

Compile-Time Generation of Custom-Precision Floating-Point IP using HLS Tools

David B. Thomas

Imperial College London

dt10@imperial.ac.uk

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*

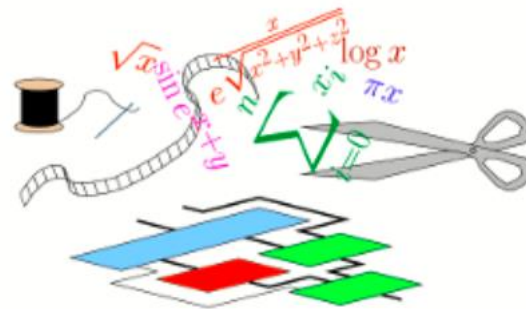
Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*
- Belle Epoque ~2005 : *we can choose **any** size floating-point?*

FloPoCo



Circuits computing just right

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*
- Belle Epoque ~2005 : *we can choose **any** size floating-point?*
- Dystopia ~2010 : *sigh, why only three floating-point types?*

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*
- Belle Epoque ~2005 : *we can choose **any** size floating-point?*
- Dystopia ~2010 : *sigh, why only three floating-point types?*



High-Level Synthesis (HLS) = hardware description using C/C++

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*
- Belle Epoque ~2005 : *we can choose **any** size floating-point?*
- Dystopia ~2010 : *sigh, why only three floating-point types?*
- Utopia ~now : *we can choose **any** size floating-point?*

Numerical IP in FPGAs : a history

- Pre-history ~1995 : *how do I build a multiplier out of LUTs?*
- Antiquity ~2000 : *can we fit a floating-point unit here?*
- Belle Epoque ~2005 : *we can choose **any** size floating-point?*
- Dystopia ~2010 : *sigh, why only three floating-point types?*
- Utopia ~now : *we can choose **any** size floating-point?*
 - ... and mix representations in one operator?*
 - ... and have it auto pipeline?*
 - ... and get identical cross-platform results?*
 - ... and add custom number formats?*
 - ... and choose rounding modes?*

Big Idea: generate floating-point IP at C++ compile-time

```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

Big Idea: generate floating-point IP at C++ compile-time

```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

```
fp<8,25> f( fp<7,23> x, fp<5,14> y )
{
    auto xy = mul<7,22>( x, y );
    return add<8,25>( xy, x );
}
```

Big Idea: generate floating-point IP at C++ compile-time

```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

```
fp<8,25> f( fp<7,23> x, fp<5,14> y )
{
    auto xy = mul<7,22>( x, y );
    return add<8,25>( xy, x );
}
```

 Three types: double, float, half

 Infinite types: any exponent + fraction width

Big Idea: generate floating-point IP at C++ compile-time

```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

```
fp<8,25> f( fp<7,23> x, fp<5,14> y )
{
    auto xy = mul<7,22>( x, y );
    return add<8,25>( xy, x );
}
```

✗ Three types: double, float, half

✗ Homogeneous operands

✓ Infinite types: any exponent + fraction width

✓ Heterogeneous operands

Big Idea: generate floating-point IP at C++ compile-time

```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

```
fp<8,25> f( fp<7,23> x, fp<5,14> y )
{
    auto xy = mul<7,22>( x, y );
    return add<8,25>( xy, x );
}
```

- ✗ Three types: double, float, half
- ✗ Homogeneous operands
- ✗ Platform dependent results

- ✓ Infinite types: any exponent + fraction width
- ✓ Heterogeneous operands
- ✓ Platform independent results

Big Idea: generate floating-point IP at C++ compile-time

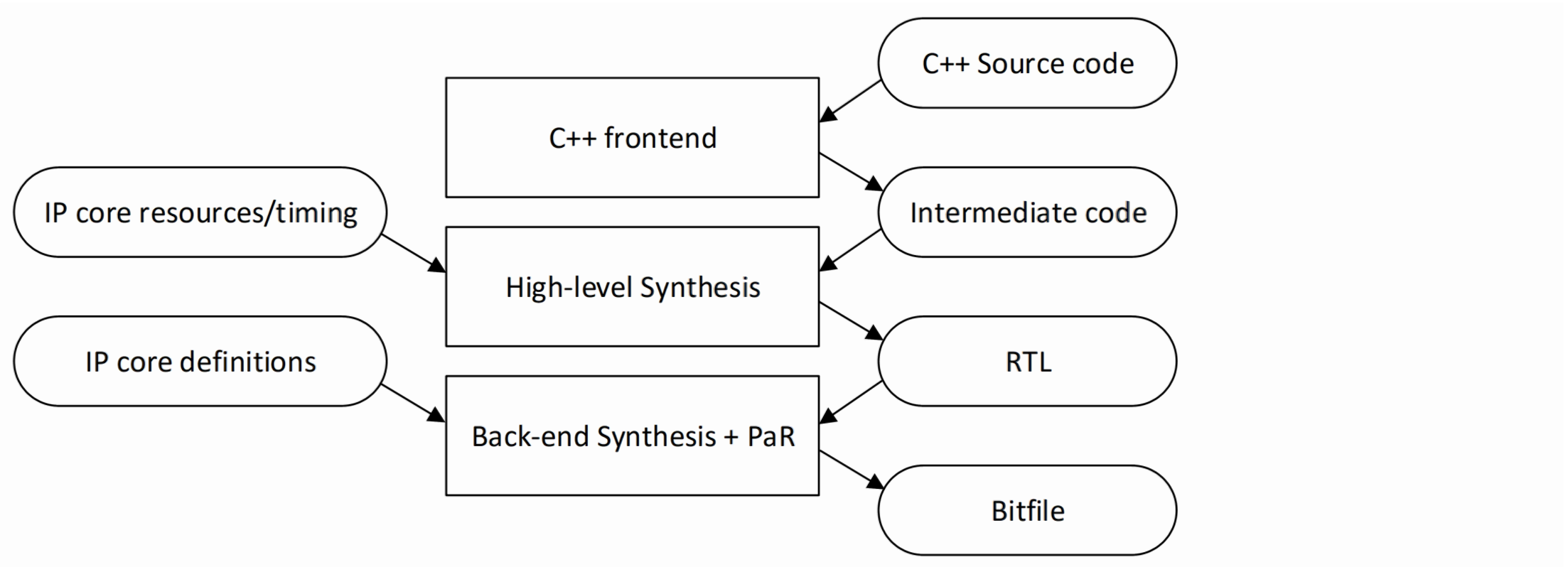
```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

```
fp<8,25> f( fp<7,23> x, fp<5,14> y )
{
    auto xy = mul<7,22>( x, y );
    return add<8,25>( xy, x );
}
```

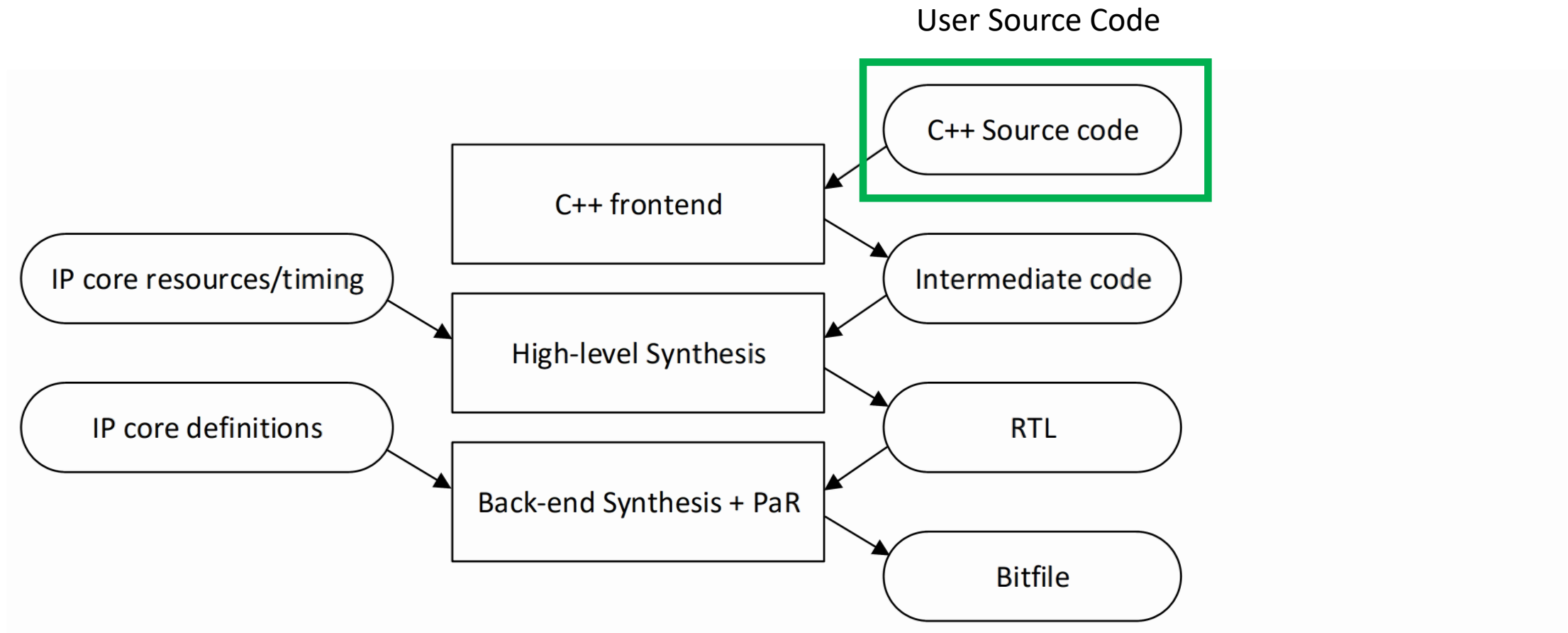
- ✗ Three types: double, float, half
- ✗ Homogeneous operands
- ✗ Platform dependent results
- ✓ Efficient in hardware

- ✓ Infinite types: any exponent + fraction width
- ✓ Heterogeneous operands
- ✓ Platform independent results
- ✓ Efficient in hardware

