Software-defined IP cores: high-level synthesis without black-box primitives

David B. Thomas
Imperial College London
dt10@imperial.ac.uk
UKDF, May 2019

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 );
}
```

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 );
}
```

✗ 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 );
}
```

- ✗ 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 );
}
```

- ✗ 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

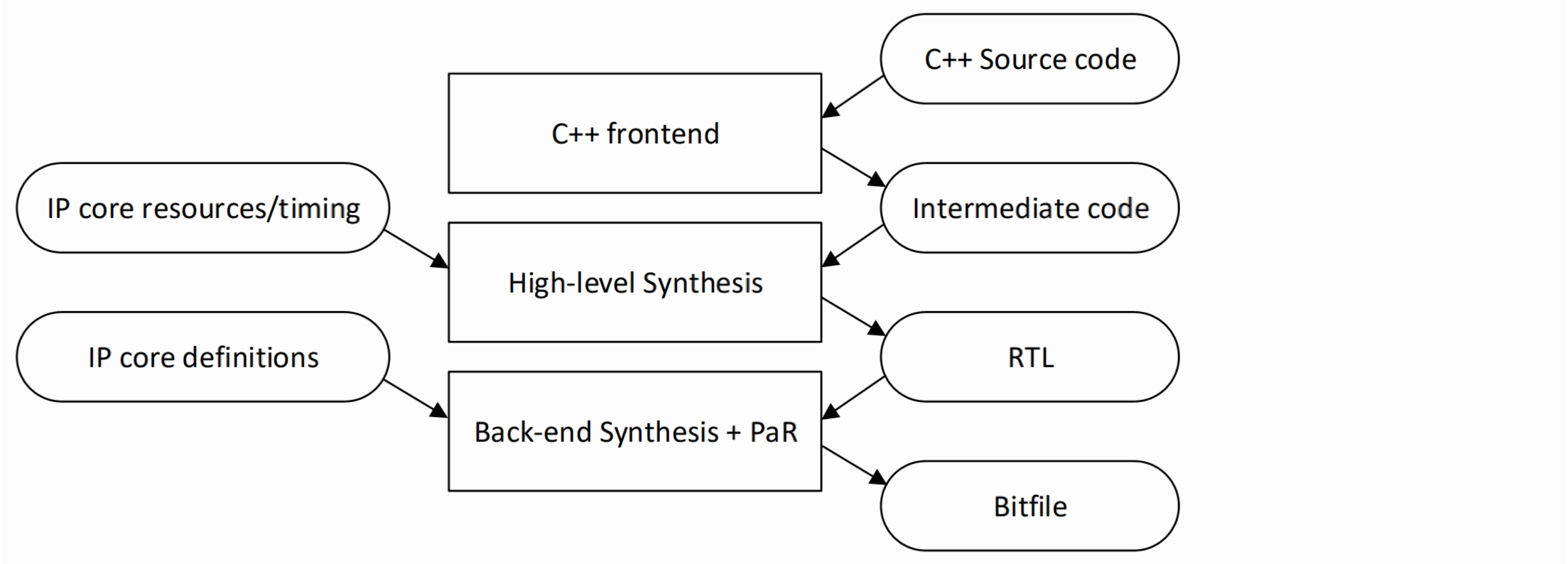
```
float f(float x, float y)
{
    float xy = x*y;
    return xy + x;
}
```

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

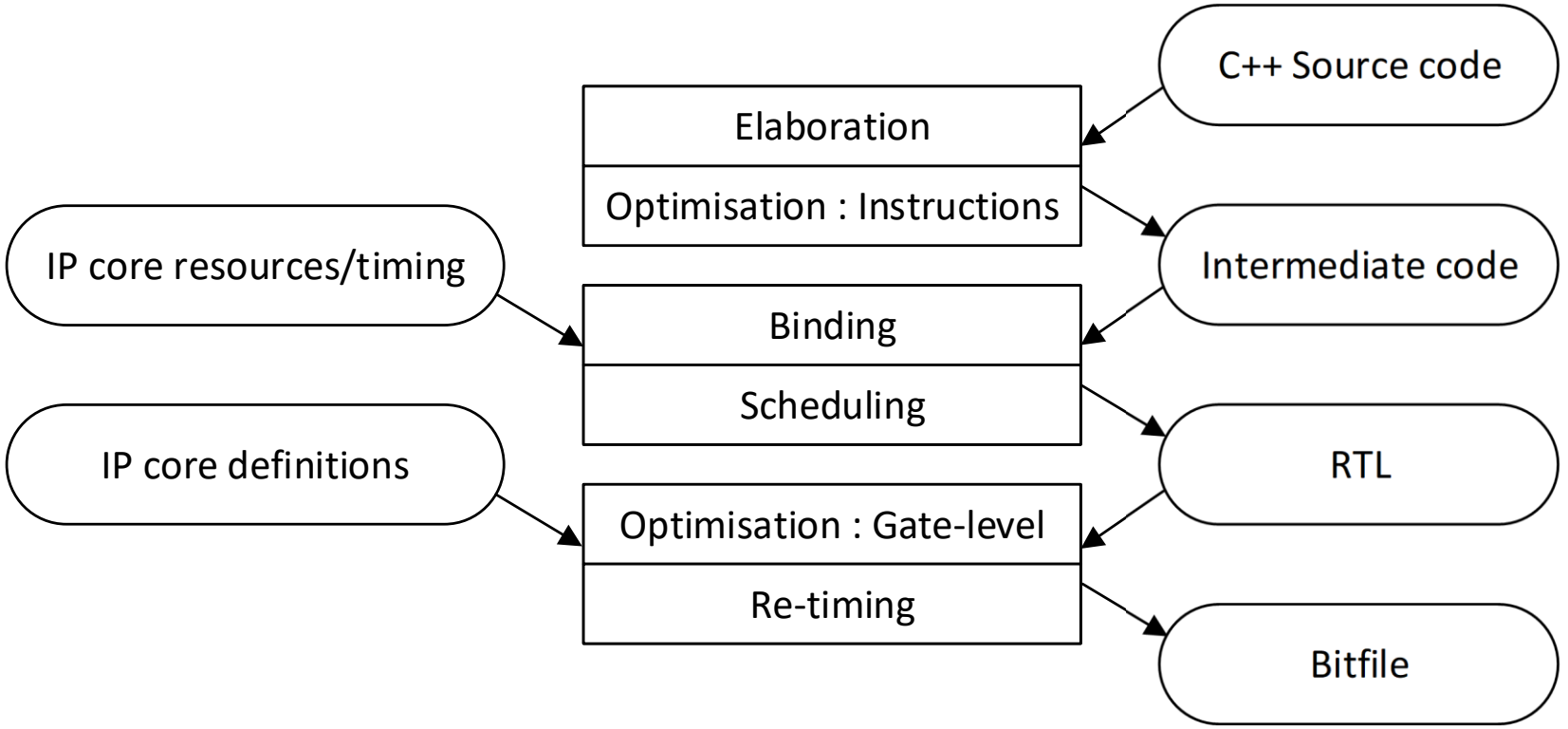
- ✗ Three types: double, float, half
- ✗ Homogeneous operands
- ✗ Platform dependent results
- ✓ Efficient in hardware

