Symbolic Crosschecking of Floating-Point and SIMD Code

Peter Collingbourne, Cristian Cadar, Paul H J Kelly

Department of Computing, Imperial College London

4 November, 2010
Outline

- SIMD
- Symbolic Execution
- KLEE-FP
- Evaluation
**SIMD**

- **Single Instruction Multiple Data**
- A popular means of improving the performance of programs by exploiting data level parallelism
- SIMD vectorised code operates over one-dimensional arrays of data called vectors

```c
__m128 c = _mm_mul_ps(a, b);
/* c = { a[0]*b[0], a[1]*b[1], a[2]*b[2], a[3]*b[3] } */
```
Symbolic Execution for SIMD

- Manually translating scalar code into an equivalent SIMD version is a difficult and error-prone task.
- We propose a novel automatic technique for verifying that the SIMD version of a piece of code is equivalent to its (original) scalar version.
Symbolic Execution

- Symbolic execution tests multiple paths through the program.
- Determines the feasibility of a particular path by reasoning about all possible values using a constraint solver.
- Can verify code correctness by verifying the absence of certain error types (such as array bounds errors or division by zero) on a per-path basis.
Symbolic Execution – Operation

- Each variable may hold either a concrete or a symbolic value.
- Symbolic value – an expression consisting of mathematical or boolean operations and symbols.
- For example, an integer variable $i$ may hold a value such as $x + 3$. 
Symbolic Execution – Example

```c
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
    ;
else
    abort();
```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
    ;
else
    abort();
Symbolic Execution – Example

unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
    ;
else
    abort();
Symbolic Execution – Example

unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
    ;
else
    abort();
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
    assert(x > x - 1);
    if (x > x - 1)
        ;
    else
        abort();
}
Symbolic Execution – Example

unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
    assert(x > x - 1);
    if (x > x - 1)
        ;
    else
        abort();
}
Symbolic Execution – Example

unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
    assert(x > x - 1);
    if (x > x - 1)
        ;
    else
        abort();
}
Symbolic Execution – Example

```c
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
    assert(x > x - 1);
    if (x > x - 1)
        ;
    else
        abort();
}
```
Issues

- SIMD vectorised code frequently makes intensive use of floating point arithmetic
- The current generation of symbolic execution tools lack or have poor support for floating point and SIMD
For the purposes of this work, we need to test *equality*, not inequality.

The requirements for equality of two floating point values are harder to satisfy than for integers.

Usually, the two numbers need to be built up in the same way to be sure of equality.

This suits us just fine for verification purposes.
KLEE

- Tool for symbolic testing of C and C++ code [Cadar, Dunbar, Engler, OSDI 2008]
- Based on the LLVM compiler framework
- `svn co http://llvm.org/svn/llvm-project/klee/trunk klee`
- Supports integer constraints only; symbolic FP not allowed
KLEE-FP: our modified version of KLEE

- Our improvements to KLEE:
  - Symbolic construction of floating-point operations
  - A set of expression matching and canonicalization rules for establishing FP equality
  - Support for SIMD instructions `insertelement`, `extractelement`, `shufflevector`
  - Semantics for a substantial portion of Intel SSE instruction set

- Related contributions to LLVM:
  - Atomic intrinsic lowering
  - An aggressive variant of phi-node folding

- Selected modifications contributed upstream to KLEE/LLVM
- `git clone git://git.pcc.me.uk/~peter/klee-fp.git`
Scalar/SIMD Implementation

```c
void zlimit(int simd, float *src, float *dst,  
            size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
```
Scalar/SIMD Implementation

```c
void zlimit(int simd, float *src, float *dst, 
    size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
```
Scalar/SIMD Implementation

```c
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```

<table>
<thead>
<tr>
<th>srcv</th>
<th>1.2432</th>
<th>-3.6546</th>
<th>2.7676</th>
<th>-9.5643</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero4</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>cmpv</td>
<td>111...111</td>
<td>000...000</td>
<td>111...111</td>
<td>000...000</td>
</tr>
<tr>
<td>dstv</td>
<td>1.2432</td>
<td>0.0000</td>
<td>2.7676</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Scalar/SIMD Implementation

