Using SIFOpt to design a fixed-point implementation of a calculation
Outline

• Fixed-point calculations
• Fixed-point values
  – Semantics of fixed-point values
  – SIF partitionings
• SIFOpt
A Calculation

• Scientists represent calculations using equations:

\[
f_1(x) = 0.0009922 \times x^6 - 0.0187 \times x^5 + \ldots
\]

\[
f_2(x_1, x_2, x_3, x_4) = 0.15 \times x_1 + 0.35 \times x_2 + 0.35 \times x_3 + 0.15 \times x_4
\]

• Programs represent calculations as an (ordered) sequence of operations to be performed:

\[
\begin{align*}
\text{con0} &= 0.15 \times x_1 \\
\text{con1} &= \text{con0} + 0.35 \times x_2 \\
\text{con2} &= \text{con1} + 0.35 \times x_3 \\
\text{con3} &= \text{con2} + 0.15 \times x_4 \\
\end{align*}
\]

\[
\begin{align*}
\text{cos0} &= -0.0187 + 0.0009922 \times x \\
\text{cos1} &= 0.1074 + \text{cos0} \times x \\
\text{cos2} &= -0.1187 + \text{cos1} \times x \\
\text{cos3} &= -0.394 + \text{cos2} \times x \\
\text{cos4} &= -0.03886 + \text{cos3} \times x \\
\text{cos5} &= 1.003 + \text{cos4} \times x \\
\end{align*}
\]
Fixed-point calculations

The sequence of steps for a fixed-point implementation is more complicated than the implementation in floating-point...

int x;
int c1 = (-1285054214 + (((66585 * x)) >> 3));
int c1s = ( c1 >> 16 );
int c2 = (922558975 + (x * c1s));
int c2s = ( c2 >> 14 );
int c3 = (-254906309 + (((x) >> 1) * c2s));
int c3s = ( c3 >> 14 );
int c4 = (-211527139 + (x * ((c3s) >> 1)));
int c4s = ( c4 >> 15 );
int c5 = (-2607850 + (((x) >> 1) * c4s));
int c5s = ( c5 >> 14 );

Fixed-point implementation

\[
c_1 = -0.0187 + 0.0009922 \times x;
\]
\[
c_2 = 0.1074 + x \times c_1;
\]
\[
c_3 = -0.1187 + x \times c_2;
\]
\[
c_4 = -0.394 + x \times c_3;
\]
\[
c_5 = -0.03886 + x \times c_4;
\]
Fixed-point calculations

• At run time:
  – all values are integers
  – all operations are integer operations

• A scaling factor \(2^n\) is associated with every run time value at design time. Every computed value will have the same scaling factor every time it is computed!

• Each run time integer represents a real value: the product of the integer and the scaling factor.
Fixed-point calculations

- The scaling factors of the arguments of a calculation are usually determined by the computational environment (at least for embedded environments.)

\[ f(x_1, x_2, y_1, y_2) = (x_1 + x_2) \cdot (y_1 + y_2) \]
Fixed-point calculations

• Compute: \((x_1 + x_2) \times (y_1 + y_2)\)

  Run time:
  \((832 + 505) \times (298 + 1224) = 2034914\)

  Scaling Factors:
  \(x_1, x_2: 2^{-7}\)
  \(y_1, y_2: 2^{-5}\)
  result: \(2^{-12}\)

  Represents:
  \((6.5 + 3.9453125) \times (9.3125 + 38.25) = 496.8\)
Scaling factors

- Scaling factors move the binary point...
  
  Integer value: 832  
  Scaling factor: $2^{-7}$  
  Represented value: 6.5  

  \[0.00000011010000000\]

  Choice of scaling factor effects value range and granularity...

  Delta: smallest positive incremental value = scaling factor.  
  \[2^{-7} = 0.0078125\]

  Range: [-256 : 255.992]
Fixed-point values

- Run time integer: I
- Design time scaling factor: $f (2^n)$

$$v = I \cdot f$$
$$v_i = I_i \cdot f_i$$

$$v_1 + v_2 = I_1 \cdot f_1 + I_2 \cdot f_2 = (I_1 + I_2) \cdot f \Leftrightarrow f = f_1 = f_2$$

$$v_1 \cdot v_2 = (I_1 \cdot f_1) \cdot (I_2 \cdot f_2) = (I_1 \cdot I_2) \cdot (f_1 \cdot f_2)$$
Fixed-point rules

• For addition:
  – Scaling factors must be the same before adding.
  – Scaling factor of the result is the same as the scaling factor of the addends.