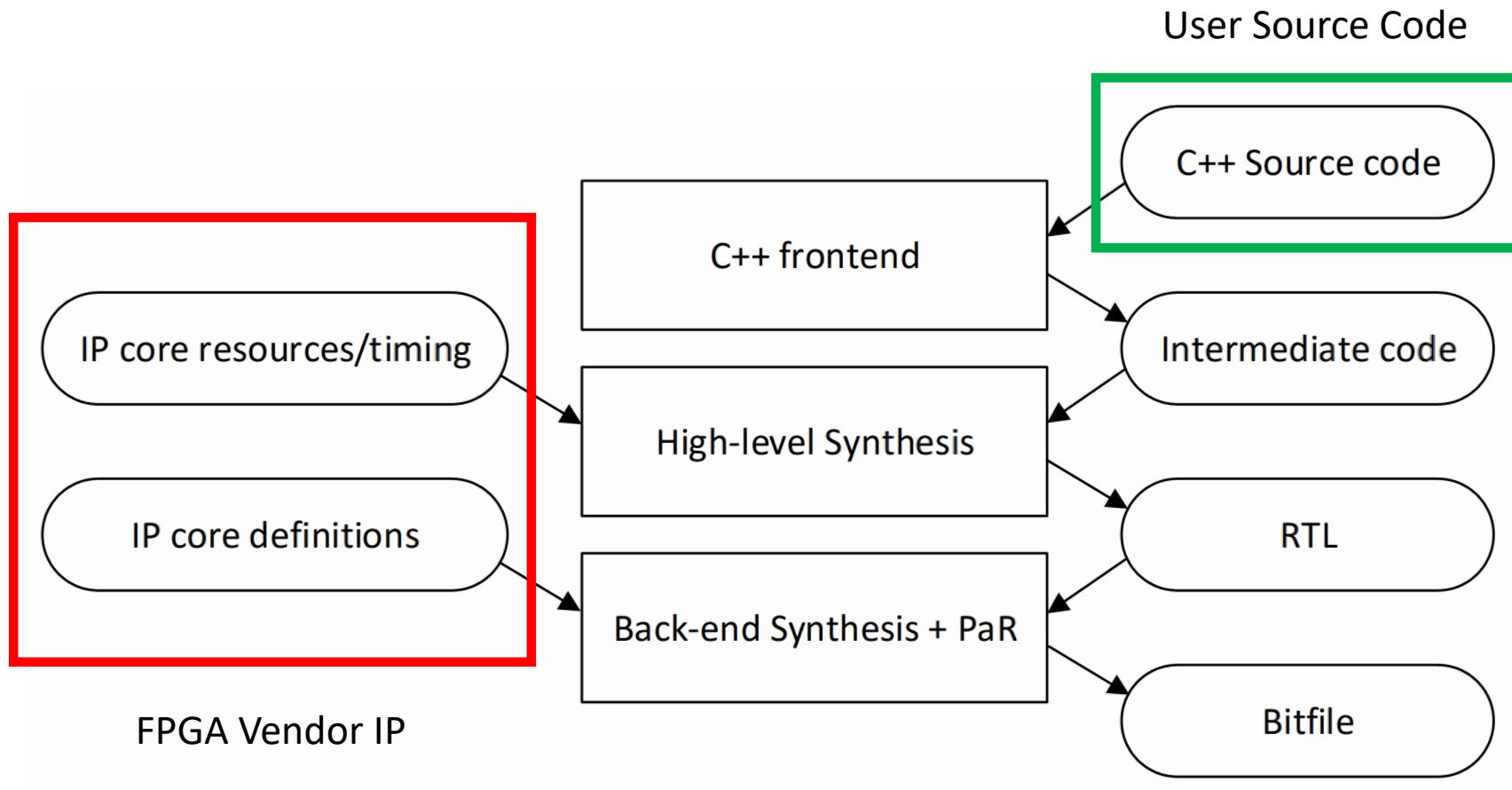
Background: floating-point in HLS



Background: floating-point in HLS

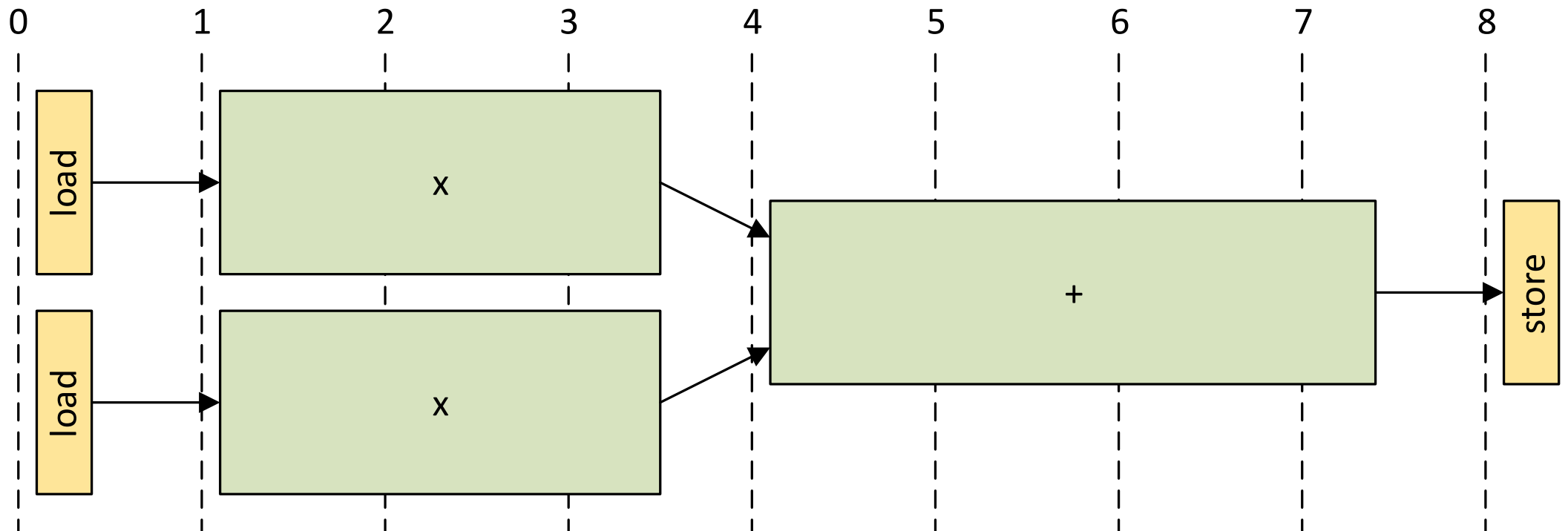


Background: floating-point in HLS



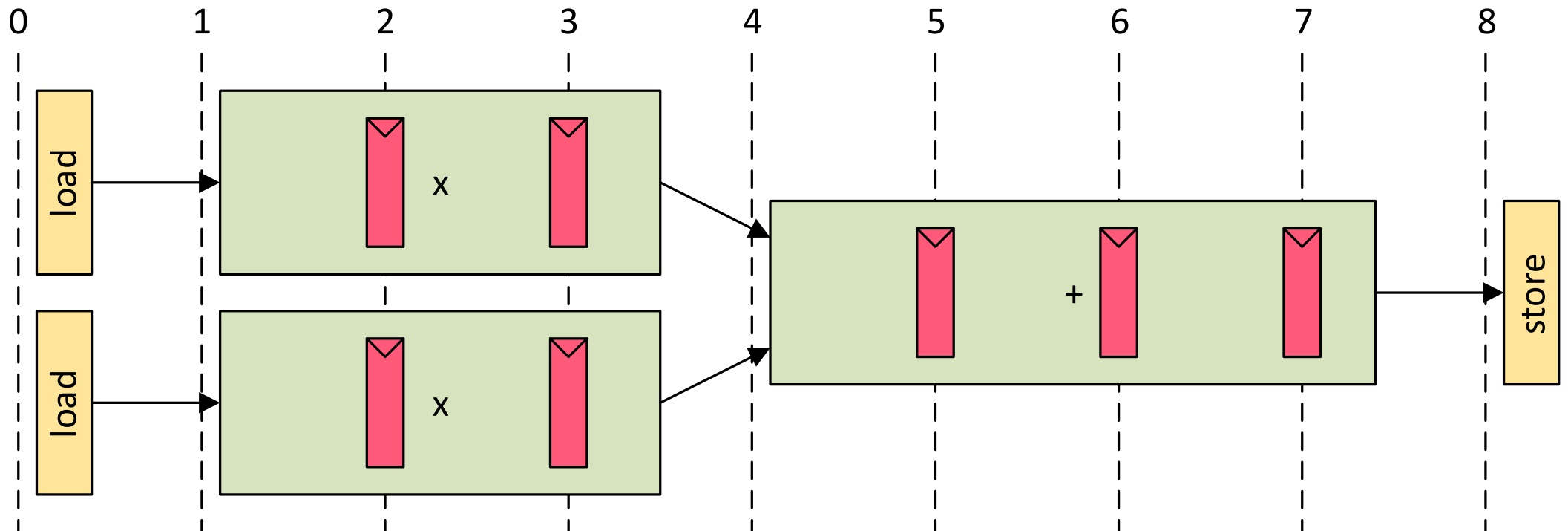
Motivation: scheduling of floating-point

Floating point is scheduled as black-box pipelines



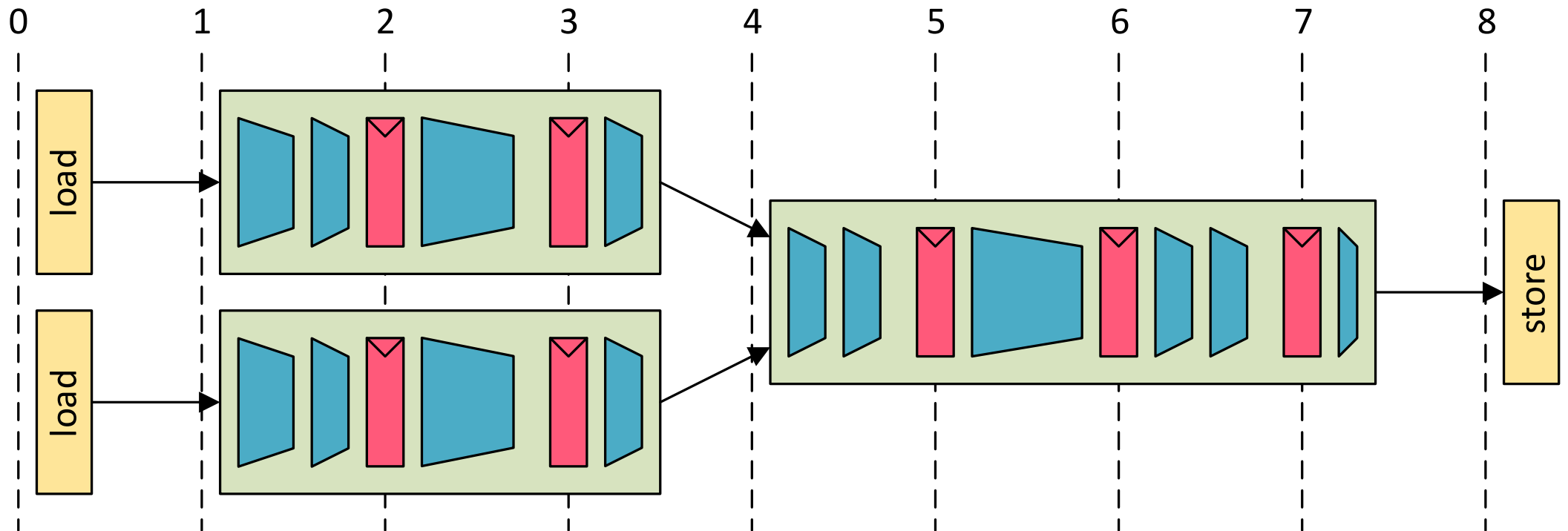
Motivation: scheduling of floating-point