```c
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```

![Diagram](attachment:diagram.png)
Scalar/SIMD Implementation

```c
void zlimit(int simd, float *src, float *dst,
           size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
```
Scalar/SIMD Implementation

while (size) {
    *dst = *src > 0.f ? *src : 0.f;
    src++; dst++; size--;
}

Select
  FOgt
  ReadLSB
  ReadLSB 0
  0 src

0 src
Scalar/SIMD Implementation

\[
\text{And}(\text{SExt}(P^1), X) \rightarrow \text{Select}(P^1, X, 0)
\]
Cross-checking SIMD Code

```c
int main(void) {
    float src[64], dstv[64], dsts[64];
    uint32_t *dstvi = (uint32_t *) dstv;
    uint32_t *dtsi = (uint32_t *) dsts;
    unsigned i;
    klee_make_symbolic(src, sizeof(src), "src");
    zlimit(0, src, dsts, 64);
    zlimit(1, src, dstv, 64);
    for (i = 0; i < 64; ++i)
        assert(dstvi[i] == dtsi[i]);
}
```
Checking Equality

\[-(I2F(x << 1) + f 2.0 *_f I2F(0 - x))
\]

\[= I2F(x \times 2) + f 2.0 *_f I2F(\sim x + 1))\]

\[\rightarrow \neg(x << 1 = x \times 2 \land 0 - x = \sim x + 1)\]
Checking Equality

\[ \neg (I2F(x << 1) + f \, 2.0 * f \, I2F(0 - x)) \]
\[ = I2F(x \times 2) + f \, 2.0 * f \, I2F(\sim x + 1)) \]
\[ \rightarrow \neg (x << 1 = x \times 2 \land 0 - x = \sim x + 1) \]
Floating Point Operations

- New nodes:
  - FAdd
  - FSub
  - FMul
  - FDiv
  - FRem
  - FPToSI
  - FPToUI
  - SIToFP
  - UIToFP
  - FPExt
  - FPTrunc
  - FCmp

- Outcome sets:
  $$O = \{<,=,>,\text{UNO}\}$$

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>CmpInst::Predicate</th>
<th>FCmp operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0eq(X, Y)</td>
<td>FCMP_OEQ = 0 0 0 1</td>
<td>FCmp(X, Y, {=} )</td>
<td>Ordered =</td>
</tr>
<tr>
<td>F0lt(X, Y)</td>
<td>FCMP_OLT = 0 1 0 0</td>
<td>FCmp(X, Y, {&lt;} )</td>
<td>Ordered &lt;</td>
</tr>
<tr>
<td>F0le(X, Y)</td>
<td>FCMP_OLE = 0 1 0 1</td>
<td>FCmp(X, Y, {&lt;,=} )</td>
<td>Ordered ≤</td>
</tr>
<tr>
<td>FUno(X, Y)</td>
<td>FCMP_UNO = 1 0 0 0</td>
<td>FCmp(X, Y, {UNO})</td>
<td>Unordered test</td>
</tr>
</tbody>
</table>
Expression Transformation Rules

