SigComp Torture Tests
draft-ietf-rohc-sigcomp-torture-tests-02.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on April 1, 2006.

Copyright Notice

Copyright (C) The Internet Society (2005).

Abstract

This document provides a set of "torture tests" for implementers of the SigComp protocol. The torture tests check each of the SigComp Universal Decompressor Virtual Machine instructions in turn, focusing in particular on the boundary and error cases that are not generally encountered when running well-behaved compression algorithms. Tests are also provided for other SigComp entities such as the dispatcher and the state handler.
Change history

Changes relative to <draft-ietf-rohc-sigcomp-torture-tests-01.txt>:
1. Removed use of ‘!’ from mnemonic code and updated Appendix byte code
2. Added SMS, DMS, CPB needed for tests - changed SHA-1 so that it only needs 16CPB
3. Corrected minor typos
4. Added NACK reason codes to decompression failures in Appendix

Changes relative to <draft-ietf-rohc-sigcomp-torture-tests-00.txt>:
1. Modified COPY, COPY-LITERAL and COPY-OFFSET, state creation, state access and input beyond message tests.
2. Added extra explanation of tests
3. Added new infinite loop test.

Changes relative to <draft-price-rohc-sigcomp-torture-tests-02.txt>:
1. Changed name to draft-ietf-rohc-sigcomp-torture-tests-00.txt
2. Addition of test for state creation and self modifying code.
3. Addition of test for reference of static dictionary RFC-3485 [3].

Changes relative to <draft-price-rohc-sigcomp-torture-tests-01.txt>:
1. Added a test for the SigComp dispatcher (covering the case where input is requested that lies beyond the end of a message).
2. Fixed a typo in the input for Section 2.16.

Changes relative to <draft-price-rohc-sigcomp-torture-tests-00.txt>:
1. Added tests for the SigComp dispatcher (covering the SigComp Useful Values, the SigComp header for message-based transports, and the record marking scheme for stream-based transports).
2. Added tests for the SigComp state handler (covering the SigComp feedback mechanism, the state memory management and the interaction between multiple compartments).
3. Updated the cost of the sorting instructions based on the new values used in SigComp RFC-3320 [2].
4. Updated the stack manipulation test to work correctly when the decompression_memory_size is only 2048 bytes.
# Table of Contents

1. Introduction ........................................... 5
2. Torture tests for UDVM ................................. 5
   2.1 Bit manipulation .................................. 5
   2.2 Arithmetic ....................................... 7
   2.3 Sorting .......................................... 8
   2.4 SHA-1 ........................................... 9
   2.5 LOAD and MULTILOAD ............................... 11
   2.6 COPY ............................................ 13
   2.7 COPY-LITERAL and COPY-OFFSET .................... 15
   2.8 MEMSET .......................................... 16
   2.9 CRC ............................................. 17
   2.10 INPUT-BITS ...................................... 18
   2.11 INPUT-HUFFMAN .................................. 20
   2.12 INPUT-BYTES .................................... 21
   2.13 Stack manipulation ............................... 23
   2.14 Program flow .................................... 25
   2.15 State creation ................................... 26
   2.16 STATE-ACCESS ................................... 29
3. Torture tests for dispatcher ........................... 31
   3.1 Useful Values .................................... 31
   3.2 Cycles checking ................................... 34
   3.3 Message-based transport ........................... 35
   3.4 Stream-based transport ............................. 37
   3.5 Input past the end of a message .................. 39
4. Torture tests for state handler ....................... 41
   4.1 SigComp feedback mechanism ........................ 41
   4.2 State memory management ........................... 44
   4.3 Multiple compartments ............................. 48
   4.4 Accessing RFC 3485 State ........................ 52
   4.5 Bytecode state creation ........................... 53
5. Security considerations ................................ 56
6. Acknowledgements ....................................... 56
7. References ............................................. 56
   Authors’ Addresses .................................. 57
A. UDVM bytecode for the torture tests ................ 57
   A.1 Instructions ....................................... 57
      A.1.1 Bit manipulation .............................. 57
      A.1.2 Arithmetic .................................... 58
      A.1.3 Sorting ...................................... 58
      A.1.4 SHA-1 ........................................ 59
      A.1.5 LOAD and MULTILOAD ........................... 59
      A.1.6 COPY ......................................... 60
      A.1.7 COPY-LITERAL and COPY-OFFSET ............... 60
      A.1.8 MEMSET ....................................... 61
      A.1.9 CRC .......................................... 61
      A.1.10 INPUT-BITS ................................... 61
1. Introduction

This document provides a set of torture tests for implementers of the SigComp protocol RFC-3320 [2]. The idea behind SigComp is to standardize a Universal Decompressor Virtual Machine (UDVM) that can be programmed to understand the output of many well-known compressors including DEFLATE and LZW. The bytecode for the chosen decompressor is uploaded to the UDVM as part of the SigComp message flow.

The SigComp User Guide [1] gives examples of a number of different algorithms that can be used by the SigComp protocol. However, the bytecode for the corresponding decompressors is relatively well behaved and does not test the boundary and error cases that may potentially be exploited by malicious SigComp messages.

This document is divided into a number of sections, each containing a piece of code designed to test a particular function of one of the SigComp entities (UDVM, dispatcher and state handler). The specific boundary and error cases tested by the bytecode are also listed, as are the output the code should produce and the number of UDVM cycles that should be used.

Each test runs in the SigComp minimum decompression memory size (that is 2K), within the minimum number of cycles per bit (that is 16) and in tests where state is stored 2K state memory size is needed.

2. Torture tests for UDVM

The following sections each provide code to test one or more UDVM instructions. In the interests of readability the code is given using the SigComp assembly language: a description of how to convert this assembly code into UDVM bytecode can be found in the SigComp User Guide [1].

The raw UDVM bytecode for each torture test is given in Appendix A.

Each section also lists the number of UDVM cycles required to execute the code. Note that this figure only takes into account the cost of executing each UDVM instruction (in particular it ignores the fact that the UDVM can gain extra cycles as a result of inputting more data).

2.1 Bit manipulation

This section gives assembly code to test the AND, OR, NOT, LSHIFT and RSHIFT instructions. When the instructions have a multitype operand the code tests the case where the multitype contains a fixed integer value, and the case where it contains a memory address at which
2-byte operand value can be found. In addition the code is designed to test that the following boundary cases have been correctly implemented:

1. The instructions overwrite themselves with the result of the bit manipulation operation, in which case execution continues normally.

2. The LSHIFT or RSHIFT instructions shift bits beyond the 2-byte boundary, in which case the bits must be discarded.

3. The UDVM registers byte_copy_left and byte_copy_right are used to store the results of the bit manipulation operations. Since no byte copying is taking place these registers should behave in exactly the same manner as ordinary UDVM memory addresses.

at (64)
:a               pad (2)
:b               pad (2)

at (128)
JUMP (start)    ; Jump to address 255

at (255)
:start

; The multitypes are values
AND ($start, 21845)     ; 448 & 21845 = 320 = 0x0140
OR ($a, 42)             ; 0 | 42 = 42 = 0x002a
NOT ($b)                ; ~0 = 65535 = 0xffff
LSHIFT ($a, 3)          ; 42 << 3 = 336 = 0x0150
RSHIFT ($b, 65535)      ; 65535 >> 65535 = 0 = 0x0000
OUTPUT (64, 4)          ; Output 0x0150 0000

; The multitypes are references
AND ($a, $start)        ; 336 & 320 = 320 = 0x0140
OR ($a, $a)             ; 320 | 320 = 320 = 0x0140
NOT ($a)                ; ~320 = 65215 = 0xfefb
LSHIFT ($b, $a)         ; 0 << 65215 = 0 = 0x0000
RSHIFT ($a, $b)         ; 65215 >> 0 = 65215 = 0xfefb
OUTPUT (64, 4)          ; Output 0xfefb 0000
END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

The output of the code is 0x0150 0000 febf 0000. Executing the code costs a total of 22 UDVM cycles.

2.2 Arithmetic

This section gives assembly code to test the ADD, SUBTRACT, MULTIPLY, DIVIDE and REMAINDER instructions. The code is designed to test that the following boundary cases have been correctly implemented:

1. The instructions overwrite themselves with the result of the arithmetic operation, resulting in continuation as if the bytes were not bytecode.

2. The result does not lie between 0 and 2^16 - 1 inclusive, in which case it must be taken modulo 2^16.

3. The divisor in the DIVIDE or REMAINDER instructions is 0 (in which case decompression failure must occur).

at (64)

:a pad (2)
:b pad (2)
:type pad (1)
:type_lsb pad (1)

at (128)

INPUT-BYTES (1, type_lsb, decompression_failure)
SUBTRACT ($type, 1)
JUMP (start)
:decomp_failure
DECOMPRESSION-FAILURE

; Now the value in $type should be 0xffff, 0x0000 or 0x0001 according to whether the input was 0x00, 0x01 or 0x02.

at (255)

:start

; The multitypes are values
; For all three messages
; $start = 1728 (first 2 bytes of ADD instr)
ADD ($start, 63809) ; 1728 + 63809 = 1 = 0x0001
SUBTRACT ($a, 1) ; 0 - 1 = 65535 = 0xffff
MULTIPLY ($a, 1001) ; 65535 * 1001 = 64535 = 0xfc17
DIVIDE ($a, 101) ; 64535 / 101 = 638 = 0x027e
REMAINDER ($a, 11) ; 638 % 11 = 0 = 0x0000
OUTPUT (64, 4) ; output 0x0000 0000

; The multitypes are references
ADD ($b, $start) ; 0 + 1 = 1 = 0x0001

; If the message is 0x00
SUBTRACT ($b, $type) ; 1 - 65535 = 2 = 0x0002
MULTIPLY ($b, $b) ; 2 * 2 = 4 = 0x0004
DIVIDE ($a, $b) ; 0 / 4 = 0 = 0x0000
REMAINDER ($b, $type) ; 4 % 65535 = 4 = 0x0004
OUTPUT (64, 4) ; output 0x0000 0004

; If the message is 0x01, $type = 0
; so decompression failure occurs at
; REMAINDER ($b, $type)

; If the message is 0x02, $type = 1 so
; $b becomes 0 and decompression failure
; occurs at DIVIDE ($a, $b)

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

If the compressed message is 0x00 then the output of the code is
0x0000 0000 0000 0004 and the execution cost should be 25 UDVM
cycles. However, if the compressed message is 0x01 or 0x02 then
decompression failure occurs.