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

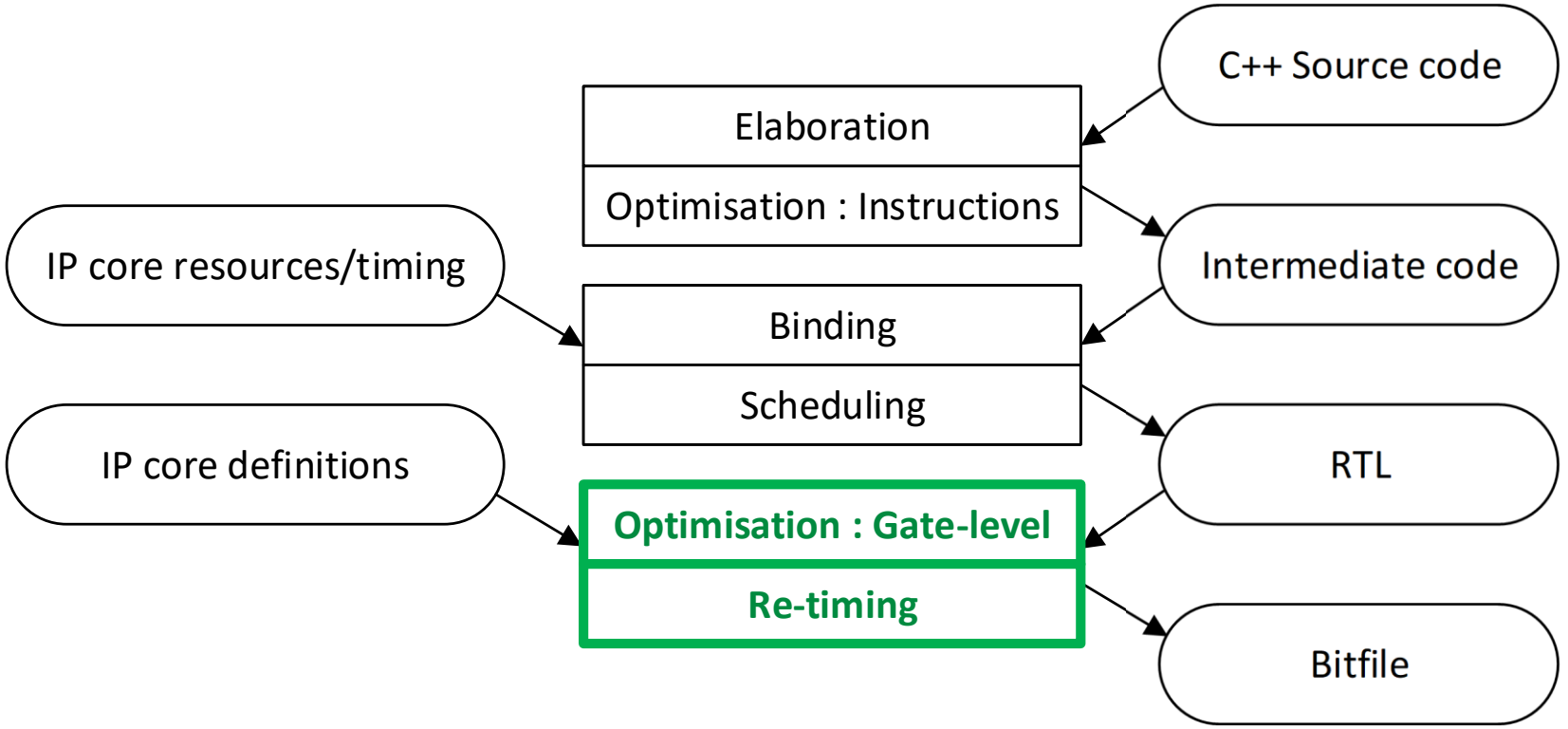
Background: floating-point in HLS



Background: floating-point in HLS

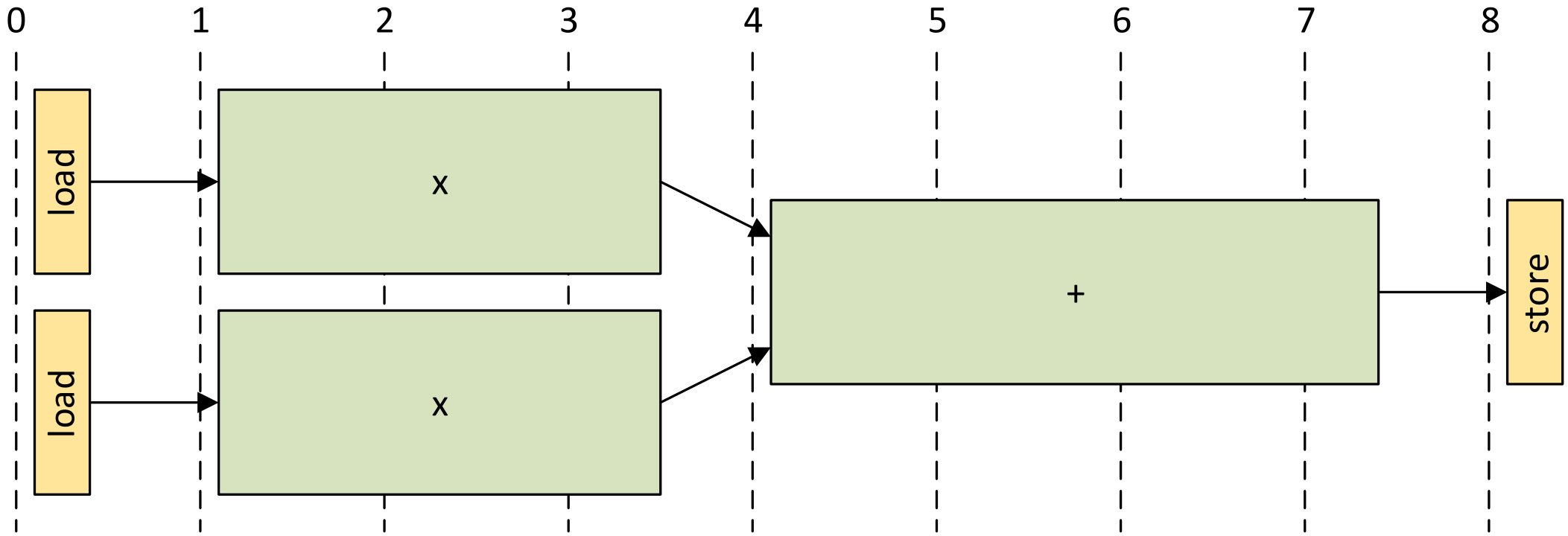


Background: floating-point in HLS



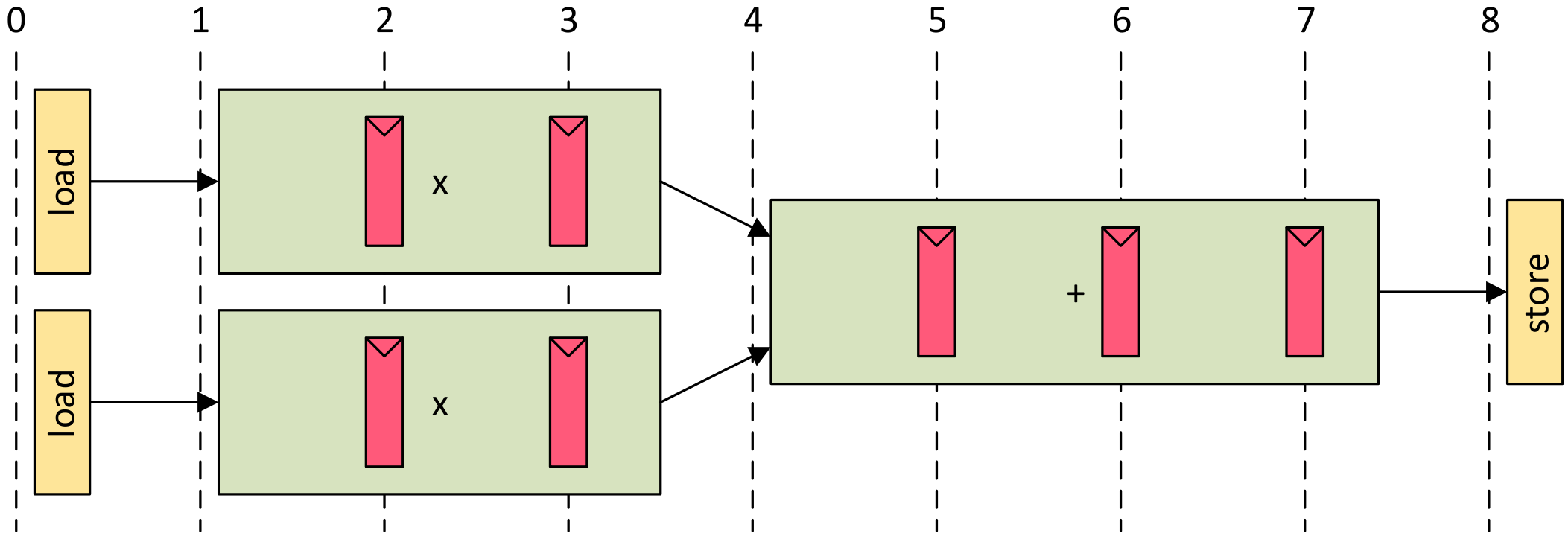
Motivation: scheduling of floating-point

Floating point is scheduled as black-box pipelines



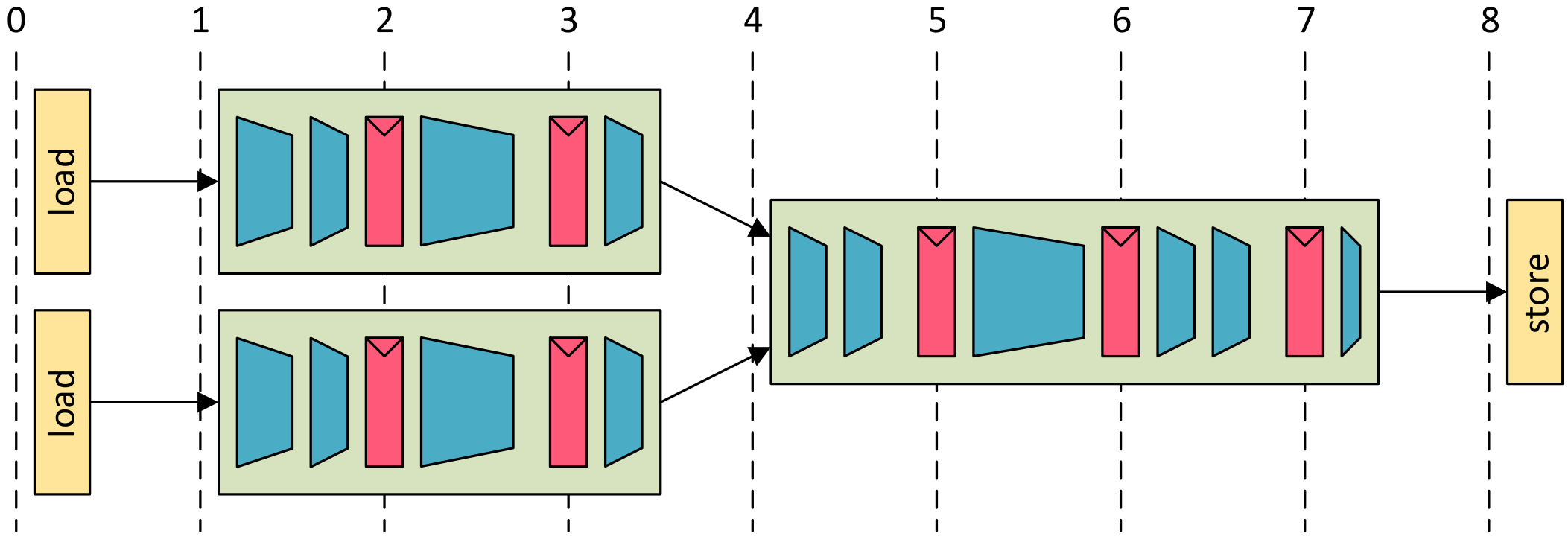
Motivation: scheduling of floating-point

Number of registers is known, but not much else



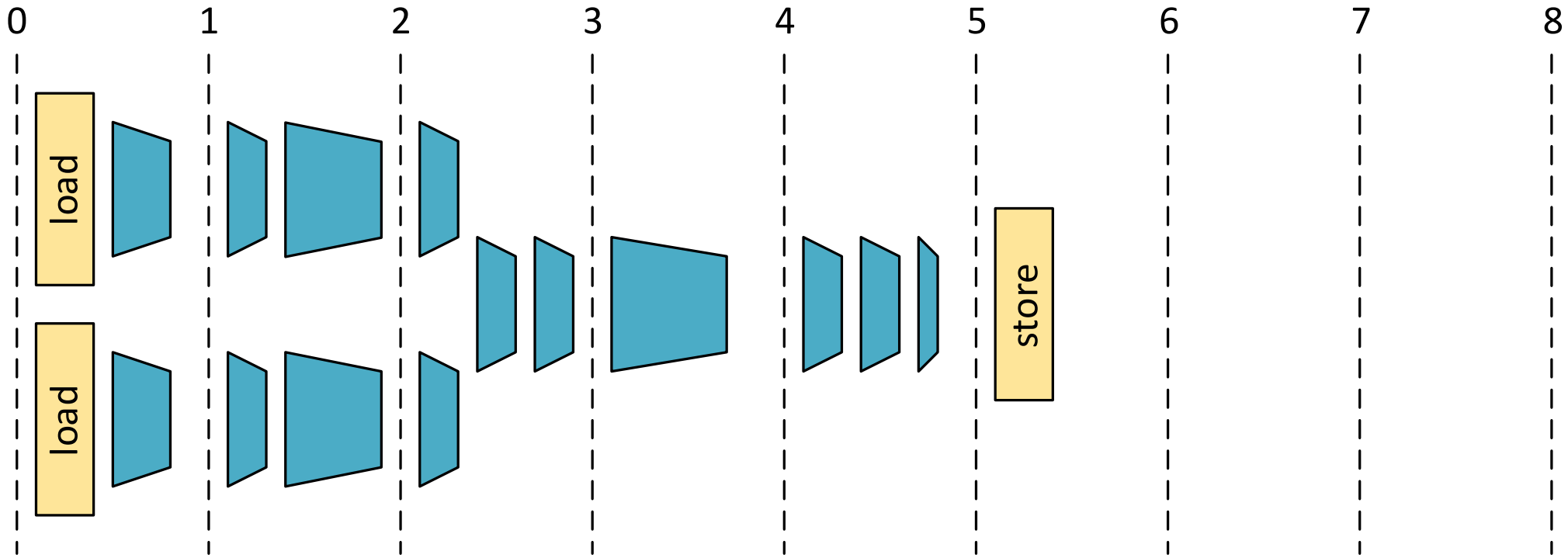
Motivation: scheduling of floating-point