18 rules, including:

\[
\text{And}(\text{FCmp}(X, Y, O_1), \text{FCmp}(X, Y, O_2)) \rightarrow \text{FCmp}(X, Y, O_1 \cap O_2)
\]
\[
\text{Or}(\text{FCmp}(X, Y, O_1), \text{FCmp}(X, Y, O_2)) \rightarrow \text{FCmp}(X, Y, O_1 \cup O_2)
\]
\[
\text{Eq}(\text{FCmp}(X, Y, O), \text{false}) \rightarrow \text{FCmp}(X, Y, O \setminus O)
\]
\[
\text{FOeq}(\text{SIToFP}(X), C) \rightarrow \text{Eq}(X, \text{FPToS}(C))
\]
\[
\text{FOeq}(\text{UIToFP}(X), C) \rightarrow \text{Eq}(X, \text{FPToUI}(C))
\]
\[
\text{And}((S\text{Ext}(P^1), X) \rightarrow \text{Select}(P^1, X, 0)
\]
Category Analysis

\[ C = \{ \text{NaN}, -\infty, -, 0, +, +\infty \} \]

\[ + \in \text{cat}(x) \quad + \in \text{cat}(y) \]
\[ \{+, +\infty \} \subseteq \text{cat}(x + y) \]

\[ \text{cat}(x) = \{0, -\} \quad \text{cat}(y) = \{0, +\} \]
\[ \neg(x > y) \]
SSE Intrinsic Lowering

- Total of 37 intrinsics supported
- Implemented via a lowering pass that translates the intrinsics into standard LLVM instructions

Input code:

\[
\%res = \text{\texttt{call}} \ <8 \times \text{i16}> \ @\text{llvm.x86.sse2.psl.li.w}(\<8 \times \text{i16}> \%\text{arg}, \text{i32} \ 1)
\]

Output code:

\[
\%1 = \text{\texttt{extractelement}} \ <8 \times \text{i16}> \%\text{arg}, \text{i32} \ 0 \\
\%2 = \text{\texttt{shl}} \ \text{i16} \ %1, \ 1 \\
\%3 = \text{\texttt{insertelement}} \ <8 \times \text{i16}> \ \text{undef}, \ \text{i16} \ %2, \ \text{i32} \ 0 \\
\%4 = \text{\texttt{extractelement}} \ <8 \times \text{i16}> \%\text{arg}, \text{i32} \ 1 \\
\%5 = \text{\texttt{shl}} \ \text{i16} \ %4, \ 1 \\
\%6 = \text{\texttt{insertelement}} \ <8 \times \text{i16}> \ %3, \ \text{i16} \ %5, \ \text{i32} \ 1 \\
\ldots \\
\%22 = \text{\texttt{extractelement}} \ <8 \times \text{i16}> \%\text{arg}, \text{i32} \ 7 \\
\%23 = \text{\texttt{shl}} \ \text{i16} \ %22, \ 1 \\
\%\text{res} = \text{\texttt{insertelement}} \ <8 \times \text{i16}> \ %21, \ \text{i16} \ %23, \ \text{i32} \ 7
\]
Phi Node Folding

\[
\text{val} = \text{silhData}[x] \ ? \ ts \ : \ \text{val} < \text{delbound} \ ? \ 0 \ : \ \text{val};
\]

```
bb156:
   ...
   \%106 = \text{load float} * \%scevgep345346, \text{align} 4
   \%107 = \text{load i8} * \%scevgep351, \text{align} 1
   \%108 = \text{icmp eq i8} \%107, 0
   \text{br i1} \%108, \text{label} \%bb158, \text{label} \%bb163

bb158:
   \%109 = \text{fcmp uge float} \%106, \%51
   \%\text{iftmp.388.0} = \text{select i1} \%109, \text{float} \%106,
      \text{float} 0.000000e+00
   \text{br label} \%bb163

bb163:
   \%\text{iftmp.387.0} = \text{phi float} [ \%\text{iftmp.388.0}, \%bb158 ],
      [ \%49, \%bb156 ]
   ...
```
Phi Node Folding

\[
\text{val} = \text{silhData}[x] \ ? \ \text{ts} : \ \text{val} < \text{delbound} \ ? \ 0 : \ \text{val};
\]

```
bb156:
...
%106 = \text{load float}* \%scevgtep345346, \text{align} 4
%107 = \text{load i8}* \%scevgtep351, \text{align} 1
%108 = \text{icmp eq i8} \ %107, 0
%109 = \text{fcmp uge float} \ %106, %51
%if\text{tmp}.388.0 = \text{select i1} \ %109, \text{float} \ %106,
\text{float} 0.000000e+00
%if\text{tmp}.387.0 = \text{select i1} \ %108, %if\text{tmp}.388.0, %49
...
```