Number of registers is known, but not much else



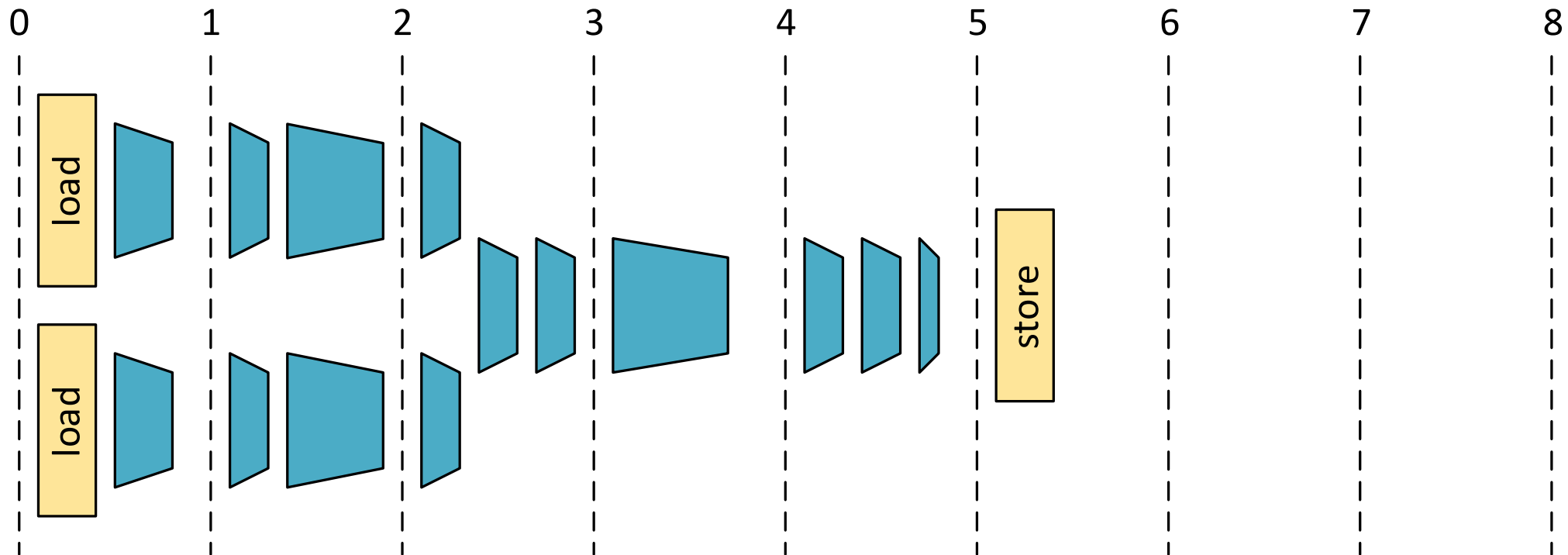
Motivation: scheduling of floating-point

Exposing internal structure makes timing visible to HLS scheduler



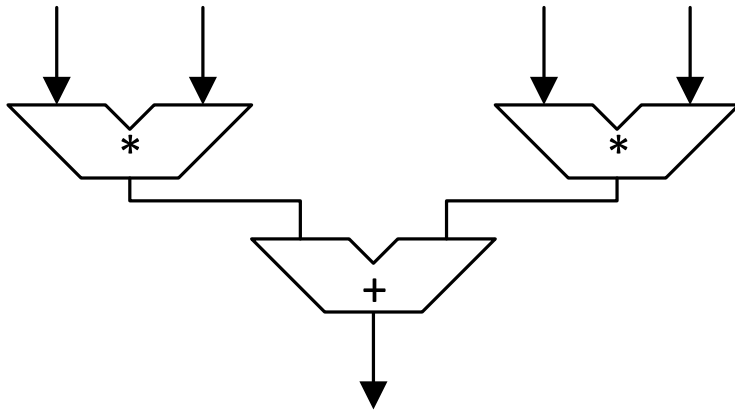
Motivation: scheduling of floating-point

HLS scheduler can pipeline based on current clock constraints



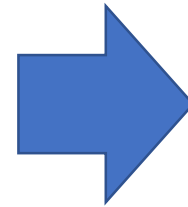
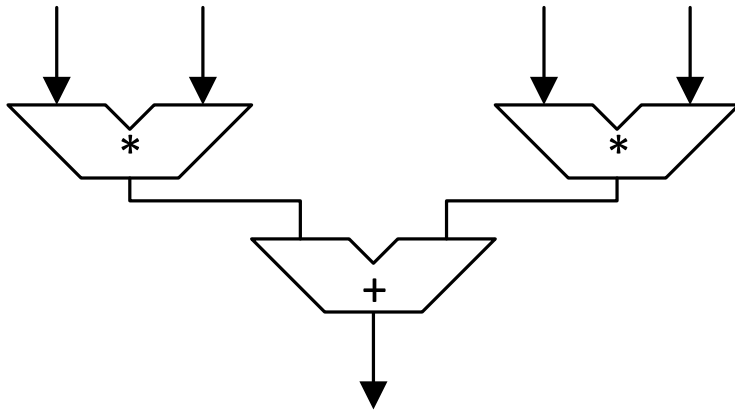
Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
    float xy0 = x[0] * y[0];  
    float xy1 = x[1] * y[1];  
    return xy0 + xy1;  
}
```

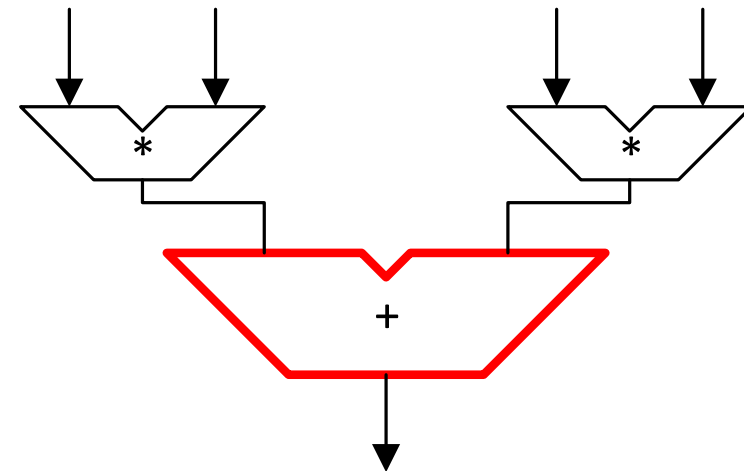


Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
    float xy0 = x[0] * y[0];  
    float xy1 = x[1] * y[1];  
    return xy0 + xy1;  
}
```

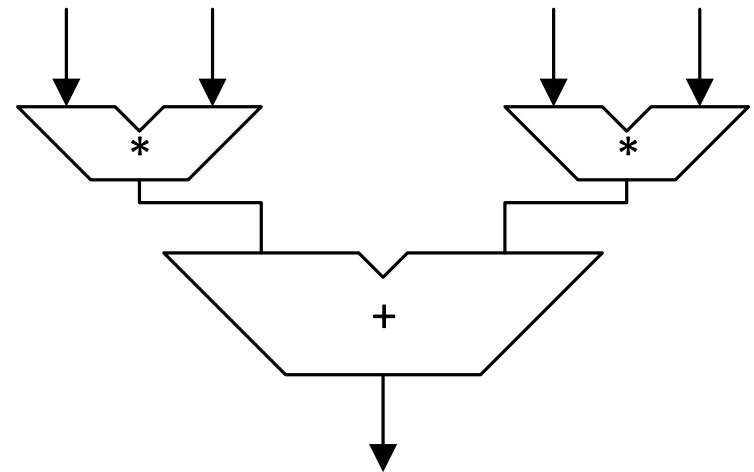


```
float f(float x[2], float y[2])  
{  
    float xy0 = x[0] * y[0];  
    float xy1 = x[1] * y[1];  
    return double(xy0) + double(xy1);  
}
```



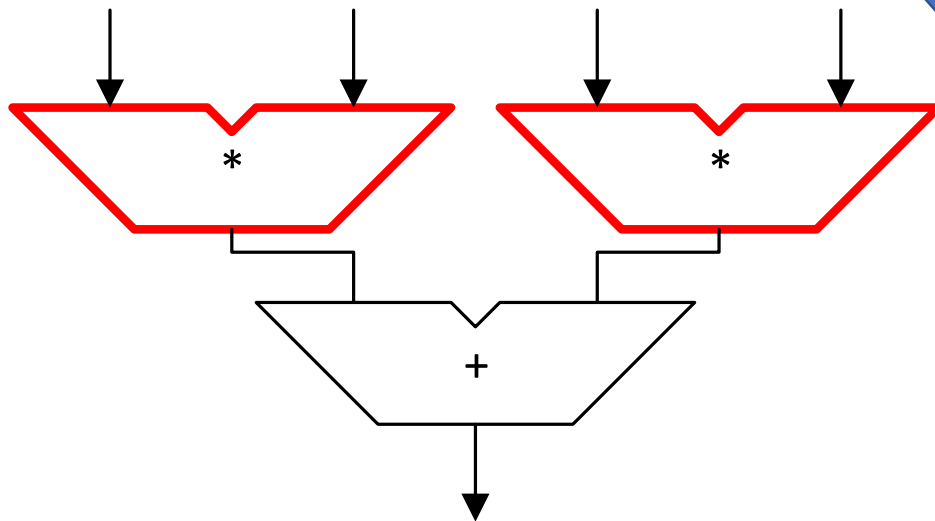
Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
    float xy0 = x[0] * y[0];  
    float xy1 = x[1] * y[1];  
    return double(xy0) + double(xy1);  
}
```

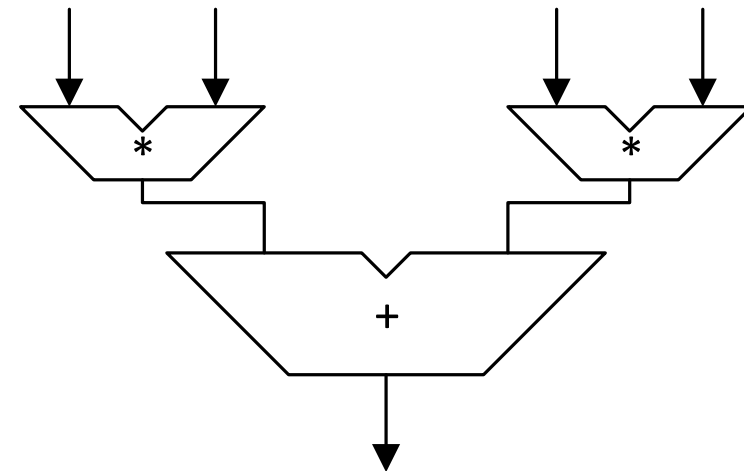


Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
  double xy0 = double(x[0]) * double(y[0]);  
  double xy1 = double(x[1]) * double(y[1]);  
  return double(xy0) + double(xy1);  
}
```