Exposing internal structure makes timing visible to HLS scheduler



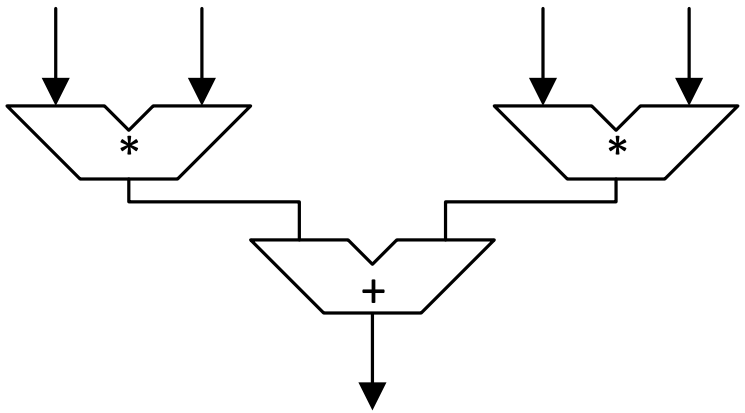
Motivation: scheduling of floating-point

HLS scheduler can pipeline based on current clock constraints



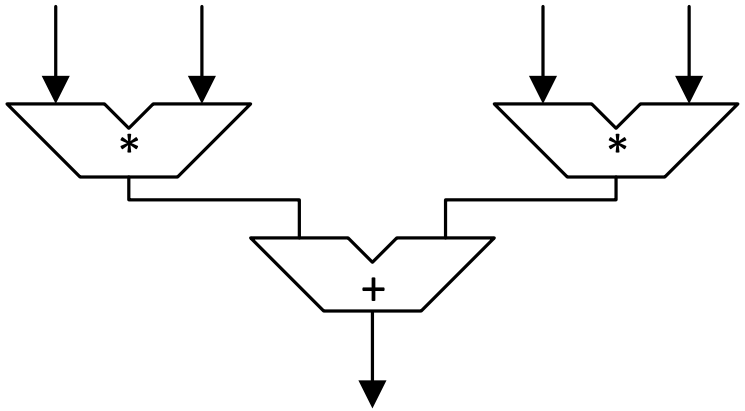
Motivation: custom precision operations

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

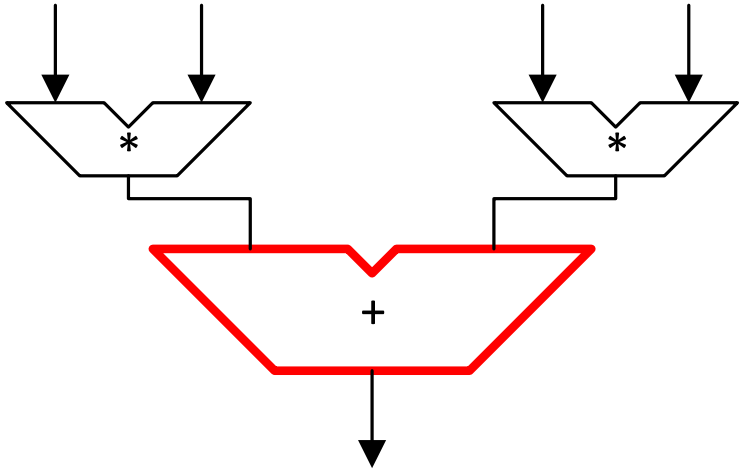


Motivation: custom precision operations

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

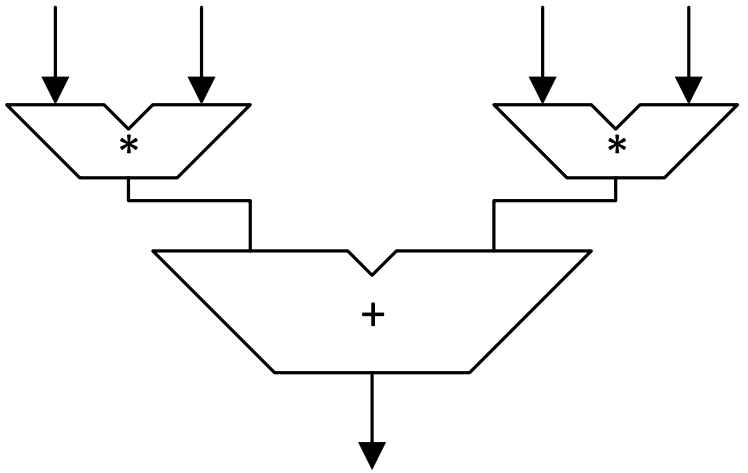



```
float f(float x[2], float y[2])
{
  float xy0 = x[0] * y[0];
  float xy1 = x[1] * y[1];
  return double(xy0) + double(xy1);
}
```



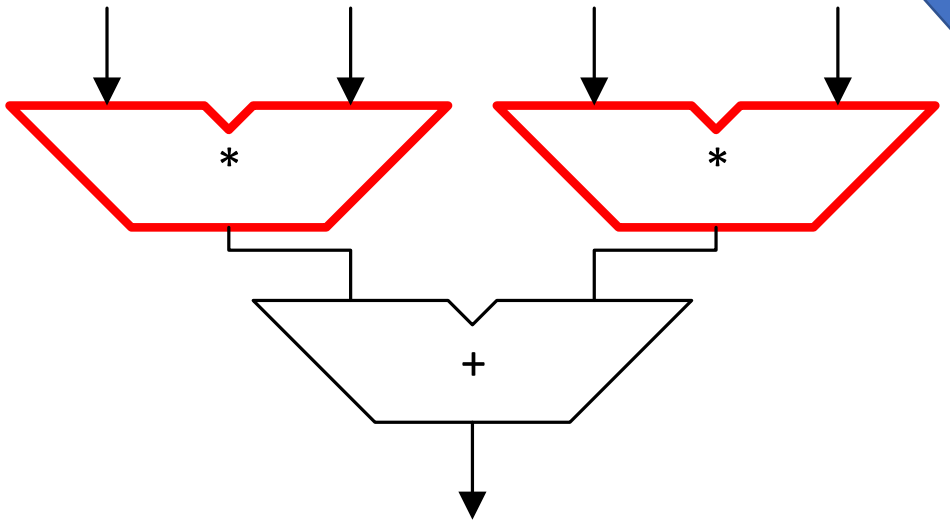
Motivation: custom precision operations

```
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);
}
```

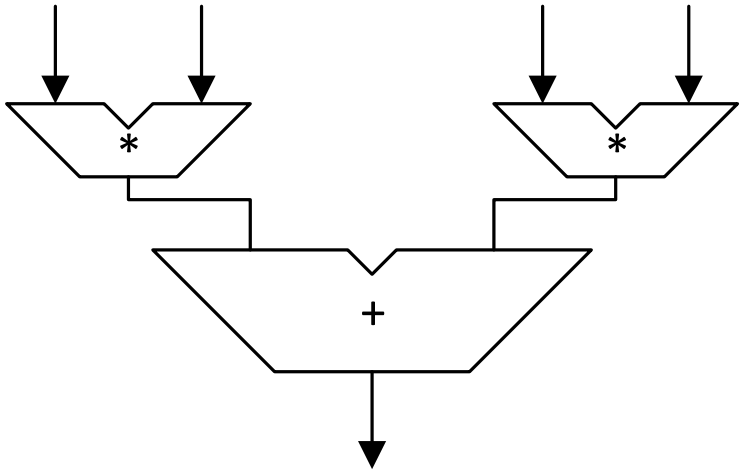


Motivation: custom precision operations

```
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);
}
```

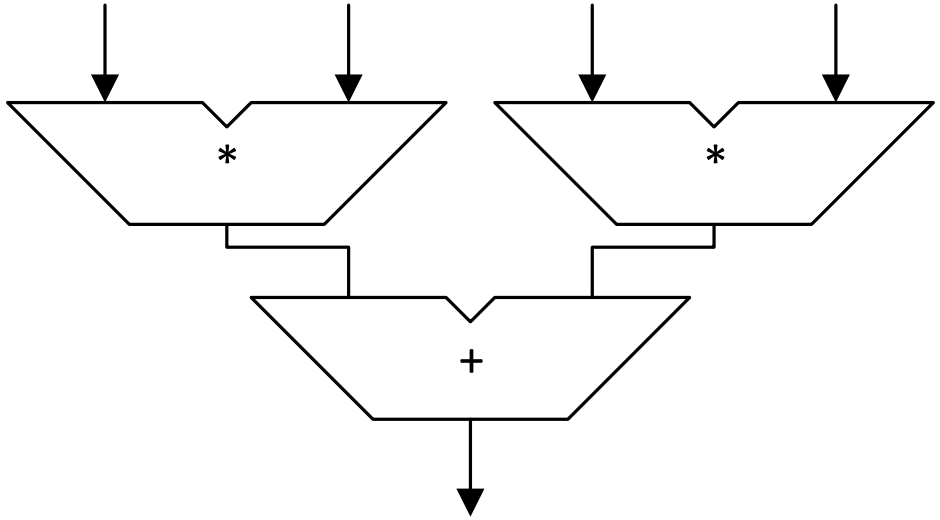



```
float f(float x[2], float y[2])
{
  float xy0 = x[0] * y[0];
  float xy1 = x[1] * y[1];
  return double(xy0) + double(xy1);
}
```



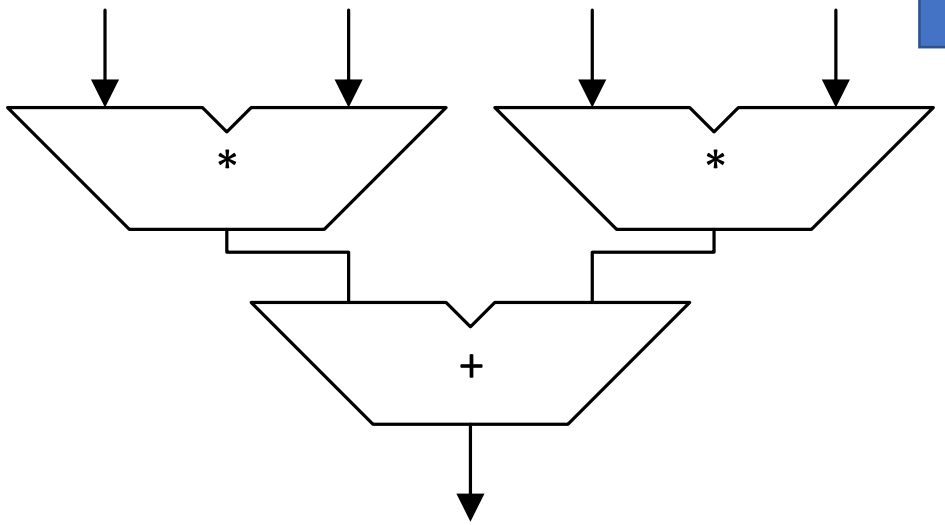
Motivation: custom precision operations

```
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);
}
```

