary space. The auxiliary space could be either another address of register or a constant offset.

In order to handle asynchronous event, the TRM supports interrupt signals. An interrupt signal is emitted by either hardware or software running on the top of CPU, and it indicates that there is a need for CPU to interrupt the currently executing thread in order handle the urgent request from the source of interrupt. TRM reserves 3 interrupt signals *irq0- irq2*. Whenever an interrupt signal arrives at CPU, the latter stores the current data from registers into memory and gets ready to handle the new event. When finished, it restores the original data from memory and continues executing from where it left off.

In this project, we use an existing Verilog [4] implementation on an FPGA development board of a single-core TRM CPU. Verilog is a widely used hardware description language that could help people build up and modify circuits on the top of FPGA board.

Computer Language Syntax

Just like the languages people speak, computer languages follow their own grammars rules as well. Compilers read the source files and tell the programmers whether they have been well formed already or are still having errors based on those rules. One of the main notions to describe the syntax of computer languages is called Backus Normal Form (BNF) [5].BNF formulates all elements involved in a language into four categories – Terminal symbols, nonterminal symbols, syntactic equations (production rule), and start symbol. "A language is, therefore, the set of sequences of terminal symbols which, starting with the start symbol, can be generated by repeated application of syntactic equations, this is substitutions" [6].

For instance, if we define a language that the start symbol are A and B; terminal symbol are a, b, c, and d; and the syntactic equations are A = a or b and B = c or d. Then the sentence “AB” can be translated into four forms – “ac”, “ad”, “bc”, and “bd”.

In our experiment, we focus on one of the special syntax form called Context Free Grammar (CFG) [7]. CFG language starts with a single nonterminal symbol V. The production translates V into a string of terminal symbols or nonterminal symbols (recursive). However, its production rules can be applied regardless of the context of a nonterminal symbol. For instance, start symbol is S, terminal symbol is a, production is S = aSa follows context free grammar.

The reason we are interested in digging into on the syntax of computer languages and CFG is they play significant role while constructing computer languages and compilers, on the other hand, parsing each sentence is based on those rules. One of the main algorithms we used is called recursive descent parsing, which is a simply and efficient way to translate CFG format input into the TRM opcode eventually.

Compiler

The compiler is a system program that translates a program represented in human-readable computer languages such as C, C++, and Java into computer-readable binary code. Since different CPUs have different instruction sets, different compilers are needed to translate programs to CPU instructions. Whenever a CPU with a new or modified instruction set comes

out, the compiler is usually the first program that needs to be built. In this project, we develop an experimental compiler for the TRM instruction set.
We select the C Language [8] as the high-level language for which to build the compiler. This for three reasons: First, the C Language is relatively simple and allows for compact compilers; second, second, C is well suited to develop low-level system software such as event-handling code or even simple operating systems; finally, there are many existing implementations of compilers for the C Language that have been ported to a variety of architectures. It is therefore comparatively simple to either develop a compiler for a subset of C from scratch or to port an existing compiler to the TRM.

Due to the significant complexity of today's compilers and due to the need to develop what amounts to unique compilers for each CPU architecture, the design of modern compilers is that of multiple interrelated components. The whole compiler works like a pipeline where the output of each component becomes the input for next component. By convention, a standard compiler has the following components – lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, machine independent code improver, target code generator, and machine dependent code optimizer.

We call the first three steps the *front end* of the compiler. Lexical analysis reads through the input text files, removes unnecessary elements such as white space and comments, and converts files into a stream of tokens. Syntax analysis processes the set of tokens and turns them into a more complex data structure called abstract syntax tree. The syntax analysis not only parser the source files into easier-processing format for following components but also is in charge of looking for syntax errors in the files. In the end, sematic analyzer checks legality rules of the language.

Conventionally, the output stream from sematic analyzer goes into intermediate code generation. Compared to real computer language, intermediate code has fewer but more restricted syntax and sematic rules, which is easier for back-end program translating later. It also optimizes the codes in favor of improving the CPU performance. Optimization varies from replacing duplicated variables to syntax simplification. The codes then come to back-end processor and get ready for execution.