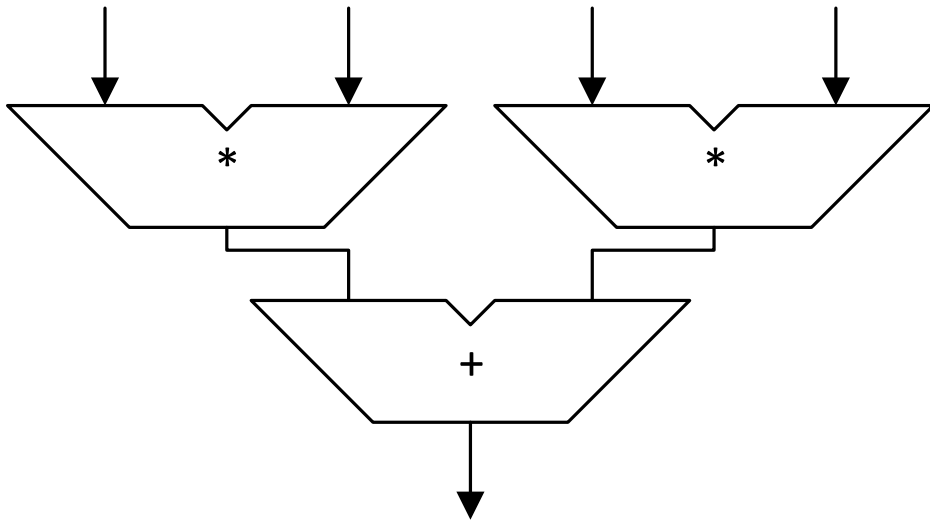


```
float f(float x[2], float y[2])  
{  
  float xy0 = x[0] * y[0];  
  float xy1 = x[1] * y[1];  
  return double(xy0) + double(xy1);  
}
```



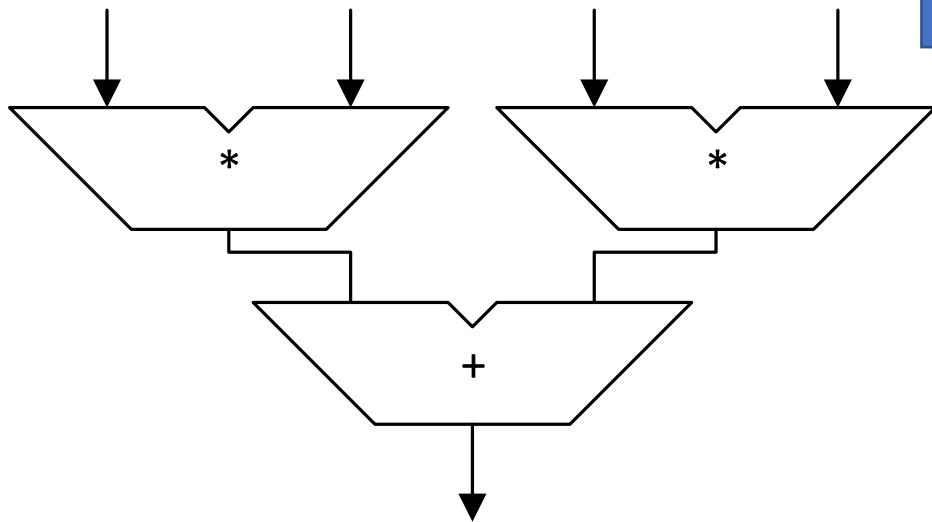
Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
  double xy0 = double(x[0]) * double(y[0]);  
  double xy1 = double(x[1]) * double(y[1]);  
  return double(xy0) + double(xy1);  
}
```

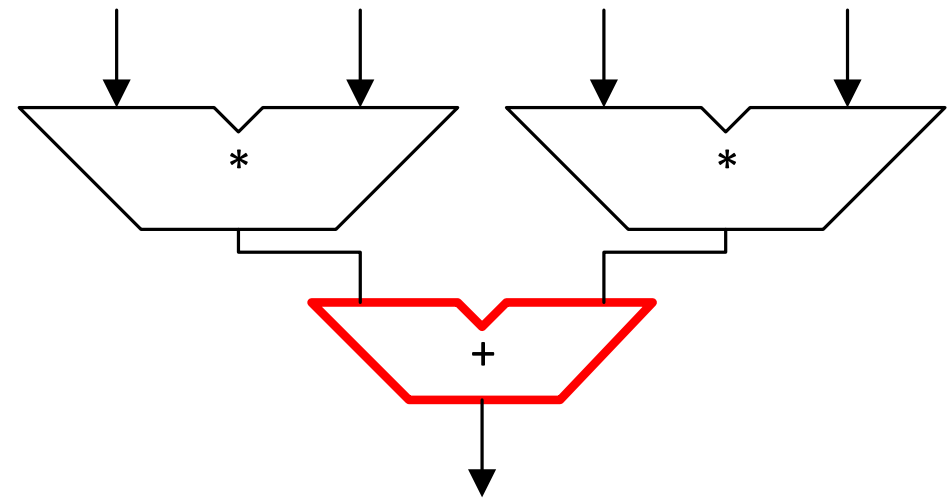


Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
    double xy0 = double(x[0]) * double(y[0]);  
    double xy1 = double(x[1]) * double(y[1]);  
    return double(xy0) + double(xy1);  
}
```

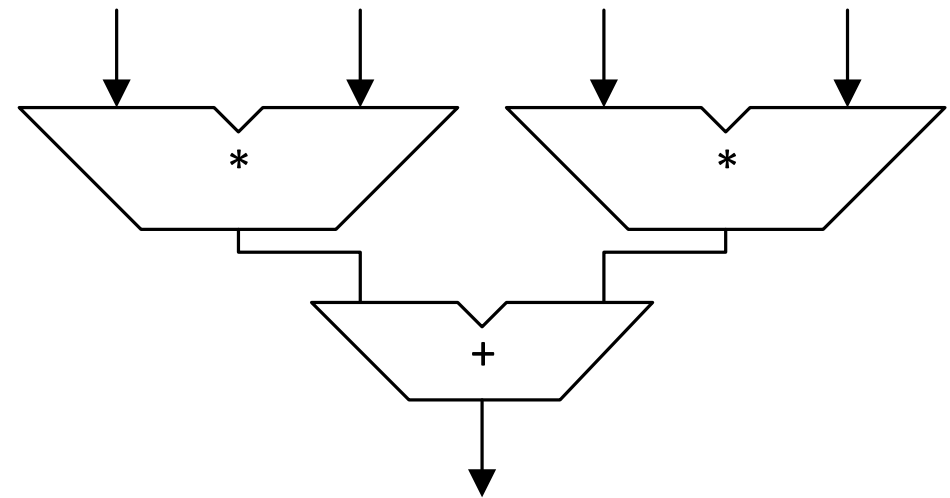


```
float f(float x[2], float y[2])  
{  
    double xy0 = double(x[0]) * double(y[0]);  
    double xy1 = double(x[1]) * double(y[1]);  
    return float41(xy0) + float41(xy1);  
}
```



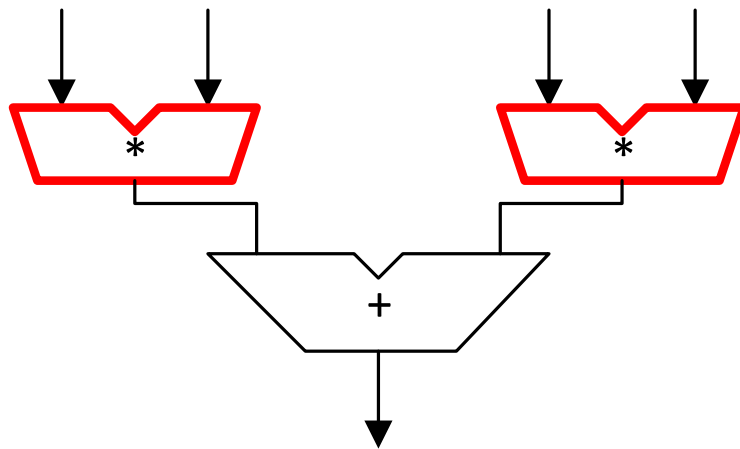
Motivation: custom precision operations

```
float f(float x[2], float y[2])  
{  
    double xy0 = double(x[0]) * double(y[0]);  
    double xy1 = double(x[1]) * double(y[1]);  
    return float41(xy0) + float41(xy1);  
}
```