• For multiplication:
  – Scaling factor of the result is the product of the scaling factors of the multiplicands.
  – If the result of multiplication is not twice as wide as the arguments, must pre-scale the arguments to avoid overflow in result.
Alignment operations

• What if we need to add two values which do not have the same scaling factor??

• For example: $2^{-9}$ and $2^{-7}$...

Add alignment operations to the computation's description...

\[ z = (x \gg 2) + y \]
Semantics of bits in a run time fixed-point integer value

Semantics is “meaning” or “interpretation”
Bit position semantics

- Without the scaling factor:
  - Sign, Mantissa, Zero pad

  Read a value from an A/D converter:
  Value is 10 bits
  Machine wordlength is 16 bits
  Value is sign extended to 16 bits

  \[
  \]

  Now – left shift the value 3 positions (for addition alignment?)

  \[
  \]
Bit position semantics

- Incorporate scaling factor:

```
S S S S I I I I I F F F F 0 0 0
```

“SIF Partitioning” value: (4/5/4)
SIF Partitionings

- **Basic SIF partitioning notation:**
  - \( S \) = Number of sign bits
  - \( I \) = Number of integer bits
  - \( F \) = Number of fraction bits

\[(S/I/F)\]

- \( (5/6/5) \)
- \( (1/0/15) \)
- \( (2/5/5) \)
SIF Partitionings

- '+' or '–' before S value:
  - '+' indicates all sign bits are 0 (the value is \( \geq 0 \))
  - '-' indicates all sign bits are 1 (the value is \(< 0\))
- "^n" after ')' indicates an additional shift of the binary point.

- \((+4/4/4)\)
  - 00001111FFFFF
- \((5/0/10)^{-3}\)
  - SSSSSFEEEEEEFF
- \((6/10/0)^{+2}\)
  - SSSSSSSIIIIIIIIIIIIIII
SIF Partitionings

• SIF Partitionings plus wordlength tell us the scaling factor, and semantics of the runtime bits of a value.
Fixed-point implementations of calculations

• A fixed-point implementation of a calculation is an integer implementation:

\[
\begin{align*}
    c0 &= (0x10419 \times x) \gg 3; \\
    c1 &= -1285054214 + c0; \\
    c1s &= c1 \gg 16; \\
    c2 &= (922558975 + (x \times c1s)); \\
    c2s &= (c2 \gg 14); \\
    c3 &= (-509812618 + (x \times c2s)); \\
    c3s &= (c3 \gg 12);
\end{align*}
\]

...it includes alignment and scaling operations which are dependent on the choice of scaling factors. The implementor has to choose scaling factors!
The SIFOpt tool takes a high-level description of a calculation and creates a fixed-point implementation:

- Scaling factors for all computed values
- Integer equivalents for all constants
- Alignment operations for all additions (subtractions)
- Prescaling operations for all multiplications

Requirement: all of the arguments of the computation must be annotated with SIF partitionings.
SIFOpt and wordlength

