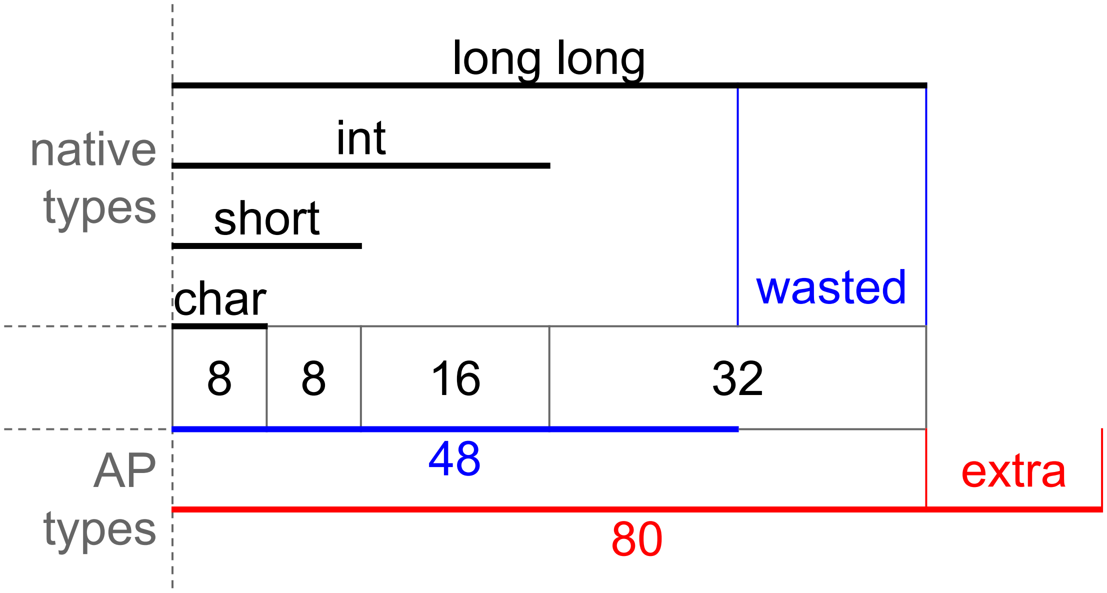
Chapter 3

Bit Accurate Data Types

https://github.com/chenhao1106/Chapter3-BitAccurateDataTypes

Why arbitrary precision types (AP types)?

- C++ native types are byte-sized and at most 8 bytes.
- However, RTL buses support arbitrary bitwidth.
- In need of C++ AP types that can be simulated and synthesized.



Mentor AC types vs Xilinx AP types

- Algorithmic C types

```
ac_int<W, true>

ac_int<W, false>

ac_fixed<W, I, true, Q, O>

ac_fixed<W, I, false, Q, O>
```

- Arbitrary Precision types

```
ap_int<W> : ap_int_base<W, true>

ap_uint<W> : ap_int_base<W, false>

ap_fixed<W, I, Q, O, N>

ap_ufixed<W, I, Q, O, N>
```