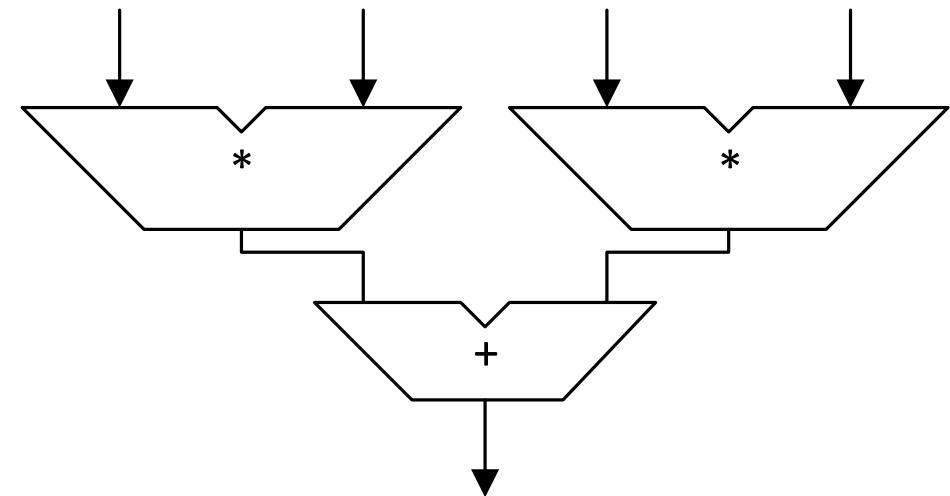


Motivation: custom precision operations

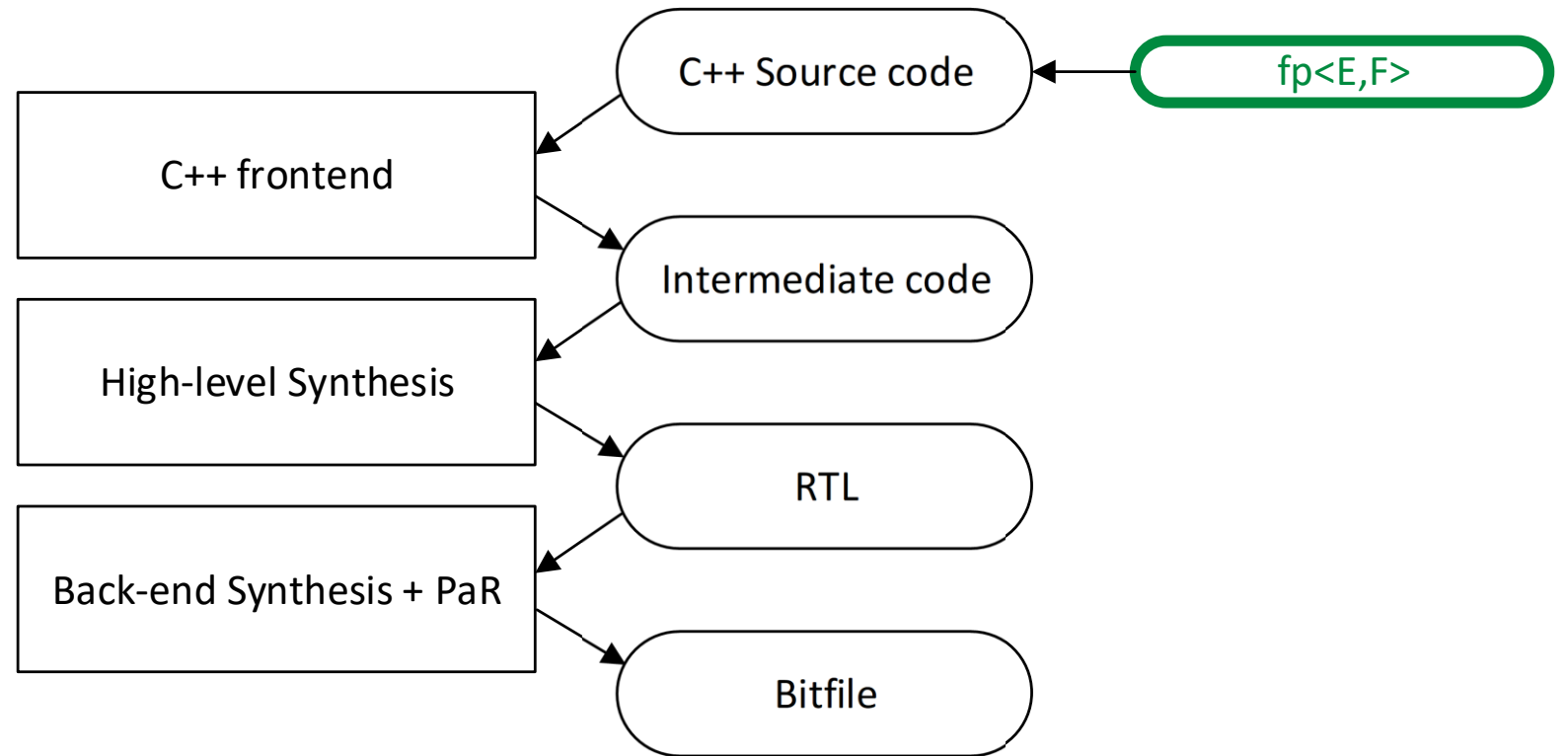
```
float f(float x[2], float y[2])  
{  
  float41 xy0 = x[0] ??*?? [0];  
  float41 xy1 = x[1] ??*?? y[1];  
  return xy0 + xy1;  
}
```



```
float f(float x[2], float y[2])  
{  
  double xy0 = double(x[0]) * double(y[0]);  
  double xy1 = double(x[1]) * double(y[1]);  
  return float41(xy0) + float41(xy1);  
}
```



Solution: templatised soft floating point



API: floating-point data-type

```
template<int E, int F>
struct fp_flopoco
{
    fw_uint< 2 + 1 + E + F > bits;
};
```

API: floating-point data-type

```
template<int E, int F>
struct fp_flopoco
{
    fw_uint< 2 + 1 + E + F > bits;
};
```

FloPoCo exception code

0 = Zero

1 = Normal

2 = Infinity

3 = NaN

API: floating-point data-type

```
template<int E, int F>
struct fp_flopoco
{
    fw_uint< 2 + 1 + E + F > bits;
};
```

Standard floating point

1 bit sign

E bit exponent

F bit fraction

API: floating-point data-type

```
template<int E, int F>
struct fp_flopoco
{
    fw_uint< 2 + 1 + E + F > bits;
};
```

```
using float32_t    = fp_flopoco<8, 23>;
using bfloat16_t   = fp_flopoco<8, 7>;
using dfloat16_t   = fp_flopoco<6, 9>;
```

API: operators

```
template< int eR,int fR, int eA,int fA, int eB,int fB >  
fp<eR,fR> mul( fp<eA,fA> a, fp<eB,fB> b);
```

API: operators

```
template< int eR, int fR, int eA, int fA, int eB, int fB >  
fp<eR, fR> mul( fp<eA, fA> a, fp<eB, fB> b );
```

API: operators

```
template< int eR, int fR, int eA, int fA, int eB, int fB >  
fp<eR, fR> mul( fp<eA, fA> a, fp<eB, fB> b );
```

API: operators

```
template< int eR, int fR, int eA, int fA, int eB, int fB >  
fp<eR, fR> mul( fp<eA, fA> a, fp<eB, fB> b);
```

API: operators

```
template< int eR,int fR, int eA,int fA, int eB,int fB >  
fp<eR,fR> mul( fp<eA,fA> a, fp<eB,fB> b);
```

Internally the function is pure platform independent C++

Compile with:

- *g++ : x86*
- *Vivado HLS : Xilinx*
- *Intel HLS : Intel*
- *Legup: Verilog*

API: operators

```
template< int eR,int fR, int eA,int fA, int eB,int fB >  
fp<eR,fR> mul( fp<eA,fA> a, fp<eB,fB> b);
```

```
fp<6,31> x;
```

```
fp<7,20> y;
```


API: operators

```
template< int eR,int fR, int eA,int fA, int eB,int fB >  
fp<eR,fR> mul( fp<eA,fA> a, fp<eB,fB> b);
```

```
fp<6,31> x;
```

```
fp<7,20> y;
```

```
fp<8,17> xy = mul<8,17,6,31,7,20>( x, y );
```

API: operators

```
template< int eR,int fR, int eA,int fA, int eB,int fB >  
fp<eR,fR> mul( fp<eA,fA> a, fp<eB,fB> b);
```

```
fp<6,31> x;
```

```
fp<7,20> y;
```

```
fp<8,17> xy = mul<8,17,6,31,7,20>( x, y );
```

```
auto xy = mul<8,17>( x, y );
```

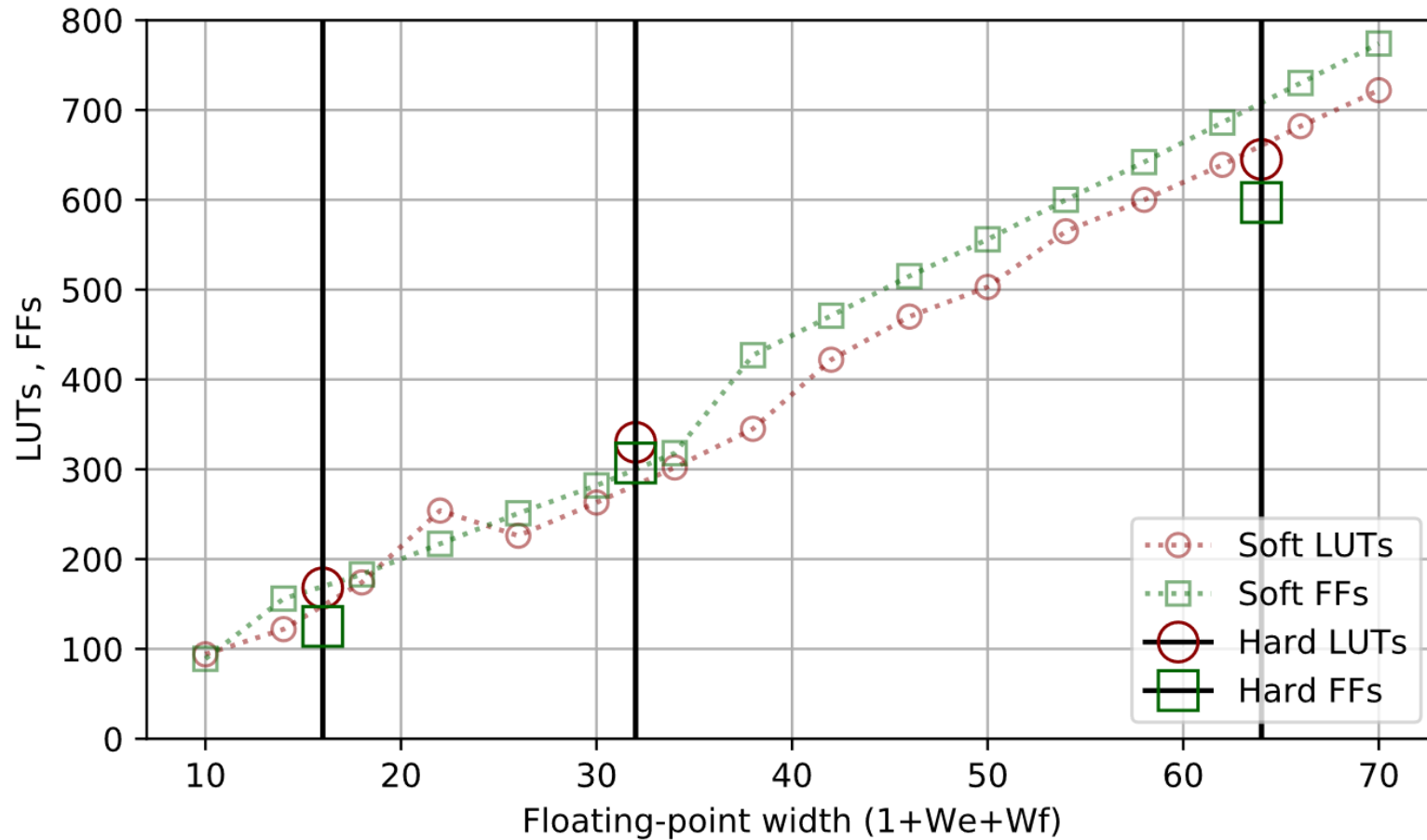
Advantages

- Platform independence
 - Plain C++ : *test applications without any HDL simulators or vendor libraries*
 - Bit-exact results: *same answer on every platform (and only sometimes 42)*