The *back-end* of the compiler provides tool to generate real target code. It usually starts from transferring intermediate code to target assemble code. The independent machine code usually indicates an assemble code that assume the hardware can provide infinite resources when needed. Those codes later on will be optimized based on the hardware condition. Once it is done, the compiler will goes through the code once more and generate machine dependent codes. These are binary executable files in the target machine.

As a result, the compiler architecture is hour-glass shaped, where well-defined interface around the intermediate representation allow a variety of front ends (for multiple languages) to be combined with a multitude of back ends (for different CPU architectures).

As a result, programmers need to only rewrite portion of an existing compiler and reuse the rest to create a new compiler. For instance, Pascal and C could share the same back-end compiler while having their own separate parsers. Two compilers running on ARM and Intel Pentium can have the same front-end parser and different code generators. In this project, we focus on the back-end compiler construction. We choose C as it is the most common computer language for designing Operating System. It is also one of the most efficient computer languages as well being very popular among users.

Since compiler is one of the most complicated application and various kinds of computer science knowledge is involved in it, we started with looking on the background of the compiler design first. *Compiler Construction* written by Dr. Wirth and *Compilers: Principles, Techniques, and Tools* written by Alfred Aho, etc give us huge help and inspiration at the beginning and let us have a big picture of the design of compiler.

After we had a general picture in my mind, we started to investigate a open source compiler called Tiny C Compiler (TCC) [9] to get a deeper view of compiler. When we look around open source compilers, we found it was different from the rest of compilers. An industrial-class compiler such as G++ or Visual Studio has over one million lines of codes. However, when we lift the veil of TCC, we realized there are only less than fifty thousand lines of codes. Although TCC is only made by author's interest for personal use, it does realize all fundamental C features, and even extends a few advanced functions. We also noticed that TCC could run faster than most of current compilers easily. Apart from astonishing the author's coding ability, we wonder how the author could manage to realize these fashions. We started to read the author's code and understand his purpose with the fear that we would not be able to fully understand and modify the codes in the end. Later on, we figured out that TCC is very special and does not have general compiler components like anyone else.

We later on started to look for suitable parser and found Pycparser [10] is a good pattern for our project. Pycparser is a front-end C language parser Pycparser consists three useful components that aim to do lexical, syntax and semantic analysis. It also provides a set of handle APIs that help users call those features. Eventually, Pycparser translates the C source files into one of the intermediate code called Abstract Syntax Tree (AST) (reference). We extent Pycaparser and add the back-end opcode generator for it. The back-end code generator retrieves each node in AST and translates each syntax structure into opcode.

CHAPTER III

COMPILER DESIGN

Symbol Table

Symbols are variables and functions in computer languages defined by programmers. In compiler design, we implement a symbol table to mark every symbols when it occurs in the source code. We store name, type, value (if it is a constant), memory address and associated register of each symbol when they are defined. When the program later on calls the symbols, the compiler searches the symbol table and acquires the necessary information needed in order to generate opcode.

Symbol table is one of the most important components that every compiler would have to implement. There are different ways to construct a symbol table. In our experiment, we used both linked list and stack data structure to implement symbol table. Due to each variable has its own scope in C language; we store variables in the different scopes in individual symbol tables. Each symbol table is a stack as well as a node in the linked list. When the code goes into a new scope, we create a new symbol table and push all variables occur into this new table. We pop the whole table out when the scope is end. The first symbol table stores global variable and function declarations and associates with a data section in the memory.

Memory Layout

When a program is parsed into opcode, the code is organized in a typical order in the memory. Instructions that play similar functions are put in the same blocks called sections. A C program usually has at least three sections – text sections, data sections, and stack section. Text section contains compiled opcode. Data section store the global variables and functions' memory address. Stack section is used to store local variables when functions are called. The more complex C program may also have heap section as well as the section for command-line arguments and environment variables.

In our experiment, we only implemented the text, data, and stack sections. We leave the rest of them in the future optimization. When program is running, we defined three special registers called program counter (PC), frame pointer (FP), and stack pointer (SP). The PC contains the address of the instruction currently being executed. PC is set to the beginning of text section and will have one increment when the current instruction is done. FP and SP are originally set to the top of the stack section. SP increases when new local variables are pushed into the stack while FP indicates the head of the stack.

