# stoc

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# stoc

Stochastic superoptimiser targetting the 6502

We've got a few different search strategies implemented actually, and these exercise the emulator, equivalence tester and everything. Some assembly language files in `examples/` contain goofy code sequences that contain obvious inefficiencies. They are just there to demonstrate stoc.

To build the system, type `make`. For each architecture, (currently a few varieties of 6502) `make` will generate the appropriate source code and compile an executable named `stoc-$arch`.

### Supported architectures

So far, we've got a few varieties of 6502. These are:

- *stoc-6502* which is a generic NMOS 6502, including the jmp indirect bug, but does not use any of the illegal opcodes
- *stoc-6510*, another NMOS 6502, and has some of the same illegal opcodes that the Commodore 64 guys use
- *stoc-65c02*, targets the later CMOS chips with extra opcodes like `phx` and so on
- *stoc-2a03*, basically the same as stoc-6502 but has no decimal mode. Dead-code elimination here will remove instructions `sed` and `cld`.

The above list is essentially what's provided by the *fake6502* submodule. If you are interested in adding other architectures, I would suggest that the easiest way would be to graft in another emulator. At build-time, a particular emulator is linked in, and this is what determines which architecture the binary supports. A separate program is built for each supported architecture.

### Theory of operation

The basic idea with this is to generate better programs than traditional compilers can, by copying a working program and making many small random successive changes to it. If the copy is found to be equivalent (or close enough), then it might get written to the standard output. Otherwise, another attempt is made, until an improvement is found.

There are a few different ways we can introduce mutations into the program, and these have names such as Dead Code Elimination, or `.dce`, Stochastic Optimisation, or `.opt`, etc. They are described in more detail below.

How can we see if two programs are equivalent? We can spam them with random numbers, and then check whether they produce the same output. Earlier versions of stoc worked in this way, but there was a slight chance that the random numbers didn't exercise the entire program. This could lead to a buggy program being output. To mitigate this risk, I've introduced the concept of testcases. A testcase is partially derived from the RNG and partially derived from the reference program. A testcase specifies what output the program should yield given a specific input. stoc will remember a number of these, and test each putative program against them all. I would guess that 99% of the garbage produced by the search algorithm is caught by the first few testcases.

### Dead Code Elimination

This search strategy looks for a more optimal rewrite by selecting random instructions for deletion; up to five at a time (this is to give pairs of instructions, such as a `pha` and corresponding `pla`, a chance to get deleted together). If the program proves to be equivalent without the selected instructions, then the instructions are deleted and the same procedure is done again. Use this procedure by using the `.dce` action.

```
$ ./stoc-2a03 examples/add_two_constants.stoc .dis .dce
; starting at $2000
; 5 instructions
; 7 bytes
; 5 clockticks
    clc
    lda #$07
    sed
    clc
    adc #$05

; starting at $2000
; 3 instructions
; 5 bytes
; 18 clockticks
    lda #$07
    clc
    adc #$05
```

It might be worth noting that the input procedure above contains two instances of the `clc` instructions, and only one is needed. Either one may be deleted, and it is picked at random. Running the same program again might have yielded the instructions `clc` and `lda #$07` in a different order. The `sed` instruction is not needed at all on the 2A03 because this is a chip variant which lacks the decimal mode. On other varieties of the 6502, as emulated by stoc-6510 for example, the `sed` instruction will be deemed necessary by the equivalence tester.

### Stochastic optimisation

This search strategy walks around the search space by trying a number of mutations at a time, at sees if these mutations together either lower the cost or increase correctness (or both). If so, then the putative program (i.e. the one including the random mutations) replaces the current starting position, and another walk begins. I don't know if this one will prove promising or not. Here are the possible mutations it does:

- Insert a random instruction
- Delete an instruction at random
- Modify a random instruction's operand
- Change a random instruction's opcode for another one, having the same addressing mode
- Pick two random instructions and swap them over
- Pick one instruction, and overwrite it entirely with another one.

This will stop searching when the random walks stop finding improvements. I.e., if it's tried *n* times without finding a more optimal program, the search stops and the last found known good program is printed out. Invoke this search with the `.opt` action. So here is an example run:

```
$ ./stoc-2a03 examples/add_two_constants.stoc .dis .opt
; starting at $2000
; 5 instructions
; 7 bytes
; 5 clockticks
    clc
    lda #$07
    sed
    clc
    adc #$05

; starting at $2000
; 1 instructions
; 2 bytes
; 1 clockticks
    lda #$0c
```

There are a few ways to reach the second program from the first; each one is a random walkk through the search space:

- Maybe stoc putatively inserted `lda #$0c` at the end of the program, and then deleted the rest of the instructions, having established that they are effectively dead code.
- Maybe stoc putatively altered `lda #$07` to `lda #$0c`, and altered `adc #$05` to something benign, and then deleted the rest of the instructions, having established that they are effectively dead code.