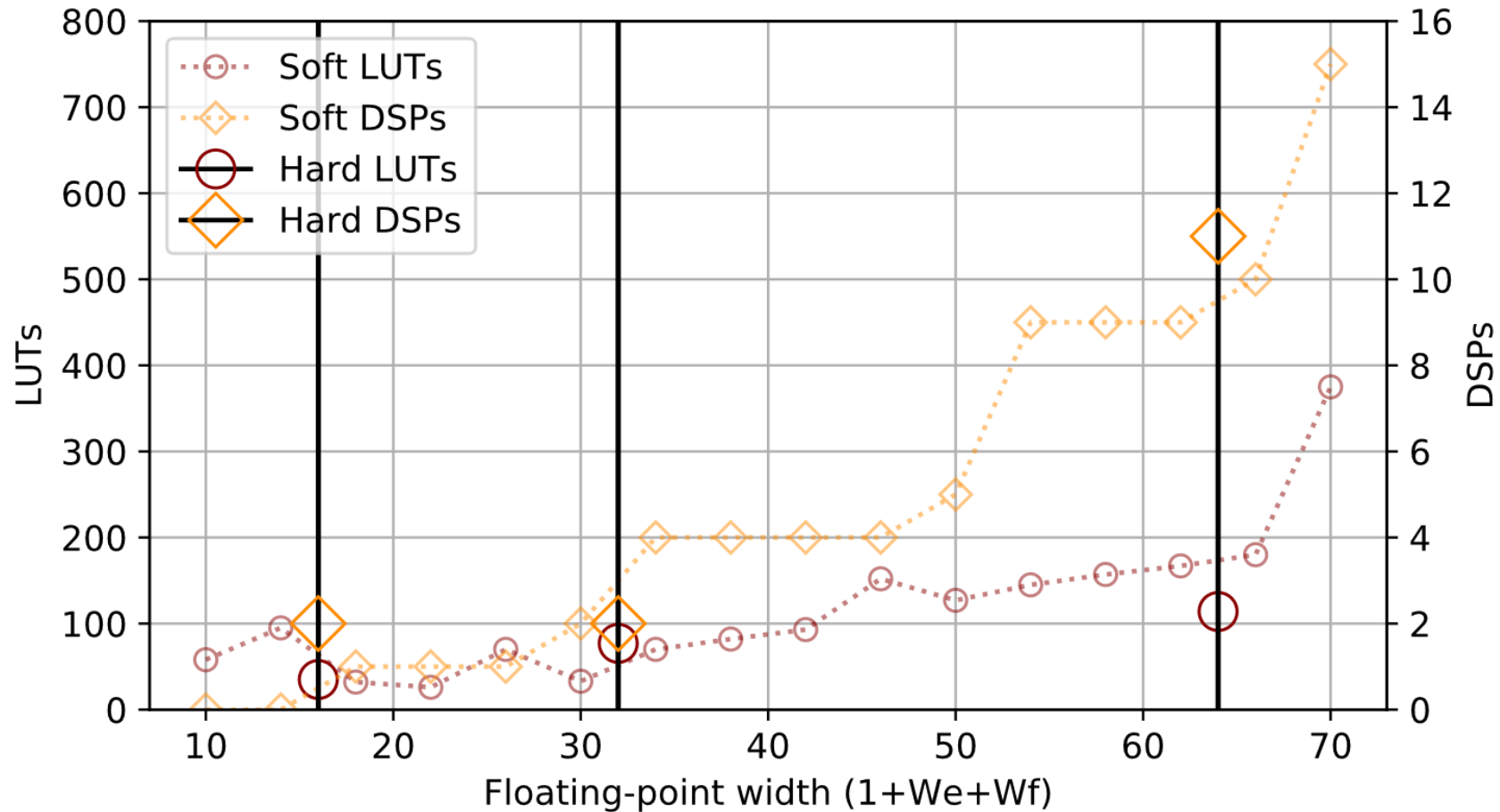
Advantages

- Platform independence
 - Plain C++ : *test applications without any HDL simulators or vendor libraries:*
 - Bit-exact results: *same answer on every platform (and only sometimes 42)*
- Performance

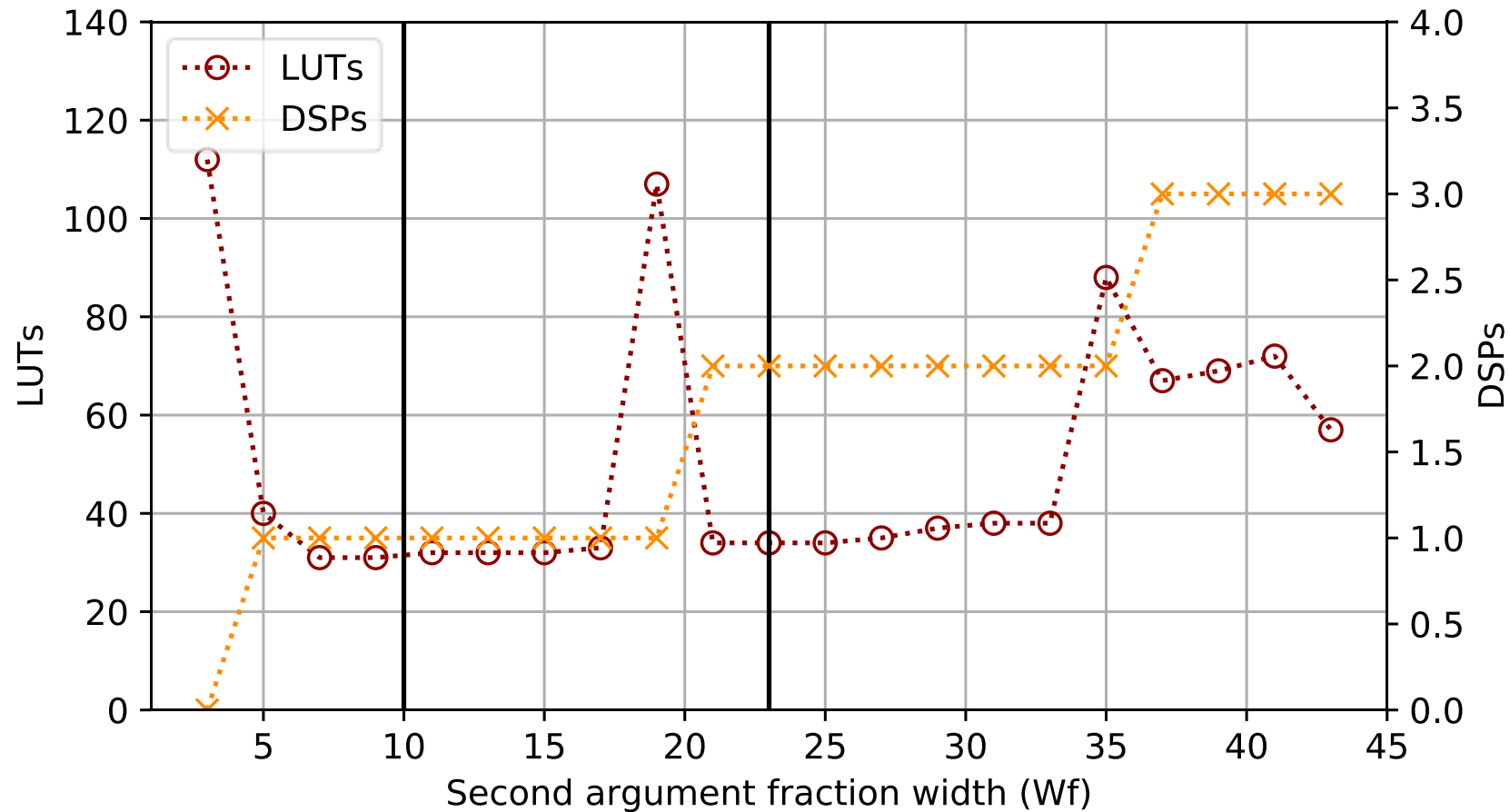
Results: homogeneous adder



Results: homogeneous multiplier



Results: *heterogeneous* multiplier



First argument is single precision; second argument is varying precision

Advantages

- Platform independence
 - Plain C++ : *test applications without any HDL simulators or vendor libraries:*
 - Bit-exact results: *same answer on every platform (and only sometimes 42)*
- Performance
 - Area and performance similar to proprietary vendor IP
 - Latency is often **lower** than standard vendor IP data-path

Advantages

- Platform independence
 - Plain C++ : *test applications without any HDL simulators or vendor libraries:*
 - Bit-exact results: *same answer on every platform (and only sometimes 42)*
- Performance
 - Area and performance similar to proprietary vendor IP
 - Latency is often **lower** than standard vendor IP data-path
- Freedom and control

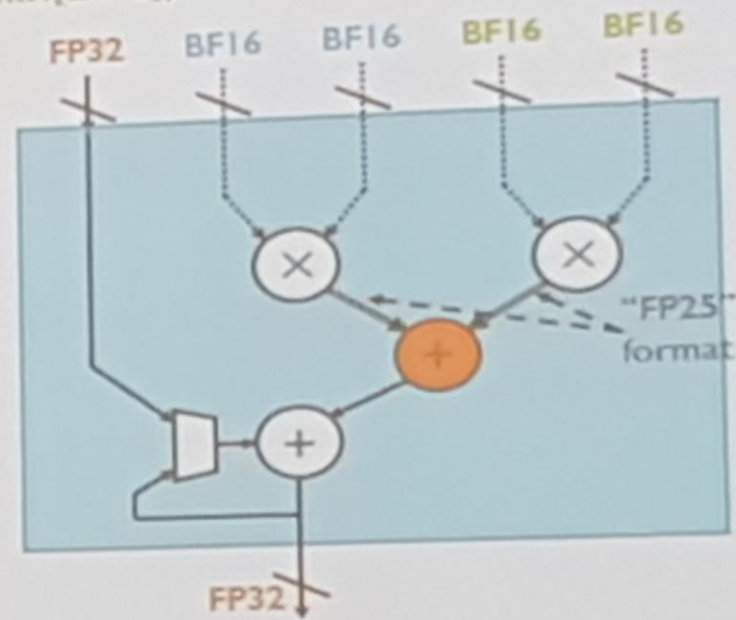
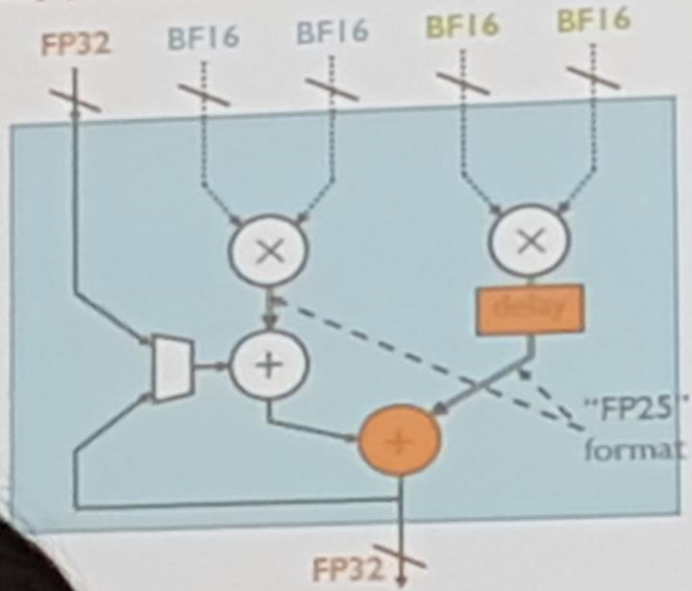
BFDOT alternatives considered

1. Accumulate products in sequence, similar to two BF16 to FP32 "FMLAL" operations

$$Z_{da.S}[k] = (Z_{da.S}[k] + Z_n.H[2k] \times Z_m.H[2k]) + Z_n.H[2k+1] \times Z_m.H[2k+1]$$

