Compile Time Computation of Constants for High Level Synthesis

Evan Albright 04-17-2024

Connecting Everyone and Everything, All the Time.

SKYWORKS®

Presentation Overview

- Traditional vs. configurable methods for creating configurable IP blocks in high-level synthesis
- Compile time coding techniques
  - Compile time recursion
  - Template argument deduction
  - `constexpr` constructors
  - Type extraction
  - Tuples
- Stratus™ HLS example: Cascaded integrator comb (CIC) decimator
- Conclusion

Traditional vs. Configurable IP Blocks

- Deriving and extracting data at *compile-time* is very powerful

Traditional

Typically, a script is run to calculate the parameters or coefficients for an IP block

Script

```
>> N = 3, R = 16, M = 1;
>> bitwidths =
ComputeAlgBitWidths(N, R, M)
bitwidths = 23 23 22 21 20 19
```

C, Python, MATLAB®

HLS Module

```
SC_MODULE(dut) {
  sc_uint<1> clk, rst;
  sc_in<DT> data_in;
  sc_int<23> state0;
  sc_int<23> state1;
  sc_int<22> state2;
  sc_int<21> state3;
  …
```

SystemC

Then a HW module is instanced with *fixed* parameters

** This means you have to maintain two codesets

Compile-Time

Using a SystemC style with *compile-time* functions gives you a module and parameter generation all in one

Easy changes. Just edit parameters and rerun HLS!

HLS Module

```
template <int N, int R, int M>
struct Alg {
  using T_tuple = AlgBWTuple<N,R,M>::T_state_tuple;
};
SC_MODULE(dut) {
  sc_uint<1> clk, rst;
  sc_in<DT> data_in;
  Alg<3, 16, 1>::T_tuple states;
```

SystemC

** No script needed!

Templated modules, recursed functions and constant expressions let you extract and derive parameters and even datatypes on the fly!

Compile Time Recursion

- Example: Fibonacci Numbers

A Fibonacci sequence starts with 0 and 1

But after that each number is the sum of the last two

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$$

The pattern of this sequence is very amenable to templated recursion

Compile Time Recursion

- Template Recursion Example

Create a template with a parameter to compute N numbers by recursing through the sequence

```
template <int N>
struct Fibonacci
{
  static constexpr int value = Fibonacci<N - 1>::value
                             + Fibonacci<N - 2>::value;
};

template <>
struct Fibonacci<0>
{
  static constexpr int value = 0;
};

template <>
struct Fibonacci<1>
{
  static constexpr int value = 1;
};
```

Then terminate the recursion with specializations for inputs 0 and 1

When N=2, both **`::values`** come from these specialization constants

Compile Time Recursion

- Constant Expression Recursion Example

`constexpr` functions can be evaluated at compile time *or* run time

```
constexpr int Fibonacci(int N)
{
  if( N==0 )
  {
    return 0;
  }
  else if( N==1 )
  {
    return 1;
  }
  else {
    return Fibonacci(N-2) + Fibonacci(N-1);
  }
};
```

N might not be a compile time constant. N can't be used in any compile time only constructs (e.g., template parameters)

** **`constexpr`** functions are not guaranteed to be evaluated at compile time, unlike template metafunctions!

Template Argument Deduction

Here the template arguments are *deduced* from the function call. Only the *type* of the function parameter matters, not the value, so that the pack “vals” can be deduced and used at compile time

std::integer_sequence defines a compile-time sequence of integers

```
template <typename T, T… vals>
constexpr T ComputeProduct(std::integer_sequence<T, vals…> )
{
  constexpr std::array< T, sizeof…(vals) > tmp = {{vals…}};
  T prod = tmp[0];
  for( int idx = 1; idx < sizeof…(vals); ++idx )
  {
    prod *= tmp[idx];
  }
  return prod;
}
```

sizeof… returns the number of elements in this parameter pack

** This style combines metaprogramming with **constexpr** functions

Constexpr Constructors

- This example computes:
  - $h_i = in\_vec_i^{\,exponent}$

The constructor is constexpr. If `in_vec` is also `constexpr`, the `h[]` array can be used at compile time

This uses `constexpr` struct types

Useful for working with compile time arrays instead of scalars

```
template <typename T, unsigned int N>
struct PowVec
{
  constexpr static unsigned int Size = N;
  T h[Size];
  constexpr PowVec( const T in_vec[Size], int exponent ) : h()
  {
    for( unsigned int idx = 0; idx < Size; ++idx ) {
      h[idx] = ComputePow( in_vec[idx], exponent );
    }
  }
}
```