2.3 Sorting

This section gives assembly code to test the SORT-ASCENDING and SORT-
DESCENDING instructions. The code is designed to test that the
following boundary cases have been correctly implemented:

1. The sorting instructions sort integers with the same value, in
which case the original ordering of the integers must be preserved.
The output of the code is 0x466f 7264 2c20 796f 7527 7265 2074 7572 6e69 6e67 2069 6e74 6f20 6120 7065 6e67 7569 6e2e 2053 746f 7020 6974 2e, and the number of cycles required is 371.

2.4 SHA-1

This section gives assembly code to test the SHA-1 instruction. The code performs four tests on the SHA-1 algorithm itself, and additionally checks the following boundary cases specific to the UDVM:

1. The input string for the SHA-1 hash is obtained by byte copying over an area of the UDVM memory.

2. The SHA-1 hash overwrites its own input string.
; Set up a 1 byte buffer
LOAD (byte_copy_left, test_three)
LOAD (byte_copy_right, test_four)

; Perform SHA-1 over a 16384 bytes in a 1 byte buffer
SHA-1 (test_three, 65535, hash_value)
OUTPUT (hash_value, 20)

; Set up an 8 byte buffer
LOAD (byte_copy_left, test_four)
LOAD (byte_copy_right, test_end)

; Perform SHA-1 over 640 bytes in an 8 byte buffer
SHA-1 (test_four, 640, test_four)
OUTPUT (test_four, 20)

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

test_one
byte (97, 98, 99)

test_two

test_three
byte (97)

test_four
byte (48, 49, 50, 51, 52, 53, 54, 55)

test_end

The output of the code is as follows:

0xa999 3e36 4706 816a ba3e 2571 7850 c26c 9cd0 d89d
0x8498 3e44 1c3b d26e baae 4aa1 f951 29e5 e546 70f1
0xed0 18d 43d3 a689 af08 8e15 6bd0 434a a0c8 31fc
0x12ff 347b 4f27 d69e 1f32 86f 4b55 73e3 666e 122f
0x4f46 0452 ebb5 6393 4f46 0452 ebb5 6393 4f46 0452
Executing the code costs a total of 17176 UDVM cycles.

2.5 LOAD and MULTILOAD

This section gives assembly code to test the LOAD and MULTILOAD instructions. The code is designed to test the following boundary cases:

1. The MULTILOAD instruction overwrites itself or any of its operands, in which case decompression failure occurs.

2. The memory references of MULTILOAD instruction operands are evaluated step-by-step rather than all at once before starting to copy data.
The INPUT-BYTES, MULTIPLY and ADD instructions give the following values for $\text{start} = 64$ just before the MULTILOADs begin:

Input $\text{start before 1st MULTILOAD}$
0x00  60
0x01  62
0x02  64
Consequently after the first MULTILOAD the values of $start are:

<table>
<thead>
<tr>
<th>Input</th>
<th>$start before 2nd MULTILOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>128</td>
</tr>
<tr>
<td>0x01</td>
<td>overlap_end = 178 = last byte of 2nd MULTILOAD instruction</td>
</tr>
<tr>
<td>0x02</td>
<td>overlap_start = 163 = 7 bytes before 2nd MULTILOAD instruction</td>
</tr>
</tbody>
</table>

Consequently execution of the 2nd MULTILOAD (and any remaining code gives):

<table>
<thead>
<tr>
<th>Input</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>MULTILOAD reads and writes operand by operand. The output is 0x0084 0084 0086 0086 002a 0080 002a 002a, and the cost of executing the code is 36 UDVM cycles.</td>
</tr>
<tr>
<td>0x01</td>
<td>The first write of the MULTILOAD instruction would overwrite the last byte of the final MULTILOAD operand, so decompression failure occurs.</td>
</tr>
<tr>
<td>0x02</td>
<td>The last write of the MULTILOAD would overwrite the MULTILOAD opcode so decompression failure occurs.</td>
</tr>
</tbody>
</table>

### 2.6 COPY

This section gives assembly code to test the COPY instruction. The code is designed to test that the following boundary cases have been correctly implemented:

1. The COPY instruction copies data from both outside the circular buffer and inside the circular buffer within the same operation.

2. The COPY instruction performs byte-by-byte copying (i.e. some of the later bytes to be copied are themselves written into the UDVM memory by the COPY instruction currently being executed).

3. The COPY instruction overwrites itself and continues executing.

4. The COPY instruction overwrites the UDVM registers byte_copy_left and byte_copy_right.

5. The COPY instruction writes to and reads from the right of the buffer beginning at byte_copy_right.
6. The COPY instruction implements byte copying rules when the destination wraps around the buffer.

at (64)

:byte_copy_left pad (2)
:byte_copy_right pad (2)

at (128) ; Set up buffer between addresses 64 & 128
LOAD (32, 16384)
LOAD (byte_copy_left, 64)
LOAD (byte_copy_right, 128)

COPY (32, 128, 33) ; Copy byte by byte starting to left of buffer, into buffer and wrapping round the buffer (inc overwriting boundaries)
LOAD (64, 16640) ; Change start of buffer to be beyond bytecode
COPY (64, 85, 65) ; Copy to the left of the buffer, overwriting this instruction
OUTPUT (32, 119) ; Output 32 * 0x40 + 86 * 0x41 + 0x55 which is 32 * '@' + 86 'A' + 'U'

LOAD (byte_copy_left, 32)
LOAD (byte_copy_right, 48)

MEMSET (32, 4, 65, 1) ; Set first 4 bytes of buffer to be "ABCD"
COPY (32, 4, 48) ; Copy from byte_copy_right (i.e. not in buffer)
OUTPUT (48, 4) ; Output 0x4142 4344 which is ‘ABCD’

COPY (48, 4, 46) ; Copy from two before byte_copy_right to wrap around the buffer
OUTPUT (32, 2) ; Output 0x4344 which is ’CD’

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

The output is above and executing the code costs a total of 365 UDVM cycles.
2.7 COPY-LITERAL and COPY-OFFSET

This section gives assembly code to test the COPY-LITERAL and COPY-OFFSET instructions. The code is designed to test similar boundary cases to the code for the COPY instruction, as well as the following condition specific to COPY-LITERAL and COPY-OFFSET:

1. The COPY-LITERAL or COPY-OFFSET instruction overwrites the value of its destination.

2. The COPY-OFFSET instruction reads from an offset that wraps around the buffer (i.e. the offset is larger than the distance between byte_copy_left and the destination).

```asm
at (64)
:byte_copy_left        pad (2)
:byte_copy_right       pad (2)
:destination           pad (2)
:offset                pad (2)

at (128)
; Set up circular buffer, source and destination
LOAD (32, 16640)
LOAD (byte_copy_left, 64)
LOAD (byte_copy_right, 128)
LOAD (destination, 33)
COPY-LITERAL (32, 128, $destination) ; Copy from the left of the buffer, overwriting bcl, bcr and destination
OUTPUT (64, 8) ; Check destination has been updated
               ; Output 0x4141 4141 0061 4141
LOAD (destination, copy)
:copy              ; Overwrite the copy instruction
COPY-LITERAL (32, 2, $destination)
OUTPUT (copy, 2) ; Output 0x4141

LOAD (byte_copy_left, 72) ; Set up new circular buffer
LOAD (byte_copy_right, 82) ; Set destination to byte_copy_right
MEMSET (72, 10, 65, 1) ; Fill buffer with 0x41 - 4A
```
COPY-OFFSET (2, 6, $destination) ; Copy from within circular
; buffer to outside buffer

LOAD (offset, 6)
COPY-OFFSET ($offset, 4, $destination)
; Copy from byte_copy_right
; so reading outside buffer
OUTPUT ($byte_copy_right, 10) ; Output 0x494A 4142 4344 494A 4142
; which is 'IJABCDIJAB'

LOAD (destination, 80) ; Put destination within the
; buffer
COPY-OFFSET (4, 4, $destination) ; Copy where destination wraps
OUTPUT (destination, 2) ; Output 0x004A

COPY-OFFSET (5, 4, $destination) ; Copy where offset wraps from
; left back round to right
OUTPUT (destination, 2) ; Output 0x004E
OUTPUT ($byte_copy_left, 10) ; Output the circular buffer
; 0x4748 4845 4647 4748 4546
; which is 'GHHEFGGHEF'

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

The output of the code is above and the cost of execution is 216 UDVM cycles.

2.8 MEMSET

This section gives assembly code to test the MEMSET instruction. The code is designed to test that the following boundary cases have been correctly implemented:

1. The MEMSET instruction overwrites the registers byte_copy_left and byte_copy_right.

2. The output values of the MEMSET instruction do not lie between 0 and 255 inclusive (in which case they must be taken modulo 2^8).
at (64)

:byte_copy_left         pad (2)
:byte_copy_right        pad (2)

at (128)

LOAD (byte_copy_left, 128) ; sets up a circular buffer
LOAD (byte_copy_right, 129) ; of 1 byte between 0x0080 and 0x0081

MEMSET (64, 129, 0, 1) ; fills up the memory in the range
; 0x0040-0x007f with 0x00, ... 0x3f;
; then it writes successively at
; 0x0080 the following values 0x40, ... 0x80
; as a side effect, the values of
; bcl and bcr are modified.
; before an during the MEMSET:
; byte_copy_left: 0x0080 byte_copy_right: 0x0081
; after the memset:
; byte_copy_left: 0x0001 byte_copy_right: 0x0203

MEMSET (129, 15, 64, 15) ; fills the memory range 0x0080-0x008f
; with values 0x40, 0x4f, ... 0xf4, 0x03, 0x12.
; as a side effect, it overwrites a
; part of the code including itself

OUTPUT (128, 16) ; outputs 0x8040 4f5e 6d7c 8b9a
; a9b8 c7d6 e5f4 0312

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

The output of the code is 0x8040 4f5e 6d7c 8b9a a9b8 c7d6 e5f4 0312.
Executing the code costs 166 UDVM cycles.

