Using Template Metaprogramming for Hardware Description

Philipp S. Käsgen
Osnabrück University of Applied Sciences
p.kaesgen@hs-osnabrueck.de

Markus Weinhardt
Osnabrück University of Applied Sciences
m.weinhardt@hs-osnabrueck.de

Abstract. When designing digital systems, the Design Space is explored well in advance to rule out as many infeasible design options as possible, as a complete enumeration would be infeasible. Still, in the end, the designer is mostly left with several similar design options, i.e. actual choice of adder, Floating Point Unit etc. In such cases, a direct comparison cannot be avoided - at least in simulation. Even though the designer might have some module descriptions to be examined at hand, it might take a while to put them to work because of differing ports and control signals.

But why should a system designer make the effort to adapt many modules to his design if he is only interested in a few aspects for a quick survey: overall power consumption, area occupation, timing, and correctness of results? This is the reason why we need a hardware description solely based on parameters. Such a description requires Metaprogramming Techniques because said parameters need to be translated to hardware behaviours. Hence, in this work we propose a parameter based hardware design paradigm based on SystemC and C++: Hardware Metadescription.

1. Introduction

More and more, metaprogramming techniques, e.g. *Template Metaprogramming (TMP)* in C++[1], establish themselves as great assets in Software development since they provide a more generic nature and better maintainability than "traditional" code. Furthermore, the compiled programs tend to run faster, since the compiler can utilize more sophisticated optimizations [1]. SystemC [5] as a C++ library can obviously exploit TMP. In doing so, the TMP advantages can be transferred to hardware descriptions. This results in better reusability of the modules and a quicker Design Space Exploration (DSE). In this work in progress[2], we research how to describe hardware modules with TMP in SystemC. We will demonstrate the design methodology of an Execution Unit and focus on the TMP techniques which are necessary to abstract the module descriptions. In the remainder of the document this description is called *Hardware Metadescription (HMD)*.

We will proceed as follows: In Section 2, we give an overview over the related work which deals with metaprogramming in the context of hardware design. Then, in Section 3, we suggest parameters to model hardware components and give a few examples. After this, we give examples on how Metaprogramming can be used with SystemC in Section 4. In Section 5, we present a

[1]ISO/IEC 14882:2017

[2]funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 283321772

few basic concepts of Template Metaprogramming which are frequently needed. And finally, we outline our future work in Section 6.

2. Related Work

The underlying idea to abstract hardware modeling in conjunction with metaprogramming techniques is already well known. Because Verilog and VHDL have never featured inherent metaprogramming, heterogeneous metaprogramming is mostly used, i.e. metalanguage and target language differ. In contrast, our approach employs homogeneous metaprogramming and only aims at simulation. Since C++11 and C++14 have been released, many new features were introduced which were adapted in part by SystemC [6].

In [4], a systematic design flow from UML to the actual hardware description is presented. Their work is broadly applicable due to the UML front-end and draws parallels to software design patterns but aims at actual hardware specification. *PragScript* serves as their metalanguage. As proof of concept, they implement a triple-redundant system using UML class diagrams. In the terms of Damasevicius et al. [4], our proposed description is a *Metaspecification*.

A more abstract approach is shown in [8]: They propose a basic Hardware library similar to the *C++ Standard Template Library (STL)* which handles different hardware implementations in a general way through metaprogramming. By this, well-known software design patterns are intended to become available for hardware design. An Iterator equivalent serves as an example. In the evaluation section, they find that pattern-based hardware implementations affect the structure of the model but not necessarily induce additional logic. Unfortunately, they do not reveal their metalanguage.

Zhou et al. use Perl as the metalanguage in [9]. They synthesize a pattern-oriented generator for a Viterbi decoder on an Altera Stratix II FPGA and compare it with Altera's corresponding IP core. On the one hand, their logic utilization is higher and clock frequency is lower, but on the other hand the power consumption and RAM occupation are lower. The evaluation of the lines of code shows that the pattern-oriented generator needed slightly less lines of code while offering more flexibility than the hard-coded one.