** Not all math library functions are `constexpr`, but will be eventually

Type Traits

- This trick determines the *type* of a template parameter at compile time

Type is determined when the specialization matches the qualities of that type

So **`is_sc_int`** matches a type that has **`&=`** and **`%=`** operators with **`int`**, a **`length()`** method with no input, and is default constructible

```
namespace pre_17
{
  template <class … >
  using void_t = void;
}

template <typename T, typename = void>
struct is_sc_int : std::false_type
{};

template <typename T>
struct is_sc_int<T, pre_17::void_t<
  std::enable_if_t<std::is_default_constructible<T>::value>,
  decltype(std::declval<T>().operator&=(int{})),
  decltype(std::declval<T>().operator%=(int{})),
  decltype(std::declval<T>().length()) >
> : std::true_type
{};
```

** Use this with **`enable_if`** if you need a function that has different behavior based on data type (e.g., **>>** for fixed point or **/** for floating point)

Type Traits

- Get the template parameters from a type

You can't get a template parameter directly from its variable!

```
template <class T>
struct sc_parameters
{};

template <int WL>
struct sc_parameters< sc_int<WL> >
{
  static constexpr int wl = WL;
};
```

Use this to take a type like **`sc_int<>`** and find out the word length parameter

```
sc_int<5> sc_tmp;
using sc_tmp_param = sc_parameters<decltype(sc_tmp)>;
std::cout << "sc_tmp WL:" << sc_tmp_param::wl << std::endl;
```

This code prints this:

```
sc_tmp WL: 5
```

Type Traits

- Create a brand new type by modifying a template parameter

resize_wordlength<> extracts word length from an **sc_int<>** and creates a new *wider* **sc_int<>** type

```
template <typename T, int wl_addend, typename enable = void>
struct resize_wordlength
{
  using type = T;
};

template <typename T, int wl_addend>
struct resize_wordlength<T, wl_addend, std::enable_if_t<is_sc_int<T>::value> >
{
  using sc_params = sc_parameters<T>;
  using type = std::conditional_t<std::is_same<T, sc_int<sc_params::wl> >::value,
    sc_int<sc_params::wl + wl_addend>,
    sc_uint<sc_params::wl + wl_addend>
  >;
};
```

** Use this to derive a wider type from an existing type to prevent overflow

Checks if it's ufixed or fixed

Again, use template specialization for cases with different types. Default simply returns the input type

This saves you from having to pass parameters and derive types internally!

Tuples

- A group of heterogeneous values defined at compile time

**Use this to iterate over a tuple

```
template <int First, int Last, int Increment>
struct static_for
{
  template <typename Func, int FirstIndex = First, int LastIndex = Last,
            std::enable_if_t<FirstIndex != LastIndex, bool> = true>
  static inline constexpr void LoopBody(Func const &f)
  {
    static_assert((Increment > 0 && LastIndex > FirstIndex) ||
                  (Increment < 0 && FirstIndex > LastIndex));
    f(std::integral_constant<int, First>{} );
    static_for<First + Increment, Last, Increment>::LoopBody(f);
  }

  template <typename Func, int FirstIndex = First, int LastIndex = Last,
            std::enable_if_t<FirstIndex == LastIndex, bool> = true>
  static inline constexpr void LoopBody(Func const &){}
};
```

`LoopBody()` function recurses on itself if indices are different

This `LoopBody()` function is the terminating case and is a no-op

Iterating over tuples example

- Create a lambda function capturing all local variables as references and pass it to **`static_for<>::LoopBody()`**

`index_value` is **`std::integral_contant<>`** type

```
T_tuple_type my_tuple;
static_for<0, std::tuple_size(T_tuple_tupe), 1>::LoopBody([&](auto index_value)
  {
    std::get<index_value.value>(my_tuple) = 0;
  }
);
```

Use **`std::get<I>(tuple)`** to extract values at compile time

Example : Cascaded Integrator Comb (CIC) Decimator

- A CIC decimator has no multipliers, only integrators and differentiators
  - The design parameters are:
    - R = Decimation rate
    - M = The number of delays in each differentiator
    - N = The order of the CIC decimator, or the number of accumulators and differentiators
    - The input bit width
    - The output bit width
  - Designs typically employ bit pruning
    - Gradually reduces bit width through the decimator
    - Keeps internal quantization noise less than the final stage
  - Using these techniques, you can create a CIC Decimator IP block fully implemented by just specifying design parameters

Example : Cascaded Integrator Comb (CIC) Decimator

`ComputeCicTuple<>` template function uses parameter pack expansion to prune each CIC stage

```
template <int… stage_idx>
constexpr auto ComputeCicTuple(double cic_gain, int in_width, int out_width,
                               std::integer_sequence<int, stage_idx…> )
{
   return std::make_tuple(ComputePrunedStageBitwidth<stage_idx, sizeof…(stage_idx)>
                         (cic_gain, in_width, out_width)… );
}
```