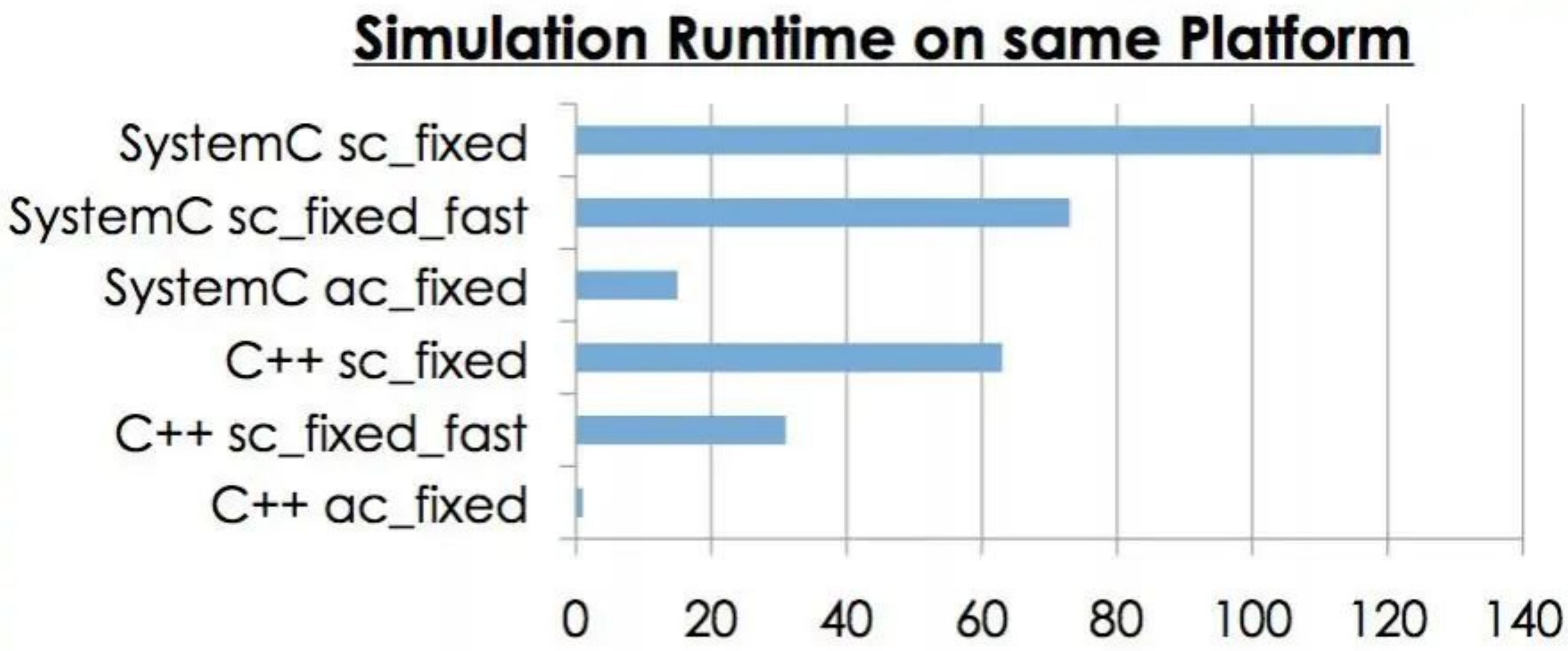
W: bitwidth
I: integral width
Q: quantization mode
O: overflow mode
N: saturation width

Outline

1. Fast switch between native C types and ap(arbitrary precision) types.
2. Supported data types.
3. Operations.
4. Useful utility/helper functions.

Switching between native/AC/AP types

Motivation: simulation performance of AC types



[Calypto 2012](). Calypto is later acquired by Mentor.

Motivation: simulation performance of AC types

- AC types hugely improves SystemC (SC) types.
- AC types are still slower than native types.
  - AC types are arrays of ints.
  - `template<int B, bool S> class iv { int v[B]; … }`
  - Directly synthesizable but suboptimal for simulation.
- Use native types for simulation and AC types for synthesis.

Mentor’s solution: define a custom macro

```
// typedef.h
#ifndef __TYPEDEF_H__
#define __TYPEDEF_H__
#include <ap_int.h>

// dType: input data type
// oType: output data type
#ifdef NATIVE_TYPES
    typedef short int dType;
    typedef int oType;
#else
    typedef ap_int<7> dType;
    typedef ap_int<14> oType;
#endif // NATIVE_TYPES

#endif // __TYPEDEF_H__
```

- Define all type aliases in a header.
- Switch between AP types and native types by defining the macro `NATIVE_TYPES`.

- Not scalable.
- Error-prone.
- Xilinx has none of these caveats.

Xilinx's solution: design a better library

```
template<int W, bool S>
class ap_private { valtype<(W + 7) / 8> };
template<int W, bool S>
struct ssdm_int_sim { ap_private<W> V; };
template<int W, bool S>
struct ssdm_int {
    int V __attribute__((bitwidth(W)));
    __attribute__((always_inline))
        ssdm_int() = default;
};

#ifdef __SYNTHESIS__
    using _AP_ROOT_TYPE = ssdm_int;
#else
    using _AP_ROOT_TYPE = ssdm_int_sim;
#endif

template<int W, bool S>
struct ap_int_base : _AP_ROOT_TYPE<W, S>;
```

- Encapsulate the hassles in the library.
- Split simulation and synthesis.