- 13 \times 13 \text{matrix:}

\[
2^{169} \rightarrow 1
\]
Evaluation

▶ We evaluated our technique on a set of benchmarks that compare scalar and SIMD variants of code developed independently by third parties

▶ The code base that we selected was OpenCV 2.1.0, a popular C++ open source computer vision library
Evaluation

- Out of the twenty OpenCV source code files containing SSE code, we selected ten files upon which to build benchmarks.
- Crosschecked 49 SIMD/SSE implementations against scalar versions.
  - Proved the *bounded* equivalence (i.e. verified) 39
  - Found inconsistencies in the other 10
# Evaluation – Coverage

<table>
<thead>
<tr>
<th>Source File (src/)</th>
<th># SIMD</th>
<th>Cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cv/cvcorner.cpp</td>
<td>44</td>
<td>100%</td>
</tr>
<tr>
<td>cv/cvfilter.cpp</td>
<td>1332</td>
<td>N/A</td>
</tr>
<tr>
<td>cv/cvimg warp.cpp</td>
<td>1070</td>
<td>74.6%</td>
</tr>
<tr>
<td>cv/cvmoments.cpp</td>
<td>35</td>
<td>100%</td>
</tr>
<tr>
<td>cv/cvmorph.cpp</td>
<td>1220</td>
<td>43.6%</td>
</tr>
<tr>
<td>cv/cvmotempl.cpp</td>
<td>43</td>
<td>100%</td>
</tr>
<tr>
<td>cv/cvpyramids.cpp</td>
<td>125</td>
<td>44.0%</td>
</tr>
<tr>
<td>cv/cvstereobm.cpp</td>
<td>270</td>
<td>53.3%</td>
</tr>
<tr>
<td>cv/cvthresh.cpp</td>
<td>238</td>
<td>100%</td>
</tr>
<tr>
<td>cxcore/cxmatmul.cpp</td>
<td>352</td>
<td>100%</td>
</tr>
</tbody>
</table>
### OpenCV – Verified up to a certain size

<table>
<thead>
<tr>
<th>#</th>
<th>Bench</th>
<th>Algo</th>
<th>K</th>
<th>Fmt</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>morph</td>
<td>dilate</td>
<td>R</td>
<td>u8</td>
<td>4 × 1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>12 × 12</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>NR</td>
<td>s16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>13 × 13</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>erode</td>
<td>R</td>
<td>s16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>NR</td>
<td>s16</td>
<td>14 × 14</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>13 × 13</td>
</tr>
<tr>
<td>11</td>
<td>pyramid</td>
<td></td>
<td></td>
<td>u8</td>
<td>8 × 2 → 4 × 1</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>nearest neighbor</td>
<td></td>
<td>u8</td>
<td>19 × 19</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>19 × 19</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>linear</td>
<td></td>
<td>u8</td>
<td>19 × 19</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>19 × 19</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>cubic</td>
<td></td>
<td>u8</td>
<td>19 × 19</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>19 × 19</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>19 × 19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Bench</th>
<th>Algo</th>
<th>K</th>
<th>Fmt</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>resize</td>
<td>linear</td>
<td>s16</td>
<td>4 × 4</td>
<td>8 × 8</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>cubic</td>
<td>s16</td>
<td>4 × 4</td>
<td>8 × 8</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td>f32</td>
<td>4 × 4</td>
<td>8 × 8</td>
</tr>
<tr>
<td>27</td>
<td>silhouette</td>
<td></td>
<td>u8 f32</td>
<td>4 × 4</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>thresh</td>
<td>BINARY</td>
<td>u8</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>BINARY_INV</td>
<td>f32</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>TRUNC</td>
<td>u8</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>TOZERO</td>
<td>f32</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>TOZERO_INV</td>
<td>u8</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
<td>f32</td>
<td>13 × 13</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>transff.43</td>
<td></td>
<td>f32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>transff.44</td>
<td></td>
<td>f32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenCV – Mismatches found