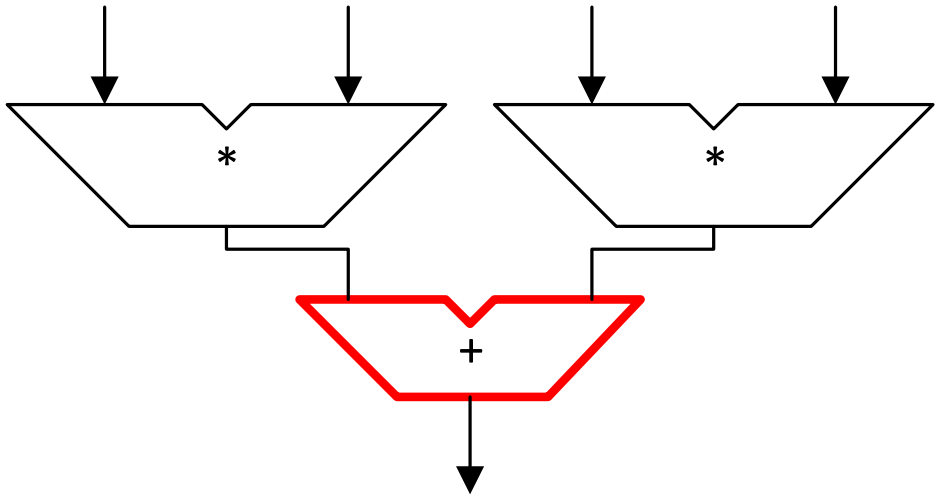


Motivation: custom precision operations

```
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);
}
```

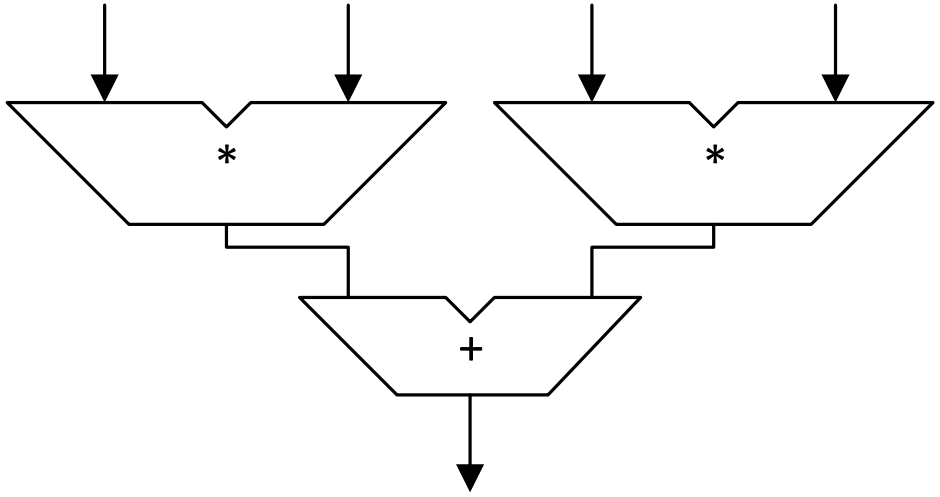



```
float f(float x[2], float y[2])
{
  double xy0 = double(x[0]) * double(y[0]);
  double xy1 = double(x[1]) * double(y[1]);
  return float41(xy0) + float41(xy1);
}
```



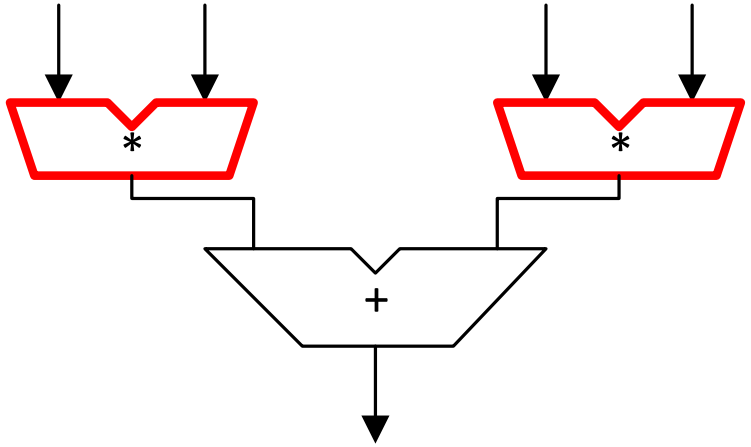
Motivation: custom precision operations

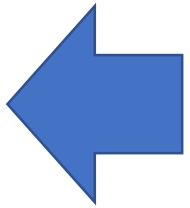
```
float f(float x[2], float y[2])
{
  double xy0 = double(x[0]) * double(y[0]);
  double xy1 = double(x[1]) * double(y[1]);
  return float41(xy0) + float41(xy1);
}
```