2.9 CRC

This section gives assembly code to test the CRC instruction. The
code does not test any specific boundary cases (as there do not
appear to be any) but focuses instead on verifying the CRC algorithm.
at (64)

:byte_copy_left               pad (2)
:byte_copy_right              pad (2)
: crc_value                   pad (2)
: crc_string_a                pad (24)
: crc_string_b                pad (20)

at (128)

MEMSET (crc_string_a, 24, 1, 1) ; sets up between 0x0046 and 0x005d
                                     ; a byte string containing 0x01,
                                     ; 0x02, ... 0x18

MEMSET (crc_string_b, 20, 128, 1) ; sets up between 0x005e and 0x0071
                                     ; a byte string containing 0x80,
                                     ; 0x81, ... 0x93

INPUT-BYTES (2, crc_value, decompression_failure)
                 ; reads in 2 bytes representing
                 ; the CRC value of the byte string
                 ; of 44 bytes starting at 0x0046

CRC ($crc_value, crc_string_a, 44, decompression_failure)
             ; computes the CRC value of the
             ; byte string crc_string_a
             ; concatenated with byte string
             ; crc_string_b (with a total
             ; length of 44 bytes).
             ; if the computed value does
             ; not match the 2 byte value read
             ; previously the program ends
             ; with DECOMPRESSION-FAILURE

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

: decompression_failure
DECOMPRESSION-FAILURE

If the compressed message is 0x62cb then the code should successfully
terminate with no output, and with a total execution cost of 95 UDVM
cycles. For different 2-byte compressed messages the code should
terminate with a decompression failure.

2.10 INPUT-BITS

This section gives assembly code to test the INPUT-BITS instruction.
The code is designed to test that the following boundary cases have
been correctly implemented:

1. The INPUT-BITS instruction changes between any of the four possible bit orderings defined by the input_bit_order register.

2. The INPUT-BITS instruction inputs 0 bits.

3. The INPUT-BITS instruction requests data that lies beyond the end of the compressed message.

```
at (64)
:byte_copy_left pad (2)
:byte_copy_right pad (2)
:input_bit_order pad (2)
:result pad (2)

at (128)
:start

INPUT-BITS ($input_bit_order, result, end_of_message) ; reads in
; exactly as many bits as the 2 byte
; value written in the input_bit_order
; register. get out of the loop when
; no more bits are available at input.

OUTPUT (result, 2) ; outputs as a 2 byte integer
; the previously read bits

ADD ($input_bit_order, 1) ; if at the beginning of this loop the
; register input_bit_order is 0,
REMAINDER ($input_bit_order, 7) ; then its value varies periodically
; like this: 2, 4, 6, 1, 3, 5, 7.
ADD ($input_bit_order, 1) ; that gives for the FHP bits: 010,
; 100, 110, 001, 011, 101, 111

JUMP (start) ; run the loop once more

:end_of_message

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)
```

An example compressed message is 0x932e ac71, which decompresses to give the output 0x0000 0002 0002 0013 0000 0003 001a 0038. Executing the code costs 66 UDVM cycles.
2.11 INPUT-HUFFMAN

This section gives assembly code to test the INPUT-HUFFMAN instruction. The code is designed to test that the following boundary cases have been correctly implemented:

1. The INPUT-HUFFMAN instruction changes between any of the four possible bit orderings defined by the input_bit_order register.

2. The INPUT-HUFFMAN instruction inputs 0 bits.

3. The INPUT-HUFFMAN instruction requests data that lies beyond the end of the compressed message.

at (64)

:byte_copy_left           pad (2)
:byte_copy_right          pad (2)
:input_bit_order          pad (2)
:result                   pad (2)

at (128)

:start

INPUT-HUFFMAN (result, end_of_message, 2, $input_bit_order, 0, $input_bit_order, $input_bit_order, $input_bit_order, 0, 65535, 0)
OUTPUT (result, 2)
ADD ($input_bit_order, 1)
REMAINDER ($input_bit_order, 7)
ADD ($input_bit_order, 1)
JUMP (start)

:end_of_message

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

An example compressed message is 0x932e ac71 66d8 6f, which decompresses to give the output 0x0000 0003 0008 04d7 0002 0003 0399 30fe. Executing the code costs 84 UDVM cycles.

As the code is run, the input_bit_order changes through all possible values to check usage of the H and P bits. The number of bits to input each time is taken from the value of input_bit_order. The
sequence is the following:

<table>
<thead>
<tr>
<th>Input_bit_order (bin)</th>
<th>Total bits input by Huffman</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>110</td>
<td>12</td>
<td>1239</td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>10</td>
<td>921</td>
</tr>
<tr>
<td>111</td>
<td>14</td>
<td>12542</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0 - not enough bits so terminate</td>
<td></td>
</tr>
</tbody>
</table>

P-bit changed, throw away 6 bits

P-bit changed, throw away 4 bits

### 2.12 INPUT-BYTES

This section gives assembly code to test the INPUT-BYTES instruction. The code is designed to test that the following boundary cases have been correctly implemented:

1. The INPUT-BYTES instruction inputs 0 bytes.

2. The INPUT-BYTES instruction requests data that lies beyond the end of the compressed message.

3. The INPUT-BYTES instruction is used after part of a byte has been input (e.g. by the INPUT-BITS instruction).
An example compressed message is 0x932e ac71 66d8 6fb1 592b dc9a 9734 d847 a733 874e 1bcb cd51 b5dc 9659 9d6a, which decompresses to give the output 0x0000 932e 0001 b166 d86f b100 1a2b 0003 9a97 34d8 0007 0001 3387 4e00 08dc 9651 b5dc 9600 599d 6a. Executing the code costs 130 UDVM cycles.

As the code is run, the input_bit_order changes through all possible values to check usage of the F and P bits. The number of bits or bytes to input each time is taken from the value of input_bit_order. For each INPUT-BYTES instruction the remaining bits of the byte are thrown away. The P-bit always changes on the byte boundary so no bits are thrown away. The sequence is the following:
### 2.13 Stack manipulation

This section gives assembly code to test the PUSH, POP, CALL and RETURN instructions. The code is designed to test that the following boundary cases have been correctly implemented:

1. The stack manipulation instructions overwrite the UDVM register stack_location.

2. The CALL instruction specifies a reference operand rather than an absolute value.

3. The PUSH instruction pushes the value contained in stack_fill onto the stack.

4. The stack_location register contains an odd integer.

\[
\text{at (64)}
\]

:byte_copy_left pad (2)
:byte_copy_right pad (2)
:input_bit_order pad (2)
:stack_location pad (2)
:next_address pad (2)

\[
\text{at (128)}
\]
LOAD (stack_location, 64)
PUSH (2)
PUSH ($64)
PUSH (66) ; Stack now contains 2, 1, 66
; so $stack_location = 66
OUTPUT (64, 8) ; Output 0x0003 0002 0001 0042
POP (64) ; Pop value 66 from address 70 to address 64
POP ($stack_location) ; Pop value 1 from address 68 to address 66
; so stack_fill is overwritten to be 1
POP (stack_location) ; Pop value 1 from address 68 to address 70
OUTPUT (64, 8) ; Output 0x0042 0000 0001 0001
JUMP (address_a)

at (192)

:address_a

LOAD (stack_location, 32)
LOAD (next_address, address_c)
SUBTRACT ($next_address, address_b) ; next_address = 64
CALL (address_b) ; push 204 on stack

at (256)

:address_b

CALL ($next_address) ; push 256 on stack

at (320)

:address_c

LOAD (stack_location, 383)
LOAD (383, 26) ; overwrite $stack_location with 26
MULTILOAD (432, 3, 1, 49153, 32768)

; write bytes so that 433 and 434
; contain 0x01c0 = 448 and
; 435 and 436 contain 0x0180 = 384
RETURN ; pop 383 from the stack and jump
; there = 384 which is 1sb of
; stack_fill which now contains 25
; which is UDVM instruction RETURN
; pop 448 from the stack and jump
The output of the code is 0x0003 0002 0001 0042 0042 0000 0001 0001, and a total of 40 UDVM cycles are used.

2.14 Program flow

This section gives assembly code to test the JUMP, COMPARE and SWITCH instructions. The code is designed to test that the following boundary cases have been correctly implemented:

1. The address operands are specified as references to memory addresses rather than as absolute values.

```assembly
at (64)

:next_address    pad (2)
:counter         pad (1)
:counter_lsb     pad (1)
:switch_counter  pad (2)

at (128)

LOAD (switch_counter, 4)

:address_a

LOAD (next_address, address_c)
SUBTRACT ($next_address, address_b)     ; address_c - address_b
OUTPUT (counter_lsb, 1)

:address_b

JUMP ($next_address)                    ; Jump to address_c

:address_c

ADD ($counter, 1)
LOAD (next_address, address_a)
SUBTRACT ($next_address, address_d)     ; address_a - address_d
OUTPUT (counter_lsb, 1)

:address_d
```
COMPARE ($counter, 6, $next_address, address_c, address_e)
; counter < 6, $next_address gives
; jump to address_a

:address_e

SUBTRACT ($switch_counter, 1) ; switch_counter = 3
LOAD (next_address, address_a)
SUBTRACT ($next_address, address_f) ; address_a - address_f
OUTPUT (counter_lsb, 1)

:address_f

SWITCH (4, $switch_counter, address_g, $next_address, address_c, address_e)
; when $switch_counter = 1,
; $next_address gives jump to
; address_a

:address_g

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

The output of the code is 0x0001 0102 0203 0304 0405 0506 0707 0708 0809, and a total of 131 UDVM cycles are used.