Integer sequence created for stage indices like in template argument deduction example

Tuple of CIC state variables created

Tuple widths are different after pruning

o

o

o

```
template <int in_width, int out_width, typename T_cic_params>
struct CicBitWidths
{
   constexpr static double cic_gain = ComputeCicGain(T_cic_params{} );
   constexpr static auto stage_widths_tuple = ComputeCicTuple(cic_gain, in_width, out_width,
                                    std::make_integer_sequence<int, 2*T_cic_params::N>{} );
};
```

`CicBitWidths<>` template helper struct calls **`ComputeCicTuple<>`** and saves result

Example : Cascaded Integrator Comb (CIC) Decimator

ScIntCicStates<> struct calls **CicBitWidths<>** to get tuple stage bitwidths

MakeStateTuple<> function uses pack expansion to make an sc_int tuple with those bitwidths

```
template <int in_width, int out_width, typename T_cic_params>
struct ScIntCicStates
{
  constexpr static auto stage_widths_tuple = CicBitWidths<in_width, out_width, T_cic_params>::stage_widths_tuple;
  constexpr static auto num_stages = std::tuple_size<decltype(stage_widths_tuple)>::value;

  template <int… indices>
  constexpr static auto MakeStateTuple(std::integer_sequence<int, indices… >)
  {
    return std::tuple<sc_dt::sc_int<std::get<indices>(stage_widths_tuple)>… >{};
  }

  using T_cic_states = decltype(MakeStateTuple(std::make_integer_sequence<int, num_stages>{}));
};
```

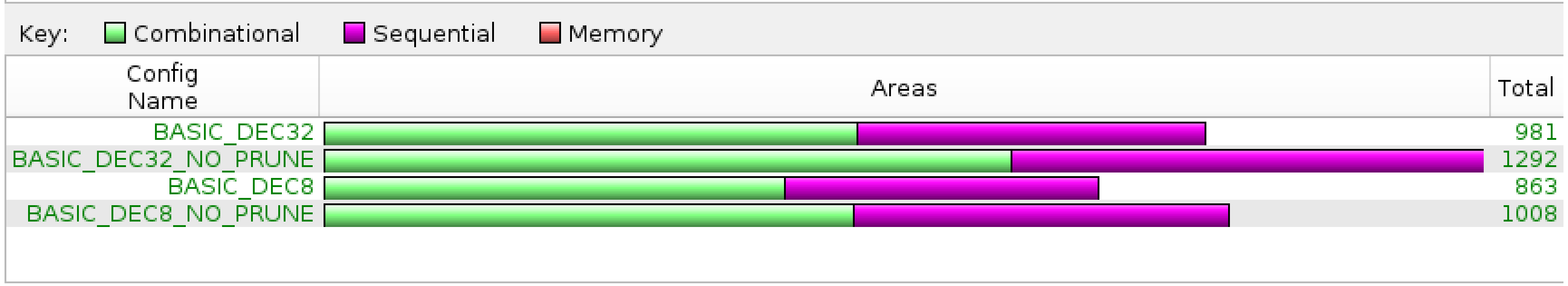
T_cic_states type is used to declare the CIC state variable in the CIC Decimator class

SKYWORKS

Example : Cascaded Integrator Comb (CIC) Decimator

- CIC Stratus HLS project synthesis exploration results for different decimation rates (8, 32) and 12 bit input/output, with and without pruning

	Config Name	Total Area	Comb. Area	Seq. Area	# FFs	Memory Area	Last Run	Total Run Time
N=3,R=32,M=1	BASIC_DEC32	981	593	388	113	0	Wed Apr 10 10:59:08 2024 PDT	0:03:20
	BASIC_DEC32_NO_PRUNE	1292	764	528	154	0	Wed Apr 10 11:02:50 2024 PDT	0:03:50
N=3,R=8,M=1	BASIC_DEC8	863	513	350	102	0	Wed Apr 10 10:51:00 2024 PDT	0:03:26
	BASIC_DEC8_NO_PRUNE	1008	589	418	122	0	Wed Apr 10 10:54:49 2024 PDT	0:03:46



SKYWORKS®

Summary

- It is possible to create SystemC algorithms with non-trivial parameters computed at compile time
- Compile time recursion with templates and constant expressions are basic computational building blocks
- Template argument deduction can simplify compile time function interfaces
- Constant expression constructors allow working with compile time derived arrays and look up tables
- Type traits make templates smarter by inspecting the template parameter types and deriving new types
- Tuples allow grouping different types together, useful for algorithms that need variables with different bit widths
- This all works in HLS, like the CIC decimator built from fundamental design parameters

skyworksinc.com

SKYWORKS®