

C-Like High Level Assembly

Joshua Thornton

Submitted for the degree of Bachelor of Engineering in the division of Software Engineering

November 2013

Mr Joshua Thornton

42334068

4/63-37 Bellevue Tce, St Lucia, 4067

4th November 2013

Prof Paul Strooper

Head of School

School of Information Technology and Electrical Engineering

University of Queensland

St Lucia, QLD, 4072

Dear Professor Strooper,

In accordance with the requirement of the Degree of Bachelor of Engineering in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled

**“C-Like High Level Assembly”**

The thesis was performed under the supervision of Professor Neil Bergmann. I declare that the work submitted in thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely

Mr Joshua Thornton

# Acknowledgments

I would like to thank Neil Bergmann for supervising my thesis, providing valuable assistance and direction with the project and assessment. I could not have completed this project without the support of my family and friends.

# Abstract

Assembly languages are the lowest level computer programming languages used by programmers to command computers. There are many assembly languages and each is specific to a particular computer architecture. An assembly language statement consists of a 2-4 letter mnemonic followed by zero or more operands. Often the meaning of an assembly statement is not immediately obvious to the programmer without consulting a reference, especially since the assembly languages are very inconsistent with one another. This project aims to simplify operation by introducing a single High Level Assembly Language with a consistent C-like syntax across multiple architectures.

The C-Like High Level Assembly Language (CHLA) makes use of C-like syntax so that it is easier to read. The language is highly consistent and much of the syntax is identical across multiple assembly languages, and thus, it is intuitive to write after learning a few simple constructs.

This project has produced an assembler and disassembler that understand CHLA as well as two specification documents. The specification documents describe the translation between an assembly architecture and CLHA. An example test program was developed to show that a CHLA program can be portable between more than one computer architecture.

**Contents**

Acknowledgments v

Abstract vii

Chapter 1: Introduction 17

1.1 Introduction and Motivation 17

1.2 Contents of Report 18

Chapter 2: Background 19

2.1 Early Assembly Languages and Assemblers 19

2.2 High Level Languages 20

2.3 Modern Assembly Languages 22

2.4 High Level Languages vs. Assembly Languages 24

2.5 High Level Assembly 25

2.6 Regular Expressions 27

2.7 Scanning 27

2.8 Decision Tree 28

2.9 Multi-pass Assembler 28

2.10 Intel Hex Format 29

2.11 SREC Format 30

2.12 CSV Format 31

2.13 AVR ATMega64 32

2.14 Freescale HCS08 33

Chapter 3: Research Plan 34

3.1 Project Aims 34

3.2 Requirements 35

Chapter 4: Design Overview 38

4.1 Initial Design 38

4.2 Specification Document Centric Design 39

Chapter 5: Specification Document Design 41

5.1 Overview 41

5.2 Specification Document Format 43

5.3 Operands 44

5.4 Equivalent Statements 46

5.5 No Ambiguity and Generality 47

Chapter 6: Implementation 50

6.1 Overview 50

6.2 Pre-processing 51

6.3 Tokenisation 52

6.4 Decision Tree and Regular Expression Generation 55

6.5 Decision-Tree Matching 58

6.6 Addresses, Labels and Jumps 59

6.7 Opcode Matching 60

6.8 Output 61

Chapter 7: Testing and Analysis 63

7.1 Test Programs 63

7.2 Correctness 64

7.3 Consistency 66

7.4 Reuse 67

Chapter 8: Evaluation 69

8.1 Limitations 69

8.2 Non-Mathematical Instructions 69

8.3 Extensibility 70

8.4 Readability 71

8.5 Use as a Learning Tool 72

Chapter 9: Conclusions 73

9.1 Summary 73

9.2 Future Work 73

References 74

Appendix A: Code Listing 76

A.1 Keywords Class 76

A.2 Operators Class 77

A.3 Assembler Class 78

Appendix B: Miscellaneous 83

B.1 ATMega64 Exhaustive Testing Diff Output 83

B.2 Tokenisation Regular Expression 90

Appendix C: Test Programs 91

C.1 ATMega64 Exhaustive Test Program Hex File 91

C.2 HCS08 Exhaustive Test Program SREC File 92

C.3 Cross-Architecture CHLA Test Program 93

**List of Figures**

Figure I: IBM 704 Mainframe Series “ADD” Opcode 17

Figure II Backus' FORTRAN example program, which compiles to over 70 machine instructions. 19

Figure III: An example assembly instruction constisting of a mnemonic, two operands and a comment. 21

Figure IV Intel Hex Record Format 27

Figure V Motorola S-Record Format [17] 28

Figure VI Assembler and Disassembler Design Overview 37

Figure VII Example Extract from Specification Document 41

Figure VIII Specification Document Field Format 41

Figure IX Specification Document Operand Example 43

Figure X Specification Document Equivalent Statements 44

Figure XI Tokeniser Workflow 53

Figure XII StatementTemplate generation 54

Figure XIII Decision-Tree Callback 54

Figure XIV Regular Expression Dictionary Callback 54

Figure XV Decision Tree Extract 55

Figure XVI Decision-Tree matching in the Assembler 56

Figure XVII Checking value lies within legal range inside Operand’s match method 57

Figure XVIII Disassembler handling variable length opcodes 59

Figure XIX Statement’s asm\_string method 60

Figure XX Statement’s chla\_string method 60

Figure XXI Exhaustive Testing Data Representation Mismatch Example 63

Figure XXII Exhaustive Testing Special Naming Mismatch Example 63

Figure XXIII Logically equivalent statements mismatch 64

**List of Tables**

Table 1 Intel Hex Record Types 28

Table 2 Field Composition of an S-Record [17] 29

Table 3 Motorola S-Record Record Types 29

Table 4 CHLA Specification Patterns 40

Table 5 Basic CHLA Keywords 41

Table 6 Generality of CHLA Assignment Statements 46

Table 7 Description of Pre-Processor Directives 49

Table 8 Example Specification Extract 55

# Introduction

## Introduction and Motivation

Assembly languages allow a programmer to take full advantage of the hardware and write highly optimised and informed code. However, there are many assembly languages and each is specific to a particular computer architecture. An assembly language statement consists of a 2-4 letter mnemonic followed by zero or more operands. Often the meaning of an assembly statement is not immediately obvious to the programmer without consulting a reference, especially since the assembly languages are very inconsistent with one another.

High Level Languages provide a number of convenient abstractions from the hardware reality that reduce the cognitive load of the programmer and the verbosity of the program. However, high level languages introduce the *abstraction penalty* which occurs when high level paradigms and objects obscure and mask the reality of the target machine paradigms and objects. In high level programming languages the mapping from virtual to physical paradigms and objects is ambiguous and thus, the programmer loses complete control over the program and cannot take full advantage of hardware features.

This thesis investigates a C-Like High Level Assembly (CHLA) language which introduces a more readable and consistent syntax whilst not introducing any of the translation ambiguity symptomatic of the *abstraction penalty*. The goal is to make assembly programming easier without sacrificing full control and power over the hardware.

## Contents of Report

<contents here>

# Background

## Early Assembly Languages and Assemblers



Figure : IBM 704 Mainframe Series “ADD” Opcode

Before the EDSAC (Electronic Delay Storage Automatic Calculator)

was developed in 1949, all computers were programmed directly using machine opcodes [1]. Programming directly in machine opcodes was very time consuming and error prone. Programmers were required to remember numeric codes and manually calculate addresses and constants [2].

With the EDSAC, Maurice Wilkes and W. Renwick introduced the first assembler called *initial orders* (Wilkes used the word ‘orders’ for what we call instructions) featuring one-letter mnemonics. Wilkes was also the first to propose the use of labels (which he called floating addresses), the first to use macros (which he called synthetic orders) and the first to develop a subroutine library [2].

The SOAP (Symbolic Optimal Assembly Program) assembly language for the IBM 650 mainframe was introduced in 1955 by Stan Poley [3]. The SOAP assembly language was the first assembly language to be very similar to present day assemblers. SOAP made use of labels to free programmers of the need to manually calculate addresses and jumps [2].

Assemblers and assembly languages were very commonly used from the 1950s through to the 1970s for all manner of programs. However, by the 1980s their use had largely been supplanted by high-level languages, except in highly optimised large projects such as operating systems [2].

## High Level Languages

The FORTRAN I Compiler was the first demonstration that it was possible to atomically generate machine code from high level languages [4]. FORTRAN was proposed by John W. Backus to his superiors at IBM in 1953, as a method to improve programmer productivity whilst maintaining program efficiency [5].

In Backus’ 1957 paper he states it is hard to quantify the “reduction of the [programmer’s] task’’, but details one anecdotal case where a programmer wrote a 47 statement program in four hours which compiled to over 1000 IBM 704 series instructions. The programmer estimates the FORTRAN language saved him three days of work [5].

Figure Backus' FORTRAN example program, which compiles to over 70 machine instructions.

Backus states that the ratio of the number of output machine instructions to input statements varied between 4 and 20. His 1957 paper also states that all programs produced by the FORTRAN compiler were not “appreciably longer” nor was there “an appreciable increase in execute time” than would have been achieved were the program produced by hand [5]. However, David Padua of the University of Illinois conducted analysis of THE FORTRAN I Compiler and found a number of inefficiencies that programming by hand would avoid, particularly concerning register reuse in loops [4].

A number of high level languages followed FORTRAN, a particularly noteworthy language is LISP (LISt Processor). John McCarthy of Massachusetts Institute of Technology published LISP in a paper entitled ‘Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I’ in 1960. LISP also targeted the IBM 704 Series but in contrast with FORTRAN, LISP was not designed to be as efficient as hand coded instructions [6].

LISP introduced a large number of features that are now a staple in many modern languages including automatic garbage collection, if-then-else conditionals, recursion, and dynamic typing [6]. These features traded machine code size and efficiency for ease and simplicity of programming [6]. Largely because of these efficiency tradeoffs LISP did not become as popular as other languages outside of the academic community.

The C Programming Language was developed between 1969 and 1973 by Dennis Ritchie. C is described by Ritchie as a “relatively low-level language” in that C deals with the same sort of objects that most computers do, namely characters, numbers and addresses [7]. Ritchie designed C to provide no operations to deal directly with composite objects such as character strings, sets, lists or arrays [7]. Nor does C provide any storage mechanism other than static allocation and the stack allocation system provided to local variables of functions, there is no heap or garbage collection [7]. Finally, C does not provide any input or output facilities [7].

The design decision to keep C very low level makes it ideal for operating system development and other critical programs where the program needs to be very tightly bound to the hardware. In “Operating System Concepts” Silberschatz states that most operating systems are implemented in C because C has high level language features, does not conceal the machine details and yet is relatively portable to other architectures [8].

## Modern Assembly Languages

All modern assemblers make use of short mnemonics to represent the CPU instructions [9]. Each single assembly instruction maps directly to one machine instruction. Typically, an assembly instruction consists of a two, three, or four letter mnemonic, followed by zero, one, or a pair of values or operands [9]. A good example is the ‘add’ instruction which is common to almost all instruction sets:

add v1, ADD v1, v2 # Add register v1 to v2 or v2 to v1

Figure : An example assembly instruction consisting of a mnemonic, two operands and a comment.

Whilst all modern assembly instructions may look similar, the syntax between them varies greatly. One of the most notable differences between architectures is the order of the operands. In Intel Syntax, the destination is placed before the source, in AT&T Syntax the source operand is placed first, followed by the destination operand [10].