- The default wordlength is 32.
- There are operators available to change the wordlength – I don't recommend using them!
- To work with 16-bit words – make sure the SIF partitionings have 17 (16 if unsigned) sign bits.
- Wordlength changing will be fixed in a future version of SIFOpt.
SIFOpt – Calculation Specification

```plaintext
var (17/0/15) in0;
var (17/0/15) in1;
var (17/0/15) in2;
var (17/0/15) in3;

var out = 0.15*in0 + 0.35*in1 + 0.35*in2 + 0.15*in3;
```

```plaintext
var (17/0/15) in0;
var (17/0/15) in1;
var (17/0/15) in2;
var (17/0/15) in3;

known w1 = 0.15;
known w2 = 0.35;
known w3 = 0.35;
known w4 = 0.15;

var out0 = w1 * in0;
var out1 = w2 * in1 + out0;
var out2 = w3 * in2 + out1;
var out = w4 * in3 + out2;
```
SIFOpt – Division by Repeated Multiplications

```
var (+24/8/0) N;
var (+24/8/0) D;

var Ns = sbpl( N, 8 );
var Ds = sbpl( D, 8 );

var N0 = Ns;
var D0 = Ds;
var R0 = 2 - D0;

var N1 = N0 * R0;
var D1 = D0 * R0;
var R1 = 2 - D1;

var N2 = N1 * R1;
var D2 = D1 * R1;
var R2 = 2 - D2;
```

```
var (+24/8/0) N;
var (+24/8/0) [1:255] D;

var Ns = sbpl( N, 8 );
var Ds = sbpl( D, 8 );

var N0 = Ns;
var D0 = Ds;
var R0 = 2 - D0;

var N1 = N0 * R0;
var D1 = D0 * R0;
var R1 = 2 - D1;

var N2 = N1 * R1;
var D2 = D1 * R1;
var R2 = 2 - D2;
```
// Input range is 0 to 2*π
var (+16/3/13) [0:6.2832] x;
var c1 = -0.0187 + 0.0009922 * x;
var c1s = asb( c1, 16 );
var c2 = 0.1074 + x*c1s;
var c2s = asb( c2, 14 );
var c3 = -0.1187 + x*c2s;
var c3s = asb( c3, 14 );
var c4 = -0.394 + x*c3s;
var c4s = asb( c4, 15 );
var c5 = -0.03886 + x*c4s;
var c5s = asb( c5, 14 );
var c6 = 1.003 + x*c5s;
var c6s = asb( c6, 14 );
SIFOpt

• Declaring an input/argument variables:

```text
var (1/8/7)      a1;
var (17/8/7)     a2;
var (17/0/15)    a3;
var (17/0/15)^-3 a4;
var (17/15/0)    a5;
var (17/15/0)^+3 a6;
var (+17/4/5)    a7;
```
SIFOpt - variables

$ sifopt -R infile

(...stuff deleted...)

********** Printing list of variables

a1 : (1/8/7) @32 [-255.992:255.992] 2^−23
a2 : (17/8/7) @32 [-255.992:255.992] 2^−7
a3 : (17/0/15) @32 [-0.999969:0.999969] 2^−15
a4 : (17/0/15)^−3 @32 [-0.124996:0.124996] 2^−18
a5 : (17/15/0) @32 [-32767:32767] 2^0
a6 : (17/15/0)^+3 @32 [-262136:262136] 2^3
a7 : (17/4/5) @32 [0:15.9688] 2^−11

********** Finished printing list of variables
SIFOpt – SIF / Range Dependencies

\[
\begin{align*}
&\text{var (}4/4/4\text{) [0:15] x1;} \\
&\text{var (+}4/4/4\text{) [-8:15] x1b;} \\
&\text{var (}4/4/4\text{) [-8:16] x2;} \\
&\text{var (}4/4/4\text{) [-16:15] x3;} \\
&\text{var (}4/4/4\text{) [-4:4.5] x4;} \\
\end{align*}
\]

\$ \text{sifopt -R infile} \\
1: The given range [0:15] is all positive, but the SIF's Ss value is not '+' \\
2: Change SIF (+4/4/4) to (4/4/4) in order to cover range [-8:15] \\
3: Change SIF (4/4/4) to (3/5/4) in order to cover range [-8:16] \\
4: Change SIF (4/4/4) to (3/5/4) in order to cover range [-16:15] \\
5: SIF (4/4/4) is sub-optimal for range [-4:4.5] (5/3/4) would be better
SIFOpt – Variables and Knowns

```plaintext
var v1;
var v2;

known k1 = 3.682;
known k2 = 71.05;
```

The SIFOpt language is a “Single Assignment” language. In a single assignment language, you can only set the value of a variable ONCE.

In a single assignment language it is preferable to always immediately assign a value to a variable when you declare it.
$ sifopt -R infile

********** Printing Tree of Optimizations
OptVar - i1: (17/6/9), [-63.998:63.998]
OptVar - i2: (17/10/5), [-1023.97:1023.97]
OptVar - v1: (12/11/9), [-1087.97:1087.97] Leaf variables: i1, i2
Initialized From:
  OptAddSub - k = -4, kmin = -15, kmax = -4
  OptBinOp: '+' result is: (12/11/9), [-1087.97:1087.97]
  OptVar - i1: (17/6/9), [-63.998:63.998]
  OptResize - m_delta = 0, m_shift = -4 (13/10/5), [-1023.97:1023.97]
  OptVar - i2: (17/10/5), [-1023.97:1023.97]
OptVar - i3: (17/3/3), [-7.875:7.875]
OptVar - v2: (16/11/5), [-1031.84:1031.84] Leaf variables: i2, i3
Initialized From:
  OptAddSub - k = 0, kmin = -15, kmax = 0
  OptBinOp: '+' result is: (16/11/5), [-1031.84:1031.84]
  OptVar - i2: (17/10/5), [-1023.97:1023.97]
  OptResize - m_delta = 0, m_shift = 7 (24/3/3), [-7.875:7.875]
  OptVar - i3: (17/3/3), [-7.875:7.875]