2. Sum of products then accumulate

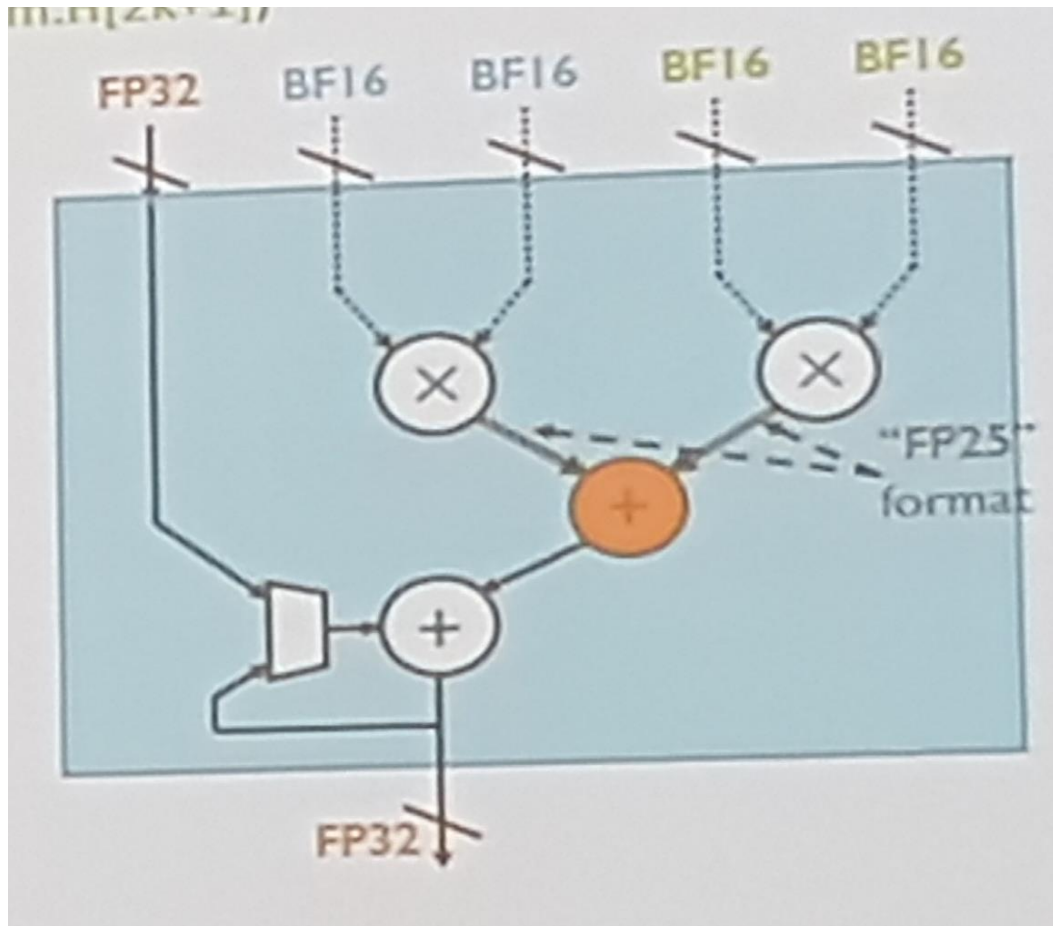
$$Z_{da.S}[k] = Z_{da.S}[k] + (Z_n.H[2k] \times Z_m.H[2k] + Z_n.H[2k+1] \times Z_m.H[2k+1])$$



arm

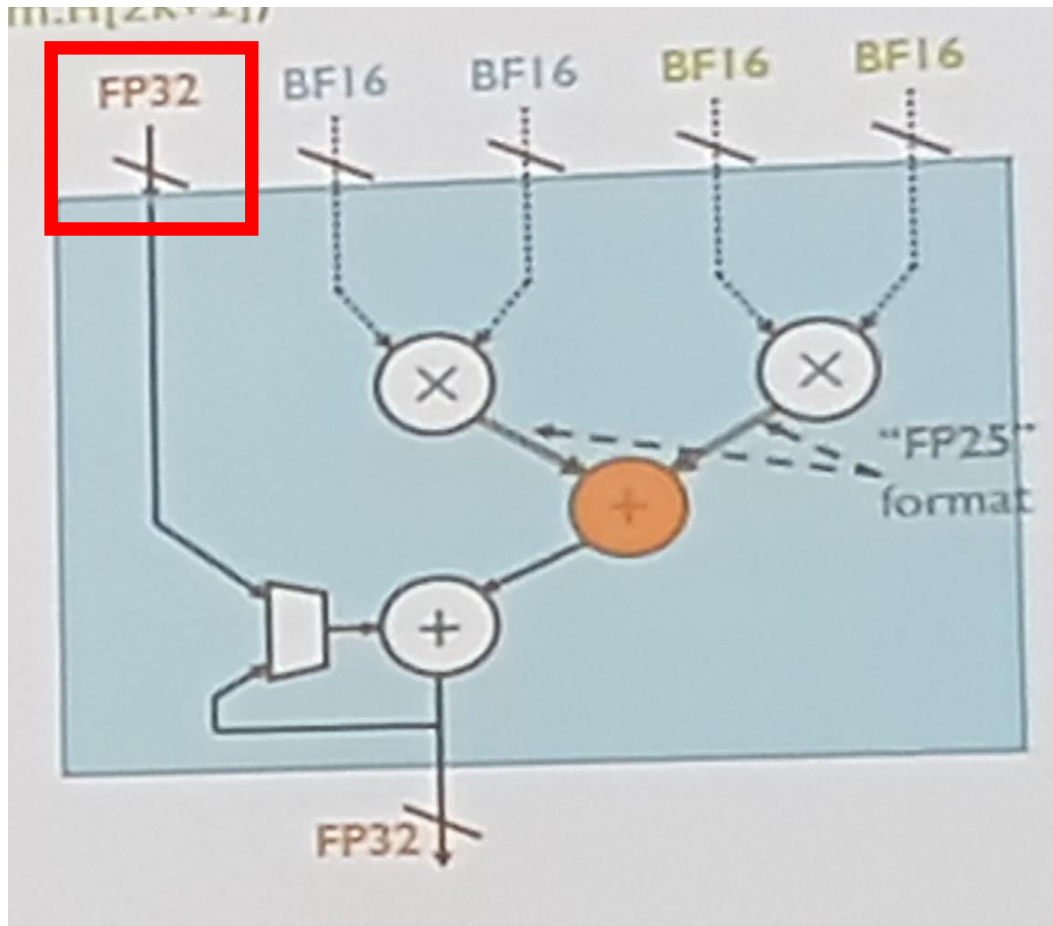
Copyright ARM, Thanks to Neil Burgess

Synthesisable BFDOT in 10 lines of code



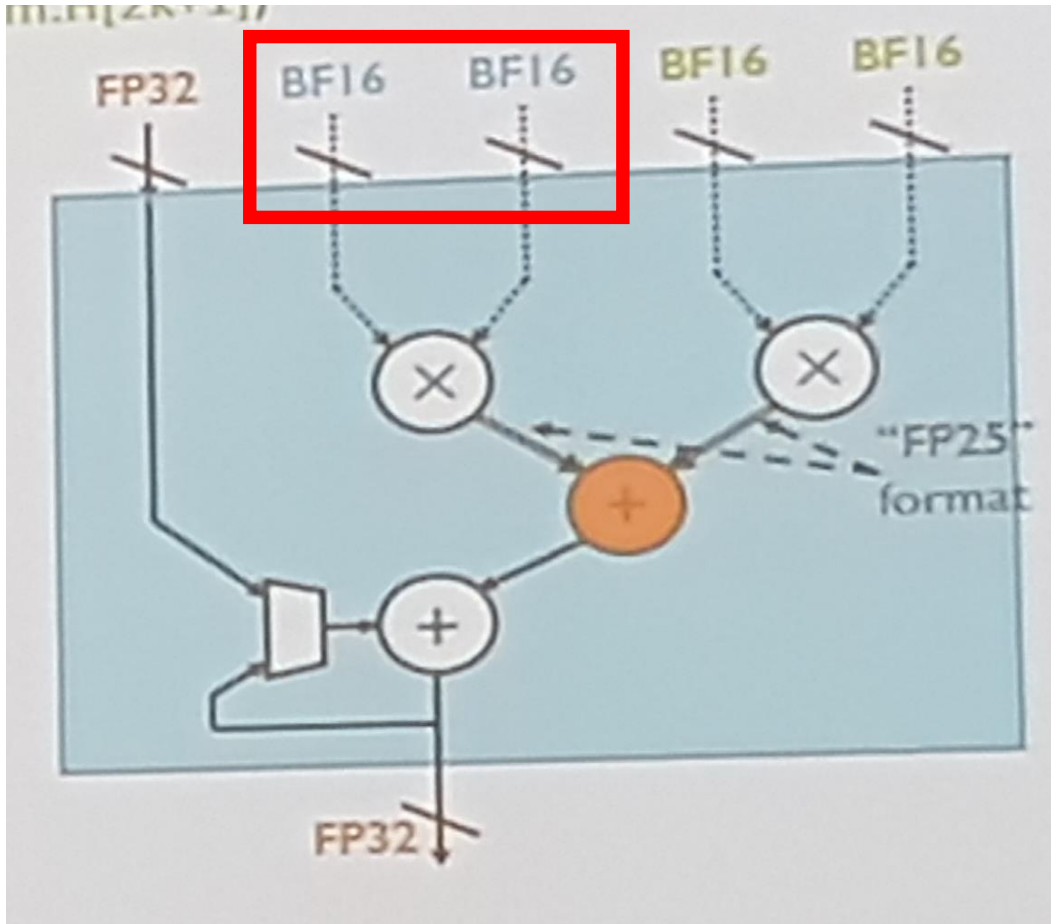
```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8,14>( x0, y0 );  
    auto xy1 = fp_mul<8,14>( x1, y1 );  
    auto sum = fp_add<8,23>( xy0, xy1 );  
    return fp_add<8,23>( acc, sum );  
}
```