All modern assemblers support directives and definitions. Directives are instructions to the assembler which are executed at assembly time (rather than execution time) to enable the program to be assembled in different ways based on parameters input by the programmer [9]. Definitions allow a programmer to re-label registers or values to make them more meaningful and thus assist in code readability [9].

Many modern assemblers have the ability to process macros (short for macroinstructions). A macro is a set of assembly instructions defined by the programmer with a name and optional parameters so that the set of instructions can be reused multiple times throughout the program. Macros are in contrast to assembly instructions in that they are not necessarily a one-to-one mapping to machine instructions [11]. M. M. Kessler discussed macros in detail in his “Implementation of Macros to Permit Structured Programming” in 1970. Kessler describes how macros benefit programmers giving the program more structure and allowing for greater abstraction, and thus increasing readability [11].

## High Level Languages vs. Assembly Languages

When discussing High Level Languages vs. Assembly Languages we are typically balancing the benefits of abstraction with the *abstraction penalty.* The abstraction penalty, is the execution and efficiency cost associated with translating programs as the programming language objects and paradigms get further away from the reality of the machine objects and paradigms [12]. Assembly languages have zero abstraction penalty as their assembly instructions map directly to machine instructions.

The benefits of abstraction are many; the same program can often run on vastly different architectures through recompilation or virtual machines, simpler control flow leads to fewer programmer mistakes (bugs), the program is often a fraction of the size and programmers can create huge complex systems by concentrating on a few issues at a time and not worrying about the underlying details. Each of these benefits can be summarised as increasing programmer output per unit time.

Randall Hyde covers the abstraction penalty extensively in his “The Art of Assembly Language” book. One example which illustrates the benefits of assembly very clearly is the ‘switch statement’. The switch statement is a method of picking one of many paths based on the value of a single variable. It can be implemented in machine code as a lookup table with the variable as a index, or as a series of conditionals or a hybrid of the each. A lookup table takes constant time, no matter how many branches there are, but if the possible ‘index’ values are sparsely distributed then there is a lot of wasted space. A series of conditionals takes linearly more execution time for each conditional and is suited to a limited number of branches [9]. Hyde argues that an assembly programmer can make a huge number of optimisations to ensure the switch statement runs optimally for their specific scenario. Conversely, the high level programmer has the implementation details hidden from them and thus, relies on the compiler to make decisions on their behalf. The compiler, Hyde contends, cannot be as efficient as the assembly programmer because it does not have as much information as the programmer [9].

Each side of the debate between High Level Languages and Low Level Languages has its argument. For projects where programmer time is the greatest constraint, higher level languages will be more beneficial. Assembly is more suited to those projects where CPU cycles and computer memory is constrained. Common uses of assembly are in operating systems, high detail graphics applications, the inner loops of complex algorithms and in critical applications such as those found in the aerospace and medical industries.

## High Level Assembly

There have a been a number of high level assembly languages since 1968, including Niklaus Wirth’s PL360, Microsoft’s Macro Assembly Language and IBM’s High Level Assembly Language [2]. Each of these languages addressed the idea that assembly languages could be made more readable without introducing the abstraction penalty.

Randall Hyde’s “The Art of Assembly Language”, the seminal book on modern assembly programming makes use of a High Level Assembly language designed by Hyde called HLA (High Level Assembly).

Hyde states that a High Level Assembly Language is characterised by a more familiar and human readable syntax, powerful macro processing and high level control structures. They must achieve this without introducing any ambiguity as to the translation to machine code so that the programmer can maintain full control over the machine code. Hyde argues that High Level Assembly Languages greatly reduces the cognitive load placed on assembly programmers.

It is worth noting that despite using a more generic and consistent syntax none of the High Level Assembly Languages listed were portable, that is, they all targeted a specific architecture. According to Randall Hyde in “The Art of Assembly Language”, the Intel x86 instruction set contains over 1000 instructions, yet most programs make use of less than 30 instructions [9]. Hyde explains that many small simple instructions can be combined to produce very complex programs. This is the principle behind RISC (Reduced Instruction Set Computing).

RISC was introduced in 1982 by David A. Patterson of the University of California, Berkely. Patterson’s paper, “A VLSI RISC” explored the “alternatives to the general trend toward architectural complexity” [13]. Patterson hypothesised that simpler instructions can provide higher performance if the simplicity enabled much faster execution of each instruction.

RISC designs have the following artificial constraints placed on them: execute one instruction per cycle, all instructions are the same size, and access memory with load and store instructions (the rest operate between registers) [13]. Almost all modern architectures make use of the RISC design strategy or a subset of the instruction set meets the RISC design constraints. The Intel x86 is an example of a CISC (Complex Instruction Set Computing) architecture with a subset that meets the RISC design constraints.

Hyde’s HLA does not cover every instruction in the x86 architecture, and thus he dedicates his later chapters to regular, non-high level assembly so his readers can leverage the full instruction set. However, he does not address the possibility of portable HLA code to other architectures using the frequently used subset of instructions.

## Regular Expressions

*“Regular Expressions represent patterns of strings of characters”*[14]. Regular expressions are formally defined by the set of strings that they match. The set of strings that match is called the ‘language’ of the regular expression. Each language consists of an ‘alphabet’ which is the legal set of symbols available in the alphabet [14].

Some symbols have special meanings and these are called ‘metasymbols’. A metasymbol is not a legal symbol and so a convention must be used to differentiate between the normal use of the symbol and the meta use of the symbol. This is usually by using a special ‘escape character’ which is used to *turn off* the special meaning of a symbol [14].

A regular expression can consist of many operations. Common operations include a choice among alternatives, concatenation, repetition, ranges of characters, and grouping [14]. All of these operations can be combined to produce complex and flexible pattern matching.

## Scanning

Scanning or lexical analysis is the first phase of an assembler or compiler. This phase is responsible for reading in the source file of the program as characters and dividing it up into tokens [14].

Tokens are like the words of a natural language, each token is a sequence of characters that represent a unit of information in the source program [14]. Typical tokens are *operators* like ‘+’ and ‘=’, *keywords* like ‘if’ and ‘goto’ and *identifiers* which are user defined strings [14].

Scanning is a special case of pattern matching and recognition. This process is typically performed using regular expressions [14]. The regular expressions split the source code up into meaningful units which are then converted to tokens. The tokens are typically some kind of enumerated type representing keywords and operators or a value such as an integer. The rest of the assembler can understand and can manipulate these types natively [14].

## Decision Tree

<decision tree background here>

## Multi-pass Assembler

M. Wilkes, the creator of Initial Orders described each instruction as having its own absolute address. Jump and branch instructions operated by adjusting the program counter by a relative amount, or to an absolute position, so that it pointed to the desired instruction [15]. However, he discovered that a system where each instruction has an absolute address is unnecessarily rigid and “*carries with it the disadvantage that a the programmer must be prepared to undertake extensive renumbering whenever extra [instructions] are inserted into the middle of the program”* [15]. In order to avoid continual renumbering, Wilkes proposed postponing all numbering until the program is in its final form. Placeholders or *labels* could be placed inline with the code to represent the eventual jump location [15].

The result of such a system is two-pass assembly, in which the first pass is used to discover all labels and calculate the address of each instruction and the second pass is used to fill in actual values for the jump and branch instructions [15].

## Intel Hex Format

Intel Hex is a hexadecimal text format suitable as input to PROM programmers. It was designed for Intel 8,16, and 32 microprocessors but has been adopted by many other architectures as well. The format was designed as hexadecimal in ASCII rather than binary so that the file can be represented in non binary mediums such as paper-tape, punch cards and CRT terminals [16].

The format is blocked into a number of ‘records’, each of which has a record type, length, address and checksum in addition to the data [16]. There are six different types of records however, 8 bit microprocessors are only interested in data records and the end of file record. 16 bit and 32 bit microprocessors use segment records for addressing values greater than the address field allows [16].

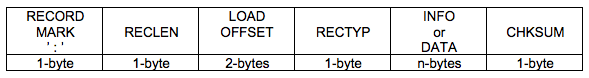


Figure Intel Hex Record Format

As seen in the diagram above, each record begins with a Record Mark field, containing the ASCII ‘:’ character to indicate the beginning of a record. The Record Length (RECLEN) field indicates the number of data bytes found in the data field. The Load Offset or Address Field specifies the 16 bit address offset from the current segment. For 8-bit microprocessors there is only one segment and thus, the 16 bit address can be thought of as an absolute address. The Record Type (RECTYP) field specifies how the information in this record should be interpreted. The different record types are listed in the table below:

|  |  |
| --- | --- |
| Record Type | Description |
| 00 | Data Record |
| 01 | End of File Record |
| 02 | Extended Segment Address Record |
| 03 | Start Segment Address Record |
| 04 | Extended Linear Address Record |
| 05 | Start Linear Address Record |

Table Intel Hex Record Types

The Data field consists of zero or more bytes as indicated by the Record Type field and the interpretation of the data field depends on the Record Type field. Finally, each record ends with a Checksum (CHKSUM) field that contains the hexadecimal representation such that if all fields except the record mark where converted to 8-bit two’s complement numbers and summed together the result would be zero. This allows for checking the integrity of the file to ensure that no errors have been introduced during its lifetime [16].

## SREC Format

The Motorola S-Record or simply SREC is a hexadecimal ASCII format for representing binary objects. It was designed to be a printable format for easy transportation and so that it could be easily edited [17].

A S-Record file consists of a number of records each record is composed of several fields. The fields in each record are the record types, record length, memory address, the data and checksum [17]. The fields are arranged as seen in the figure below:



Figure Motorola S-Record Format [17]

The role and length of each field is described in the table below:

|  |  |  |
| --- | --- | --- |
| Field | Field Length | Contents |
| Type | 1 byte | S-Record Type: S0, S1 etc. |
| Record Length | 1 byte | Specifies the number of data bytes |
| Address | 2-4 bytes | The 2,3, or 4 byte address at which the data field is to be loaded into memory |
| Data | n bytes | From 0 to n bytes of executable code, memory loadable data or descriptive information. |
| Checksum | 1 byte | The least significant byte of the one’s complement of the sum of the bytes making up the record length, address and data fields. |

Table Field Composition of an S-Record [17]

There are eight different Record Types to accommodate encoding, transportation and decoding. The different types and their purpose are listed in the table below:

|  |  |
| --- | --- |
| Record Type | Description |
| 00 | Data Record |
| 01 | End of File Record |
| 02 | Extended Segment Address Record |
| 03 | Start Segment Address Record |
| 04 | Extended Linear Address Record |
| 05 | Start Linear Address Record |

Table Motorola S-Record Record Types

## CSV Format

CSV stands for comma separated values. There is no common standard for the CSV format but it is described in RFC 4180 [18]. A CSV file is used to transport tabular data using the following format:

* Each record is located on a separate line delimited by a line break [18].
* The last record in the file may or may not have a trailing line break [18].
* In each record there may be one or more fields separated by commas. Each line should contain the same number of fields throughout the file [18].
* Fields may or may not be enclosed in double quotes [18].
* Fields containing line breaks, double quotes and commas should be enclosed in double quotes [18].

The RFC does not describe the text encoding that should be used for the file.

## AVR ATMega64

The ATMega64 is a high performance 8-bit AVR RISC-based microprocessor developed by the Atmel Corporation. It is part of the broader ATMega family which share the same instruction set and much of their functionality [19]. AVR is the name of the instruction set that these chips use and does not stand for anything.

The ATMega64 is used heavily in industrial automation, metering and other embedded applications requiring a large code base and low power.

The ATMega64 has a 16 bit address bus used to address the 64KB of on-board flash and 4KB of SRAM. It also has 32 memory mapped general purpose registers taking up addresses 0-31. And 64 memory-mapped input/output ports taking up address 32-95 [19].