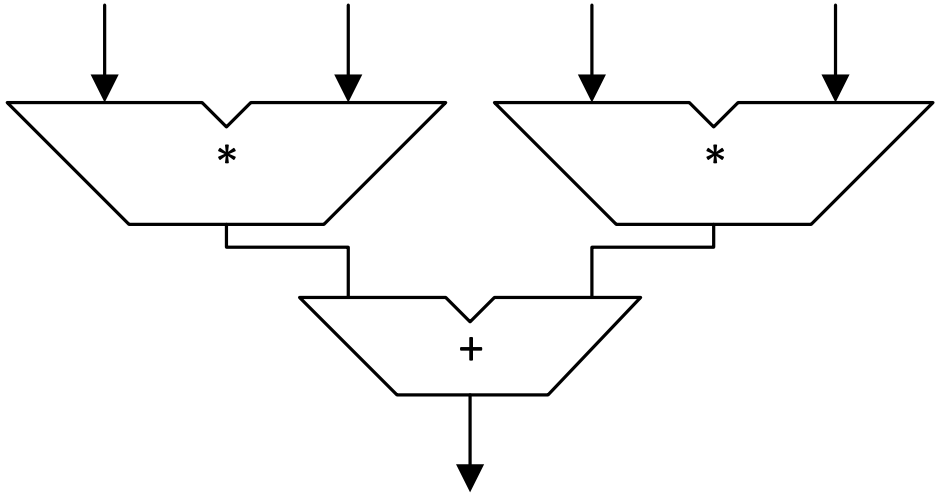
Motivation: custom precision operations

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

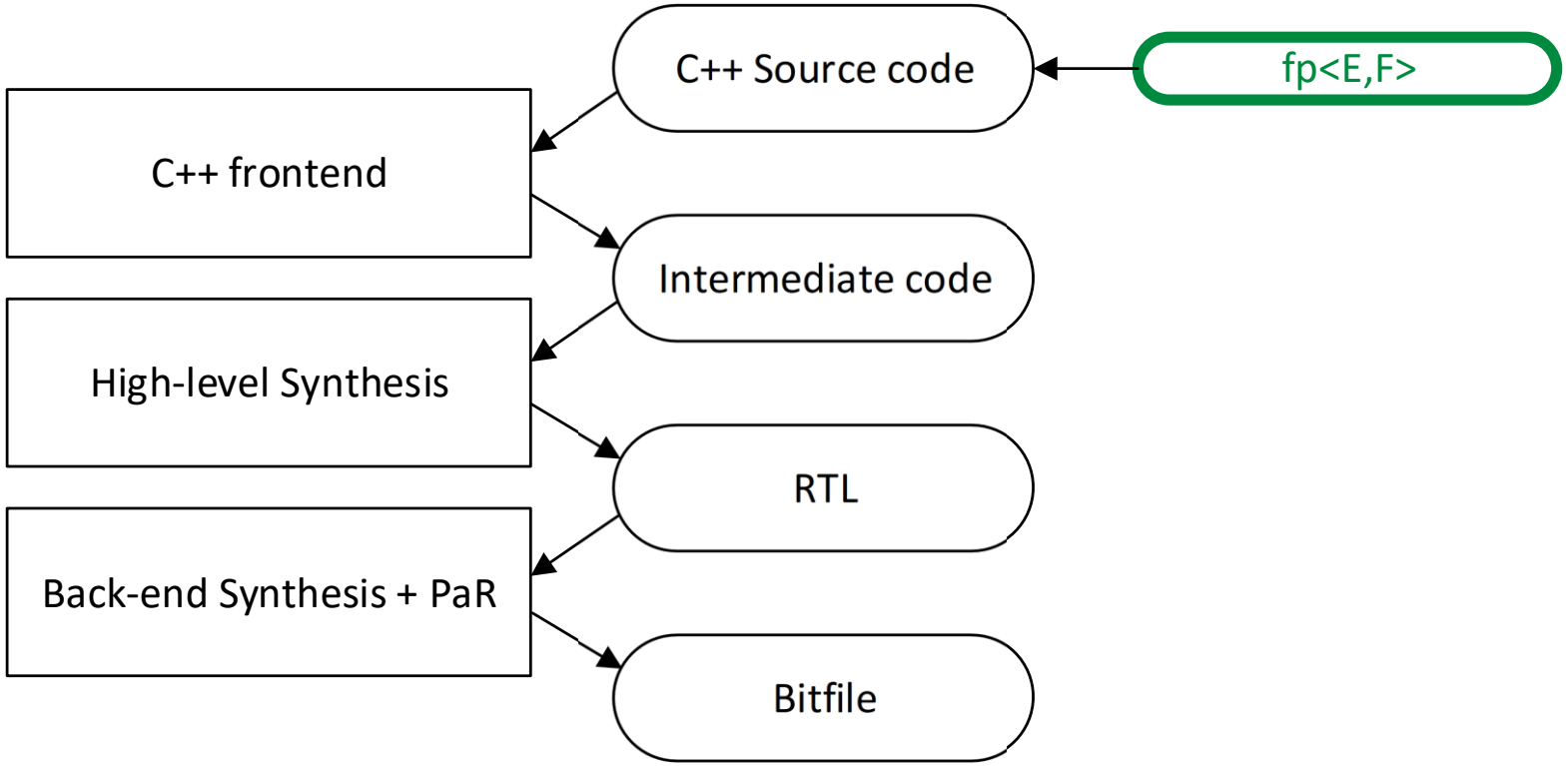




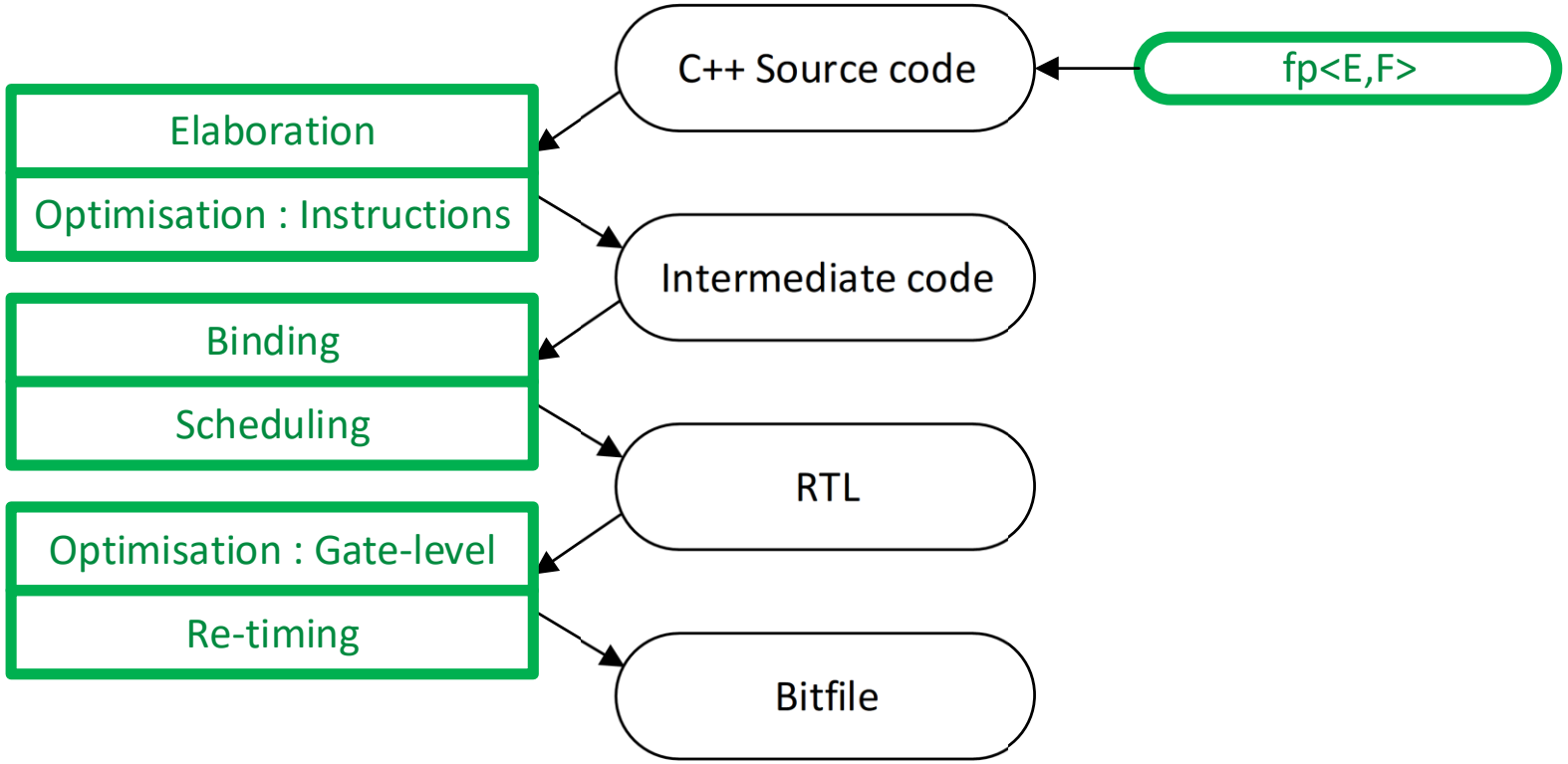
```
float f(float x[2], float y[2])
{
  double xy0 = double(x[0]) * double(y[0]);
  double xy1 = double(x[1]) * double(y[1]);
  return float41(xy0) + float41(xy1);
}
```



Solution: templatised soft floating point



Solution: templatised soft floating point



Method

FloPoCo : standing on the shoulders of giants

- FloPoCo is a tool for generating floating-point IP
  - Algorithms produce high performance operators
  - Has been extensively tested and widely used

- Algorithms need to be adapted for templatised floating-point
  - Uses run-time logic to specialise operators for exponent and fraction
  - Supports homogenous width operators

Conversion process

1. Pick an operator
   - Started with FloPoCo operators: add, mul, div
   - Now adding non FloPoCo operators: reciprocal, constant multipliers

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time

```
vhdl << decl("excExpFracX",2+wE+wF)
     << " <= X"<<range(wE+wF+2, wE+wF+1)
     << " & X"<<range(wE+wF-1, 0)<<";"<<endl;
```

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time

```
vhdl << decl("excExpFracX",2+wE+wF)
     << " <= X"<<range(wE+wF+2, wE+wF+1)
     << " & X"<<range(wE+wF-1, 0)<<";"<<endl;


-- In architecture declarations signal
excExpFracX : std_logic_vector(32 downto 0);
-- In architecture statements
excExpFracX <= X(33 downto 32) & X(30 downto 0);
```

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time

```
vhdl << decl("excExpFracX",2+wE+wF)
     << " <= X"<<range(wE+wF+2, wE+wF+1)
     << " & X"<<range(wE+wF-1, 0)<<";"<<endl;


fw_uint<2+wE+wF> excExpFracX = concat(
    get_bits<wE+wF+2, wE+wF+1>(X) ,
    get_bits<wE+wF-1, 0>(X)
);
```

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
   - Utility functions for padding, conversion, …
   - Compile-time initialised ROMs
   - Combined shift and normalisation

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
   - Utility functions for padding, conversion, …
   - Compile-time initialised ROMs
   - Combined shift and normalisation

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
4. Get it working in plain C++
   - Verify numerical correctness in C++ before even thinking about HLS
   - Compile operator for cross-product of:
     - Exponents: 4..12
     - Fractions: 4..64
   - Run each compiled operator for edge cases plus ~1M random cases

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
4. Get it working in plain C++
5. Fight the HLS tool front-end

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
4. Get it working in plain C++
5. Fight the HLS tool front-end
6. Plead with the HLS tool back-end

Conversion process

1. Pick an operator
2. Convert each generator line from run-time to compile-time
3. Create helpers for new integer operations
4. Get it working in plain C++
5. Fight the HLS tool front-end
6. Plead with the HLS tool back-end
7. Go back to 1

Challenges: correctness

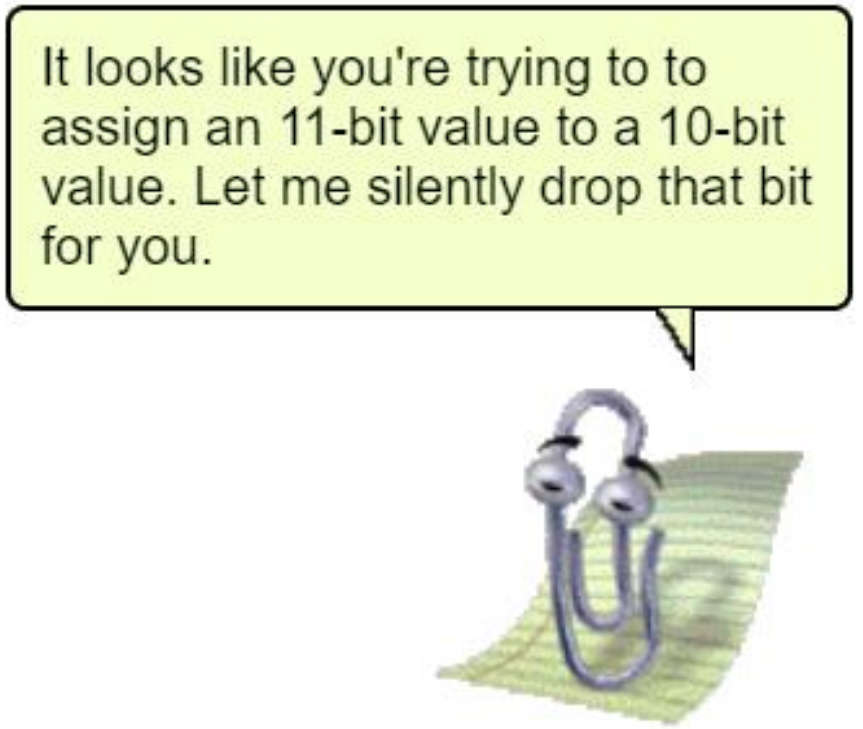
Initially tried to convert to Xilinx's ap_int directly

```
ap_uint<log2(W+1)> b;
// …
ap_uint<log2(W)> a;
// …
a=b;
```

Challenges: correctness

Initially tried to convert to Xilinx's ap_int directly

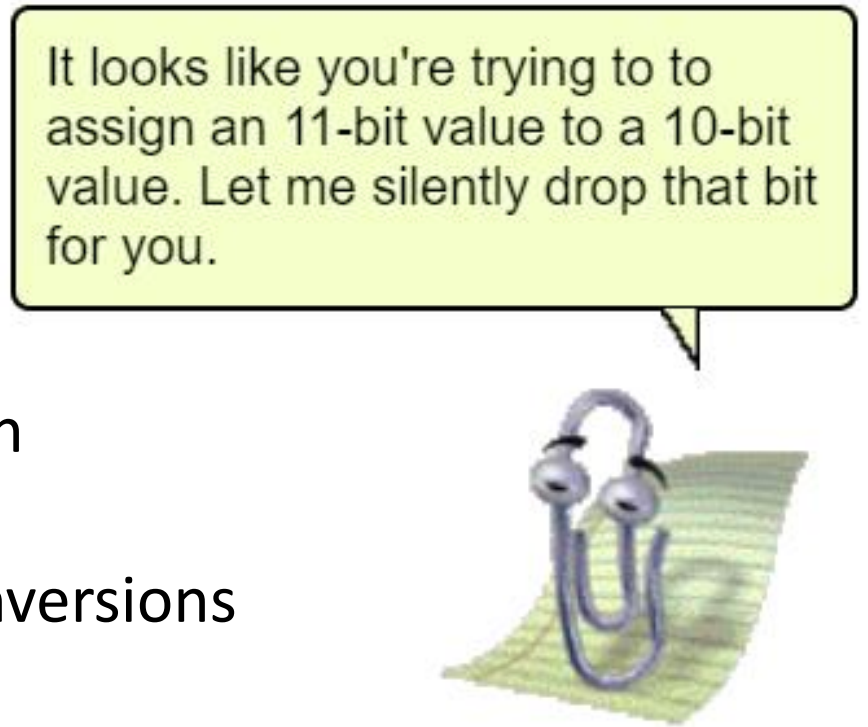
```
ap_uint<log2(W+1)> b;
// …
ap_uint<log2(W)> a;
// …
a=b;
```



Challenges: correctness

Initially tried to convert to Xilinx's ap_int directly

```
ap_uint<log2(W+1)> b;
// …
ap_uint<log2(W)> a;
// …
a=b;
```



In a floating-point core the widths ***must*** match

We want very strict typing and no implicit conversions

Challenges: portability

- We want to write and verify the algorithms once in plain C++
  - Identical bit-exact results on all platforms: Xilinx, Intel, Lattice, …
  - Efficient on known platforms
  - Extensible to future platforms

Challenges: portability

- We want to write and verify the algorithms once in plain C++
  - Identical bit-exact results on all platforms: Xilinx, Intel, Lattice, …
  - Efficient on known platforms
  - Extensible to future platforms

- Main barrier is custom width integers
  - What types do we use?
  - What are the semantics of operators?
  - How do we express shifts and normalisations?

