

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

Insert abstract here.

**Contents**

Acknowledgments v

Abstract vii

Chapter 1: Introduction 12

1.1 Introduction and Motivation 12

1.2 Contents of Report 13

Chapter 2: Background 14

2.1 Early Assembly Languages and Assemblers 14

2.2 High Level Languages 15

2.3 Modern Assembly Languages 17

2.4 High Level Languages vs. Assembly Languages 19

2.5 High Level Assembly 20

2.6 Regular Expressions 22

2.7 Scanning 22

2.8 Decision Tree 23

2.9 Multi-pass Assembler 23

2.10 Intel Hex Format 24

2.11 SREC Format 25

2.12 CSV Format 26

2.13 AVR ATMega64 27

2.14 Freescale HCS08 28

Chapter 3: Research Plan 29

3.1 Project Aims 29

3.2 Requirements 30

Chapter 4: Design Overview 33

4.1 Initial Design 33

4.2 Specification Document Centric Design 33

Chapter 5: CHLA Specification Design 34

5.1 Overview 34

5.2 No Ambiguity and Generality 34

5.3 Assembly and CHLA 34

5.4 Operands 34

5.5 Equivalent Statements 34

Chapter 6: Implementation Design 35

6.1 Overview 35

6.2 Initial Approach 35

6.3 Tokenisation 35

6.4 Decision Tree Generation 35

6.5 Preprocessing 35

6.6 Matching 35

6.7 Addresses, Labels and Jumps 35

6.8 Output 35

Chapter 7: Testing and Analysis 36

7.1 Test Programs 36

7.2 Ambiguity 36

7.3 Correctness 36

7.4 Consistency 36

7.5 Reuse 36

Chapter 8: Evaluation 37

8.1 Non-Mathematical Instructions 37

8.2 Extensibility 37

8.3 Readability 37

8.4 Use as a Learning Tool 37

Chapter 9: Conclusions 38

9.1 Future Work 38

Chapter 10: References 39

**List of Figures**

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

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

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

Figure IV Intel Hex Record Format xxiv

# 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 looking it up, especially since the assembly languages are very inconsistent.

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

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

Figure : An example assembly instruction constisting 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 relabel 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 details 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.

RISC (Reduced Instruction Set Computing) 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 like 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

A decision tree can be used for parsing

## 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].

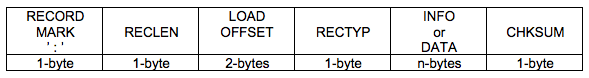


Figure Intel Hex Record Format

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 there 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 53 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.

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 and thus, a programmer should be able to read and write programs written for different architectures in the same syntax even though the programs would 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

A syntax for the 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 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.

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 distinct 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 line was read in as CHLA, processed and written out as assembly based some rules without storing it.

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 is provided with the language grammar and some basic constructs, and it produces code to parse the source code. 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 both are 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 program 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.

# CHLA 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 between 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 | Values are assumed to be 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.

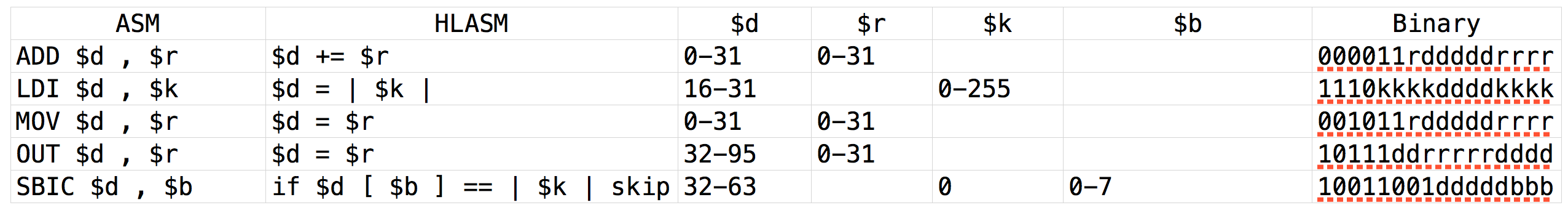


Figure Example Extract from Specification Document

## No Ambiguity and Generality

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

DDRA = r16

Scenario two is concerned with CHLA 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 one 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).

## 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 exist for every record. The operand fields must exist if that operand is found in either the CHLA or assembly. The operand field must exist if the instruction is to be disassembled (See Note 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. 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 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 value is multiplied by two, and then for both singles and pairs the binary value is offset by the first value in the range.

The choice of which of the five placeholders to choose is completely arbitrary. However, many instruction set manuals use the characters ‘d’ and ‘r’ for destination and source registers, ‘k’ for constant values, ‘b’ to represent bit positions and ‘l’ for labels. 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

# Implementation Design

## Overview

## Initial Approach

## Tokenisation

## Decision Tree Generation

## Preprocessing

## Matching

## Addresses, Labels and Jumps

## Output

# Testing and Analysis

## Test Programs

## Ambiguity

Inc vs add

## Correctness

Program assembles and disassembles correctly.

## Consistency

Same HLASM instruction does the same thing.

## Reuse

How much of the code can be shared among architectures

# Evaluation

## Non-Mathematical Instructions

Do not map to operators/operands. Just look like function calls. Not really scalable.

## Extensibility

Add more architectures -> better specification framework. Byte ordering and format

## Readability

Anecdotally better. More obvious what an instruction does. More verbose however,

## Use as a Learning Tool

# Conclusions

## Future Work

Formal spec:

* hints as to the recommended range (i.e. 256-65535 despite it supporting 0-255) (similar for add and sub with ‘1’ )

# References

[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