Most logical, arithmetic and conditional instructions operate on the 32 general purpose registers. The CPU core has two inputs and one output leading directly to the register bank and most of these instructions will execute in a single clock cycle [19]. In line with the RISC model there are also load and store instructions to interface with the memory. The AVR instruction set also includes some special instructions to move values between the general purpose registers and the memory-mapped input/output registers [19].

## Freescale HCS08

The HCS08 is a high performance and low power 8 bit microprocessor developed by Freescale Semiconductors. The HCS08 uses the HC08 instruction set [20].

The HCS08 is widely used in industry today. Current applications of this microprocessor include power steering, automotive door/window/seat control, engine control units, airbag systems, in-vehicle networking, heating control and motor control [20].

The HCS08 has a 16-bit address bus divided up into a number of blocks which, combined, addresses: the registers, the ram, the flash and any peripherals. Every register, peripheral and input/output port is memory mapped with the exception of special registers. The HCS08 has 5 special registers connected directly to the CPU core. These are the 8-bit accumulator, H:X 16-bit index register, the stack pointer, the program counter and the status register. Most instructions use the accumulator as one of the operands and most results are stored in the accumulator [20].

# Research Plan

## Project Aims

Broadly speaking, the current problem with assembly languages is that if an average programmer were to pick up a piece of random assembly code they would not be able to decipher its meaning or purpose.

The difficulty is that there are many assembly languages and each is specific to a given architecture. Most of these languages are very inconsistent with each other and consist of mnemonics whose meaning is not clear without consulting a reference. In order to be read an assembly program a programmer must be very familiar with the particular architecture or the code must be heavily commented on each line.

The goal of this project is to investigate whether it is possible to produce a generic C-Like High Level Assembly (CHLA) language that can represent many assembly languages in a consistent and familiar syntax. Programs written in a CHLA language should be readable by those who are not experts in the given architecture.

The project aimed to produce a useable development environment to test the CHLA language. The development environment was to consist of both an assembler and a disassembler. The assembler would facilitate the translation from CHLA to a specific architecture’s assembly language. The disassembler would take some binary program data from a specific architecture and translate it into CHLA. These two tools should allow programmers to write new programs in CHLA and target a microprocessor as well as convert existing code to CHLA so that it may be more easily read.

It is very important to make the distinction between the goal of CHLA, which is to provide a consistent syntax across multiple architectures, and the goal of ‘C’, which is meant to provide completely portable code between multiple architectures. It is not a goal to be able to write a program in CHLA and have it work on every microprocessor architecture. That goal would be very unrealistic as not all architectures have the same instructions or the same memory/register locations. Rather, if two architectures share an instruction, the goal is to have that instruction have the same CHLA syntax. With this goal, the syntax for a given instruction is consistent across all architectures. A programmer should be able to read and write programs written for different architectures in the same syntax even though the programs might not be strictly cross-compatible. A single program written in CHLA should only be able to target multiple architectures if the program makes use of instructions from a subset of instructions shared among both architectures.

## Requirements

The syntax for CHLA needs to be developed. The syntax must be flexible so as to have the potential to describe the entire functionality of all microprocessor instructions. The syntax must also be obvious in its meaning. In order to make the syntax ‘obvious’ it is modelled on the C Programming language, which is very mature and well recognised.

There should be no ambiguity as to which instruction a given CHLA statement corresponds to. An expert assembly programmer who is familiar with the instruction set should be able to unambiguously understand how the CHLA code translates to assembly. Assembly is often used in environments where a high degree of certainty and exactness is required, therefore, CHLA cannot be allowed to introduce any ambiguity into such environments.

The CHLA language will be analysed for correctness, consistency, and reuse. Correctness is defined as having both the assembler and disassembler translate CHLA free of errors. Consistency is defined as equivalent instructions across multiple architectures having the same CHLA syntax. Reuse is a measure of how many CHLA statements can be targeted at more than one platform.

It is beyond the scope of this project to develop a translation from CHLA to every assembly language architecture as there are simply far too many. The initial goal was to investigate the possibility of a translation with a single architecture. Once a single architecture had been shown to work a second architecture would be developed.

Two distinct and varied microprocessor architectures are necessary in order to investigate whether the CHLA syntax has the potential to encapsulate many assembly languages. The more different the two architectures, the better, as it will demonstrate the breadth of a CHLA language’s potential.

An assembler must be developed that has pre-processing capabilities including “definitions” and “conditional assembly”. Pre-processing allows for constants (such as addresses) to be represented in a human friendly way. Conditional assembly allows locations such as an output register to be defined conditionally based on the architecture and thus, allows the code to be more portable. Without such features CHLA would have limited readability and portability.

A disassembler must be developed that supports a variety of binary formats and the ability to introduce names for constants during disassembly. Parsing a variety of assembly programs would be exceptionally difficult as they contain directives directed at the assembler and macros that need to be processed. In contrast, parsing binary is relatively easy as only the instructions and values remain. Thus, in order to be scalable to many architectures in the long term, the disassembler will support binary data formats. However, parsing binary data loses some of the semantic meaning of values such as the name of a register. In order to achieve the goal of readability, the disassembler must have the capability to recover the name of a register from its address when given a file describing such names.

# Design Overview

## Initial Design

The initial approach involved a design that had all of the logic for parsing CHLA and translating it inside the Assembler. There was no internal representation of the data. Instead, each source line was individually read-in, parsed, and written out as assembly to a destination file. There was no attempt at understanding the program, the assembler simply manipulated characters based on some rules.

This approach was very naïve and was not readily scalable to future architectures. Parsing statements required very nested conditional code with lots of special casing to handle every scenario. The structure was unclear and the code difficult to follow.

To overcome these problems research was undertaken into scalable assembler design. In the related field of compiler development, it was discovered that many academic compilers involve some sort of generator which, given a grammar for some language and some basic constructs, will produce code to parse the source. The compiler approach does not translate directly to assemblers, but the concept of using an external document and generated code is appealing to a system that needs to be extensible and flexible.

## Specification Document Centric Design

The assembler and disassembler are both designed around Specification Documents. A specification document is a CSV file that describes the translation between an architecture’s assembly and CHLA for a given architecture. When assembling or disassembling a program, the specification document must also be provided to the assembler or disassembler so that it can understand and interpret the code.



Figure Assembler and Disassembler Design Overview

Figure VI provides an overview of the two core workflows of assembly and disassembly. Each involves some sort of source file and the specification document in order to produce the output. The details of the specification documents, the assembler and disassembler will be discussed in the chapters to come.

# Specification Document Design

## Overview

This chapter describes the CHLA syntax in general as well as the specification documents that describe specific CHLA statements and their translation to an architecture’s assembly.

C-Like High Level Assembly is made up of the following constructs: **keywords**, **operators**, and **operands**. There is also an additional concepts of **ranges** and **values** concerning operands. **Ranges** describe the range of possible values that an operand may have. For example, 8-bit operands typically must have a value in the range of 0-255. **Values** come into play during assembly and disassembly when an operand takes on a specific value such as an address or constant.

How CHLA is to be applied to every architecture is not defined in a strict set of rules. Instead, patterns are followed so that many distinct architectures can all appear very similar. The general approach is to convert the instruction to the equivalent ‘C’ syntax with the destination and/or source address as the operands.

|  |  |  |
| --- | --- | --- |
| Pattern | Description | Example |
| Constants | All values are assumed to represent addresses, not constants, unless enclosed in ‘|’ | r0 = |10| |
| Logical and Arithmetic Assignment | Destination, followed by ‘C’ logical/arithmetic assignment operator, followed by operand | A += |10| r0 &= r1 |
| Logical and Arithmetic with no assignment | Operand, followed by test, followed Operand | r0 & r1  A <= |0| |
| Load and Store from Memory | RISC Load and Store instructions should use ‘C’ array like syntax with the keyword MEM | r0 = MEM[x + 2] |
| Conditional | IF keyword, followed by conditional operator or expression, then skip or goto with a label | If >= goto loop  If PINA[0] == |1| skip |

Table CHLA Specification Patterns

Table 5 lists the basic CHLA keywords. These are used in all architectures to help describe instructions. Each architecture likely requires additional keywords to describe those entities that are not memory mapped and thus do not have addresses. For example, the HCS08 architecture requires additional keywords for its Accumulator register and Index registers.

|  |  |
| --- | --- |
| Name | Symbol |
| Transfer Bit | T |
| Carry Bit | C |
| Interrupt Bit | I |
| Status Register | SREG |
| Swap | SWAP |
| If | IF |
| Goto | GOTO |
| Skip | SKIP |
| Return | RET |
| Interrupt Return | RETI |
| Stack Pop | POP |
| Stack Push | PUSH |
| Memory | MEM |
| Program Memory | PMEM |

Table Basic CHLA Keywords

From these general rules and syntax a specification document for each architecture must be developed. A single statement in a specification document consists of the assembly instruction, the CHLA statement, the binary opcode and the allowed range of any operands. Figure VII shows some example statements taken from the ATMega64 specification document.

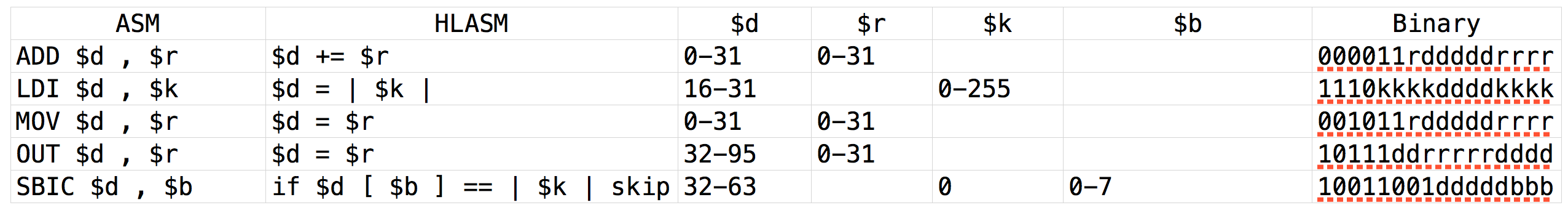


Figure Example Extract from Specification Document

## Specification Document Format

The specification document is a comma separated file with a “.spec” extension. It consists of 8 fields representing the assembly, CHLA, five operands and the opcode.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Assembly | CHLA | ‘d’ Operand | ‘r’ Operand | ‘k’ Operand | ‘b’ Operand | ‘l’ Operand | Opcode |

Figure Specification Document Field Format

Figure VIII details the fields found in the specification document. The “assembly” and “CHLA” fields must be present for every record. The “operand fields” must exist if that operand is found in either the CHLA or assembly. The “opcode” field must exist if the instruction is to be disassembled (See **Disassembly Note** one paragraph below).

Records should be ordered in ascending order based on the number of clock cycles an operation takes to execute. The assembler and disassembler can take advantage of this fact to unambiguously translate between Assembly and High Level Assembly for statements that could be performed using more than one instruction.

**Disassembly Note:**

Due to disassembly limitations, some opcodes for which there are multiple possible corresponding instructions can only be disassembled to one of the instructions. For example, the two instructions below both double the value of r16 and store the result in r16.

LSL r16

ADD r16,r16

In fact, during assembly the LSL instruction is just converted into the ADD instruction. It is therefore very difficult to get the LSL back from the ADD instruction opcode. For simplicity, LSL has an empty opcode field and during disassembly any LSL instructions encountered will be just disassembled to its ADD equivalent.