Problem: too many integer libraries

```
#include <cstdint>
#include “boost/cpp_int.hpp”
#include “ap_int.h”
#include “ac_int.h”
#include “sc_int.h”
```

Solution: yet another integer library

```
#include <cstdint>
#include “boost/cpp_int.hpp”
#include “ap_int.h”
#include “ac_int.h”
#include “sc_int.h”
```

Solution: yet another integer library

```
#include <cstdint>
#include “boost/cpp_int.hpp”
#include “ap_int.h”
#include “ac_int.h”
#include “sc_int.h”

#include “fw_uint.h”
```

Solution: yet another integer library

```
#include <cstdint>
#include “boost/cpp_int.hpp”
#include “ap_int.h”
#include “ac_int.h”
#include “sc_int.h”

#include “fw_uint.h”
```

```
template<int W>
class fw_uint<W>;
```

- Type designed for IP development
  - Strict width semantics
  - No implicit conversions
- Zero overhead abstraction
  - Inlines away completely
  - Maps to efficient platform API

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>;
```

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);

fp<6,31> x;
fp<7,20> 7;
```

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> 7;

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> 7;

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

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

Results

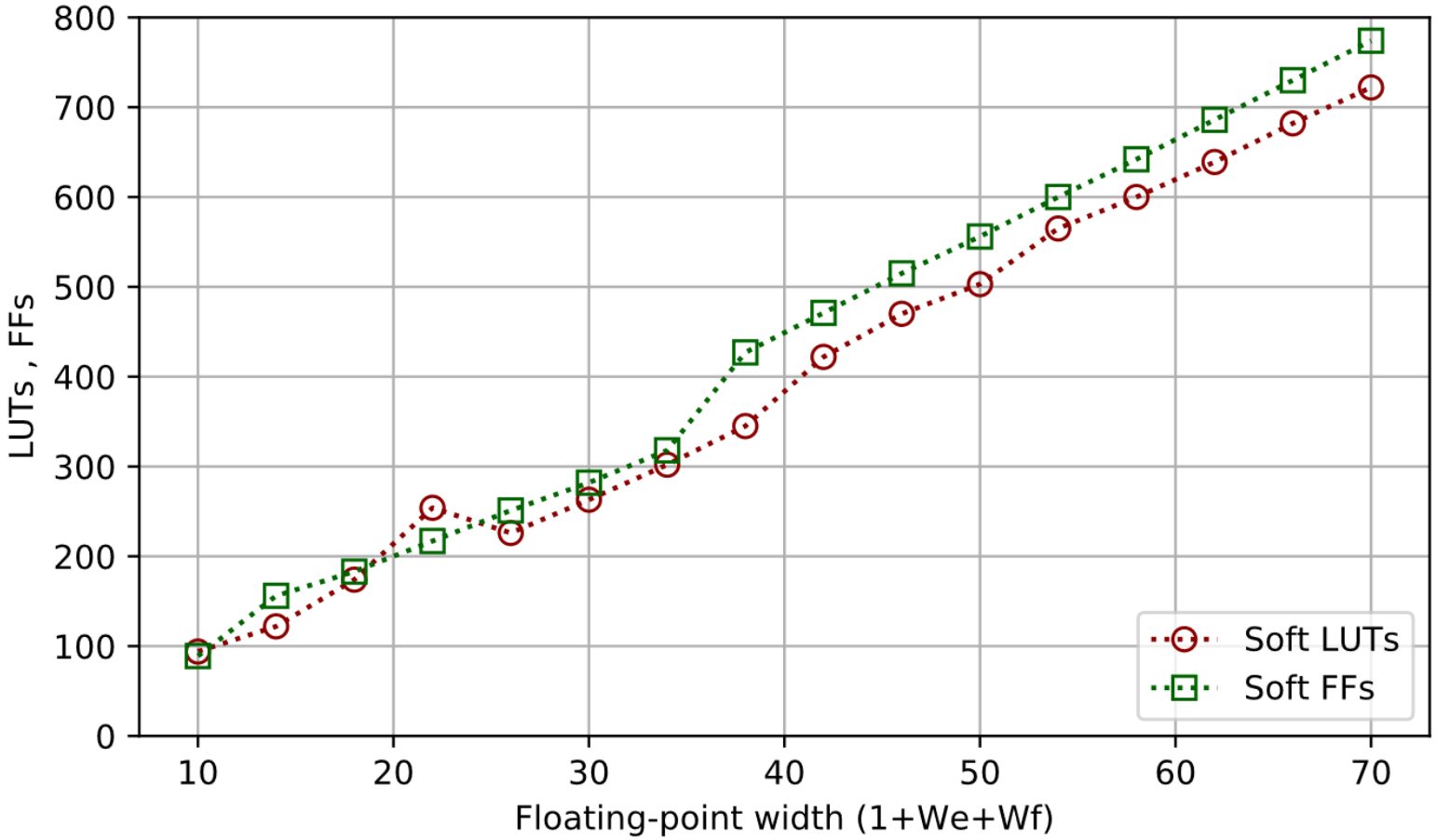
Results: setup

HLS + PaR: Vivado 2018.3
Part : Virtex 7
Target clock: 200MHz

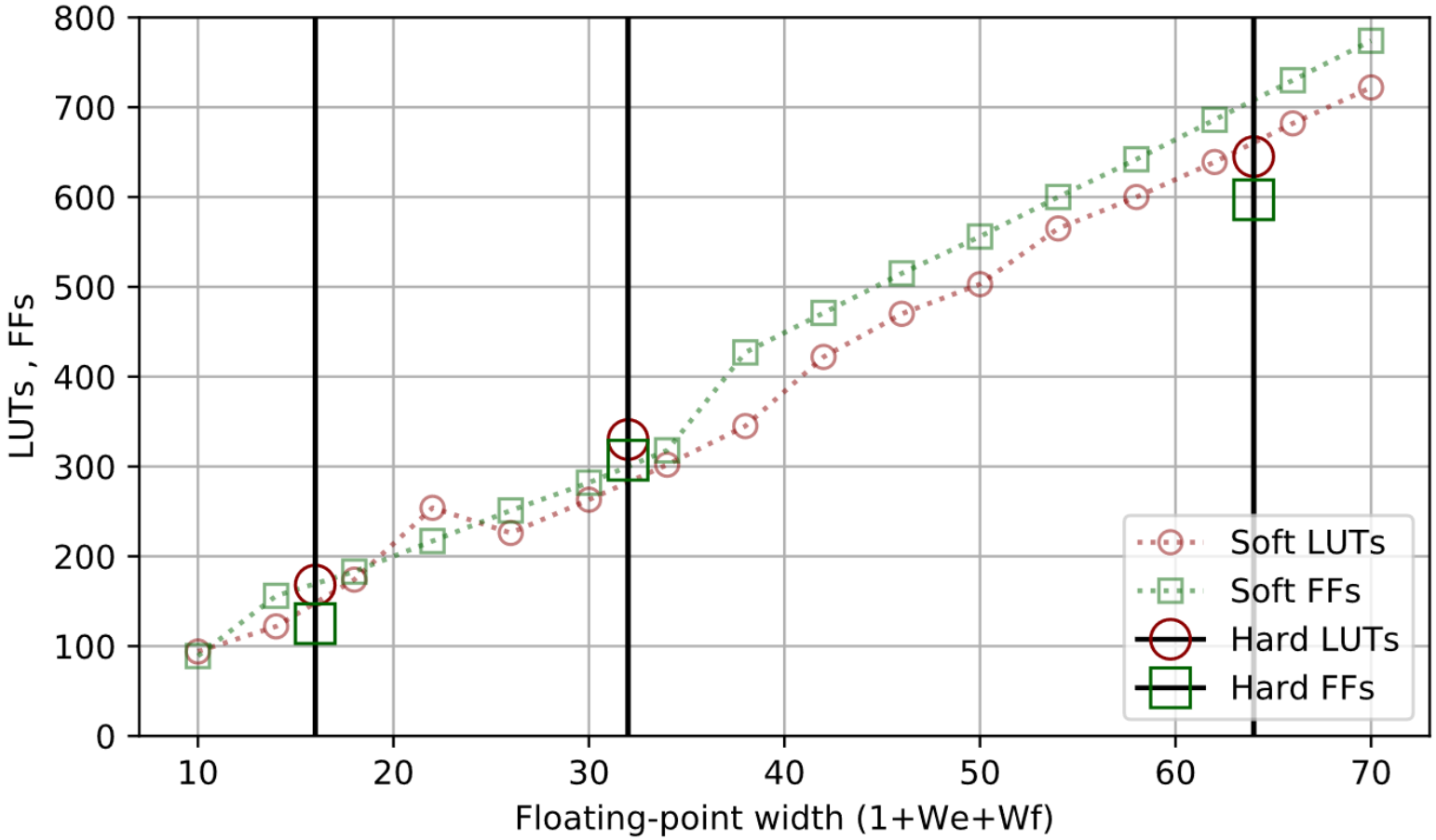
Chose vendor operators with similar LUT/DSP balance