2.15 State creation

This section gives assembly code to test the STATE-CREATE and STATE-FREE instructions. The code is designed to test that the following boundary cases have been correctly implemented:

1. An item of state is created that duplicates an existing state item.

2. An item of state is freed when the state has not been created.

3. An item of state is created and then freed by the same message.

4. The STATE-FREE instruction frees a state item by sending fewer bytes of state_identifier than the minimum_access_length.

5. The STATE-FREE instruction has partial_identifier_length operand shorter than 6 or longer than 20.

6. The STATE-FREE instruction specifies a partial_identifier that matches with two state items in the compartment.
7. Write the bytes of the identifier to the position specified in the STATE-FREE instruction after the STATE-FREE instruction has been run (and before END-MESSAGE).

```
at (64)
:byte_copy_left     pad (2)
:byte_copy_right    pad (2)
:states             pad (1)
:states_lsb         pad (1)
:min_len            pad (1)
:min_len_lsb        pad (1)
:state_identifier   pad (20)

set (state_length, 10)

at (127)
:decompression_failure
at (128)

INPUT-BYTES (1, states_lsb, decompression_failure)

:test_one
LSHIFT ($states, 11)
COMPARE ($states, 32768, test_two, create_state_a2, create_state_a2)

:create_state_a2
STATE-CREATE (state_length, state_address2, 0, 20, 0)

:test_two
LSHIFT ($states, 1)
COMPARE ($states, 32768, test_three, create_state_a, create_state_a)

:create_state_a
STATE-CREATE (state_length, state_address, 0, 20, 0)

:test_three
LSHIFT ($states, 1)
COMPARE ($states, 32768, test_four, free_state, free_state)

:free_state
INPUT-BYTES (1, min_len_lsb, decompression_failure)
STATE-FREE (state_identifier, $min_len)
COPY (identifier1, $min_len, state_identifier)
```
LSHIFT ($states, 1)
COMPARE ($states, 32768, test_five, free_state2, free_state2)

:free_state2
STATE-FREE (identifier1, 6)

:test_five
LSHIFT ($states, 1)
COMPARE ($states, 32768, end, create_state_b, create_state_b)

:create_state_b
END-MESSAGE (0, 0, state_length, state_address, 0, 20, 0)

:end
END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

:identifier1
byte (67, 122, 232, 10, 15, 220, 30, 106, 135, 193, 182, 42, 118, 118, 185, 115, 49, 140, 14, 245)
at (256)
:state_address
byte (192, 204, 63, 238, 121, 188, 252, 143, 209, 8)

:state_address2
byte (101, 232, 3, 82, 238, 41, 119, 23, 223, 87)

Upon reaching the END-MESSAGE instruction the UDVM does not output any decompressed data, but instead may make one or more state creation or state free requests to the state handler. Assuming that the application does not veto the state creation request (and that sufficient state memory is available) the code results in 0, 1 or 2 state items being present in the compartment.

The following table lists ten different compressed messages, the states created and freed by each, the number of states left after each message and the number of UDVM cycles used. There are 3 state creation instructions:

create state_a which has hash identifier1
create state_b (in END-MESSAGE) which is identical to state_a
create state_a2 which has a different identifier, but the first 6 bytes are the same as those of identifier1.
2.16 STATE-ACCESS

This section gives assembly code to test the STATE-ACCESS instruction. The code is designed to test that the following boundary cases have been correctly implemented:

1. A subset of the bytes contained in a state item is copied to the UDVM memory.

2. Bytes are copied from beyond the end of the state value.

3. The state_instruction operand is set to 0.

4. The state cannot be accessed because the partial state identifier is too short.

5. The state identifier is overwritten by the state item being accessed.

The following bytecode needs to be run first to set up the state for the rest of the test.
END-MESSAGE (0, 0, state_length, state_start, 0, 20, 0)

; The bytes between state_start and state_end are derived from
; translation of the following mnemonic code:
; at (512)
; OUTPUT (data, 4)
; END-MESSAGE (0,0,0,0,0,0)
; :data
; byte (116, 101, 115, 116)

:state_start
byte (34, 162, 12,4, 35, 0, 0, 0, 0, 0, 116, 101, 115, 116)
:state_end
set (state_length, (state_end - state_start))

This is the bytecode for the rest of the test.

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)
:type                           pad (1)
:type_lsb                       pad (1)
:state_value                    pad (4)

:decompression_failure
at (127)
INPUT-BYTES (1, type_lsb, decompression_failure)
COMPARE ($type, 1, execute_state, extract_state, error_conditions)

:execute_state
STATE-ACCESS (state_identifier, 20, 0, 0, 0, 512)

:extract_state
STATE-ACCESS (state_identifier, 20, 12, 4, state_value, 0)
OUTPUT (state_value, 4)
JUMP (end)
:error_conditions

COMPARE ($type, 3, state_not_found, id_too_short, state_too_short)

:state_not_found

STATE-ACCESS (128, 20, 0, 0, 0, 0)
JUMP (end)

:id_too_short

STATE-ACCESS (state_identifier, 19, 6, 4, state_value, 0)
JUMP (end)

:state_too_short

STATE-ACCESS (state_identifier, 20, 12, 5, state_value, 0)
JUMP (end)

at (484)

:end

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)
at (512)

:state_identifier

byte (0x5d, 0xf8, 0xbc, 0x3e, 0x20, 0x93, 0xb5, 0xab, 0xe1, 0xf1, 0x70, 0x13, 0x42, 0x4c, 0xe7, 0xfe, 0x05, 0xe0, 0x69, 0x39)

If the compressed message is 0x00 then the output of the code is 0x7465 7374 and a total of 21 UDVM cycles are be used. If the compressed message is 0x01 then the output of the code is also 0x7465 7374 but in this case using a total of 15 UDVM cycles. If the compressed message is 0x02, 0x03 or 0x04 then decompression failure occurs.

3. Torture tests for dispatcher

The following sections give code to test the various functions of the SigComp dispatcher.

3.1 Useful Values

This section gives assembly code to test that the SigComp "Useful
Values" are correctly initialized in the UDVM memory. It also tests that the UDVM is correctly terminated if the bytecode uses too many UDVM cycles or tries to write beyond the end of the available memory.

The code tests that the following boundary cases have been correctly implemented:

1. The bytecode uses exactly as many UDVM cycles as are available (in which case no problems should arise) or one cycle too many (in which case decompression failure should occur). A liberal implementation could allow more cycles to be used than are strictly available, in which case decompression failure will not occur. This is an implementation choice. If this choice is made, the implemene must be sure that the cycles are checked eventually and that decompression failure does occur when bytecode uses an excessive number of cycles. This is tested in Section 3.2.

2. The bytecode writes to the highest memory address available (in which case no problems should arise) or to the memory address immediately following the highest available address (in which case decompression failure must occur).

:udvm_memory_size    pad (2)
:cycles_per_bit      pad (2)
:sigcomp_version     pad (2)
:partial_state_id_length pad (2)
:state_length        pad (2)

at (64)

:byte_copy_left      pad (2)
:byte_copy_right     pad (2)
:remaining_cycles     pad (2)
:check_memory         pad (1)
:check_memory_lsb     pad (1)
:check_cycles         pad (1)
:check_cycles_lsb     pad (1)

at (127)
:decompression_failure
at (128) ; Set up a 1 byte buffer

LOAD (byte_copy_left, 32)
LOAD (byte_copy_right, 33)

:test_version
COMPARE ($sigcomp_version, 1, decompression_failure, test_state_access, decompression_failure)

:test_state_access

COMPARE ($partial_state_id_length, 0, decompression_failure, test_length_equals_zero, test_state_length)

:test_length_equals_zero
; No state was accessed so state_length should be zero (first message)
COMPARE ($state_length, 0, decompression_failure, end, decompression_failure)

:test_state_length
; State was accessed so state_length should be 960
COMPARE ($state_length, 960, decompression_failure, test_udvm_memory, decompression_failure)

:test_udvm_memory
; Copy one byte to
; udvm_memory_size + input - 1
; Succeed when input byte is 0x00
; Fail when input byte is 0x01

INPUT-BYTES (1, check_memory_lsb, decompression_failure)
ADD ($check_memory, $udvm_memory_size)
SUBTRACT ($check_memory, 1)
COPY (32, 1, $check_memory)

:test_udvm_cycles

INPUT-BYTES (1, check_cycles_lsb, decompression_failure)
; Work out the total number of cycles available to the UDVM
; total_UDVM_cycles = cycles_per_bit * (8 * message_size + 1000)
; = cycles_per_bit * (8 * (partial_state_id_length + 3) + 1000)
LOAD (remaining_cycles, $partial_state_id_length)
ADD ($remaining_cycles, 3)
MULTIPLY ($remaining_cycles, 8)
ADD ($remaining_cycles, 1000)
MULTIPLY ($remaining_cycles, $cycles_per_bit)
ADD ($remaining_cycles, $check_cycles)
The bytecode must be executed a total of four times in order to fully test the SigComp Useful Values. In the first case the bytecode is uploaded as part of the SigComp message (no compressed data is required in this case). This causes the UDVM to request creation of a new state item, and uses a total of 966 UDVM cycles.

Subsequent tests access this state by uploading the state identifier as part of the SigComp message. Note that the SigComp message should not contain a returned feedback item (as this would cause the bytecode to calculate the total number of available UDVM cycles incorrectly).

A 2-byte compressed message is required for the second and subsequent cases: if the message is 0x0000 then the UDVM should successfully terminate using exactly the number of available UDVM cycles. However, if the message is 0x0001 then the UDVM should use too many cycles and hence terminate with decompression failure. Furthermore if the message is 0x0100 then decompression failure must occur because the UDVM attempts to write beyond its available memory.