## Operands

Operands fulfil two roles in a specification document: “constants” or “values”. Where a value or constant would appear in the Assembly or CHLA fields, an operand placeholder should be put instead. Then, for each placeholder that occurs in the Assembly or CHLA fields the range of acceptable values for that operand must be entered in the appropriate field. For example:



Figure Specification Document Operand Example

Two different subtraction statements are shown in Figure IX. There are a number of things operands are being used for in this example. In each of the Assembly and CHLA fields, operands have been used as placeholders for values or constants. In the DEC instruction example, the $d operand is used as a placeholder for an address in the range 0-31. The $k operand is being used to represent the constant ‘1’. It is important that constants be represented by operands and not the actual constant in case the programmer provides the constant in a different form such as ‘0x01’ or as a pre-processor definition. By using an operand any expression that evaluates to ‘1’ will also parse correctly instead of just the character ‘1’.

The acceptable range for a value or constant can be one of four different types. The different types are shown in Table 7 below.

|  |  |
| --- | --- |
| Type | Example |
| Constant Single | 1 |
| Ranged Single | 0-255 |
| Constant Pair | 25:24 |
| Ranged Pair | 25:24-31:30 |

During assembly the range of an operand helps unambiguously determine the correct instruction that a statement corresponds to. During disassembly it is impossible for a value to be outside its allowable range as there is not enough bits to represent numbers outside this range. However, the range still plays a vital role. Consider the SBIW example instruction above, the destination register is represented by just two bits or the numbers 0-3. In order to recover the actual address the binary value must be calculated from the range. For pairs, the bit-value is multiplied by two, and then, for both singles and pairs, the binary value is offset by the start of the range. For example, the SBIW instruction has a bit-value of ‘1’, this should map to the second pair of values: ‘27:26’. The original bit-value is multiplied by two, and then offset by the start of the range: ‘25+2 and 24+2’ to give the correct range of ’27:26’.

The choice of which of the five placeholders to choose is not completely arbitrary. Most of the operands are interchangeable, however, the assembler must be informed if the operand is a label, as the label’s ‘address’ is filled in at a later point. Thus, all labels must use the ‘$l’ operand. Many instruction set manuals use the characters ‘d’ and ‘r’ for destination and source registers, ‘k’ for constant values and ‘b’ to represent bit positions. It is not necessary but this convention has been followed when developing the two specifications for this project as it aids readability and maintainability.

## Equivalent Statements

It is sometimes desirable to have more than one CHLA statement correspond to a single instruction. To achieve this, two separate records for the same instruction are provided in the specification document, each with different CHLA syntax. With this system either form of the syntax will result in the same instruction.

Equivalent statements are a very good system for providing generic, consistent and flexible syntax whilst also providing convenient syntax. These two outcomes can be demonstrated by examining an example instruction:

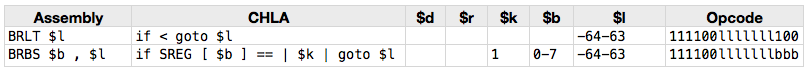


Figure Specification Document Equivalent Statements

The two statements above are actually the exact same instruction when executed on the microprocessor. The instruction checks whether a bit is set in the status register, and if it is, advances the program counter to a new location. The BRBS form of the instruction, allows any bit to be examined, whereas the BRLT instruction is a special case of the BRBS instruction with the SIGN bit hardcoded into the opcode. Both versions of the instruction have strong use cases and may be more descriptive to a programmer in certain scenarios. CHLA allows you to have both and thus, wherever possible, equivalent statements have been added so as to increase readability and portability.

## Generality without Ambiguity

The requirements of this project specify that there must be no ambiguity between the CHLA statement and the assembly that it translates to. However, it is very convenient and assists portability if the assembler can maintain some generality and interpret a statement and insert the correct assembly. Allowing generality and maintaining no ambiguity may seem mutually exclusive but are actually achievable. Let us examine two scenarios where this arises and how it can be overcome.

Scenario One considers ‘assignment statements’. Table 6 contains a number of assignment statements in CHLA and their corresponding assembly. The addition of pre-processor definitions is to emphasise the generality of the assignment statement as just ‘operand = operand’.

|  |  |
| --- | --- |
| Assembly | CHLA |
| MOV r16, r17 | #define A r16  #define B r17  A = B |
| OUT DDRA, r16 | #define A DDRA  #define B r16  A = B |
| IN r16, DDRA | #define A r16  #define B DDRA  A = B |
| LDA 16 | #define B 16  A = B |

Table Generality of CHLA Assignment Statements

In the above example, the assembler is performing some work on behalf of the programmer. The assembler analyses the source and destination registers and unambiguously determines the only possible instruction. In this way, the assembler achieves some form of generality whilst meeting the requirement of being unambiguous. The goal is to aid reading and writing by performing some of the work for the programmer. For example, an ATMega64 programmer can write the following two statements one after the other without having to worry about the exact instruction that will be used behind the scenes.

r16 = r17 # MOV instruction

DDRA = r16 # OUT instruction

Scenario two is concerned with CHLA statements where more than one assembly statement could accomplish the task. Consider incrementing the accumulator register by the constant value ‘1’ in the HCS08 architecture.

A += |1|

This could be accomplished by both the ADD instruction or the INCA instruction. To avoid ambiguity, a convention must be used to determine the which instruction to use. The convention used in CHLA is to choose the instruction with the smallest number of clock cycles, and if they have equal number of clock cycles, then the instruction with the smallest opcode. Currently there is no way to unambiguously determine the correct instruction if there exists two instructions in a single architecture that perform the same task in the same number of clock cycles with the same opcode length (Note that no two such instructions have been encountered yet).

# Implementation

## Overview

The assembler and disassembler are written in python3 using object oriented concepts.

The assembler comprises of four key steps:

1. pre-processing
2. tokenisation
3. decision-tree generation
4. matching

Pre-processing and tokenisation prepare the source CHLA program so that the assembler can understand it. The decision-tree is generated from the specification document and it allows the assembler to match CHLA tokens to assembly instructions.

The disassembler comprises of three key steps:

1. binary parsing
2. regular expression generation
3. matching

Binary parsing converts the hexadecimal binary format into a list of byte length opcodes. Regular expressions for the opcodes are generated from the specification document. The regular expressions are then used to match the binary data to instructions.

## Pre-processing

The pre-processor scans through the source file linearly looking for pre-processor directives. If an external file inclusion directive is encountered, the included external file is opened, and pre-processing is continued from within the included file. When the end of the included file has been reached, pre-processing resumes from where the inclusion directive was encountered. The programmer must be careful to not recursively include a file as the pre-processor will continue to open the include files to infinite depth.

Table 7 details the pre-processor directives supported by the assembler.

|  |  |
| --- | --- |
| Directive | Description |
| #include “<file>” | Includes an external definition file. The contents of the file included will also be pre-processed. |
| #define <name> <expression> | Defines <name> with the value of expression. The expression may be any python evaluable expression and can make use of previously defined values. |
| #ifdef <definition> | Conditionally processes statements depending on whether a value is defined. |
| #ifndef <definition> | Conditionally processes statements depending on whether a value is not defined. |
| #else | Reverse the condition of an #ifdef or #ifndef. Not allowed outside of a conditional pre-processor directive. |
| #endif | Stops conditionally processing statements. |

Table Description of Pre-Processor Directives

The pre-processing directives are quite flexible and powerful. The #define directive uses python3’s eval() function and puts in the scope all current definitions. This allows complex expressions to be built up and aids in readability. For example, it is more readable to describe the value of a control register as the sum of its bits rather than just a seemingly random constant. Consider setting up an ATMega64 Timer:

#define WGM12 3

#define CS11 1

#define TIMER1B (1<<WGM12 | 1<<CS11)

OR:

#define TIMER1B 10

The two TIMER1B statements are equivalent. The first example should provide a programmer with a better understanding as to why the value is what it is compared to the second example with a seemingly random constant.

## Tokenisation

Tokenisation is the process of breaking up the source file into discrete units that the assembler can understand. The individual discrete units are called tokens. Tokens are to source code, as words are to natural language.

The tokeniser breaks the source code up into four different types of tokens: *Keywords*, *Operators*, *Values* and *Labels*.

*Keywords* are reserved strings with special meaning attached to them. For example, the “IF” keyword indicates the beginning of a conditional statement. Refer to Appendix A.1 for a full list of the keywords contained in the Keywords Class.

*Operators* are symbols with special meaning attached to them. The CHLA operators are based on the programming language C’s operators. Refer to Appendix A.2 for a full list of the operators contained in the Operators Class.

*Values* may be hexadecimal numbers preceded by ‘0x’ or decimal numbers. A value may be representative of a constant number, an address, or register/pair. A pair is a special case of a value where two values are separated by a colon. In assembly, pairs are two registers combined together to produce a double width register. The internal representation of a pair is literally two values:

p = Pair( firstValue, secondValue )

All entities in the source code not found to be a *keyword*, *operator* or *value* are assumed to be a *label*. If this is an incorrect assumption, it will be detected later during matching.

Regular expressions are used to identify different tokens. Appendix B.2 shows a graphical representation of the primary tokenisation regular expression. There are four main groups in the regular expression to take note of: Group #1, Group #4, Group #5 and Group #6. These groups represent the code to detect labels, pairs, keywords/values and operators respectively.

The output from the regular expression is then given to the ‘parse\_token’ method which resolves it to a Keyword, Operator, Value or Label. The parse\_token function is shown below:

171 def parse\_token( self, token ):

172

173

174 # definition

175 if token in self.definitions.keys():

176 if isinstance( self.definitions[ token ], int ):

177 return Value( token, self.definitions[ token ] )

178 else:

179 token = self.definitions[ token ]

180

181 # keyword

182 for k in Keywords():

183 if token.lower() == str( k ):

184 return k

185

186 # value

187 if re.match( "(0x[0-9A-Fa-f]+)|(-?[0-9]+)", token ):

188 return Value( "", int( token, 0 ) )

189

190 # operator

191 if re.match( "[\+!=&~\-\^\.\|<>\[\]\(\)\\*:/]+", token ):

192 for op in Operators():

193 if token == str( op ):

194 return op

195 raise ValueError( "Operator not recognised: " + token )

196

197 # pair

198 if re.match( "([a-zA-Z0-9]+:([a-zA-Z0-9]+))", token ):

199 tokens = token.split(":")

200 first = self.parse\_token( tokens[0] )

201 second = self.parse\_token( tokens[1] )

202 if not isinstance( first, Value ):

203 raise ValueError( "Pair must contain two values " )

204 if not isinstance( second, Value ):

205 raise ValueError( "Pair must contain two values " )

206 return Pair( first, second )

207

208 # assume label

209 return Label( token, None )

The parse\_token function attempts to parse the token in a particular order. Definitions take highest precedence as it is possible to redefine anything. Keywords have the second highest precedence as they are reserved words in the language. Smaller regular expressions are then used to determine if the characters of the token match a value, operator or pair. Failure to match at this point results in the function assuming the token is a Label.