All results are post place-and-route

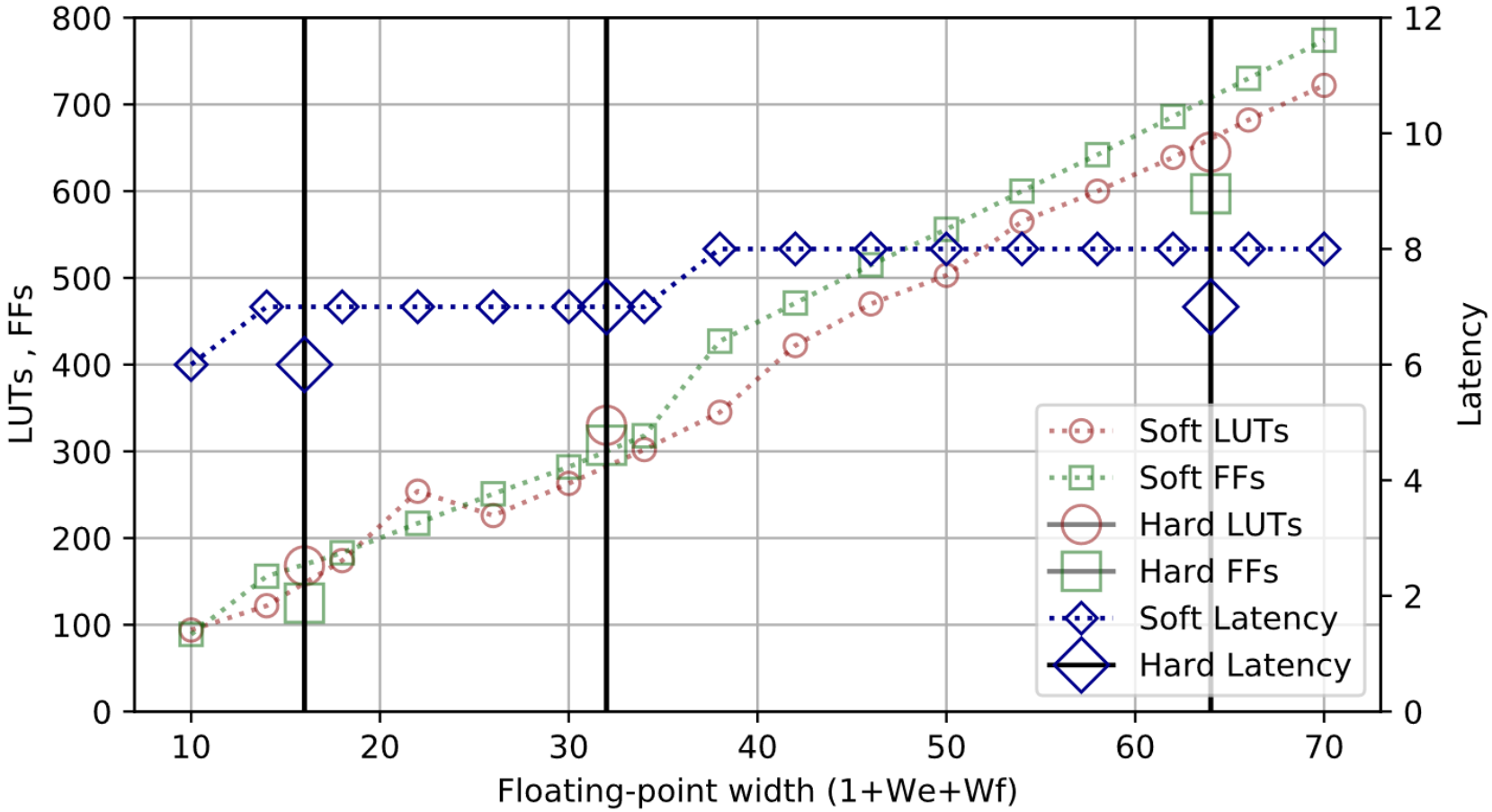
Results: homogeneous adder



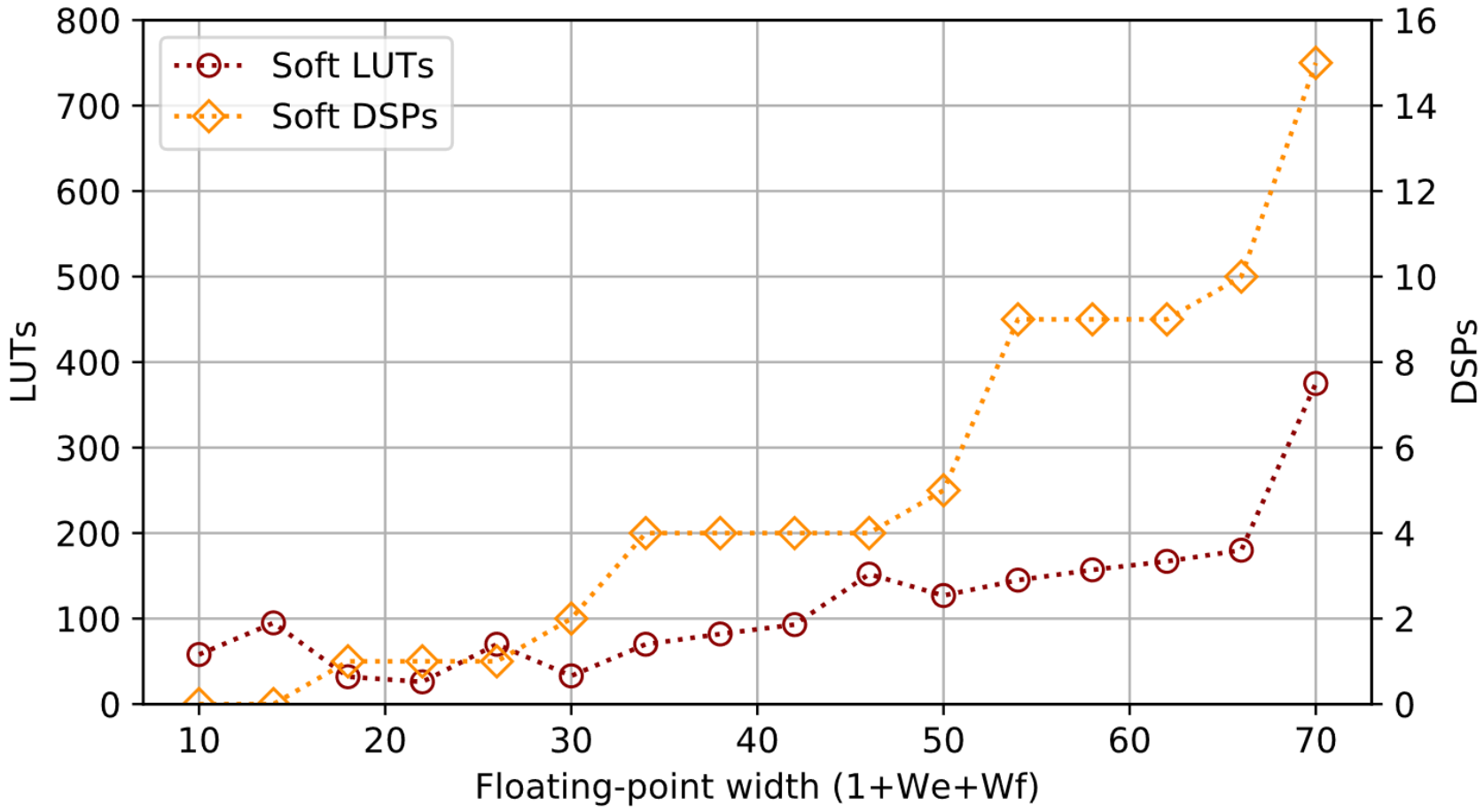
Results: homogeneous adder



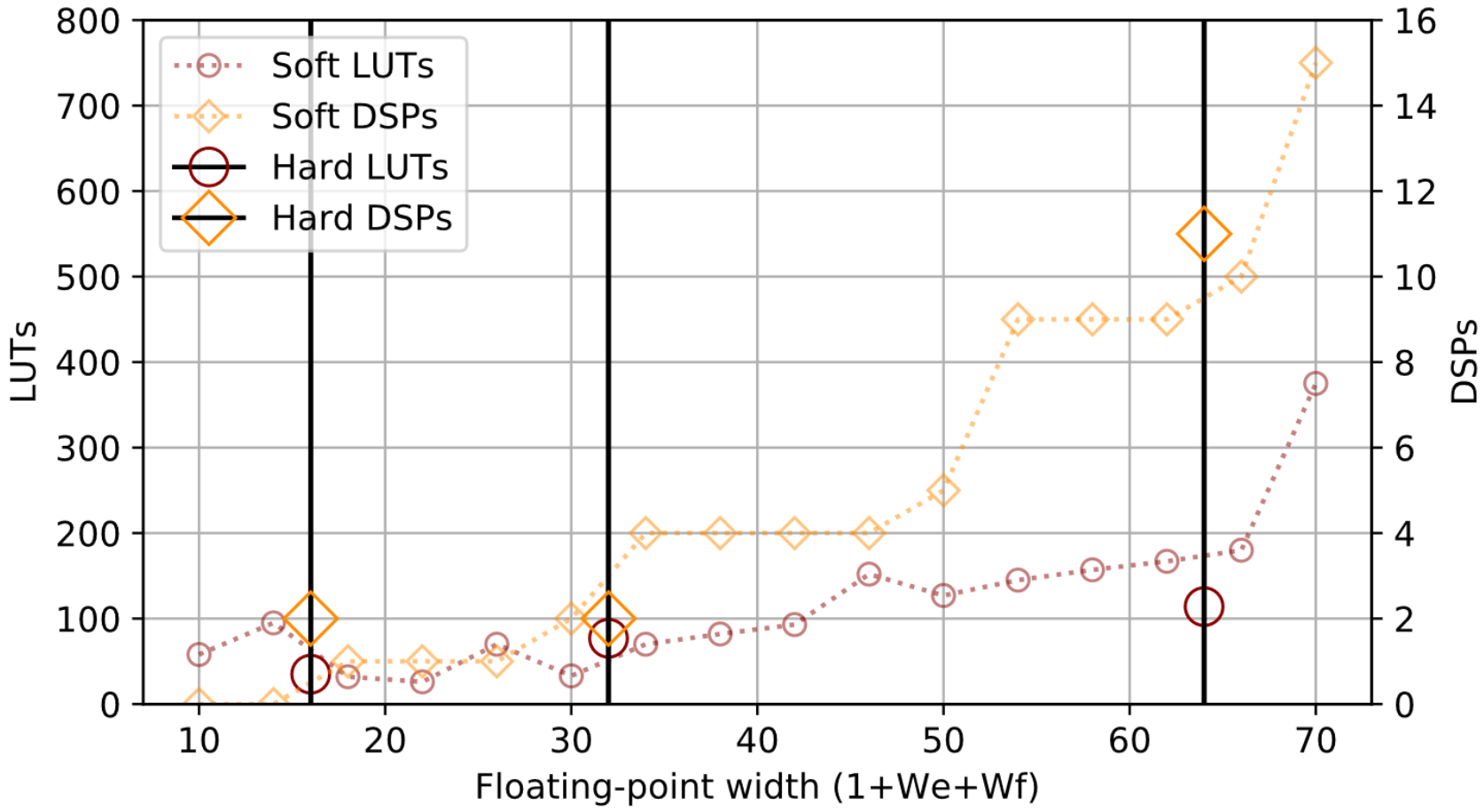
Results: homogeneous adder



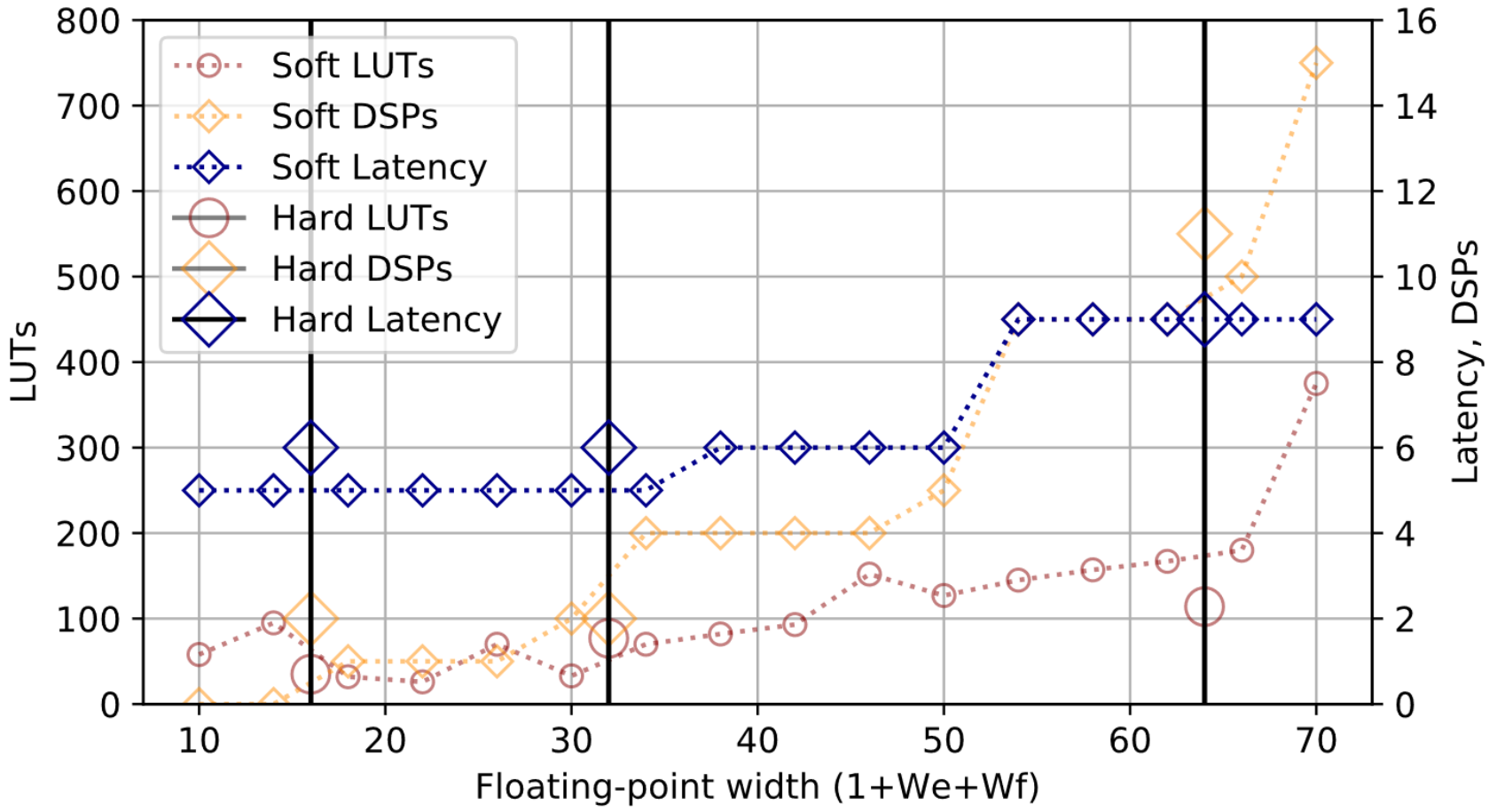
Results: homogeneous multiplier



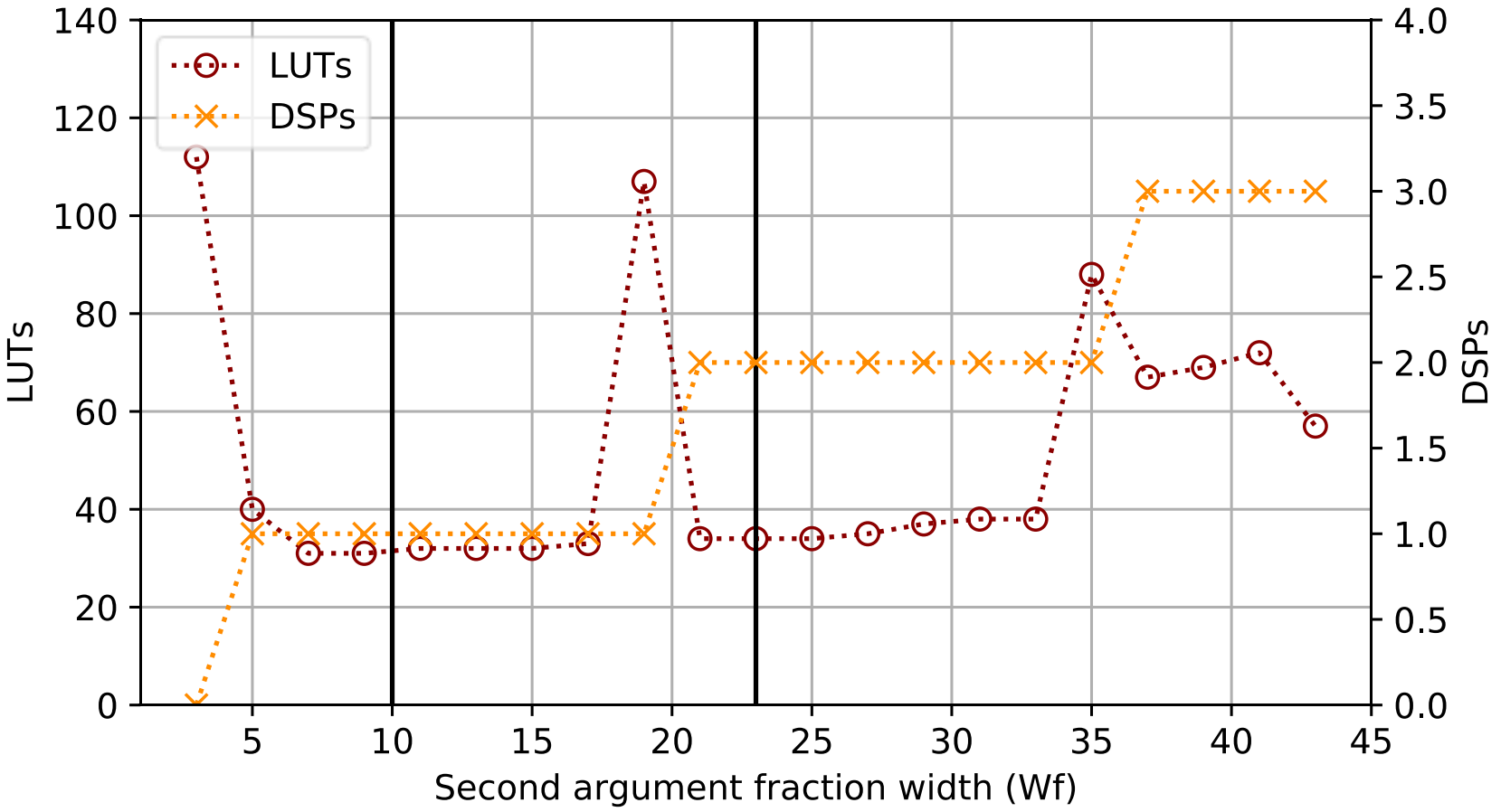
Results: homogeneous multiplier



Results: homogeneous multiplier



Results: ***heterogeneous*** multiplier



First argument is single precision; second argument is varying precision

Results: four element dot-product

Parameters					Results			
Variant	TA	TL	TR	Round	LUT	FF	DSP	Delay
Soft	half	half	half	round	520	642	4	13
	single	half	half	full	1428	1255	4	12
	single	half	half	round	722	943	4	14
	single	single	half	round	900	1165	4	14
	single	single	single	round	912	1241	8	14
Vendor	half	half	half	round	655	770	8	14
	single	half	half	round	1195	1459	8	20
	single	single	half	round	1404	1672	8	19
	single	single	single	round	1313	1577	8	16

Results: four element dot-product

Parameters					Results			
Variant	TA	TL	TR	Round	LUT	FF	DSP	Delay
Soft	half	half	half	round	520	642	4	13
	single	half	half	full	1428	1255	4	12
	single	half	half	round	722	943	4	14
	single	single	half	round	900	1165	4	14
	single	single	single	round	912	1241	8	14
Vendor	half	half	half	round	655	770	8	14
	single	half	half	round	1195	1459	8	20
	single	single	half	round	1404	1672	8	19
	single	single	single	round	1313	1577	8	16

Results: four element dot-product

Parameters					Results			
Variant	TA	TL	TR	Round	LUT	FF	DSP	Delay
Soft	half	half	half	round	520	642	4	13
	single	half	half	full	1428	1255	4	12