### 3.2 Cycles checking

As discussed in Section 3.1, it is possible to write an implementation which takes a liberal approach to checking the cycles used and allow some extra cycles. The implementer must be sure that decompression failure does not occur too early and that in the case of excessive use of cycles, decompression failure does eventually occur. This test checks that:

1. Decompression failure occurs eventually when there is an infinite loop.
Internet-Draft            SigComp Torture Tests           September 2005

at (64)
:byte_copy_left           pad (2)
:byte_copy_right          pad (2)
:value                    pad (2)
:copy_next                pad (2)

at(128)
MULTILOAD (byte_copy_left, 4, 32, 41, 0, 34)
   ; Set up a 10 byte buffer
   ; Set the value to copy
   ; Copy it 100 times,
   ; output the value,
   ; increment the counter

:loop
COPY (value, 2, $byte_copy_left)
COPY-OFFSET (2, 100, $copy_next)
OUTPUT (value, 2)
ADD ($value, 1)
JUMP (loop)

If the cycles are counted exactly and cpb = 16 then decompression
failure will occur at COPY-OFFSET when value = 180 = 0xB4.  If cpb =
32 then decompression failure will occur when value = 361 = 0x0169.
If they are not counted exactly, then decompression failure MUST
occur eventually.

3.3 Message-based transport

This section provides a set of messages to test the SigComp header
over a message-based transport such as UDP.  The messages test that
the following boundary cases have been correctly implemented:

1.  The UDVM bytecode is copied to different areas of the UDVM
memory.

2.  The decompression memory size is set to an incorrect value.

3.  The SigComp message is too short.

4.  The destination address is invalid.

The basic version of the code used in the test is given below.  Note
that the code is designed to calculate the decompression memory size
based on the Useful Values provided to the UDVM:
:udvm_memory_size                 pad (2)
:cycles_per_bit                  pad (2)
:sigcomp_version                 pad (2)
:partial_state_id_length        pad (2)
:state_length                   pad (2)

at (128)

:code_start

; udvm_memory_size for message based transport
;    = DMS - total_message_size

ADD ($udvm_memory_size, total_message_size)
OUTPUT (udvm_memory_size, 2)
END-MESSAGE (0, 0, 0, 0, 0, 0, 1)

:code_end

set (header_size, 3)
set (code_size, (code_end - code_start))
set (total_message_size, (header_size + code_size))

A number of complete SigComp messages are given below, each containing some or all of the above code. In each case it is indicated whether the message will successfully output the decompression memory size or whether it will cause a decompression failure to occur (together with the reason for the failure):

SigComp message:       Effect:
0xf8                  Fails (message too short)
0xf800                Fails (message too short)
0xf800 e106 0011 2200 0223 Outputs the decompression_memory_size
0x0000 0000 0000 0000 01
0xf800 f106 0011 2200 0223 Fails (message too short)
0x0000 0000 0000 0000 01
0xf800 e006 0011 2200 0223 Fails (invalid destination address)
0x0000 0000 0000 0000 01
The messages should be decompressed in the order given to check that an error in one message does not interfere with the successful decompression of subsequent messages.

The two messages that successfully decompress each use a total of 5 UDVM cycles.

3.4 Stream-based transport

This section provides a byte stream to test the SigComp header and delimiters over a stream-based transport such as TCP. The byte stream tests all of the boundary cases covered in Section 3.2, as well as the following cases specific to stream-based transports:

1. Quoted bytes are used by the record marking scheme.

2. Multiple delimiters are used between the same pair of messages.

3. Unnecessary delimiters are included at the start of the stream.

The basic version of the code used in the test is given below. Note that the code is designed to calculate the decompression memory size based on the Useful Values provided to the UDVM:

```plaintext
:udvm_memory_size   pad (2)
:cycles_per_bit     pad (2)
:sigcomp_version    pad (2)
:partial_state_id_length pad (2)
:state_length       pad (2)

at (128)

; udvm_memory_size for stream based transport = DMS / 2

MULTIPLY ($udvm_memory_size, 2)
OUTPUT (udvm_memory_size, 2)
OUTPUT (test_record_marking, 5)
END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

:test_record_marking

byte (255, 255, 255, 255, 255)
```
The above assembly code has been compiled and used to generate the following byte stream:

0xffff f801 7108 0002 2200 0222 a092 0523 0000 0000 00ff 00ff 0x03ff ffff ffff ffff ffff f801 7e08 0002 2200 0222 a3d2 0523 0000 0000 0x0000 00ff 04ff ffff ffff ffff ff

Note that this byte stream can be divided into five distinct portions (two SigComp messages and three sets of delimiters) as illustrated below:

Portion of byte stream:                                Meaning:

0xffff                                                 Delimiter
0xf801 7108 0002 2200 0222 a092 0523                   First message
0x0000 0000 0000 00ff 00ff 03ff ffff
0xffff ffff                                            Delimiter
0xf801 7e08 0002 2200 0222 a3d2 0523                   Second message
0x0000 0000 0000 00ff 04ff ffff ff
0xffff ffff ffff                                       Delimiter

When the complete byte stream is supplied to the decompressor dispatcher, the record marking scheme must use the delimiters to partition the stream into two distinct SigComp messages. Both of these messages successfully output the decompression memory size (as a 2-byte value), followed by five consecutive 0xff bytes to test that the record marking scheme is working correctly. A total of 11 UDVM cycles are used in each case.

It must also be checked that the dispatcher can handle the same error cases as covered in Section 3.2. Each of the following byte streams should cause a decompression failure to occur for the reason stated:
Byte stream:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Reason for failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf8ff ff</td>
<td>Message too short</td>
</tr>
<tr>
<td>0xf800 ffff</td>
<td>Message too short</td>
</tr>
<tr>
<td>0xf801 8108 0002 2200 0222 a092 0523 ffff</td>
<td>Message too short</td>
</tr>
<tr>
<td>0x0000 0000 0000 00ff 00ff 03ff ffff</td>
<td>Invalid destination</td>
</tr>
<tr>
<td>0xf801 7008 0002 2200 0222 a092 0523 ffff</td>
<td></td>
</tr>
<tr>
<td>0x0000 0000 0000 00ff 04ff ffff ff</td>
<td></td>
</tr>
</tbody>
</table>

Note that when a decompression failure occurs it is an implementation decision whether to close the entire stream or whether to ignore the error and attempt to decompress subsequent messages in the stream.

### 3.5 Input past the end of a message

This section gives assembly code to test that the implementation correctly handles input past the end of a SigComp message. The code is designed to test that the following boundary cases have been correctly implemented:

1. An INPUT instruction requests data that lies beyond the end of the message. In this case the dispatcher should not return any data to the UDVM. Moreover, the message bytes held by the dispatcher should still be available for retrieval by subsequent INPUT instructions.

2. The INPUT-BYTES instruction is used after part of a byte has been input (e.g. by the INPUT-BITS instruction). In this case the remaining partial byte must be discarded, even if the INPUT-BYTES instruction requests data that lies beyond the end of the message.

```assembly
at (64)
:byte_copy_left   pad (2)
:byte_copy_right  pad (2)
:input_bit_order  pad (2)
:result           pad (1)
:result_lsb       pad (6)
:right

at (128)
LOAD (byte_copy_left, result)
```
LOAD (byte_copy_right, right)

:start

; Input bits to ensure that the remaining message is not byte aligned
INPUT-BITS (9, result, decompression_failure1) ; Input 0x1FF (9 bits)

; Attempt to read 7 bytes
INPUT-BYTES (7, result, next_bytes) ; This should fail, throw away
  ; 7 bits with value 0x7a and
  ; jump to next_bytes

:decompression_failure1
DECOMPRESSION-FAILURE ; This instruction is never
  ; executed but is used to
  ; separate success and failure
  ; to input bytes.

:next_bytes

; Read 7 bits - this removes the byte alignment of the message

; If the bits have not been thrown away where they should then the
; message will be 1 byte longer than necessary and the output will
; be incorrect.
INPUT-BITS (7, result, decompression_failure1) ; Input 0x00 (7 bits)

; Read 2 bytes
INPUT-BYTES (2, result, decompression_failure1)
  ; Throw away 1 bit value 0
  ; Input 0x6869
OUTPUT (result, 2)
  ; Output 0x6869

INPUT-BITS (16, result, bits) ; Attempt to read more bits than
  ; there are to ensure they
  ; remain available

:decompression_failure2
DECOMPRESSION-FAILURE ; This instruction is never
  ; executed but is used to
  ; separate success and failure
  ; to input bits.

:bots
; Read 8 bits
INPUT-BITS (8, result, decompression_failure2) ; Input 0x21 or fail
OUTPUT (result_lsb, 1) ; Output 0x21
:end_message

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

If the compressed message is 0xffff 0068 6921 then the code terminates successfully with the output 0x6869 21, and a total of 23 UDVM cycles are used. However, if the compressed message is 0xffff 0068 69 then decompression failure occurs (at the final INPUT-BITS).

4. Torture tests for state handler

The following sections give code to test the various functions of the SigComp state handler.