The output of the tokeniser is a dictionary containing the line number as the key and a list of tokens as the value. Here is a graphical representation of the tokeniser’s workflow:

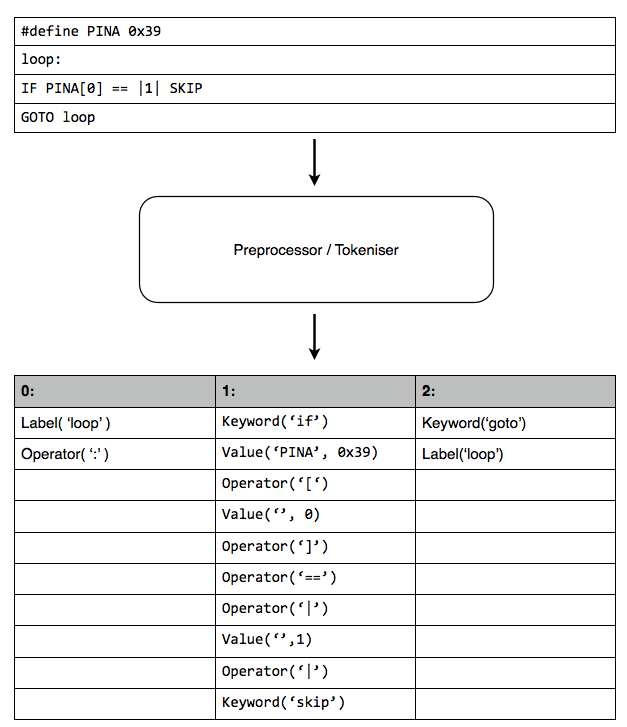


Figure Tokeniser Workflow

## Decision Tree and Regular Expression Generation

The Generator class is responsible for parsing a specification document and generating a Decision Tree for the assembler and regular expressions for the disassembler. Parsing of the specification document is done on a line by line basis.

Statements are CHLA’s equivalent to assembly’s instructions. Each line of the specification document corresponds to a single CHLA statement. A statement describes an operation and the operands the operation is performed on. Statements are the core data structure used by the assembler and disassembler. Statements are implemented in two fundamental classes: the *StatementTemplate* class and the *Statement* class. A *StatementTemplate* has no values attached to its operands. A *Statement* class is created from a *StatementTemplate* when values are provided for the operands during matching.

Each line of the specification document contains an assembly field, CHLA field, five operand fields and the opcode field. Each of these fields are separately parsed and then collected together to form a *StatementTemplate*. An instance of the *Statement* class represents a single instruction and stores the assembly representation, the CHLA representation, the operands and their values, and the opcode.

67 # create template

68 # statement templates are filled in when the values are added

69 s = StatementTemplate( asm, chla, opcode, operandPositions, operands )

70

71 # callback

72 callback( s )

Figure StatementTemplate generation

The template is created and passed to a callback. For the assembler, the callback adds the template to a decision tree (Figure XIII), for the disassembler, the callback adds it to a dictionary with a regular expression based on the opcode as the key (Figure XIV).

98 def add\_to\_tree( statement ):

99 node = tree

100 for token in statement.chla():

101 if token not in node:

102 node[token] = {}

103 node = node[token]

104 if node.get( Keywords.LEAF ):

105 raise ValueError( "Invalid Specification document" )

106 node[ Keywords.LEAF ] = statement

Figure Decision-Tree Callback

85 statements = {}

86 def regex\_statements( statement ):

87 opcode = statement.opcode()

88 regex = re.sub( "[drklb]", "[01]", opcode )

89 statements[ regex ] = statement

Figure Regular Expression Dictionary Callback

Table 8 is an example extract from the ATMega64 specification document (opcodes left out for brevity). Figure XV shows an example decision tree for the statements in the specification document extract. The *StatementTemplate* would be placed at the leaf node of each branch.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ASM | CHLA | $d | $r | $k | $b | $l |
| BRBC $b , $l | if SREG [ $b ] == | $k | goto $l |  |  | 0 | 0-7 | -64-63 |
| BRBS $b , $l | if SREG [ $b ] == | $k | goto $l |  |  | 1 | 0-7 | -64-63 |
| SBIC $d , $b | if $d [ $b ] == | $k | skip | 32-63 |  | 0 | 0-7 |  |
| SBIS $d , $b | if $d [ $b ] == | $k | skip | 32-63 |  | 1 | 0-7 |  |
| SBRC $d , $b | if $d [ $b ] == | $k | skip | 0-31 |  | 0 | 0-7 |  |
| SBRS $d , $b | if $d [ $b ] == | $k | skip | 0-31 |  | 1 | 0-7 |  |

Table Example Specification Extract

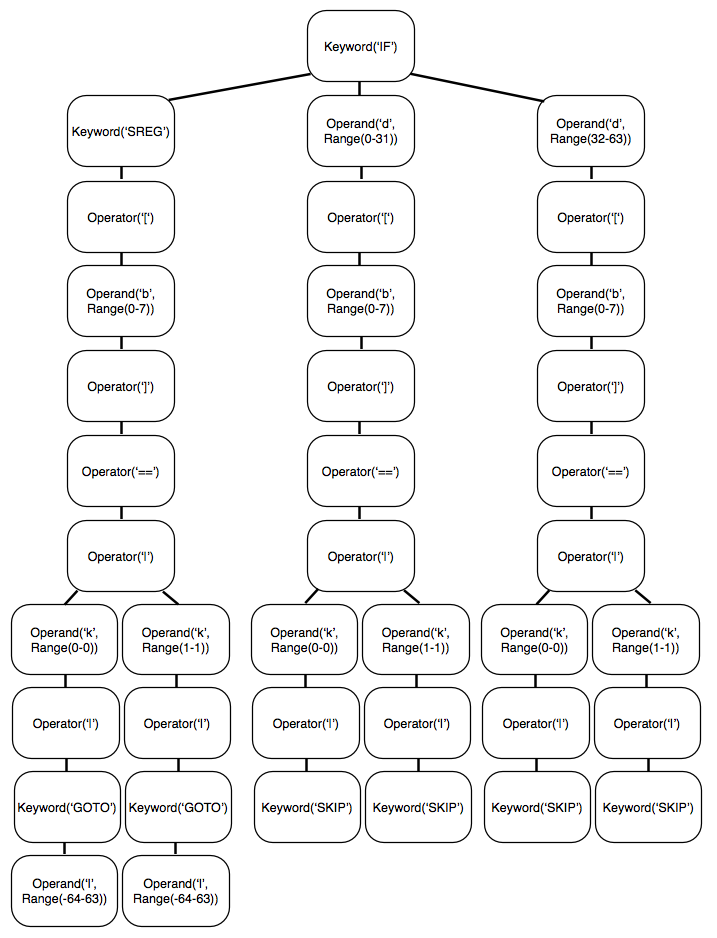


Figure Decision Tree Extract

During matching, the source code tokens should match the nodes of a branch and reach the leaf node of the instruction it corresponds to. At the leaf node the tokens are converted into a *Statement* by adding the *Values* from the tokens to *Operands* of the *StatementTemplate*. An error with be thrown during generation should there be two templates at one leaf. Two templates at one leaf would mean it is impossible for the assembler to differentiate two different instructions, and thus, impossible to assemble.

## Decision-Tree Matching

Matching is very simple. When the assembler is ready for matching it already has statements from the tokeniser, and a decision tree from the Generator.

The tokeniser has converted the source code statements into lists comprised of Keywords, Operators, Values and Labels.

The generator has converted the specification document into a decision tree comprised of Keywords, Operators and Operands. The StatementTemplate for each branch is located at the leaf.

150 def match( node, chla ):

151 """recursively matches chla against the parse tree"""

152

153 if len( chla ) == 0:

154 # no more tokens to match

155 # Return the leaf node if it exists

156 return node.get( Keywords.LEAF )

157

158 for n in node:

159 # try all branches at current node

160 if n.match( chla[0] ):

161 # next token matches

162 # continue down branch

163 r = Assembler.match( node[n], chla[1:] )

164 if r:

165 # successfully matched statement at leaf

166 return r

167

168 return None

Figure Decision-Tree matching in the Assembler

Figure XVI shows the simple recursive algorithm which takes a line of source code and traverses the tree matching the CHLA tokens to the nodes. Matching Keywords to Keywords and Operators to Operators is trivial because all the entities are just constants. Matching a Value to an Operand however, requires checking if the Value lies within the legal Range of the Operand. All labels are assumed to match the ‘l’ operand. It is impossible to determine the address of the label at this point. If a label is incorrectly matched, it will be corrected during the second pass. Figure XVII shows the Operand’s special match method which checks if the Value lies within a valid range.

14 def match( self, other ):

15

16 """Special method designed to match operands

17 to legal values as well as equate two operands."""

18

19 if isinstance( other, Operand ):

20 return self.\_name==other.\_name and self.\_range==other.\_range

21

22 if isinstance( other, ( Value, Pair) ) and self.\_name != "l":

23 return self.\_range.check( other )

24

25 if isinstance( other, ( Label ) ) and self.\_name == "l":

26 if other.value():

27 return self.\_range.check( other )

28 return True

29

30 return False

Figure Checking value lies within legal range inside Operand’s match method

## Addresses, Labels and Jumps

The assembler is a two pass assembler. The length of each instruction, and thus the absolute position of each instruction is determined in the first pass. When an address is located during the first pass, its address is stored in a separate dictionary. After the first pass, every instruction has an absolute position and the value of every label is known.

The second pass involves locating any branch or jump instructions and filling in the appropriate value. At this point the address value can have its range checked to see if it lies in the legal range. If it is found to be out of the legal range, a second round of matching is attempted. During the first round of matching, all labels were assumed to match because the value of the label was unknown, now that the value is known accurate matching can take place. It is likely that, if a new match is found, that a larger opcode will be required to store the larger address. If a larger opcode is used for the newly matched statement, all addresses and labels from that point must be updated. See Appendix A.3 for the Assembler class for exact implementation.

## Opcode Matching

When the disassembler is ready for matching it already has the opcodes from the binary parser and the opcode regular expressions from the generator. Each opcode is tested against the regular expressions until a match it found. When a match is found, the operand values are extracted from the opcode and a Statement is created.

Not all opcodes are the same length. The HCS08 can have opcodes of length 8, 16, 24 or 32 bits and the ATMega64 can have opcodes of 16 or 32 bits. To ensure that a match is found despite variable lengths being acceptable, each opcode is first tested at the minimum size, and if no match is found another opcode is added to it to find a larger size match. If no match has been found after the maximum size then an error is thrown. Figure XVIII shows this in detail below.

41 # get next byte

42 byteAddress = address[i]

43 i += 1

44

45 # instruction words vary in size so:

46 # look for larger word instruction if didn't match smaller word

47 if not found:

48 # maximum word size

49 if wordBytes >= 4:

50 raise ValueError( "Unrecognised opcode: {0}".format( opcode ) )

51 # increase size

52 word.append( sourceBinary[byteAddress] )

53 wordBytes += 1

54

55 # if last word did match, start new word

56 else:

57 word = [ sourceBinary[ byteAddress] ]

58 found = False

59 wordBytes = 1

60 wordAddress = byteAddress

61

62 # some architectures only support specific word sizes

63 # only match word size instructions

64 if wordBytes % bytesInWord != 0:

65 continue

66

67 # convert word to opcode

68 # words are big endian

69 # instructions are little endian

70 # e.g. ATMega64 32-bit instruction is bytes [ 1, 0, 3, 2 ]

71 opcode = ""

72 for w in range( 0, wordBytes, bytesInWord ):

73 for b in range( 0, bytesInWord ):

74 opcode += word[ w + bytesInWord - b - 1 ]

Figure Disassembler handling variable length opcodes

## Output

After matching, both the disassembler and the assembler have the program stored as a dictionary with addresses as the keys and Statements as the values. The Statements are an abstract representation of the program and can be output in a number of ways.

Each statement class has two convenience methods: chla\_string and asm\_string which output the statement in CHLA and assembly respectively. Outputting the statement as a string is as simple as printing out the tokens in order whilst substituting any operands with their value. The methods are shown below in Figures XIX and XX.

