CoreASM Documentation

Documentation on the CoreASM compiler

Blake Loring

May 4, 2015

Introduction

In order to make building code for the CoreVM virtual machine easier a simple assembler was constructed.

Similarly to the VirtualMachine CoreASM supports an integer only syntax and instructions (no floats).

The assembler itself uses a LL(1) grammar and subsequently has a very simple parser which is able to generate machine code for CoreVM in a single pass.

Syntax

The syntax of the assembly language is very simple, supporting only labels, data declarations and instructions.

Labels can be used to reference a point in the file. If a label is referenced before it is used then the address will be resolved when the label is defined. This allows for blocks of code like

```
jmp start
start:
```

to be valid.

If a label is overwritten then any references to it after the point where it is redefined will reference the second label. This means that for code such as

```
jmp start

start:
load A 5
start:
jmp start
```

The second jump will jump to the second label, but the first jump will still jump to the first label.

Instructions follow the general pattern INSTRUCTION DEST OtherOperands, when applicable. With the conditional jump instructions such as JEQ and JNE they follow the pattern INSTRUCTION Register Operand (Register or value) DESTINATION.

The data instructions can be used to allocate space in the binary which can be used to store data at runtime using the GET and SET instructions. For example, the code

```
load A 5
set A variable
variable:
db 4
exit:
```

sets the value of the memory allocated at label variable to 5. This can then be loaded into registers later using the GET instructions.

Usage

The compiler only supports single files, and execution on the generated objects will begin from the first line of the file.

To compile an assembly file into CoreVM machine code the command coreasm INPUTFILE OUTPUTFILE should be invoked.

Implementation

The implementation of the CoreASM parser is split into three core sections. The Tokenizer handles the conversion of portions of the input strings into defined tokens. The parser takes the tokens and works out what they mean. Finally, the ByteBuffer takes the generated ByteCode, handling additions to it, and it's content its written out to a file after parsing.

Tokenizer

The tokenizer takes an array of characters as an input and breaks it into a set of Assembler::Token elements.

Parser

ByteBuffer

Grammar

TODO: Instruction Specifics (When Finalized).

```
NUM: [0-9]+
ID: [a-zA-Z][a-zA-Z0-9]*

Program: Block [Program]
Block: Label | Instruction | Data
Label: ID ':'
Data: 'db' NUM
Instruction:
Add | Sub | Mul | Div | Load | Move | Set | Get | JumpEqual | JumpNotEqual | Interrupt
```