- Choose between `ssmd_int` and `ap_private` via `__SYNTHESIS__`
- `ssmd_int` instructs the compiler backend to generate synthesizable target code.
- `ap_private` picks best-fitting native types; transparent inlining of operator overloads.

Placeholder type specifier: C++11 `auto` keyword

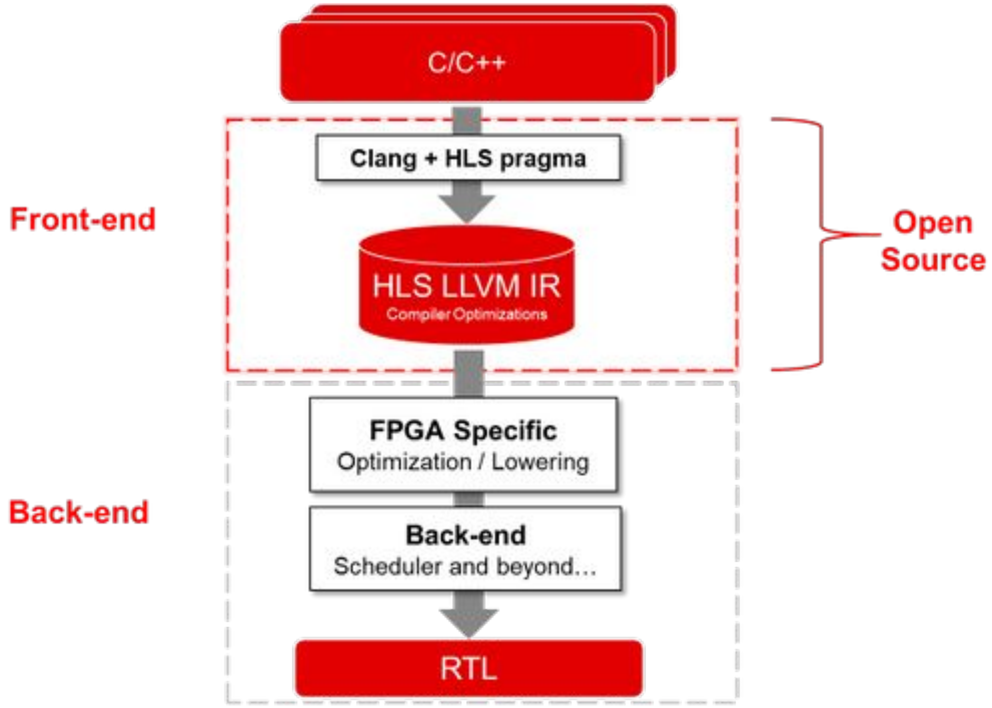
```
ap_int<8> add(ap_int<7> a, ap_int<7> b)
{ return a + b; }
ap_int<14> mul(ap_int<7> a, ap_int<7> b)
{ return a * b; }
ap_int<?> sub(ap_int<7> a, ap_int<7> b)
{ return a - b; }
ap_int<?> div(ap_int<7> a, ap_int<7> b)
{ return a / b; }
ap_fixed<?,?> addf(ap_fixed<14,7> a,
                   ap_fixed<14,7> b)
{ return a + b; }

/**************************************/
auto mulf(ap_fixed<14,7> a,
          ap_fixed<14,7> b)
{ return a * b; }
```

- Mentor’s solution requires constant redefinition of `oType`.
- Can you figure out the ‘?’s.

- Use `auto`?
- It doesn’t hinder synthesis; `auto` is deduced in the clang frontend, long before HLS.
- ap_int rules
- ap_fixed rules

General style guide: use modern C++



- UG902, UG1399
- xilinx_hls_lib_<release_number>.tgz
- Vitis HLS LLVM open source

What C++14 constructs are valid?