Whatever the case, the search has discovered that adding two constants together is equivalent to loading the sum of those constants, and suggested a replacement program that does so.

# Chapter 2

# Data Structure Index

## 2.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

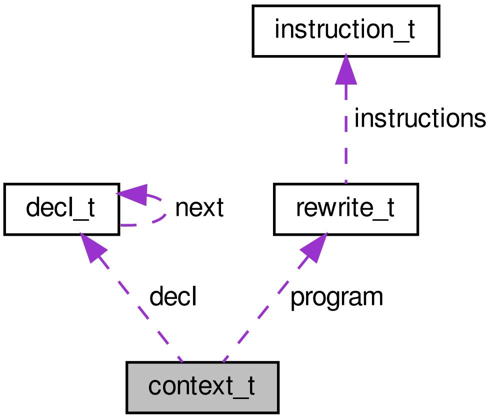
Here is a list of all documented files with brief descriptions:

# Chapter 4

# Data Structure Documentation

## 4.1 context_t Struct Reference

Collaboration diagram for context_t:



### Data Fields

- uint8_t **a**
- uint8_t **x**
- uint8_t **y**
- uint8_t **flags**
- uint8_t **s**
- uint16_t **pc**
- long long int **clockticks**
- rewrite_t **program**
- data_t **mem** [ADDR_SPACE]
- uint8_t **memf** [ADDR_SPACE]

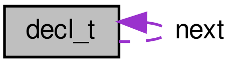
- uint16_t **ea**
- uint8_t **opcode**
- int **exitcode**
- struct _decl_t ∗ **decl**

The documentation for this struct was generated from the following file:

- stoc.h

## 4.2 decl_t Struct Reference

Collaboration diagram for decl_t:



### Data Fields

- int(∗ **fn** )(context_t ∗c, struct _decl_t ∗d, uint8_t ∗∗scram)
- int(∗ **setup** )(context_t ∗c, struct _decl_t ∗d, uint8_t ∗∗scram)
- char **label** [LABEL_LEN]
- uint16_t **start**
- unsigned int **length**
- struct _decl_t ∗ **next**

The documentation for this struct was generated from the following file:

- decl.h

## 4.3 instruction_t Struct Reference

### Data Fields

- addr_t **address**
- uint8_t **opcode**
- uint16_t **operand**

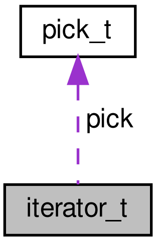
The documentation for this struct was generated from the following file:

- stoc.h

## 4.4 iterator_t Struct Reference

```
#include <pick.h>
```

Collaboration diagram for iterator_t:



### Data Fields

- pick_t * pick

  *Pointer to the pick_t object over which we're iterating.*
- int current

  *Current offset.*

### 4.4.1 Detailed Description

An iterator

This one is for iterating over a pick_t.

The documentation for this struct was generated from the following file:

- pick.h

## 4.5 pick_t Struct Reference

```
#include <pick.h>
```

### Data Fields

- int count

  *Number of members in the set.*
- uint16_t vals [MAXPICKSIZE]

  *the members*

### 4.5.1 Detailed Description

A set of values.

This is intended to be a set, which we can pick a number at random from, or which we can iterate over

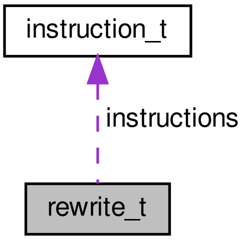
The documentation for this struct was generated from the following file:

- pick.h

## 4.6 rewrite_t Struct Reference

```
#include <stoc.h>
```

Collaboration diagram for rewrite_t:



### Data Fields

- uint16_t org

  *Where the rewrite starts.*
- uint16_t length

  *Number of instructions in the rewrite.*
- uint16_t end

  *The first address after the last instruction.*
- instruction_t instructions [REWRITE_LEN]

  *Array of instructions.*
- long double fitness

  *Fitness, or "how correct is the rewrite".*
- long double mcycles

  *machine cycles, or "how long does the program take"*
- int blength

  *The program's length, in bytes.*

### 4.6.1 Detailed Description

rewrite_t

A rewrite is a list of instructions, plus associated data.