71 def asm\_string( self ):

72 s = []

73

74 for token in self.\_asm:

75 if isinstance( token, Operand ):

76 s.append( self.\_values[ token.name() ].asm() )

77 else:

78 s.append( token.asm() )

79

80 return "{0} {1}".format( s[0], "".join( s[1:] ) )

Figure Statement’s asm\_string method

82 def chla\_string( self ):

83

84 s = []

85

86 for token in self.\_chla:

87 if isinstance( token, Operand ):

88 s.append( self.\_values[ token.name() ].chla() )

89 else:

90 s.append( token.chla() )

91

92 return " ".join( s )

Figure Statement’s chla\_string method

# Testing and Analysis

## Test Programs

During development many small test programs were built to test specific features. Jump and Branch instructions which require the calculation of absolute and relative addresses are complex and varied, and thus, a large proportion testing was concentrated on those instructions. Once development was complete, an exhaustive test program was developed for both architectures to ensure that every instruction worked.

The exhaustive tests were created in each architecture’s native development environment and written in assembly. Each test program exhaustively tests all instructions in the instruction set. Most HCS08 instructions have several different acceptable ranges and versions, so every possible range and version was tested. These programs were then assembled to produce the HEX and SREC binary formats of the ATMega64 and HCS08 respectively. The hexadecimal binary data files for these two tests can be found in Appendices C.1 and C.2.

The test programs are put through the disassembler to produce CHLA programs. The CHLA programs are then fed into the Assembler to produce the native assembly. At this point the native assembly can be compared to the original assembly.

In addition to the two exhaustive tests, a third test program was developed to investigate the consistency and reusability of CHLA. This program was specially designed to assemble to both the ATMega64 architecture and HCS08 architecture. It makes use of conditional pre-processor directives to provide code specific to one architecture or another. The CHLA code for this program can be found in Appendix C.3.

## Correctness

The exhaustive test programs allow for easy testing of correctness. The test assembly programs go through the following process:

1. Native assembly to binary data representation
2. Disassembled from binary to CHLA
3. Assembled from CHLA back to native assembly

This sequence of events exhaustively tests every possible instruction and range and produces an output file that facilitates easy testing.

A Unix tool called ‘diff’ is used to compare the original and final assembly files. The tool is configured to ignore whitespace and capitalization. The results of this diff tool for the ATMega64 exhaustive test program are available in Appendix B.1.

Most of the instructions match perfectly character for character and therefore do not show up in the output. Some instructions show as different due to different representation of the same data. For example consider the first mismatch in the output:

6c6

< andi r16,240

---

> ANDI r16 , 0xF0

Figure Exhaustive Testing Data Representation Mismatch Example

These instructions are equivalent but the native assembly has the number represented in Hexadecimal format whilst the computed assembly has the number represented in decimal format.

Character differences with no semantic difference are not restricted to just constants. Figure XXII shows each of the load instructions presenting as a mismatch because the native assembly uses special names to represent register pairs and the computed assembly does not. The assembler and disassembler are written very flexibly to support as many architectures as possible and therefore output the data in the most generic format possible.

101,112c111,123

< ld r16,r27:26

< ld r16,r27:26+

< ld r16,-r27:26

---

> ld r16 , X

> ld r16 , X+

> ld r16, -X

Figure Exhaustive Testing Special Naming Mismatch Example

There are also differences in output that cannot be avoided. Names of labels are discarded by the assembler during assembly and just their address remains. Thus, when reintroducing labels, the name cannot be recovered. Instead, the labels are just labels sequentially ‘l0’, ‘l1’, ‘l2’ etc. These generated labels obviously do not match the original semantic labels.

The final class of mismatch that occurs is when an equivalent statement is used instead of the original. This is most evident in the branching instructions where there are many equivalent statements. Consider Figure XXIII for example, a ‘branch if equal’ has been translated as a ‘branch if specified bit in the status register is set’. These are logically equivalent, as the specified bit is the ‘zero bit’ in the status register but appear as a mismatch.

41,42c41,42

< l0:

< brbs 1,l0

---

> lbreq:

> BREQ lbreq

Figure Logically equivalent statements mismatch

Despite many apparent mismatches, the code is provably correct. The computed assembly is once again natively assembled (after some minor character changes) to produce a second binary data file. This second binary file was then shown to be identical to the original binary file. This proves that during the disassembly and assembly round trip the semantic meaning of the program was unaltered.

## Consistency

In Section 3.2, consistency is defined as “equivalent instructions across multiple architectures having the same CHLA syntax”. Consistency is entirely concerned with the quality of the multiple architectures’ specification documents and not with the actual assembler or disassembler.

The ATMega64 and HCS08 specification documents were designed with consistency in mind. The ATMega64 specification was developed first and its syntax was modelled on the C programming language. The HCS08 was developed second and whenever an equivalent instruction was encountered the syntax was copied from the ATMega64 specification to ensure consistency. To further aid consistency, equivalent statements were introduced wherever possible so that one architecture could model the limitations of another. Consider, the “skip if bit is set” ATMega64 instruction, there is no equivalent in HCS08. To counter this, a special case of the “branch if bit is set” HCS08 instruction was introduced with the branch hard coded to skip a single instruction.

The cross-architecture CHLA program (Appendix C.3) was used to empirically demonstrate the consistency between the two specifications. The test program demonstrates consistency by having the exact same CHLA statements assemble and run on both architectures. With the exception of some conditional register definitions and some architecture specific interrupt code 100% of the code is completely generic.

Registers and interrupt locations are never the same even between microprocessors of the same architecture and therefore, some setup boilerplate for each target architecture is necessary at the beginning of each program. Despite this limitation, this CHLA test program proves it is possible to write a CHLA program with 100% generic code for the actual program logic.

## Reuse

In Section 3.2, Reuse was defined as a measure of how many CHLA statements can be ‘targeted at’ or ‘reused on’ multiple architectures. Reuse is important if a programmer wishes to write a single program and have it assembled to multiple architectures. Reuse is closely related to Consistency. A high degree of consistency should contribute to a high degree of reuse.

Reuse is difficult to calculate as different architectures support different instructions and therefore, the limitation on instruction reuse lies mostly with the architectures themselves, and not with CHLA. At best, CHLA’s instruction reuse can only be as good as the number of shared instructions between two architectures. To aid in analysis, there is a set of instructions that almost every architecture supports. This report will refer to this shared subset of instructions as the ‘RISC subset’.

RISC is not a strict set of instructions, but instead a microprocessor design concept encouraging generic, small and simple instructions over specialised and complex instructions. In reality this translates to a set of instructions that perform basic arithmetic, logic, branching and memory loading and storing.

With careful design it is possible to achieve 100% reuse when using the RISC subset of instructions. The cross-architecture CHLA program (Appendix C.3) attempts to provide some empirical evidence of this. This test program has 100% reuse and only makes use of instructions that would be considered within the RISC subset.

Instructions that perform complex tasks or are specific to the architecture are very unlikely to be reusable on a second architecture and CHLA does very little to improve this. Thus, the percentage of instruction reuse for a given architecture is roughly equivalent to the percentage of instructions that are within the RISC subset.

# Evaluation

## Data Segment Limitation

No attempt has been made to implement data segment functionality. The data segment allows a programmer to define constant data to be used throughout a program. The data segment is regularly in industry for defining things like a cryptographic decoding matrix or an array of waveform values.

Data segment functionality is very difficult to implement generically and consistently as it is not related to the assembly instructions but instead is comprised of directives to the architecture’s assembler. Data segment definitions are directives to the assembler to organise some data at some memory location.

In order to support data segment functionality, an extension to the document specification format is needed to facilitate the various assembler pre-processor directives. Furthermore, it will be very difficult to ensure correctness of the program as the address of the data cannot be known until the assembler has organised it.

## Non-Mathematical Instructions

The goal of CHLA is to provide a readable and intuitive assembly language modelled on the C programming language. It is very easy to model mathematical and logical operations on the C Programming language as they map directly to C operators. Branches and Jumps also map relatively intuitively to C keywords with similar functionality. However, there are a number of assembly instructions that do not have any equivalent functionality in C. Thus, these instructions cannot be intuitively determined.

Two instructions found in every assembly architecture are PUSH and POP. These instructions allow values to be pushed onto the top of the memory stack and values to be popped off the top of the memory stack. In CHLA they have been implemented as “function calls” as this is the closest approximation to C syntax. All similar instructions that do not have a C equivalent have also been implemented as a function call.

Implementing instructions as a function call is an acceptable compromise to a point. However, if too many instructions are implemented as function calls the entire purpose of creating an intuitive operand/operator based language is lost as the programmer is reduced to just remembering a different set of arbitrary commands.

In order for instructions to map to C operators and keywords, they need to be simple and mathematical in nature. Instructions that fall into this category are generally those that are modelled on the RISC philosophy. For this reason, non-RISC architectures are unlikely to benefit from CHLA.

## Extensibility

For CHLA to be truly useful, many architectures need to be supported. This report has demonstrated that it is possible to implement two different architectures. It is anticipated that further architectures should not be significantly difficult to implement, however, changes to the specification document may be necessary.

As more architectures are implemented, the likelihood of encountering more complexity increases. In order to accommodate further complexity, the specification document format needs to offer more detail. The current specification document assumes that bits are in order in the opcode. This assumption does not hold for all architectures and so more sophisticated operand/opcode formatting would be necessary. Similar changes would likely be required as other assumptions are found to not hold across many different architectures.

## Readability

The main goal of CHLA is to improve the readability of CHLA. However, it is very difficult to measure how ‘readable’ a computer program is. Particularly given that easier to understand statements do not necessarily correlate to an easier understanding of the overall program. However, the inverse can be assumed true, if a programmer cannot understand the individual statements, then they cannot understand the whole program. Thus, easier to understand statements likely contribute to program readability.

Anecdotal evidence suggests that the functionality of CHLA statements are more obvious than their instruction equivalent. For example, consider a simple branch instruction in CHLA and in assembly.

if < goto exit # CHLA Branch

BRBS 4, exit # Assembly Branch

A programmer can likely read the first instruction in English and determine the meaning: “if less than goto exit”. However, it is unlikely that a programmer not family with AVR Assembly with be able to determine that BRBS stands for “branch if bit in status register is set” and that the 4th bit is the sign bit and thus, infer that this is a test for “less than”. This instruction is a very good example of how CHLA improves readability. However, not all instructions benefit from C like syntax. For example the “RET” instruction, for returning from a subroutine is identical to the CHLA “RET” statement.

Whilst difficult to demonstrate quantitatively, CHLA appears to greatly improve readability for instructions that are mathematical or logical in nature. In the worst case scenario CHLA cannot map an instruction to a C-Like equivalent and it is implemented as simply a keyword or function call that is at least as good as the equivalent instruction. CHLA further aids readability by providing the same consistent syntax across multiple architectures so that programmers may become familiar with its statements.

## Use as a Learning Tool

One application of CHLA that became evident during development is its use as a tool for C programmers to learn assembly. For example, if a programmer wants to learn how to write assembly to add two 16-bit integers on the ATMega64 architecture, they can make an example C program, compile it and disassemble to CHLA. The resulting CHLA code should be familiar to the C Programmer and often demonstrates the most efficient method of performing a particular task.

CHLA’s use as a learning tool is most useful for isolated examples of specific tasks. Larger programs become unwieldy to navigate and understand as disassembled code is not necessarily logically structured. CHLA is ideal when a C programmer knows what they want to achieve but they are unsure how to complete the same task in assembly.