- User guides (UG) are ambiguous; check against the open-sourced HLS LLVM code.
- `#pragma`
- `__attribute__(())`

Supported Data Types

Supported arbitrary precision types - `<ap_int.h>`

1. Integer data types `<ap_int.h>`
    - Signed: `ap_int<W> (W = bitwidth)`
        - range:

$$-2^{W-1} \leq var \leq 2^{W-1} - 1$$

    - Unsigned: `ap_uint<W>`
        - range:

$$0 \leq var \leq 2^{W} - 1$$

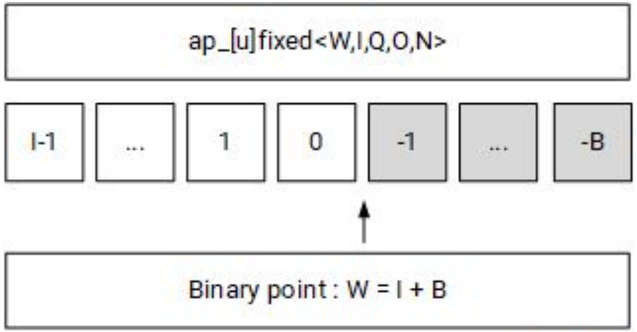
Supported arbitrary precision types - <ap_fixed.h>

2. Fixed-point data type <ap_fixed.h>
   - Signed: ap_fixed<W, I, Q, O, N>
     - range:

$$-0.5 \times 2^{I} \leq var \leq (0.5 - 2^{-W})2^{I}$$

   - Unsigned : ap_ufixed<W, I, Q, O, N>
     - range:

$$0 \leq var \leq (1 - 2^{-W})2^{I}$$



W = bitwidth
I = integral Width
Q = quantization mode
O = overflow mode
N = saturation width

Quantization and Overflow mode

- Only support fixed point data type (ap_fixed).
- `ap_[u]fixed<bitwidth, decimal_place, quant_mode, overflow_mode> variable_name.`

Quantization mode		Overflow mode	
AP_RND	Round to +∞.	AP_SAT	Saturation.
AP_RND_ZERO	Round to 0.	AP_SAT_ZERO	Saturation to 0.
AP_RND_MIN_INF	Round to -∞.	AP_SAT_SYM	Symmetrical saturation.
AP_RND_INF	Round to ∞.	AP_WRAP	Wrap around (default).
AP_RND_CONV	Convergent rounding.	AP_WRAP_SM	Sign magnitude wrap around.
AP_TRN	Truncation to -∞ (default).	※AP_SAT modes can result in higher resource usage as extra logic will be needed to perform saturation.	
AP_TRN_ZERO	Truncation to 0.		

https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/fixe_point_types.html#tam1539734262627

Operations

Operator

- Common C++ operators are supported for ap types.
    - Arithmetic
    - Comparison
    - Output (cout, not for synthesis)

- Bit-wise operation
    - Bit-selection, range-selection
    - Concatenation
    - Reduction

Division and Modulus

- Mentor implements division via Knuth’s algorithm D. If directly synthesized, it will lead to large LUT. Thus, MentorBB suggests that we manually instantiate a separate highly-optimized divider IP, and wires it to where division occurs.
- Xilinx automatically generates divider LogiCORE IPs for synthesis. No need to instantiate dividers separately. The divider LogiCORE IP is based on Radix-2 non-restoring division and High Radix division with prescaling.
- Likewise for modulus.

https://www.xilinx.com/products/intellectual-property/divider.html

Bit-selection operator

The least significant bit has index 0.

- Bit-selection operator: [ ]

```
ap_uint<5> var = 21;
cout << var[2] << “\n”;  // 1
var[2] = 0;
cout << var << “\n”;     // 17
```

- Range-selection operator: (,)

```
ap_uint<5> var = 21;
cout << var(4, 2) << “\n”;  // 5
var(4, 2) = 0;
cout << var << “\n”;        // 1
```

	MSB				LSB
var	1	0	1	0	1
idx	4	3	2	1	0

Shift operator

- Shfit left: `<<` . Shift right: `>>` .
- Unsigned
    - pad zero on both side

```
ap_uint<5> var = 21;           // 21
cout << (var >> 1) << "\n";  // 10
cout << (var << 1) << "\n";  // 10
```