3. Metamodeling of Hardware Components

Metamodeling of hardware means to describe hardware only in terms of a few parameters which entirely define the behaviour of a hardware module. For demonstration purposes, we will illustrate the HMD concept by the example of an Execution Unit. Of course, Control Units can be implemented with this methodology, too. Yet, an Execution Unit is a bit more challenging as we also show how to assign the functionality with a Template Parameter in Section 4 which turns out to be non-trivial.

In general, arithmetic units take certain data, process them for a certain amount of time, and finally output a result. We can identify the following parameters:

- *Latency*: Clock Cycles which need to pass between issuing an operation and getting the result
- *Initiation Interval (II)*: Clock Cycles which need to pass between issuing two consecutive operations

- *Number Representation*: Can be Floating Point, Integer, etc.
- *Number of Operands*: For supporting unary, binary and ternary operators
- *Operand Width*: in bits
- *Result Width*: in bits
- *Operators*: A set of operators which are supported by the module
- *Area*: Occupied area on the target technology
- *Power Consumption*: Estimated Power Consumption on the target technology

The last two parameters do not contritbute to the functionality and, hence, are not required for a simulation but become interesting when investigating further trade-offs between design choices. They are optional as this would ultimately require to have the data for all target technologies. A set of data for one target technolgy might be translated to another using the estimation approach in [3] if necessary. Yet, they are left out for the models in the next paragraph.

For instance, a simple Arithmetic Logic Unit (ALU) can be modeled with

$$ALU = \{\ Latency = 0,\ II = 1,\ Number\ Representation = Integer,\ Number\ of\ Operands = 2,\ Operand\ Width = 32,\ Result\ Width = 32,\ Operators = \{\ +, -, <, \neg, \cup, \cap\ \}\ \}\ .$$

The resulting hardware module can be seen as a black box which provides the ports and behaviour implied by the parameters. Assuming we would like to have a model of a Floating Point Divider we could choose either a blocking one ({ Latency=24, II=25, Number Representation=Float, Number of Operands=2, Operand Width=32, Result Width=32, Operators={ / } } or a pipelined one ({ Latency=24, II=1, Number Representation=Float, Number of Operands=2, Operators={ / } } just by changing II. A simple shift operation which is only cheap wiring could be modeled with { Latency=0, II=0, Number Representation=Integer, Number of Operands=1, Operand Width=32, Result Width=32, Operators={ *Leftshift* } }. HMD modules cover multiple different traditionally described modules with one description, while being less verbose and, hence, less error-prone.

4. Hardware Metadescription

In Section 3, we identified a set of parameters which describe the behaviour of an arbitrary Execution Unit. When we want to simulate the model, we have to derive a working SystemC module from these parameters. First, we need a template class:

```
template <size_t LATENCY, size_t II, typename SCALAR, size_t OPERAND_NUMBER,
  typename... OPERATORS>
class fu_module : public sc_module{ /* Ports, Processes, ... */};
```

where `LATENCY` denotes latency, `II` denotes Initiaion Interval, `SCALAR` denotes the Number Representation, and `OPERAND_NUMBER` is the number of operands. The set of operators is given as a Variadic Template named `OPERATORS...`. In this case, the Variadic Template is supposed to comprise function objects such as `std::plus<int>` or even new ones:

```
#include <cmath>
template <typename T> struct fma_t{       //Fused-Multiply-Add Function Object
  constexpr T operator()(const T &arg1, const T &arg2, const T &arg3) const {
    return std::fma(arg1,arg2,arg3);
  }
};
```

For clarification, for the way the module is defined here, the `OPERATORS` are required to be of the same kind, i.e. latency etc. are the same for each operator in an instance of `fu_module`. In case a designer desires to implement two functions with differing parameters, he will have to instantiate two distinct units.