********** Finished printing tree of optimizations

var (17/6/9) i1;
var (17/10/5) i2;
var v1 = i1 + i2;
var (17/3/3) i3;
var v2 = i2 + i3;

********** Printing list of Optimization Trees
var (17/6/9) i1;
var (17/10/5) i2;
var (12/11/9) [-1087.97:1087.97] v1 = (i1 + rsb( i2, 4 ));
var (17/3/3) i3;
var (16/11/5) [-1031.84:1031.84] v2 = (i2 + asb( i3, 7 ));

********** Finished printing list of optimization trees

********** Printing list of variables
i1 : (17/6/9) @32 [-63.998:63.998] 2^-9
i2 : (17/10/5) @32 [-1023.97:1023.97] 2^-5
v1 : (12/11/9) @32 [-1087.97:1087.97] 2^-5
i3 : (17/3/3) @32 [-7.875:7.875] 2^-12
v2 : (16/11/5) @32 [-1031.84:1031.84] 2^-5

********** Finished printing list of variables
SIFOpt - Addition

```
var (17/6/9) i1;
var (17/10/5) i2;
var v1 = i1 + i2;
var (17/3/3) i3;
var v2 = i2 + i3;
```

```
$ sifopt -C infile
********** Printing generated C++ code
int i1;
int i2;
int v1 = (i1 + ((i2) << 4));
int i3;
int v2 = (i2 + ((i3) >> 7));
********** Finished printing generated C++ code
```
SIFOpt - Multiplication

```
var (17/6/9)    i1;
var             c1 = i1 * i1;

var (1/10/21)   i2;
var             c2 = i2 * i2;

var (24/4/4)    i3;
var             c3 = i3 * i2;
```

```
$ sifopt -C infile
********** Printing generated C++ code
int i1;
int c1 = (i1 * i1);
int i2;
int c2 = (((i2) >> 16) * ((i2) >> 15));
int i3;
int c3 = (i3 * ((i2) >> 8));
********** Finished printing generated C++ code
```
SIFOpt - Multiplication

********** Printing Tree of Optimizations
OptVar - i1: (17/6/9), [-63.998:63.998]
OptVar - c1: (2/12/18), [-4095.75:4095.75] Initialized From:
  OptMult
    OptBinOp: '*' m_SharedVars = true result is: (2/12/18), [-4095.75:4095.75]
      OptVar - i1: (17/6/9), [-63.998:63.998]
      OptVar - i1: (17/6/9), [-63.998:63.998]
OptVar - i2: (1/10/21), [-1024:1024]
OptVar - c2: (1/20/11), [-1.04856e+06:1.04856e+06] \47.9995/ Initialized From:
  OptMult
    OptBinOp: '*' m_SharedVars = true result is: (1/20/11), [-1.04856e+06:1.04856e+06] \47.9995/
      OptResize - m_delta = 0, m_shift = 16, m_tbits = 16 (17/10/5), [-1024:1023.97] \0.0312495/
        OptVar - i2: (1/10/21), [-1024:1024]
      OptResize - m_delta = 0, m_shift = 15, m_tbits = 15 (16/10/6), [-1024:1023.98] \0.0156245/
        OptVar - i2: (1/10/21), [-1024:1024]
OptVar - i3: (24/4/4), [-15.9375:15.9375]
OptVar - c3: (1/14/17), [-16320:16320] \0.0019379/ Leaf variables: i3, i2
Initialized From:
  OptMult
    OptBinOp: '*' result is: (1/14/17), [-16320:16320] \0.0019379/
      OptVar - i3: (24/4/4), [-15.9375:15.9375]
      OptResize - m_delta = 0, m_shift = 8, m_tbits = 8 (9/10/13), [-1024:1024] \0.000121593/
        OptVar - i2: (1/10/21), [-1024:1024]
********** Finished printing tree of optimizations
SIFOpt - Constants