Recursive Descent Parsing

Abstract syntax tree is a tree representation widely used in the design of compiler. Each node denotes an elements occurring in the source code while the children nodes denotes the relationship between different nodes. In our project, we read C source codes from the users and construct an abstract syntax tree with the help of Pycparser. Since the Astract syntax tree follows the rules defined in context-free grammars, recursive descent parsing is applied here to translate abstract syntax tree.

“Recursive descent paring is a top-down process for a set of mutually recursive productions in which the parser attempts to verify that the syntax of the input stream is correct as it is read from left to right.” [11]

In our project, we denoted those main methods written in Python below to implement recursive descent paring procedures.

fileAST: parse the root node in the abstract syntax tree. Each parsing starts from FileAST. There are three possible children nodes may occur – decl, funcDecl, and funcDef.

decl/ funcDecl: is called when the program declares a new variable/function. It stores the variable into symbol table.

funcDef: is called when the program define a new function. Generating function definition is one of the most important and complex produce when parsing. There are several things need to be done here and we implement them in individual sub-retinues.

prolog: is called when function is defined. ‘prolog’ create new symbol table for local variables in the function and push the function arguments into it. In addition of all arguments, ‘prolog’ creates two special symbols in order to store FP value and return values. Since the local variables have their own scopes, when the program goes to a new sub-routine (function), it stores the current FP value into memory and move increase the FP to the tail of the stack (this is done in the method funcCall).

block: is called during the function definition. ‘block’ goes to each lines in the function definitions and keep recursive descent parsing. It generates opcode for the main segment of a function and store then into text section.

epilog: is called when the function definition is finished. ‘epilog’ restores FP to the previous address stored in the stack and pass the return value to the previous routine. It then pops the all variables in the current scope and set the SP to the tail of previous stack.

funcCall: is called when functions are called. It sets up the FP and SP values and jumps SP to the address in the text section where the functions begin.

CHAPTER IV

CONCLUSION

Compiler is one of the most significant and fundamental software that each CPU architecture would have to use. It is the cornerstone of other software. During almost one year's experiment, we not only learned of how to build up our own compiler, but also had a deeper understanding of the relationship between hardware and software. Although the difficulty is beyond our expectation and we had to changed our original plan and look for new approaches sometimes, it is very fun and educational. During the experiment, we came up with the questions and problems that we would not had if only reading textbook. It is also a huge jump from understanding what it is said in book to handcrafting our own weapon.

However, due to limitation of our knowledge and time, we also leave several checkpoints that we would like to improve or extend our project in the later experiment. They come from three places. 1. Although we have built our compiler, we did not consider any way to optimize the code generation. The compiler is also designed to parse only a subset of C language to opcode. It is not fully compatible with standard C. We would like to refactor our code later to make it fully compatible as well as efficient. 2. We would like to make use of compiler and build an advanced operating system on the top TRM such as FreeRTOS. This will enable people to later run a POSIX compliant OS on top of the TRM. 3. The flexibility of FPGA gives us the chance to modify the hardware features as well. We would like to dig into the design of TRM and add more features to TRM. One possible improvement would be changed it to a multi-cores architecture.

TRM gave us much more fun and we hope we would have more time playing around it. Little chip, big world.

REFERENCES

[1] Niklaus Wirth (2010 Aug). *Experiments in Computer System Design.* Retrieved from http://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/ComputerSystemDesign.pdf

[2] Real Time Engineers Ltd. *FreeRTOS*. Retrieved from http://www.freertos.org/

[3] *FPGA.* Retrieved from http://en.wikipedia.org/wiki/Field-programmable_gate_array/

[4] Niklaus Wirth.*The Design of a RISC Architecture and its Implementation with an FPGA*

[5] *Verilog.* Retrieved from http://www.verilog.com/

[6] W.Floyd, (1963 June). *Syntactic Analysis and Operator Precedence*

[7] Niklaus Wirth. (1996). *Compiler Construction.* Retrieved from http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf

[8] Noam Chomsky (1959). *On Certain Fromal Properties of Grammars*

[9] Dennis M. Ritchie. *The Development of the C Language.* Retrieved from http://cm.bell-labs.com/who/dmr/chist.html

[10] Fabrice Bellard. *Tiny C Compiler*. Retrieved from http://bellard.org/tcc/

[11] Eli Bendersky. *Pycparser.* Retrieved from https://github.com/eliben/pycparser

[12] *Recursive Descent Parsing.* Retrieved from http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html