In summary, all parameters from Section 3 are provided (the width of the operands and result are omitted for now) and a hardware designer could, for instance, specialize this module to recreate the behaviour of an existing module. The realization of such a generic module is explained in the remainder of this Section. Essentially, we have to derive an architecture of an HMD module from said parameters. The presented challenges might not be easy to recognize as such but a detailed explanation is out of the scope of this document. Thus, we will focus on the proposals for overcoming them. For our implementation, we wrote C++/SystemC 2.3.1 code compiled by the GNU Compiler Chain 6.3.0.

4.1. Floating Point Operations in SystemC

Since SystemC 2.3.1 does not provide a floating point data type, we came up with the idea of a simple `float_cast` union which consists of one field that can be either interpreted as an `int` or `float` data type.

```
typedef union{ float f; int i; } float_cast;
```

Then the `float` value can be "encoded" as an `int` value, transmitted via the SystemC signals and "decoded" at the desired function unit which harnesses the simulation platform's FPUs.

Regarding the special treatment of `float` values, naively, we could try to implement a member function which handles the input data only depending on the class template `SCALAR`. Unfortunately, this would require partial template specialization of this member function which is not supported by C++. The only option left would be to partially specialize the class for each case and implement the according member function multiple times. But then the same problem arises later when handling unary, binary, and ternary operators which would lead to an enormous overhead due to the different combinations of template specializations.

C++11 introduced the concept of *Substitution Failure Is Not An Error (SFINAE)* which can be exploited to achieve a similar behaviour as partial specialization of member functions: By using the `std::enable_if<>` statement, different implementations of the same function can be written. As long as the premisses of each `std::enable_if<>` statement mutually exclude each other, the actually compiled function is unambiguous, for instance, using the following description for the otherwise identical member function signature:

```
template <typename T=SCALAR>
typename std::enable_if_t<std::is_integral<T>::value,void> process_data();

template <typename T=SCALAR>
typename std::enable_if_t<!std::is_integral<T>::value,void> process_data();
```

Note the negation in the second premiss.

4.2. Pseudo-Dynamic Dispatch of templated functions

Another challenge is the *dynamic* selection of a template parameter out of a static parameter set because, technically, they need to be *statically* indexed.

The workaround to this is a switch statement. The switch variable is the dynamic part and the case statement contains the static indexing part. But being required to write one case statement for each case lacks of generality, especially since the number of case statements has to be adapted to the size of the parameter set. To introduce more generality, the preprocessor in conjunction with the Boost Library[3] is employed. Firstly, the adaption to the different parameter set sizes can be achieved again with SFINAE. Secondly, for each possible parameter set size a specialized function is created which, thirdly, unrolls the according case statements. In the end, two nested `BOOST_PP_REPEATS` unroll the possible implementations which are later enabled or disabled by SFINAE. This procedure guarantees easily maintainable code. In the following an example is given for the dynamic selection of an operator:

```
#define MAX_SUPPORTED_NUMBER_OF_OPERATORS 10
...

std::tuple<OPERATORS...> tuple_operators;
int datum1, datum2, result;
...

template<size_t index>
void process_data(){
  auto private_operator = std::get<index>(tuple_operators);
  result = private_operator(datum1, datum2);
}

#define OPERATOR_SELECT_CASE(rep,n,_) case n: \
process_data<n>(); \
break;

#define OPERATOR_SELECT(rep,n,_) template <unsigned MAX=sizeof...(OPERATORS)>\
typename std::enable_if_t<(MAX==n),void> operator_select(size_t i){ \
  switch(i){ \
  BOOST_PP_REPEAT(n,OPERATOR_SELECT_CASE,_) \
  } \
}

BOOST_PP_REPEAT(MAX_SUPPORTED_NUMBER_OF_OPERATORS, OPERATOR_SELECT,_)
```

The `MAX_SUPPORTED_NUMBER_OF_OPERATORS` parameter is the only restriction to this approach. But a designer can simply increase it if he needs to provide more operators in a module. In the end, for processing `datum1` and `datum2` into `result` with the first operator in the variadic template `OPERATORS...`, simply the method `operator_select(0)` has to be called, for instance.