<table>
<thead>
<tr>
<th>#</th>
<th>Bench</th>
<th>Algo</th>
<th>K</th>
<th>Fmt</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>eigenval</td>
<td>f32</td>
<td>4</td>
<td>4 × 4</td>
<td></td>
<td>Precision</td>
</tr>
<tr>
<td>2</td>
<td>harris</td>
<td>f32</td>
<td>4</td>
<td>4 × 4</td>
<td></td>
<td>Precision, associativity</td>
</tr>
<tr>
<td>3</td>
<td>morph</td>
<td>dilate</td>
<td>R</td>
<td>f32</td>
<td>4 × 1</td>
<td>Order of min/max operations</td>
</tr>
<tr>
<td>4</td>
<td>morph</td>
<td>dilate</td>
<td>NR</td>
<td>f32</td>
<td>4 × 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>morph</td>
<td>erode</td>
<td>R</td>
<td>f32</td>
<td>4 × 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>thresh</td>
<td>TRUNC</td>
<td>f32</td>
<td>4 × 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>pyramid</td>
<td>f32</td>
<td>16</td>
<td>16 × 2 → 8 × 1</td>
<td>Associativity, distributivity</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>resize</td>
<td>linear</td>
<td>u8</td>
<td>4 × 4 → 8 × 8</td>
<td>Precision</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>transsf.43</td>
<td>s16 f32</td>
<td></td>
<td></td>
<td></td>
<td>Rounding issue</td>
</tr>
<tr>
<td>10</td>
<td>transcf.43</td>
<td>u8 f32</td>
<td></td>
<td></td>
<td></td>
<td>Integer/FP differences</td>
</tr>
</tbody>
</table>
**morph (f32) and thresh (TRUNC, f32)**

- **std::{min, max}, {MIN, MAX}PS:**

  \[
  \begin{align*}
  \min(X, Y) &= \text{Select}(\text{F0lt}(X, Y), X, Y) \\
  \max(X, Y) &= \text{Select}(\text{F0lt}(Y, X), X, Y) \\
  \min(X, \text{NaN}) &= \text{NaN} \\
  \min(\text{NaN}, Y) &= Y \\
  \min(\min(X, \text{NaN}), Y) &= \min(\text{NaN}, Y) = Y \\
  \min(X, \min(\text{NaN}, Y)) &= \min(X, Y)
  \end{align*}
  \]
morph (f32) and thresh (TRUNC, f32)

- std::{min, max}, {MIN, MAX} PS:

  \[
  \begin{align*}
  \text{min}(X, Y) &= \text{Select}(\text{F0lt}(X, Y), X, Y) \\
  \text{max}(X, Y) &= \text{Select}(\text{F0lt}(Y, X), X, Y) \\
  \text{min}(X, \text{NaN}) &= \text{NaN} \\
  \text{min}(\text{NaN}, Y) &= Y \\
  \text{min}(\text{min}(1, \text{NaN}), 2) &= \text{min}(\text{NaN}, 2) = 2 \\
  \text{min}(1, \text{min}(\text{NaN}, 2)) &= \text{min}(1, 2) = 1
  \end{align*}
  \]
Conclusion and Future Work

- KLEE-FP extends KLEE with floating point/SIMD capabilities
- Has been used to find bugs in code extracted from the wild
- Future work may involve:
  - Inequalities
    - Interval arithmetic
    - Affine arithmetic
  - Floating point counterexamples
  - GPUs: CUDA/OpenCL?
git clone git://git.pcc.me.uk/~peter/klee-fp.git