4.1 SigComp feedback mechanism

This section gives assembly code to test the SigComp feedback mechanism. The code is designed to test that the following boundary cases have been correctly implemented:

1. Both the short and the long versions of the SigComp feedback item are used.

2. The chain of returned SigComp parameters is terminated by a non-zero value.

   at (64)

   :type pad (1)
   :type_lsb pad (1)

   :requested_feedback_location pad (1)
   :requested_feedback_length pad (1)
   :requested_feedback_bytes pad (127)

   :returned_parameters_location pad (2)
   :length_of_partial_state_id_a pad (1)
   :partial_state_identifier_a pad (6)
   :length_of_partial_state_id_b pad (1)
   :partial_state_identifier_b pad (12)
   :length_of_partial_state_id_c pad (1)
:partial_state_identifier_c  pad (20)
:terminate_returned_parameters  pad (1)

align (128)

set (q_bit, 1)
set (s_bit, 0)
set (i_bit, 0)
set (flags, (((4 * q_bit) + (2 * s_bit)) + i_bit))

INPUT-BYTES (1, type_lsb, decompression_failure)
COMPARE ($type, 1, short_feedback_item, long_feedback_item, decompression_failure)

:short_feedback_item

set (requested_feedback_data, 127)
set (short_feedback_value, ((flags * 256) + requested_feedback_data))
LOAD (requested_feedback_value, short_feedback_value)
JUMP (return_sigcomp_parameters)

:long_feedback_item

set (requested_feedback_field, 255)
set (long_feedback_value, ((flags * 256) + requested_feedback_field))
LOAD (requested_feedback_value, long_feedback_value)
MEMSET (requested_feedback_bytes, 127, 1, 1)

:return_sigcomp_parameters

set (cpb, 0)
set (dms, 1)
set (sms, 0)
set (sigcomp_version, 1)

set (parameters_msb, (((64 * cpb) + (8 * dms)) + sms))
set (sigcomp_parameters, ((256 * parameters_msb) + sigcomp_version))
LOAD (returned_parameters_location, sigcomp_parameters)
LOAD (length_of_partial_state_id_a, 1536) ; length 6 first byte 0
LOAD (length_of_partial_state_id_b, 3072) ; length 12 first byte 0
LOAD (length_of_partial_state_id_c, 5120) ; length 20 first byte 0
LOAD (terminate_returned_parameters, 5376) ; length 21
; used to terminate
MEMSET (partial_state_identifier_a, 6, 0, 1)
MEMSET (partial_state_identifier_b, 12, 0, 1)
MEMSET (partial_state_identifier_c, 20, 0, 1)

END-MESSAGE (requested_feedback_location,
returned_parameters_location, 0, 0, 0, 0, 0)
:decompression_failure
DECOMPRESSION-FAILURE

When the above code is executed it supplies a requested feedback item to the state handler. If the compressed message is 0x00 then the short (1-byte) version of the feedback is used. Executing the bytecode in this case costs a total of 52 UDVM cycles. Assuming that the feedback request is successful the feedback item should be returned in the first SigComp message to be sent in the reverse direction. The SigComp message returning the feedback should begin as follows:

+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1 |   X   | first header byte
+---+---+---+---+---+---+---+---+
| 0 |            127            | returned feedback field
+---+---+---+---+---+---+---+---+

So the first 2 bytes of the returning SigComp message should be 0xfn7f where n = c, d, e or f (the choice of n is determined by the compressor generating the returning SigComp message, which is not under the control of the above code).

If the compressed message is 0x01 then the long version of the feedback item is used. Executing the bytecode in this case costs a total of 179 UDVM cycles and the SigComp message returning the feedback should begin as follows:

+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1 |   X   | first header byte
+---+---+---+---+---+---+---+---+
| 1 |            127            | returned feedback length
+---+---+---+---+---+---+---+---+          ^
| 1 |                                           |
+---+---+---+---+---+---+---+---+ |
| 2 |                                           |
+---+---+---+---+---+---+---+---+ |
| 3 | returned feedback field
+---+---+---+---+---+---+---+---+
So the first 129 bytes of the SigComp message should be 0xfnff 0102 0304 ... 7e7f where n = c, d, e or f.

As well as testing the requested and returned feedback items, the above code also announces values for each of the SigComp parameters. The supplied version of the code announces only the minimum possible values for the cycles_per_bit, decompression_memory_size, state_memory_size and SigComp_version (although this can easily be adjusted to test different values for these parameters).

The code should also announce the availability of state items with the following partial state identifiers:

0x0001 0203 0405 0x0001 0203 0405 0607 0809 0a0b 0x0001 0203 0405 0607 0809 0a0b 0c0d 0e0f 1011 1213

Note that different implementations may make use of the announcement information in different ways. It is a valid implementation choice to simply ignore all of the announcement data and use only the minimum resources that are guaranteed to be available to all endpoints. However the above code is useful for checking that an endpoint interprets the announcement data correctly (in particular ensuring that it does not mistakenly use resources that have not in fact been announced).

### 4.2 State memory management

The following section gives assembly code to test the memory management features of the state handler. The code checks that the correct states are retained by the state handler when insufficient memory is available to store all of the requested states.

The code is designed to test that the following boundary cases have been correctly implemented:

1. A state item is created that exceeds the total state_memory_size for the compartment.

2. States are created with a non-zero state_retention_priority.

3. A new state item is created that has a lower state_retention_priority than existing state items in the compartment.

For the duration of this test it is assumed that all states will be saved in a single compartment with a state_memory_size of 2048 bytes.
at (64)

:byte_copy_left pad (2)
:byte_copy_right pad (2)
:order pad (2)
:type pad (1)
:type_lsb pad (1)
:state_length pad (2)
:state_retention_priority pad (2)

at (127)
:decompression_failure
at (128)

MULTILOAD (byte_copy_left, 2, state_start, order_data)

INPUT-BYTES (1, type_lsb, decompression_failure)
COMPARE ($type, 5, general_test, large_state, verify_state)

:general_test

COMPARE ($type, 3, start, state_present, state_not_present)

:start

MULTIPLY ($type, 6)
ADD ($type, order_data)
LOAD (order, $type)
ADD ($type, 6)

; Finish with the value (order_data + 6*n) in order where
; n is the input value 0x00, 0x01 or 0x02
; type = order + 6
; These values are used to index into the 'order_data'
; that is used to work out state retention priorities and lengths

:loop

COPY ($order, 2, state_retention_priority)
COMPARE ($order, $type, continue, end, decompression_failure)

:continue

; Set up a state creation each time through the loop

LOAD (state_length, $state_retention_priority)
MULTIPLY ($state_length, 256)
STATE-CREATE ($state_length, state_start, 0, 6,
$state_retention_priority

ADD ($order, 2)
JUMP (loop)

:state_present

; Access the states that should be present
STATE-ACCESS (state_identifier_a, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_b, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_c, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_e, 6, 0, 0, 0, 0)
JUMP (end)

:state_not_present

; Check that the state that shouldn’t be present is not present.
STATE-ACCESS (state_identifier_d, 6, 0, 0, 0, 0)
JUMP (end)

:large_state

STATE-CREATE (2048, state_start, 0, 6, 0)
JUMP (end)

:verify_state

STATE-ACCESS (large_state_identifier, 6, 0, 0, 0, 0)
JUMP (end)

:end

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)
at (512)

:state_start

byte (116, 101, 115, 116)

:order_data
; This data is used to generate the retention priority
; and state length of each state creation.

word (0, 1, 2, 3, 4, 3, 2, 1, 0)

:state_identifier_a
byte (142, 234, 75, 67, 167, 135)
:state_identifier_b
byte (249, 1, 14, 239, 86, 123)
:state_identifier_c
byte (35, 154, 52, 107, 21, 166)
:state_identifier_d
byte (180, 15, 192, 228, 77, 44)
:state_identifier_e
byte (212, 162, 33, 71, 230, 10)
:large_state_identifier
byte (239, 242, 188, 15, 182, 175)

The above code must be executed a total of 7 times in order to complete the test. Each time the code is executed a 1-byte compressed message should be provided as below. The effects of the messages are given below. States are described in the form (name, x, y) where name corresponds to the name of the identifier in the mnemonic code, x is the length of the state and y is the retention priority of the state.

<table>
<thead>
<tr>
<th>Message</th>
<th>Effect:</th>
<th>#cycles:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>create states: (a,0,0), (b,256,1), (c,512,2)</td>
<td>811</td>
</tr>
<tr>
<td>0x01</td>
<td>create states: (d,768,3), (e,1024,4) - deleting a, b, c</td>
<td>2603</td>
</tr>
<tr>
<td>0x02</td>
<td>create states: (c,512,2), - deleting d (b,256,1), (a,0,0)</td>
<td>811</td>
</tr>
<tr>
<td>0x03</td>
<td>access states a,b,c,e</td>
<td>1805</td>
</tr>
<tr>
<td>0x04</td>
<td>access state d - not present so decompression failure</td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>create states: (large, 2048,0) - deleting a, b, c, e</td>
<td>2057</td>
</tr>
</tbody>
</table>
0x06 access large state

Note that as new states are created some of the existing states will be pushed out of the compartment due to lack of memory.

4.3 Multiple compartments

This section gives assembly code to test the interaction between multiple SigComp compartments. The code is designed to test that the following boundary cases have been correctly implemented:

1. The same state item is saved in more than one compartment.

2. A state item stored in multiple compartments has the same state identifier but a different state_retention_priority in each case.

3. A state item is deleted from one compartment but still belongs to a different compartment.

4. A state item belonging to multiple compartments is deleted from every compartment to which it belongs.

The test requires a total of three compartments to be available, which will be referred to as Compartment 0, Compartment 1 and Compartment 2. Each of the three compartments should have a state_memory_size of 2048 bytes.