The documentation for this struct was generated from the following file:

- stoc.h

# Chapter 5

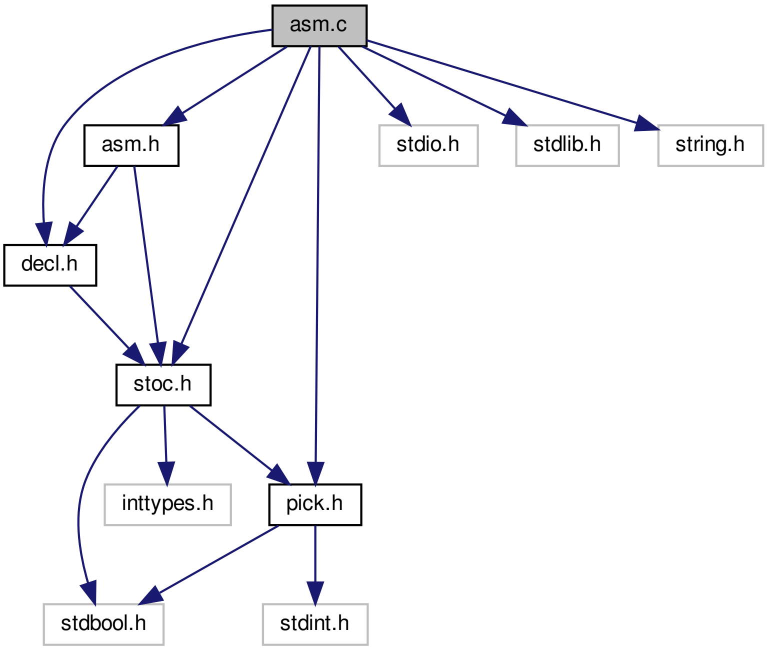
# File Documentation

## 5.1 asm.c File Reference

Parser for .stoc files.

```
#include "asm.h"
#include "decl.h"
#include "pick.h"
#include "stoc.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Include dependency graph for asm.c:



## Macros

- #define ASMBUFLEN 255

  *Maximum length for each line.*
- #define DELIM "\n\t "

  *Whitespace, used for delimiting tokens, passed to strtok.*

## Functions

- void **endofline** ()
- decl_t ∗ **parse_register_in** (int(∗fn)(context_t ∗c, decl_t ∗d, uint8_t ∗∗scram))
- decl_t ∗ **parse_register_out** (int(∗fn)(context_t ∗c, decl_t ∗d, uint8_t ∗∗scram), int(∗setup)(context_t ∗c, decl_t ∗d, uint8_t ∗∗scram))
- decl_t ∗ **parse_stack_in** ()
- decl_t ∗ **parse_stack_out** ()
- uint8_t **parse_byte** (char ∗t)
- decl_t ∗ **parse_run** (rewrite_t ∗r)
- decl_t ∗ **parseline** (char ∗line, context_t ∗c)
- void readfile (char ∗filename, context_t ∗reference)

### 5.1.1 Detailed Description

Parser for .stoc files.

### 5.1.2 Function Documentation

#### 5.1.2.1 readfile()

```
void readfile (
            char * filename,
            context_t * reference )
```

Load a .stoc file

**Parameters**

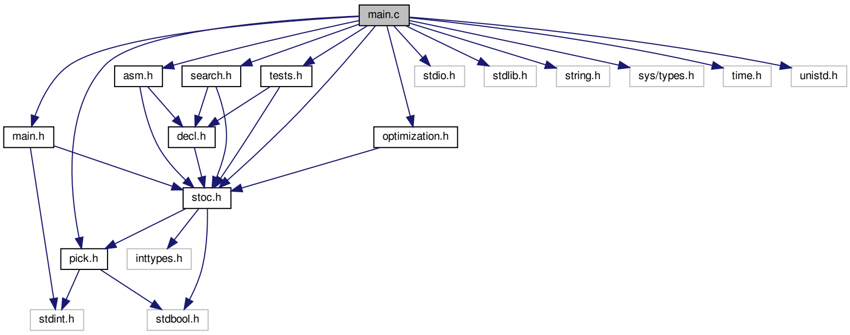
| | |
|---|---|
| *filename* | The name of the file to load |
| *reference* | A pointer to the context_t to load the file into |

## 5.2 main.c File Reference

Main file for stoc.

```
#include "main.h"
#include "asm.h"
#include "optimization.h"
#include "pick.h"
#include "search.h"
#include "stoc.h"
#include "tests.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
```

Include dependency graph for main.c:



## Functions

- uint8_t **mem_read** (context_t ∗c, uint16_t address)
- void **mem_write** (context_t ∗c, uint16_t address, uint8_t val)
- void **hexdump** (context_t ∗c)
- void **help** (char ∗progname)
- void **parseoption** (char ∗opt, char ∗progname)
- void **actions** (char ∗opt, context_t ∗c, char ∗progname)
- int main (int argc, char ∗∗argv)

  *The program's enntrypoint.*

### 5.2.1 Detailed Description

Main file for stoc.

# Index