	single	half	half	round	722	943	4	14
	single	single	half	round	900	1165	4	14
	single	single	single	round	912	1241	8	14
Vendor	half	half	half	round	655	770	8	14
	single	half	half	round	1195	1459	8	20
	single	single	half	round	1404	1672	8	19
	single	single	single	round	1313	1577	8	16

Results: four element dot-product

Parameters					Results			
Variant	TA	TL	TR	Round	LUT	FF	DSP	Delay
Soft	half	half	half	round	520	642	4	13
	single	half	half	full	1428	1255	4	12
	single	half	half	round	722	943	4	14
	single	single	half	round	900	1165	4	14
	single	single	single	round	912	1241	8	14
Vendor	half	half	half	round	655	770	8	14
	single	half	half	round	1195	1459	8	20
	single	single	half	round	1404	1672	8	19
	single	single	single	round	1313	1577	8	16

Observation: HLS C++ front-ends are lagging

- Got add/mul/div working in 2014-ish Vivado HLS
  - …required some workarounds to fix HLS front-end
- C++11 made this feasible and practical with basic constexpr
  - Vivado HLS supported C++11 (from 2011) around in 2018
- Future revisions really help with templatised IP too
  - C++14: full constexpr
  - C++17: if constexpr / return type deduction
  - C++20: template lambdas
- The “H” in HLS should keep getting more H over time

Current and future work

- *Current work*
  - Performance in large scale applications
  - More operators: *exp, log, …*
- Future work
  - Formal verification backend for `fw_uint`
  - Function approximation at compile-time

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
- Library is available as open-source:

  https://github.com/template-hls/template-hls-float

  It works, but is quite alpha-level at the moment