The assembly code for the test is given below:

```
at (64)
:byte_copy_left pad (2)
:byte_copy_right pad (2)
:type pad (1)
:type_lsb pad (1)

at (127)
:decompression_failure

at (128)

MULTILOAD (byte_copy_left, 2, state_start, state_end)
INPUT-BYTES (1, type_lsb, decompression_failure)
COMPARE ($type, 3, create_state, overwrite_state, temp)
```

:temp
COMPARE ($type, 5, overwrite_state, access_state, error_conditions)

:create_state
; starting byte identified by $type according to input:
; Input  0x00  0x01  0x02
; $type  512   513   514

ADD ($type, state_start)
STATE-CREATE (448, $type, 0, 6, 0)

; create state again, beginning in different place in buffer
; starting byte identified by $type according to input:
; Input  0x00  0x01  0x02
; $type  515   516   517

ADD ($type, 3)
STATE-CREATE (448, $type, 0, 6, 0)

; create a third time beginning in different place again
; starting byte identified by $type according to input:
; Input  0x00  0x01  0x02
; $type  516   517   515

SUBTRACT ($type, temp_one)
REMAINDER ($type, 3)
ADD ($type, temp_two)
STATE-CREATE (448, $type, 0, 6, 0)

:common_state

STATE-CREATE (448, temp_three, 0, 6, $type)
JUMP (end)

:overwrite_state

STATE-CREATE (1984, 32, 0, 6, 0)
JUMP (end)

:access_state

STATE-ACCESS (state_identifier_c, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_d, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_f, 6, 0, 0, 0, 0)
STATE-ACCESS (state_identifier_g, 6, 0, 0, 0, 0)

:end

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)
:error_conditions

COMPARE ($type, 7, access_a, access_b, access_e)

:access_a

STATE-ACCESS (state_identifier_a, 6, 0, 0, 0, 0)  JUMP (end)

:access_b

STATE-ACCESS (state_identifier_b, 6, 0, 0, 0, 0)  JUMP (end)

:access_e

STATE-ACCESS (state_identifier_e, 6, 0, 0, 0, 0)  JUMP (end)

at (512)

:state_start

byte (0, 1, 2, 3, 4, 5, 6)

:state_end

set (temp_one, (state_start + 2)); = 514
set (temp_two, (state_start + 3)); = 515
set (temp_three, (state_end - 1)); = 518

:state_identifier_a ; start state at 512
byte (172, 166, 11, 142, 178, 131)

:state_identifier_b ; start state at 513
byte (157, 191, 175, 198, 61, 210)

:state_identifier_c ; start state at 514
byte (52, 197, 217, 29, 83, 97)

:state_identifier_d ; start state at 515
byte (189, 214, 186, 42, 198, 90)

:state_identifier_e ; start state at 516
The above code must be executed a total of 9 times in order to complete the test. Each time the code is executed a 1-byte compressed message \( N \) should be provided, taking the values 0x00 to 0x08 in ascending order (so the compressed message should be 0x00 the first time the code is run, 0x01 the second and so on).

If the code makes a state creation request then the state must be saved in Compartment \( (N \mod 3) \).

When the compressed message is 0x00, 0x01 or 0x02 the code makes four state creation requests in compartments 0, 1 and 2 respectively. This creates a total of seven distinct state items referred to as State a through to state g. The states should be distributed amongst the three compartments as illustrated in Figure 1 (note that some states belong to more than one compartment).

When the compressed message is 0x03 or 0x04 the code overwrites all of the states in compartments 0 and 1 respectively. This means that states a, b and e will be unavailable because they are no longer present in any of the three compartments.

When the compressed message is 0x05 the code checks that the states c, d, f and g are still available. Decompression should terminate successfully in this case.

When the compressed message is 0x06, 0x07 or 0x08 the code attempts to access states a, b and e respectively. Decompression failure should occur in this case because the relevant states are no longer available.

The cost in UDVM cycles for each compressed message is given below (except for messages 0x06, 0x07 and 0x08 where decompression failure should to occur):

<table>
<thead>
<tr>
<th>Compressed message:</th>
<th>Cost in UDVM cycles:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08</td>
<td>1809 1809 1809 1993 1994 1804 N/A N/A N/A</td>
</tr>
</tbody>
</table>
Figure 1: States created in the three compartments

4.4 Accessing RFC 3485 State

This section gives assembly code to test accessing SIP-SDP static dictionary state [3]. The code first accesses the state and then outputs result.
at (32)

:input pad (1)
:input2 pad (1)
:input3 pad (1)

at (128)

STATE-ACCESS (sip_dictionary, 20, 0xcfe, 1, input, 0)
STATE-ACCESS (sip_dictionary, 6, 0xcff, 1, input2, 0)
STATE-ACCESS (sip_dictionary, 12, 0xd00, 1, input3, 0)

OUTPUT (input, 3)

END-MESSAGE (0, 0, 0, 0, 0, 0, 0)

:sip_dictionary
byte (0xfb, 0xe5, 0x07, 0xdf, 0xe5, 0xe6)
byte (0xaa, 0x5a, 0xf2, 0xab, 0xb9, 0x14)
byte (0xce, 0xaa, 0x05, 0xf9, 0xc, 0xe6)
byte (0x1b, 0xa5)

The output of the code is 0x5349 50, and the cost is 11 UDVM cycles.

4.5 Bytecode state creation

This section gives assembly code to test storing bytecode using END-MESSAGE and later loading the bytecode using a partial state identifier within the SigComp header. The assembly code is designed so that it includes testing changing bytes to be stored after the state create request, changing the bytes of the bytecode before they are executed, loading byte code in special memory areas and correct initialization order of the UDVM.

1. Four items of bytecode state are created. The bytes to be saved change after the first state create request. The uploaded bytecode is modified before execution.

2. The bytecode is loaded using partial state identifier and modified before execution.

3. The bytecode is loaded before 128 using partial state identifier.

4. The bytecode is loaded using partial state identifier. Part of the loaded memory is reserved area, which is overwritten after loading the bytecode.
5. The byte loading fails because the partial state identifier is too short.

```
at (30)
 :save_area1
 set (saved_instr1, (save_area1 + (code_start2 - start_saved))) ; = 33

at (80)
 :save_area2
 set (saved_instr2, (save_area2 + (code_start2 - start_saved))) ; = 83

at (128)
 :code_start

COPY (start_saved, saved_len, save_area1)
 ; copy ‘ok2’, OUTPUT (save_area2,3) END-MESSAGE
 ; to position 30 and create as state
STATE-CREATE (saved_len, save_area1, saved_instr1, 6, 10)

set (modify1, (save_area1 + 5)) ; = 35
LOAD (modify1, 0x1e03)
 ; modify save_area2 to be save_area1 in the
 ; created state

COPY (start_saved, saved_len, save_area2)
STATE-CREATE (saved_len, save_area2, saved_instr2, 20, 10)
STATE-CREATE (saved_len, save_area2, saved_instr2, 12, 10)
 ; copy ‘ok2’, OUTPUT (save_area2,3) END-MESSAGE
 ; to position 80 and create as state twice with
 ; min access len 20 and 12

JUMP (modify)

:ok1
byte (0x4f, 0x4b, 0x31)

set (after_output_minus1, (after_output - 1))

:modify
INPUT-BYTES (1, after_output_minus1, decompression_failure)
 ; Input overwrites the next instruction
OUTPUT (ok1, 3) ; Now is OUTPUT (ok1, 2) so output is 0x4f4b

:after_output

; Save from ok1 to the opcode of END-MESSAGE
```
set (modify_len, ((after_output + 1) - ok1)) ; = 13

END-MESSAGE (0, 0, modify_len, ok1, modify, 6, 10)
    ; Save 'ok1', INPUT-BYTES, OUTPUT as state

set (saved_len, (end_saved - start_saved)) ; = 8

:start_saved
byte (0x4f, 0x4b, 0x32)

code_start2

; Translated bytecode for OUTPUT (save_area2, 3)
byte (0x22, 0xa0, 0x50, 0x03)

; Translated bytecode for END-MESSAGE (0, 0, 0, 0, 0, 0, 0)
; The zeros do not need to be sent because UDVM is initialised to 0
byte (0x23)

:end_saved
decompression_failure

The outputs and cycle usages are:

<table>
<thead>
<tr>
<th>Message</th>
<th>Output</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4f4b</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>0x4f4b 31</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0x4f4b 32</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0x0000 32</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>None</td>
<td>Decompression failure</td>
</tr>
</tbody>
</table>

First message: mnemonic code annotated above

0xf804 6112 aobe 081e 2008 1e21 060a 0e23 be03 12a0 be08 a050 2008
0xa050 a053 140a 2008 a050 a053 0c0a 1606 004f 4b31 1c01 a0b3 fc22
0xa0a8 0323 0000 0da0 a8a0 ab06 0a4f 4b32 22a0 5003 2302

Second message: access and run last state saved by previous message - 'ok1', INPUT-BYTES, OUTPUT, END-MESSAGE.

0xf905 b88c e72c 9103

Third message: access and run state from save_area2 with 12 bytes of state identifier - 'ok2', INPUT-BYTES, OUTPUT, END-MESSAGE.
Fourth message: access and run state from save_area1. The state is 'ok2', INPUT-BYTES, OUTPUT, END-MESSAGE but the first two bytes should be overwritten when initialising UDVM memory.

Fifth Message: attempt to access state from save_area2 with fewer than 20 bytes of state identifier.

5. Security considerations

This document describes torture tests for the SigComp protocol RFC-3320 [2]. Consequently the security considerations for this document match those of SigComp.

6. Acknowledgements

Thanks to Richard Price and Pekka Pessi for test contributions and to Pekka Pessi and Cristian Constantin who served as committed working group document reviewers.

7. References


Appendix A. UDVM bytecode for the torture tests

The following sections list the raw UDVM bytecode generated for each test. The bytecode is presented in the form of a complete SigComp message, including the appropriate header. It is followed by input messages, the output they produce and where the decompression succeeds the number of cycles used.

In some cases the test is designed to be run several times with different compressed messages appended to the code. In the cases where multiple whole messages are used for a test e.g. Appendix A.2.3, these are supplied. In the case where decompression failure occurs, the high level reason for it is given as a reason code defined in NACK [6].

Note that the different assemblers can output different bytecode for the same piece of assembly code, so a valid assembler can produce results different from those presented below. However, the following bytecode should always generate the same results on any UDVM.