# Conclusions

## Summary

The goal of CHLA is to bring order and consistency to the disorganized, unclear world of assembly. CHLA achieves this by introducing a consistent C-like syntax which makes the function of instructions clear and intuitive. CHLA can be applied to multiple architectures allowing the programmer to use a single syntax across many platforms and relieving them of the task of remembering hundreds of 2-4 letter mnemonics.

An assembler and disassembler were developed. The assembler allows translation between CHLA and native assembly. The disassembler allows translation from binary to CHLA. Specification Documents are used to inform the assembler and disassembler on how to translate between the native assembly and binary to CHLA.

Specification Documents contain numerous records. Each record describes a translation between a CHLA statement and an assembly instruction. Equivalent statements are multiple CHLA statements mapping to the same assembly instruction. Equivalent statements allow for both flexible and generic syntax as well as convenient and intuitive syntax. Specification documents were developed for two distinct architectures: AVR ATMega64 and HCS08.

Three test programs were developed to test correctness and consistency/reuse. Two of the test programs are provided as a binary files and contain every possible instruction for the ATMega64 and HCS08 respectively. These provably show the correctness of the translation through the disassembler to CHLA and then through the assembler back to assembly. The third test program is written in CHLA and involves conditional definitions to demonstrate that it is possible for a single CHLA program to target multiple architectures. This cross-architecture example program demonstrates the syntax consistency and instruction reuse possible in CHLA.

## Future Work

This project has demonstrated the merits of a generic high level assembly language. However, before a generic high level assembly language could see widespread real world use it would require data segment support and a more extensive and formalised design specification format.

Data segment support would require significant work. The CHLA assembler needs to be aware of the native assembler data segment directives. The disassembler would need to understand the memory organisation of each architecture so that it can differentiate between the code segment and the data segment.

The specification document would need to be formalised and extended to support all possible architectures. An additional field should be added describing a recommended value range in addition to the legal range. For example, a two byte add instruction would have a recommended range of 256-65535 in addition to its legal range of 0-65535 because a one byte add instruction should be used for values in the 0-255 range. There should be a field specifying the number representation of an operand. For example the ATMega64 instruction set makes use of three different number representations: Unsigned, Two’s Complement and Excess-N. As described in section 8.3, a more advanced opcode description is required to allow for out of order operand bits.

# References

[1] M. V. Wilkes, D. J. Wheeler, and S. Gill, *The preparation of programs for an electronic digital computer*, 1984.

[2] D. Salomon, *Assemblers and loaders*: Ellis Horwood, 1992.

[3] S. P. a. G. L. Mitchell, "SOAP, IBM 650 Symbolic Optimal Assembly Program," 1955.

[4] D. Padua, "The fortran I compiler," *Computing in Science & Engineering,* vol. 2, pp. 70-75, 2000.

[5] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick*, et al.*, "The FORTRAN automatic coding system," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, 1957, pp. 188-198.

[6] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM,* vol. 3, pp. 184-195, 1960.

[7] B. W. Kernighan and D. M. Ritchie, *The C programming language*: Prentice Hall, 1988.

[8] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating system concepts* vol. 4: Addison-Wesley, 1998.

[9] R. Hyde, *The art of assembly language*. San Francisco :: No Starch Press, 2003.

[10] A. M. Devices, "AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions," ed, 2012.

[11] M. M. Kessler, *Implementation of Macros to Permit Structured Programming in OS/360*: Federal Systems Division, International Business Machines Corporation, 1970.

[12] A. Kuketayev, "The Data Abstraction Penalty (DAP) Benchmark for Small Objects in Java," ed, 2001.

[13] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE computer,* vol. 15, pp. 8-21, 1982.

[14] I. B. M. Corporation, "IBM 704 Programmers Model," ed, 1954.

[14] Louden Compilers Text Book

[15] floating addresses pdf

[16] Intel Hex

[17] SREC pdf

[18] <http://tools.ietf.org/html/rfc4180>

[19] atmega64 reference manual

[20] hcs08 reference manual

1. Code Listing
   1. Keywords Class

import inspect

class Keyword:

"""Abstract representation of a CHLA keyword."""

def \_\_init\_\_( self, name ):

self.\_name = name

def \_\_str\_\_( self ):

return self.\_name

def \_\_repr\_\_( self ):

return "Keyword: {0}".format( self.\_name )

def match( self, other ):

if isinstance( other, Keyword ):

return self.\_name == other.\_name

return False

def \_\_key( self ):

return ( self.\_name )

def \_\_hash\_\_( self ):

return hash( self.\_\_key() )

def asm( self ):

return self.\_name

def chla( self ):

return self.\_name

class Keywords:

SREG = Keyword( "sreg" )

TRANSFER = Keyword( "t" )

SWAP = Keyword( "swap" )

IF = Keyword( "if" )

GOTO = Keyword( "goto" )

SKIP = Keyword( "skip" )

RET = Keyword( "ret" )

RETI = Keyword( "reti" )

POP = Keyword( "pop" )

PUSH = Keyword( "push" )

MEM = Keyword( "mem" )

PMEM = Keyword( "pmem" )

CARRY = Keyword( "c" )

X = Keyword( "x" )

END = Keyword( "end" )

SP = Keyword( "sp" )

HX = Keyword( "h:x" )

HA = Keyword( "h:a" )

XA = Keyword( "x:a" )

NOP = Keyword( "nop" )

A = Keyword( "a" )

H = Keyword( "h" )

INTERRUPT = Keyword( "i" )

def \_\_iter\_\_(self):

return ( v for k, v in inspect.getmembers( self ) if not k.startswith( '\_\_' ) )

* 1. Operators Class

import inspect

class Operator:

"""Abstract representation of CHLA operator."""

def \_\_init\_\_( self, name ):

self.\_name = name

def \_\_str\_\_( self ):

return self.\_name

def \_\_repr\_\_( self ):

return "Operator: {0}".format( self.\_name )

def match( self, other ):

if isinstance( other, Operator ):

return self.\_name == other.\_name

return False

def \_\_key( self ):

return ( self.\_name )

def \_\_hash\_\_( self ):

return hash( self.\_\_key() )

def asm( self ):

return self.\_name

def chla( self ):

return self.\_name

class Operators:

PLUS\_EQ = Operator( "+=" )

PLUS\_PLUS = Operator( "++" )

PLUS = Operator( "+" )

AND\_EQ = Operator( "&=" )

NOT\_EQ = Operator( "!=" )

NOT = Operator( "~=" )

EQ\_EQ = Operator( "==" )

SUB\_EQ = Operator( "-=" )

SUB\_SUB = Operator( "--" )

SUB = Operator( "-" )

EOR = Operator( "^=" )

EQ = Operator( "=" )

FMUL = Operator( "." )

MUL = Operator( "\*" )

ABS = Operator( "|" )

LSL = Operator( "<<" )

ASR = Operator( ">>>" )

LSR = Operator( ">>" )

OR = Operator( "|=" )

OPEN\_BRACE = Operator( "[" )

OPEN\_BRACKET = Operator( "(" )

CLOSE\_BRACE = Operator( "]" )

CLOSE\_BRACKET = Operator( ")" )

PAIR = Operator( ":" )

COMMA = Operator( "," )

HASH = Operator( "#" )

GEQ = Operator( ">=" )

LEQ = Operator( "<=" )

LT = Operator( "<" )

GT = Operator( ">" )

AND = Operator( "&" )

DIV = Operator( "/" )

def \_\_iter\_\_(self):

return ( v for k, v in inspect.getmembers( self ) if not k.startswith( '\_\_' ) )

* 1. Assembler Class

from chla.Generator import \*

from chla.Tokeniser import \*

class Assembler:

def assemble( spec, source, out, bytesInWord, definitions ):

"""assemble( spec, source, out, bytesInWord, definitions ): takes a specification document and source file as arguments and assembles the program into the output file"""

# initialise tokeniser

t = Tokeniser( source, definitions )

# initialise parse tree

tree = Generator.generate\_tree( spec, t )

# read in program

\_,program = t.tokenise()

# first pass - calculate labels

statements = {}

byteLabels = {}

nameLabels = {}

wordCounter = 0

for byte in sorted( program.keys() ):

# get statement

chla = program[ byte ]

# match statement to assembly instruction

template = Assembler.match( tree, chla )

if template:

# fill in operands with values

statements[ wordCounter ] = template.statement\_from\_chla( chla )

# increment byte counter

wordCounter += int( len( template.opcode() ) / ( 8 \* bytesInWord ) )

# not a statement, check if label

elif isinstance( chla[0], Label ) and chla[1] == Operators.PAIR:

# get all labels at that byte address

labels = byteLabels.get( wordCounter, list() )

labels.append( chla )

# save address

byteLabels[ wordCounter ] = labels

nameLabels[ chla[0].name() ] = wordCounter

# no match

else:

raise ValueError( "ERROR byte offset {0}: Could not match '{1}'".format( byte, " ".join( [ repr( t ) for t in chla ] ) ) )

# second pass - fill in label addresses

# we stop if we reach this address

maxByte = sorted( list( set ( list( statements.keys() ) + list( byteLabels.keys() ) ) ) )[-1]

b = 0

while b <= maxByte:

# get next statement

if statements.get( b ):

s = statements[b]

else:

b += 1

continue

# get the range and value for the label operand

labelRange,label = s.operand( 'l' )

if label:

# absolute address

if s.instruction() == Instructions.JMP or s.instruction() == Instructions.CALL or s.instruction() == Instructions.JSR:

addressValue = nameLabels[ label.name() ]

# relative address

else:

addressValue = nameLabels[ label.name() ] - ( b + 1 )

# update the label's value to the new address

label.update\_value( addressValue )

# new value maybe outside range

if not labelRange.check( label ):

# try upgrade 0-255 jump to 0-65535 jump if possible

# this could not be done earlier as jump address was not known

template = Assembler.match( tree, s.chla() )

if template:

# fill in operands with values

statements[b] = template.statement\_from\_chla( s.chla() )

# re-check range

labelRange,label = statements[b].operand( 'l' )

if labelRange.check( label ):

# must relocate all bytes after this point

mb = sorted( list( set ( list( statements.keys() ) + list( byteLabels.keys() ) ) ) )[-1]

maxByte = mb + 1

while mb > b:

if byteLabels.get( mb ):

byteLabels[ mb + 1 ] = byteLabels[ mb ]

del byteLabels[ mb ]

if statements.get( mb ):

statements[ mb + 1 ] = statements[ mb ]

del statements[ mb ]

mb -= 1

b += 1

continue

raise ValueError( "Byte Offset {0} Label '{1}' with value {2} out of range {3} in line {4}".format( b, l.name(), value, repr( r ), repr( s ) ) )

# increment byte counter

b += 1

# output the assembly

f = open( out, "w" )

keys = sorted( list( set ( list( statements.keys() ) + list( byteLabels.keys() ) ) ) )

for byteNo in keys:

# Write label

if byteLabels.get( byteNo ):

f.write( "\n" )

# write each label at this address

for label in byteLabels.get( byteNo ):

for token in label:

f.write( token.asm() )

f.write( "\n" )

# Write each statement at this address

if statements.get( byteNo ):

# Write ASM

asm = statements[ byteNo ].asm\_string()

# Write chla comment

chla = ( ( 4 - int( len( asm ) / 8 ) ) \* "\t" ) + "; " + statements[ byteNo ].chla\_string()

# Write line

f.write( asm + chla + "\n" )

# close file

f.close()

def match( node, chla ):

"""recursively matches chla against the parse tree"""

if len( chla ) == 0:

# no more tokens to match

# Return the leaf node if it exists

return node.get( Keywords.LEAF )

for n in node:

# try all branches at current node

if n.match( chla[0] ):