[3] http://www.boost.org/

4.3. Templated modeling

For our implementation, we define an Operator Model Pattern with the previously mentioned parameters:

```
template <typename SCALAR, size_t LATENCY, size_t INITIATION_INTERVAL,
size_t NUMBER_OF_OPERANDS, typename FUNCTOR>
struct operator_model;
```

An actual model could look like this:

```
using adder_f_t = operator_model<float, 3, 1, 2,typename std::plus<float>>;
```

All Models have to be managed by a central structure which we call Operator Model Container:

```
template <typename SCALAR, size_t LATENCY, size_t INITIATION_INTERVAL,
size_t NUMBER_OF_OPERANDS,  typename FUNCTOR, typename... REST>
struct operator_model_container<operator_model<SCALAR, LATENCY,
INITIATION_INTERVAL, NUMBER_OF_OPERANDS, FUNCTOR>, REST...>
{...};
```

An example could look like this:

```
using all_models = operator_model_container<adder_f_t/*...*/>;
```

This container provides functions to look up parameters of each contained operator model. For instance:

```
size_t l = all_models::get_latency<adder_f_t>::value;
```

These models may be used within an Execution Unit of a microprocessor or in the Processing Elements of a Reconfigurable Processor. Next, we wish to assign an Opcode to such an operator model. This can be achieved with the following:

```
#define ADD_FP 5 //Example Opcode
using opcode2model = static_map< static_map_entry<ADD_FP, adder_f_t>/*...*/>;
```

Now, we can configure an Execution Stage by simply passing a set of Opcodes as Template Parameters. The Execution Unit has to take care of the instantiation of the `fu_module` instances and the wiring. The behaviour is provided by the models. In particular, for emulating latency, the `fu_module` must privately generate an accordingly deep pipeline for delaying the results of the function object calls. In addition, it has to abide by the Initiation Interval. Ideally, the designer does not have to care about these implementational details, since they solely rely on the Operator Models. His responsibility is just assigning the Operator Models to Opcodes, and then configuring the Execution Units with said Opcodes.

5. Template Metaprogramming

As we have seen in Section 4, it is necessary to look up parameters or types by other parameters or types. In the following, we give minimal examples on how these look-ups can be achieved during compile-time. In particular, we will describe four static maps which map a number, i.e. a non-type, to a type, a type to a non-type, a type to a type, and a non-type to a non-type. First we need structures which assign the respective key to a value, i.e. static map entries:

```
//Entry mapping a non-type KEY to a type VALUE
template<size_t KEY, typename VALUE> struct map_entry_nontype_to_type;

//Entry mapping a type KEY to a non-type VALUE
template<typename KEY, size_t VALUE> struct map_entry_type_to_nontype;

//Entry mapping a type KEY to another type VALUE
template<typename KEY, typename VALUE> struct map_entry_type_to_type;

//Entry mapping a non-type KEY to another non-type VALUE
template<size_t KEY, size_t VALUE> struct map_entry_nontype_to_nontype;
```

Then we define a general case for each kind of static map using a *Variadic Template Parameter* REST:

```
template<typename... REST> struct map_nontype_to_type;

template<typename... REST> struct map_type_to_nontype;

template<typename... REST> struct map_type_to_type;

template<typename... REST> struct map_nontype_to_nontype;
```

Variadic Templates can represent an arbitrary number of possibly different types. This is useful for the intended generic character of our static maps.