- “Known” declarations only associate a real value with a name – the individual occurrences of the known are optimized individually!

```plaintext
var (24/4/4) v1;
known k1 = 7.2;
var sum = v1 + k1;
var prod = v1 * k1;
```

$ sifopt -C infile

********** Printing generated C++ code
int v1;
// known k1 = 7.2
int sum = (v1 + 115);
int prod = (v1 * 7549747);
********** Finished printing generated C++ code
SIFOpt – Constants and Addition

- SIFOpt tries to create constants which match the scaling factor of the non-variable argument. This can result in constants dropping to 0!

```javascript
var (25/3/2) x;

var s1 = x + 2.5;
var s4 = x + 300;
var s2 = x + 2.2;
var s3 = x + 0.01;
```
SIFOpt – Constants and Addition

$ sifopt -C infile
6: WARNING – value 0.01 loses all mantissa bits when aligned with (25/3/2)
6: WARNING – Initializer for variable 's3' has a (0/0/0) SIF!
********** Printing generated C++ code
int x;
int s1 = (x + 40);
unsigned int s4 = (x + 4800);
int s2 = (x + 35);
unsigned int s3 = (x + 0);
********** Finished printing generated C++ code
SIFOpt – Constants and Addition

********** Printing Tree of Optimizations
OptVar - x: (25/3/2), [-7.75:7.75]
OptVar - s1: (24/4/2), [-5.25:10.25] Initialized From:
    OptAddSub - k = 0, kmin = -23, kmax = 2
    OptBinOp: '+' result is: (24/4/2), [-5.25:10.25]
    OptVar - x: (25/3/2), [-7.75:7.75]
    OptLit - OptDouble - 2.5 as 40=0x28 in 32 bits with (+26/2/1) = 2.5
OptVar - s4: (+19/9/2), [292.25:307.75] Initialized From:
    OptAddSub - k = 0, kmin = -19, kmax = 2
    OptBinOp: '+' result is: (+19/9/2), [292.25:307.75]
    OptVar - x: (25/3/2), [-7.75:7.75]
    OptLit - OptDouble - 300 as 4800=0x12c0 in 32 bits with (+19/7/0)^+2 = 300
OptVar - s2: (24/4/4), [-5.5625:9.9375] \0.0125/ Initialized From:
    OptAddSub - k = 0, kmin = -23, kmax = 0
    OptBinOp: '+' result is: (24/4/4), [-5.5625:9.9375] \0.0125/ 
    OptVar - x: (25/3/2), [-7.75:7.75]
    OptLit - OptDouble - 2.2 as 35=0x23 in 32 bits with (+26/2/4) = 2.1875 \0.0125/ 
OptVar - s3: (0/0/0), [0:0] Initialized From:
    OptAddSub - k = 0, kmin = 0, kmax = 0
    OptBinOp: '+' result is: (0/0/0), [0:0]
    OptVar - x: (25/3/2), [-7.75:7.75]
    OptLit - OptDouble - 0.01 as 0=0x0 in 32 bits with (32/0/0) = 0 \0.01/ 
********** Finished printing tree of optimizations
SIFOpt – Absolute Error Estimates

- SIFOpt computes maximum absolute error values possible due to truncation of bits... these estimates become extremely pessimistic for anything but simple, short calculations.
SIFOpt – Mixed-point Computation

• One output type is C++ code which uses a “Mixed-point” C++ class to implement the calculations. This class can be used to perform test runs of the calculation and extract actual truncation error amounts from run-time computations.
SIFOpt – Advanced SIFOpt

• Built-in functions
  - NC( expr )
  - SIF manipulation functions
  • asb( val, n ) - this is a right shift
  • rsb( val, n ) - this is a left shift
  • ...many others...
SIFOpt - NC() function

• Addition can mistakenly believe that a carry-out can occur from an addition when it cannot because:
  – Run-time data dependence
  – Common terms in the expression: \((x - x)\)

\[
\begin{array}{c}
\text{SSSSIIIIIIFFFFFEF} \\
+ \quad \text{SSSSSSSSSSFFFFFEF} \\
\hline
\text{SSSIIIIIIFFFFFEF}
\end{array}
\]