# next token matches

# continue down branch

r = Assembler.match( node[n], chla[1:] )

if r:

# successfully matched statement at leaf

return r

return None

1. Miscellaneous
   1. ATMega64 Exhaustive Testing Diff Output

4c4

< adiw r26:r24,1

---

> ADIW r25:r24 , 1

6c6

< andi r16,240

---

> ANDI r16 , 0xF0

10,11c10,11

< cpi r16,240

< subi r16,1

---

> CPI r16 , 0xF0

> DEC r16

17c17

< add r16,r16

---

> LSL r16

24c24

< ori r16,240

---

> ORI r16 , 0xF0

26,27c26,27

< sbci r16,240

< sbiw r26:r24,1

---

> SBCI r16 , 0xF0

> SBIW r25:r24, 1

29c29

< subi r16,240

---

> SUBI r16 , 0xF0

36c36

< adc r16,r16

---

> ROL r16

41,42c41,42

< l0:

< brbs 1,l0

---

> lbreq:

> BREQ lbreq

44,45c44,45

< l1:

< brbc 4,l1

---

> lbrge:

> BRGE lbrge

47,48c47,48

< l2:

< brlt l2

---

> lbrlt:

> BRLT lbrlt

50,51c50,51

< l3:

< brbs 0,l3

---

> lbrlo:

> BRLO lbrlo

53,54c53,54

< l4:

< brne l4

---

> lbrne:

> BRNE lbrne

56,57c56,57

< l5:

< brbs 2,l5

---

> lbrmi:

> BRMI lbrmi

59,60c59,60

< l6:

< brpl l6

---

> lbrpl:

> BRPL lbrpl

62,63c62,63

< l7:

< brsh l7

---

> lbrsh:

> BRSH lbrsh

65,66c65,66

< l8:

< brne l8

---

> lbrbc:

> BRBC 1 , lbrbc

68,71c68,69

< l9:

< brbs 1,l9

< rcall l10

< rcall l11

---

> lbrbs:

> BRBS 1 , lbrbs

73,74c71,75

< l10:

< l11:

---

>

> CALL subr

> RCALL subr

>

> subr:

76,77c77,79

< ldi r31,0

< ldi r30,52

---

>

> ldi r31, HIGH(subr)

> ldi r30, LOW(subr)

79,80c81,83

< ldi r31,0

< ldi r30,59

---

>

> ldi r31, HIGH(lijmp)

> ldi r30, LOW(lijmp)

81a85,86

> lijmp:

>

84,85c89,94

< l12:

< jmp l12

---

> ljmp:

> JMP ljmp

>

> lrjmp:

> RJMP lrjmp

>

87,88d95

< l13:

< rjmp l13

92a100

>

95c103

< movw r26:r24,r31:r30

---

> MOVW r25:r24 , r31:r30

96a105

>

98a108

>

101,112c111,123

< ld r16,r27:26

< ld r16,r27:26+

< ld r16,-r27:26

< ldd r16,r29:r28+0

< ld r16,r29:r28+

< ld r16,-r29:r28

< ldd r16,r29:r28+1

< ld r16,r31:r30

< ld r16,r31:r30+

< ld r16,-r31:r30

< ldd r16,r31:r30+1

< ldi r16,255

---

>

> ld r16 , X

> ld r16 , X+

> ld r16, -X

> ld r16 , Y

> ld r16, Y+

> ld r16, -Y

> ldd r16, Y + 1

> ld r16, Z

> ld r16, Z+

> ld r16 , -Z

> ldd r16 , Z + 1

> ldi r16 , 0xFF

116,117c127,130

< lpm r16,r31:r30

< lpm r16,r31:r30+

---

> lpm r16, Z

> lpm r16,Z+

>

>

119,129c132,145

< st r27:26,r16

< st r27:26+,r16

< st -r27:26,r16

< st r29:r28,r16

< st r29:r28+,r16

< st -r29:r28,r16

< sts 127,r16

< st r31:r30,r16

< st r31:r30+,r16

< st -r31:r30,r16

< sts 119,r16

---

> st X, r16

> st X+, r16

> st -X, r16

>

> st Y, r16

> st Y+, r16

> st -Y, r16

> std Y + 63, r16

>

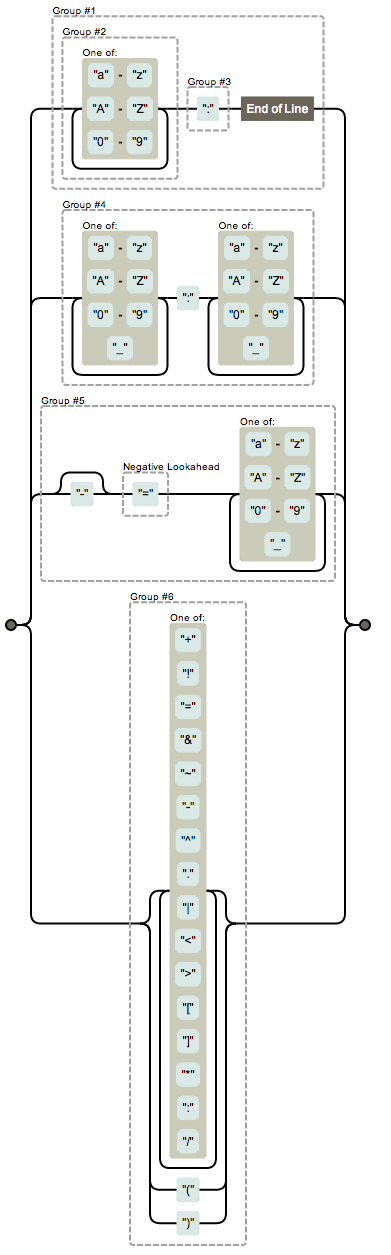
> st Z, r16

> st Z+, r16

> st -Z, r16

> std Z + 63, r16

* 1. Tokenisation Regular Expression



1. Test Programs
   1. ATMega64 Exhaustive Test Program Hex File

:020000020000FC

:100000000000011F010F01960123007F00950117D9

:100010000107003F0A95012709038103890303951E

:10002000000F0695019F010201030195012B006F4E

:10003000010B004F0197011B005F0595989401F992

:10004000189411FBD998001F0795D99A0295F9F3D6

:10005000FCF7FCF3F8F3F9F7FAF3FAF7F8F7F9F726

:10006000F9F30E94340000D00895F0E0E4E309952C

:10007000F0E0EBE3099401130C943C00FFCFC99925

:10008000C99B01FD01FF09B3012FCF011ABB0F91DD

:100090001F938895A8950C910D910E910881099157

:1000A0000A91098100810191029101810FEF009174

:1000B00018000091FEFFC89504910591E8950C93F6

:1000C0000D930E93088309930A930FAF0083019356

:0C00D000029307AF00937F000093FEFF37

:00000001FF

* 1. HCS08 Exhaustive Test Program SREC File

S0170000616C6C5F696E737472756374696F6E732E616273B7

S123E000450140949AA919B910C90100D90100E910F99ED901009EE910AB02BB10CB0100D4

S123E020DB0100EB10FB9EDB01009EEB10A77FAF7FA4FFB410B410C40100D40100E410F4F6

S123E0409ED401009EE410371047576710779E671024FE131025FE27FE90FE92FE28FE29D5

S123E060FE22FE24FE2FFE2EFEA5FFB510C50100D50100E510F59ED501009EE51093FE2557

S123E080FE23FE91FE2CFE2BFE2DFE26FE2AFE20FE0310FD0210FD1210ADFE3110FD41FF7C

S123E0A0FD51FFFD6110FD6100FD9E6110FC989A3F104F5F8C6F107F9E6F10A1FFB110C143

S123E0C00100D10100E110F19ED101009EE110331043536310739E63103E010065FF4A7556

S123E0E0109EF310A3FFB310C30100D30100E310F39ED301009EE3103B10FD4BFE5BFE6B30

S123E10010FD7BFE9E6B10FC3A104A5A6A107A9E6A1052A8FFB810C80100D80100E810F813

S123E1209ED801009EE8103C104C5C6C107C9E6C10CCE131DCE131FCCDE138DDE138DDE166

S123E14038FDA6FFB610C60100D60100E610F69ED601009EE61045FFFF55103201009EAE61

S123E1609EBE01009ECE109EFE10AEFFBE10CE0100DE0100EE10FE9EDE01009EEE10381090

S123E18048586810789E6810341044546410749E64104E10115E106EFF107E104230104055

S123E1A0506010709E60109D62AAFFBAFFCA0100DA0100EA10FA9EDA01009EEA10878B8976

S123E1C0868A88391049596910799E6910361046566610769E66108081A2FFB210C20100A6

S123E1E0D20100E210F29ED201009EE210999BB710C70100E710F79ED701009EE71035FF73

S123E2009601009EFF10BF10CF0100DF0100EF10FF9EDF01009EEF10A0FFB010C00100E01E

S11CE22010F09ED001009EE0108497853D104D5D6D107D9E6D10959F9470

S105FFFEE0001D

S9030000FC

* 1. Cross-Architecture CHLA Test Program

#define GREEN\_BIT 0

#define YELLOW\_BIT 1

#define RED\_BIT 2

#define GREEN\_STATE (1<<GREEN\_BIT)

#define YELLOW\_STATE (1<<YELLOW\_BIT)

#define RED\_STATE (1<<RED\_BIT)

#define INTERRUPT\_LOW (int(20000/8)&0x00FF)

#define INTERRUPT\_HIGH ((int(20000/8)&0xFF00)>>8)

#ifdef AVR

#include "test/atmega64.def"

#define STATE r16

#define COUNTER r17

#define TEMP r18

#else

#include "test/hcs08.def"

#define STATE 16

#define COUNTER A

#define TEMP X

#endif

#define OUTPUT PORTA

setup:

TEMP = |0x07|

DDRA = TEMP

#ifdef AVR

TEMP = |0x00|

TCCR1A = TEMP

TEMP = |0x0A|

TCCR1B = TEMP

TEMP = |INTERRUPT\_HIGH|

OCR1AH = TEMP

TEMP = |INTERRUPT\_LOW|

OCR1AL = TEMP

TEMP = |0x10|

TIMSK = TEMP

TEMP = |3|

MEM[ EICRA ] = TEMP

TEMP = |1|

EIMSK = TEMP

SREG [ I ] = | 1 |

#else

TEMP = |0x17|

SRTISC = TEMP

TEMP = |0x12|

IRQSC = TEMP

#endif

COUNTER = |0|

STATE = |GREEN\_STATE|

busy:

goto busy

button:

if STATE[GREEN\_BIT] == |1| skip

goto exit

COUNTER <= |0|

if != goto exit

STATE = |YELLOW\_STATE|

COUNTER = |3|

#ifndef AVR

IRQSC[2] = |1|

#endif

goto exit

timer:

if STATE[GREEN\_BIT] == |0| skip

goto green

if STATE[YELLOW\_BIT] == |0| skip

goto yellow

if STATE[RED\_BIT] == |0| skip

goto red

yellow:

OUTPUT = STATE

COUNTER <= |0|

if == goto yellowNext

yellowDec:

COUNTER -= |1|

goto exit

yellowNext:

STATE = |RED\_STATE|

COUNTER = |10|

goto exit

red:

OUTPUT = STATE

COUNTER <= |0|

if == goto redNext

redDec:

COUNTER -= |1|

goto exit

redNext:

STATE = |GREEN\_STATE|

COUNTER = |10|

goto exit

green:

OUTPUT = STATE

COUNTER <= |0|

if == goto exit

COUNTER -= |1|

goto exit

exit:

reti