Synthesisable BFDOT in 10 lines of code



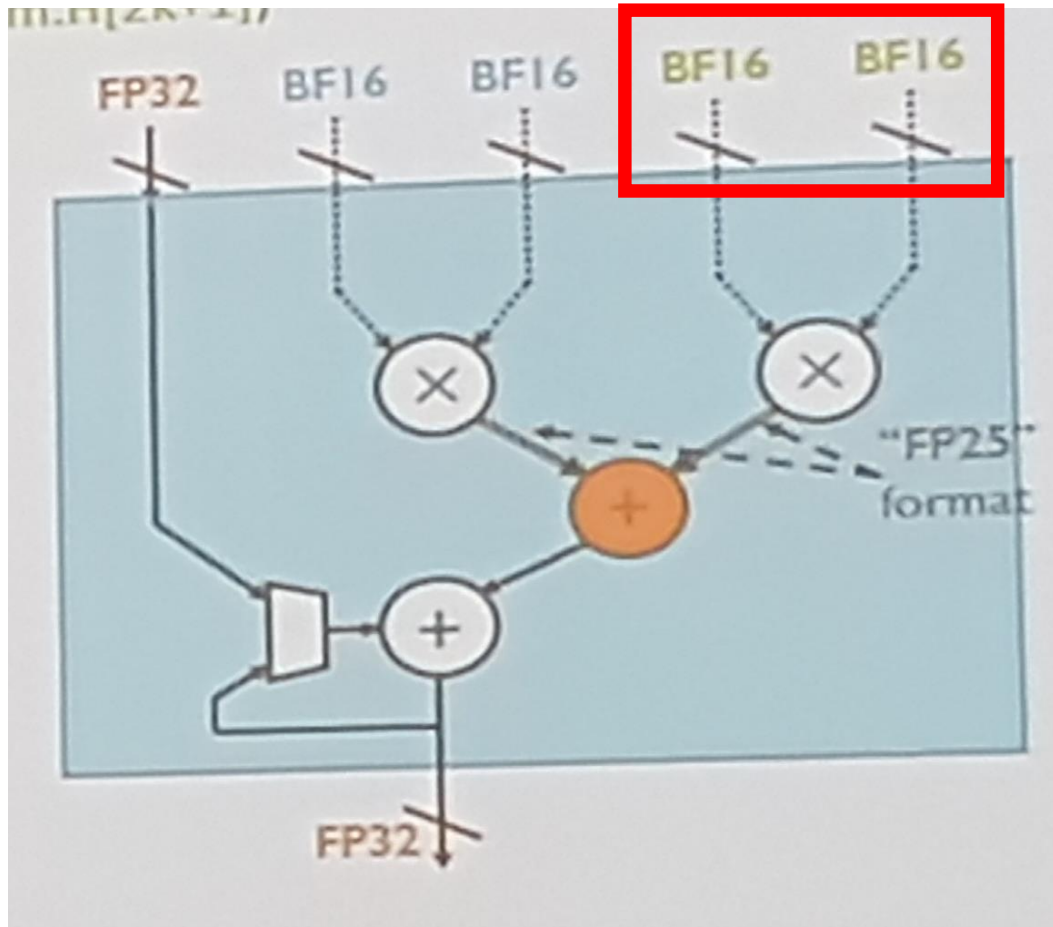
```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8,14>( x0, y0 );  
    auto xy1 = fp_mul<8,14>( x1, y1 );  
    auto sum = fp_add<8,23>( xy0, xy1 );  
    return fp_add<8,23>( acc, sum );  
}
```

Synthesisable BFDOT in 10 lines of code



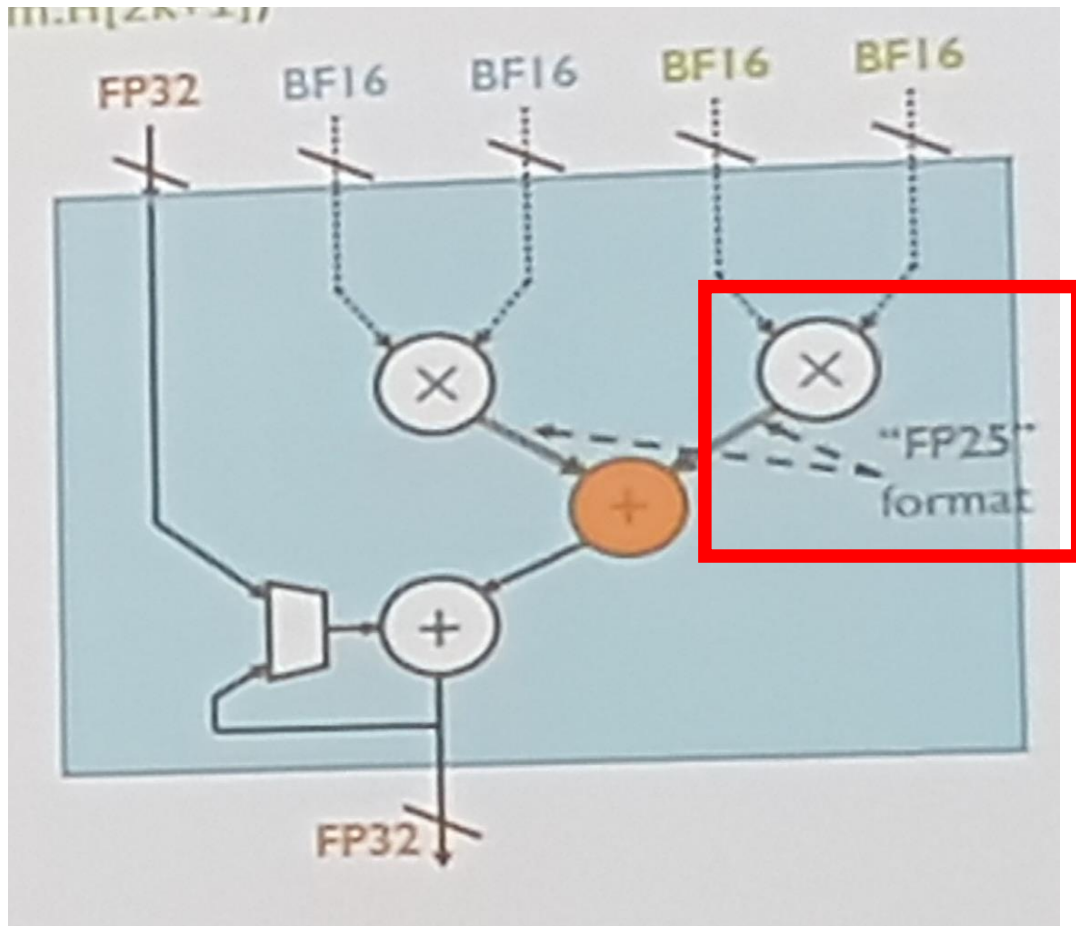
```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8,14>( x0, y0 );  
    auto xy1 = fp_mul<8,14>( x1, y1 );  
    auto sum = fp_add<8,23>( xy0, xy1 );  
    return fp_add<8,23>( acc, sum );  
}
```

Synthesisable BFDOT in 10 lines of code



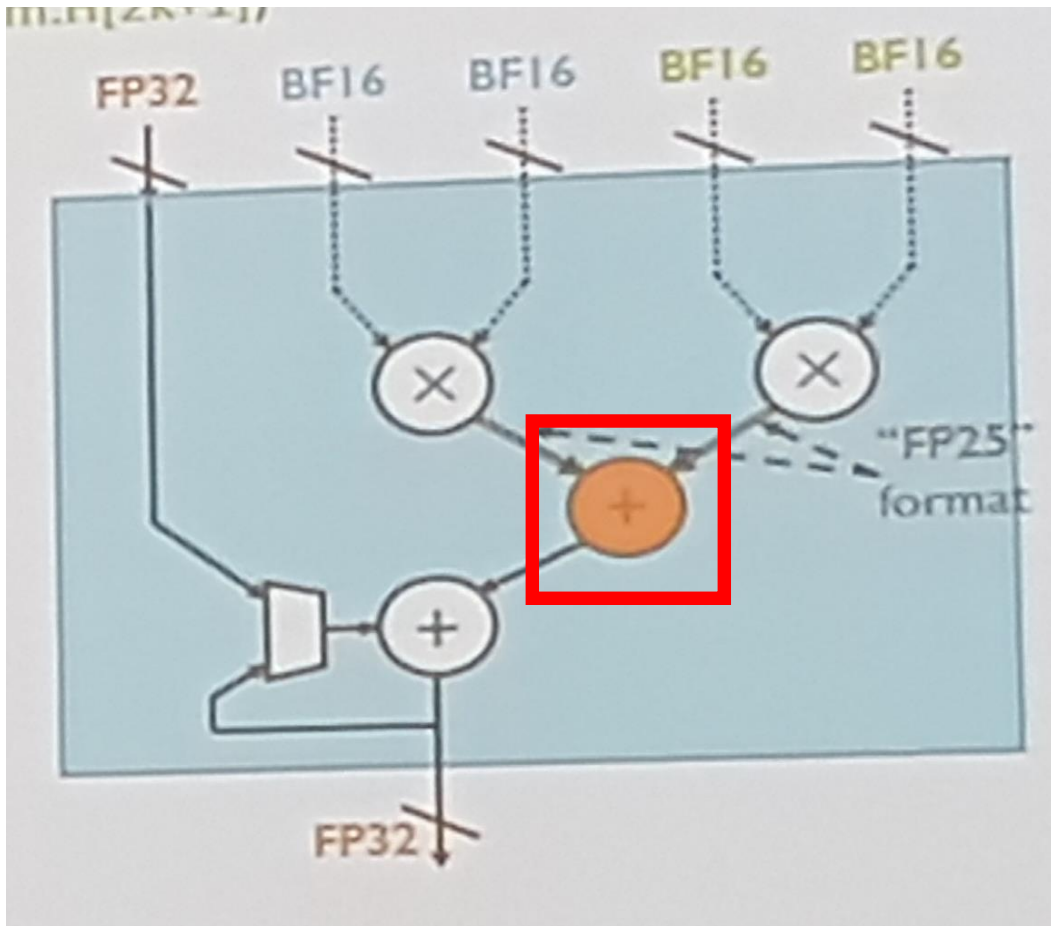
```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8,14>( x0, y0 );  
    auto xy1 = fp_mul<8,14>( x1, y1 );  
    auto sum = fp_add<8,23>( xy0, xy1 );  
    return fp_add<8,23>( acc, sum );  
}
```

Synthesisable BFDOT in 10 lines of code



```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8, 14>( x0, y0 );  
    auto xy1 = fp_mul<8, 14>( x1, y1 );  
    auto sum = fp_add<8, 23>( xy0, xy1 );  
    return fp_add<8, 23>( acc, sum );  
}
```

Synthesisable BFDOT in 10 lines of code



```
float32_t BFDOT2(  
    float32_t acc,  
    bfloat16_t x0, bfloat16_t y0,  
    bfloat16_t x1, bfloat16_t y1  
) {  
    auto xy0 = fp_mul<8, 14>( x0, y0 );  
    auto xv1 = fp_mul<8, 14>( x1, y1 );  
    auto sum = fp_add<8, 23>( xy0, xv1 );  
    return fp_add<8, 23>( acc, sum );  
}
```


Advantages

- Platform independence
 - Plain C++ : *test applications without any HDL simulators or vendor libraries:*
 - Bit-exact results: *same answer on every platform (and only sometimes 42)*
- Performance
 - Area and performance similar to proprietary vendor IP
 - Latency is often **lower** than standard vendor IP data-path
- Freedom and control
 - Easily create precisely sized operators
 - Add new data-types to the language: e.g. Posits (see FPL 2019, Dinechin et. al.)

Challenges

- Verification
 - The possible space of operators is huge: *millions of combinations*
 - Many more corner cases than heterogeneous operators
- How to pick the custom number formats?
 - Error analysis is hard enough for us – how do “normal” people do it?
 - Tools like FPTaylor and daisy assume standard-size homogeneous types
 - How can we help users *safely* get the benefit of custom precision?

Conclusion

- Templatised soft floating point is feasible and efficient
 - Works in production HLS tools
 - Produces similar quality-of-results to vendor IP blocks
- Generating at compile-time has some unique advantages
 - Fully heterogenous types increases efficiency and accuracy
 - Provides more optimisation opportunities in the HLS scheduler
- Floating-point library is available as open-source:
<https://github.com/template-hls/template-hls-float>
It works, but is quite alpha-level at the moment