A.1 Instructions

A.1.1 Bit manipulation
A.1.2 Arithmetic

Input: 0x00
Output: 0x0000 0000 0000 0004
Cycles: 25

Input: 0x01
DECOMPRESSION-FAILURE DIV_BY_ZERO

Input: 0x02
DECOMPRESSION-FAILURE DIV_BY_ZERO

A.1.3 Sorting
A.1.4 SHA-1

Input: None
Output: 0xa999 3e36 4706 816a ba3e 2571 7850 9c26 9c0 9cd0 d89d
       0xe89b 3e44 1c3b d26e bae f93 29e5 e546 70f1
       0x12ff 347b 4f27 d69e 1f32 8ef 4b55 73e3 66e1 122f
       0x4f46 0452 ebb5 3639 4f46 0452 ebb5 3639 4f46 0452
Cycles: 17176

A.1.5 LOAD and MULTILOAD
A.1.6 COPY

Input: None
Output: 0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Cycles: 216

A.1.7 COPY-LITERAL and COPY-OFFSET

Input: None
Output: 0x0141 1414 1411 1414 1414 1414 1414 1414 1414 1414 1414 1414 1414 1414 1414 1414
Cycles: 216
A.1.8  MEMSET

0xf801 810e 8687 0ea0 42a0 8115 86a0 8100 0115 a081 0f86 0f22 8710 0x23

Input: None
Output: 0x8040 4f5e 6d7c 8b9a a9b8 c7d6 e5f4 0312
Cycles: 166

A.1.9  CRC

0xf801 8115 a046 1801 0115 a05e 1487 011c 02a0 4413 1b62 a046 2c0e 0x23

Input: 0x62cb
Output: None
Cycles: 95

Input: 0xabcd
DECOMPRESSION FAILURE USER_REQUESTED (CRC mismatch)

A.1.10 INPUT-BITS

0xf801 511d 62a0 4614 22a0 4602 0622 010a 2207 0622 0116 ee23

Input: 0x932e ac71
Output: 0x0000 0002 0002 0013 0000 0003 001a 0038
Cycles: 66

A.1.11 INPUT-HUFFMAN

0xf801 d11e a046 1c02 6200 6262 6200 ff00 22a0 4602 0622 010a 2207 0x0622 0116 e623

Input: 0x932e ac71 66d8 6f
Output: 0x0000 0003 0008 04d7 0002 0003 0399 30fe
Cycles: 84
A.1.12  INPUT-BYTES

Input: 0x932e ac71 66d8 6fb1 592b dc9a 9734 d847 a733 874e
       0x1bcb cd51 b5dc 9659 9d6a
Output: 0x0000 932e 0001 b166 d86f b100 1a2b 0003 9a97 34d8
        0x0007 0001 3387 4e00 08dc 9651 b5dc 9600 599d 6a
Cycles: 130

A.1.13  Stack Manipulation

Input: None
Output: 0x0003 0002 0001 0042 0042 0001 0001 0000 0001 0001
Cycles: 40

A.1.14  Program Flow

Input: None
Output: 0x0001 0102 0203 0304 0405 0506 0707 0708 0808 0909
Cycles: 131
A.1.15 State creation

Input: 0x01
Output: None
Cycles: 23

Input: 0x02
Output: None
Cycles: 14

Input: 0x03
Output: None
Cycles: 24

Input: 0x0405
DECOMPRESSION-FAILURE  INVALID_STATE_ID_LENGTH

Input: 0x0415
DECOMPRESSION-FAILURE  INVALID_STATE_ID_LENGTH

Input: 0x0406
Output: None
Cycles: 23

Input: 0x09
Output: None
Cycles: 34

Input: 0x1e06
Output: None
Cycles: 46

Input: 0x1e07
Output: None
Cycles: 47

Input: 0x1e14
Output: None
Cycles: 60
A.1.16 STATE-ACCESS

Set up bytecode:

```
0xf819 0123 0000 1089 0014 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Input: None

```
Output: 0x7465 7374
Cycles: 26
Input: 0x01
Output: 0x7465 7374
Cycles: 15
```
A.2 Dispatcher tests

A.2.1 Useful Values

```
0xf805 a10e 8620 0ea0 4221 1742 01f8 06f8 1743 00f2 060c 1744 00ec
0x3fec 1744 a3c0 e607 e61c 01a0 479f df06 2340 0723 0112 2001 631c
0x01a0 499f cf0e a044 4306 2203 0822 0806 22a3 e808 2241 0622 6407
0x22a3 d612 2062 2023 0000 a3c0 8687 06

Input: None
Output: None
Cycles: 966

0xf939 b90f db05 4b

Input: 0x0000
Output: None
Cycles: cycles_per_bit * 1072

Input: 0x0001
DECOMPRESSION-FAILURE CYCLES_EXHAUSTED

Input: 0x0100
DECOMPRESSION-FAILURE SEGFAULT
```

A.2.2 Cycles Checking

```
0xf801 a10f 8604 2029 0022 12a0 4402 6014 02a0 6423 22a0 4402 0622
0x0116 ef

Input: None
DECOMPRESSION-FAILURE CYCLES_EXHAUSTED
```
A.2.3 Message-based transport

0xf8

Input: None
DECOMPRESSION-FAILURE MESSAGE_TOO_SHORT

0xf800

Input: None
DECOMPRESSION-FAILURE MESSAGE_TOO_SHORT

0xf800 e106 0011 2200 0223 0000 0000 0000 01

Input: None
Output: decompression_memory_size
Cycles: 5

0xf800 f106 0011 2200 0223 0000 0000 0000 01

Input: None
DECOMPRESSION-FAILURE MESSAGE_TOO_SHORT

0xf800 e006 0011 2200 0223 0000 0000 0000 01

Input: None
DECOMPRESSION-FAILURE INVALID_CODE_LOCATION

0xf800 ee06 0011 2200 0223 0000 0000 0000 01

Input: None
Output: decompression_memory_size
Cycles: 5

A.2.4 Stream-based transport
The above stream contains two messages:

Output: decompression_memory_size
Cycles: 11

Output: decompression_memory_size
Cycles: 11

0xf8ff ff

Input: None
DECOMPRESSION-FAILURE          MESSAGE_TOO_SHORT

0xf800 ffff

Input: None
DECOMPRESSION-FAILURE          MESSAGE_TOO_SHORT

0xf801 8108 0002 2200 0222 a092 0523 ffff 0000 0000 00ff 00ff

Input: None
DECOMPRESSION-FAILURE          MESSAGE_TOO_SHORT

0xf801 7008 0002 2200 0222 a092 0523 ffff 0000 0000 00ff 00ff

Input: None
DECOMPRESSION-FAILURE          INVALID_CODE_LOCATION

A.2.5  Input past the end of a message

0xf803 210e 86a0 460e a042 a04d 1d09 a046 0alc 07a0 4606 001d 07a0
0x46ff 1c02 a046 fa22 a046 021d 10a0 4606 001d 08a0 46ff 22a0 4701
0x23

Input: 0xffffa 0068 6921
Output: 0x6869 21
Cycles: 23

Input: 0xffffa 0068 69
DECOMPRESSION-FAILURE USER_REQUESTED (not enough bits)

A.3 State handler tests

A.3.1 SigComp feedback mechanism

```
0xf805 031c 01a0 41a0 5517 6001 070e a04f 0ea0 42a4 7f16 0e0e a042
0xa4ff 15a0 44a0 7f01 010e a0c3 a801 0ea0 c5a6 000e a0cc ac00 0ea0
0xd9b4 000e a0ee b500 15a0 c606 0001 15a0 cd0c 0001 15a0 da14 0001
0x23a0 42a0 c3
```

Input: 0x00
Output: None
Cycles: 52

Input: 0x01
Output: None
Cycles: 179

A.3.2 State memory management
Input: 0x00
Output: None
Cycles: 811

Input: 0x01
Output: None
Cycles: 2603

Input: 0x02
Output: None
Cycles: 811

Input: 0x03
Output: None
Cycles: 1805

Input: 0x04
DECOMPRESSION-FAILURE STATE_NOT_FOUND

Input: 0x05
Output: None
Cycles: 2057

Input: 0x06
Output: None
Cycles: 1993
A.3.3 Multiple compartments

0xf81b 110f 8602 89a2 071c 01a0 45f9 1762 030d 3d06 1762 0537 86a0
0x6806 2289 20a1 c062 0006 0006 2203 20a1 c062 0006 0007 22a2 020a
0x2203 0622 a203 20a1 c062 0006 0020 a1c0 a206 0006 6216 2b20 a7c0
0x2000 0600 1622 1fa2 1306 0000 0000 1fa2 1906 0000 0000 1fa2 2506
0x0000 0000 1fa2 2b06 0000 0000 2300 0000 0000 0000 1762 0706 101a
0x1fa2 0706 0000 0000 16ea 1fa2 0d06 0000 0000 16e0 1fa2 1f06 0000
0x0000 169f d600 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Input: 0x00
Output: None
Cycles: 1809

Input: 0x01
Output: None
Cycles: 1809

Input: 0x02
Output: None
Cycles: 1809

Input: 0x03
Output: None
Cycles: 1993

Input: 0x04
Output: None
Cycles: 1994

Input: 0x05
Output: None
Cycles: 1804

Input: 0x06
DECOMPRESSION-FAILURE  STATE_NOT_FOUND

Surtees & West Expires April 1, 2006 [Page 70]
A.3.4 Accessing RFC 3485 State

Input: None
Output: 0x5349 50
Cycles: 11

A.3.5 Bytecode state creation

Input: None
Output: 0x5349 50
Cycles: 11
0xf804 6112 a0be 081e 2008 1e21 060a 0e23 be03 12a0 be08 a050 2008
0xa050 a053 140a 2008 a050 a053 0c0a 1606 004f 4b31 1c01 a0b3 fc22
0xa0a8 0323 0000 0da0 a8a0 ab06 0a4f 4b32 22a0 5003 2302

Input: None
Output: 0x4f4b
Cycles: 67

0xf905 b88c e72c 9103

Input: None
Output: 0x4f4b 31
Cycles: 7

0xfb24 63cd ff5c f8c7 6df6 a289 ff

Input: None
Output: 0x4f4b 32
Cycles: 5

0xf95b 4b43 d567 83

Input: None
Output: 0x0000 32
Cycles: 5

0xf9de 8126 1199 1f

Input: None
DECOMPRESSION-FAILURE STATE_NOT_FOUND
Internet-Draft            SigComp Torture Tests           September 2005

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.

This Internet-Draft will expire on April 1, 2006.