- Signed
    - pad zero when shifting left
    - pad sign bit when shifting right

```
ap_int<5> var = 21;           // -11
cout << (var >> 1) << "\n";  // -6
cout << (var << 1) << "\n";  // 10
```

	MSB				LSB
var	1	0	1	0	1
idx	4	3	2	1	0

Useful Utility/Helper Function

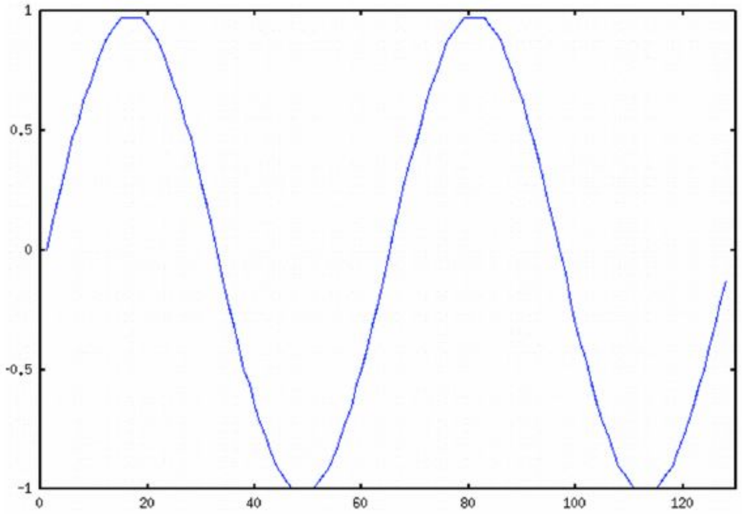
Useful utility/helper functions

`<hls_math.h>`

- Overload math functions in `<cmath>` for `ap_int<>` and `ap_fixed<>`
- Accuracy is worse than that of `<cmath>`
- For simulation, it is a drop-in replacement of `<cmath>`
- For synthesis, it is a collection of optimized IPs
- Restrictions:
    - `ap_fixed<W, I>` where I<=33 and W-I<=32
    - `ap_ufixed<W, I>` where I<=32 and W-I<=32
    - `ap_int<I>` where I<=33
    - `ap_uint<I>` where I<=32

Accuracy Experiment

- Calculate a sine function using cmath and hls_math.



Result

- Comparison

c_math	hls_math	error
+0.0000000000000000	+0.0000000000000000	+0.0000000000000000
+0.0960082560777664	+0.0960082560777664	+0.0000000000000000
+0.1910928487777710	+0.1910928338766098	+0.0000000149011612
+0.2843389511108398	+0.2843389511108398	+0.0000000000000000
+0.3748495280742645	+0.3748494982719421	+0.0000000298023224
+0.4617536962032318	+0.4617536664009094	+0.0000000298023224
+0.5442155003547668	+0.5442154407501221	+0.0000000596046448
+0.6214414238929749	+0.6214413642883301	+0.0000000596046448
+0.6926887035369873	+0.6926886439323425	+0.0000000596046448
+0.7572717070579529	+0.7572716474533081	+0.0000000596046448

Useful utility/helper functions

<hls_vector.h> (Vitis HLS 2020 only)

- SIMD optimized vector of numbers: `hls::vector<T,N>`
  - ```
    hls::vector<int, 1000> a = 10;
    ```
- Compared to that of Mentor:
  - ```
    static int a[1000];                                    ;
    static bool dummy = ac::init_array<100>(a,1000);
    ```
  - dummy is always `true`.

Reference

- Xilinx Arbitrary Precision Data Types Library
  - https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/ogi1585574074269.html
- Vivado Design Suite User Guide: High-Level Synthesis (UG902)
- Divider Generator v5.1 LogiCORE IP Product Guide (PG151)
- Vitis High-Level Synthesis User Guide (UG1399)
- HLS Bluebook, Mentor
- Algorithmic C (AC) Datatypes, Mentor

GitHub

- https://github.com/chenhao1106/Chapter3-BitAccurateDataTypes