For the actual maps we partially specialize the respective general definitions which handle the static map entries. For the actual look-up we need a `struct` that behaves like a "getter function". Putting these pieces together so far, this might look like this:

```
template <size_t KEY, typename VALUE, typename... REST>
struct map_nontype_to_type< map_entry_nontype_to_type< KEY, VALUE>, REST... >{
  //"getter"-struct which looks up a value by a key
  /*...*/
};
```

We are also required to cover the case that a key is not contained in the map:

```
//default implementation
template <typename...>
struct map_nontype_to_type{ /*...*/ };
```

When we want to set up and use such a static map, e.g.

```
using static_map = map_nontype_to_type</*(map entries)...,*/
                     map_entry_nontype_to_type<3,int>/*, (more entries)...*/
                   >;
```

and wish to access a map entry in the program, we just have to write:

```
//...
static_map::get<3>::type a = 5;
//...
```

which in this case is equivalent to writing:

```
int a = 5;
```

Now that we have defined the scope of the examples, let us investigate on how the look-ups can be done. Depending on the nature of the key and the value, i.e. non-type or type, slightly different concepts have to be used.

5.1. Look-up of a type by a non-type

For this look-up, we recursively check whether a given argument ARG matches with a KEY by using the std::conditional<> statement. Basically, it behaves like the ternary operator (?:) but only for types: If the value of KEY is equal to the value of ARG, the type VALUE will be "returned". Otherwise, the look-up is continued with the remaining template parameter REST which implicitly comprises the map entries not yet considered.

```
template<size_t KEY, typename VALUE, typename... REST>
struct map_nontype_to_type< map_entry_nontype_to_type<KEY, VALUE>, REST... >{
  //"getter"-struct which looks up a value by a key
  template<size_t ARG> struct get{
    using type = typename std::conditional<KEY==ARG,     //Premise
      VALUE,                                             //if premise is true
      typename map_nontype_to_type<REST...>::            //if premise is false
        template get<ARG>::type>::type;
  };
};
```

If the above structure fails to find a matching key, void will be returned as a type.

```
//default implementation
template<typename...>
struct map_nontype_to_type{
  template<size_t> struct get{ using type = void; };
};
```

5.2. Look-up of a non-type by a type

Given a type, the strategy looks similar to the previous one. The std::is_same<> compares the types of its arguments. If the condition is satisfied, VALUE will be "returned". Otherwise the search continues. As a default value, -1 is chosen.

```
template<typename KEY, size_t VALUE, typename... REST>
struct map_type_to_nontype<map_entry_type_to_nontype<KEY,VALUE>, REST...>{
  template<typename ARG> struct get{
    static const size_t value = (std::is_same<ARG,KEY>::value) ?
      VALUE :
      map_type_to_nontype<REST...>::template get<ARG>::value;
  };
};

template<typename...>
struct map_type_to_nontype{
  template<typename> struct get{ static const size_t value = -1; };
};
```

5.3. Look-up of a type by a type

Combining the previously introduced concepts std::conditional<> and std::is_same<>, we can look up a type by another type like this:

```
template<typename KEY, typename VALUE, typename... REST>
struct map_type_to_type<map_entry_type_to_type<KEY,VALUE>, REST...>{
  template<typename ARG> struct get{
    using type = typename std::conditional<std::is_same<ARG,KEY>::value,
      VALUE,
      typename map_type_to_type<REST...>::template get<ARG>::type>::type;
  };
};
```

As default, `void` will be returned.

5.4. Look-up of a non-type by a non-type

This last example looks up a non-type by another non-type. This is straight-forward:

```
template<size_t KEY, size_t VALUE, typename... REST>
struct map_nontype_to_nontype<map_entry_nontype_to_nontype<KEY,VALUE>,REST...>
{
  template<size_t ARG> struct get{
    static const size_t value = (ARG==KEY) ? VALUE :
      map_nontype_to_nontype<REST...>::template get<ARG>::value;
  };
};
```

Again, -1 is used as default.

6. Conclusion and Future Work

Summarizing, we introduced the HMD design methodology on the basis of an Execution Unit in Section 3 and Section 4 and gave examples of useful TMP concepts in Section 5. We refrained from comparing a hard-coded hardware module with a metadescribed one because in the example in Section 3, the arbitrary number of function objects allows to cover as many hard-coded implementations as wanted at once. The potential of the HMD proposal is obvious: By changing a few parameters in the example, different hardware behaviours can be realized. Thus, the design space can be rapidly explored and design decisions can be swiftly reconsidered without further changes of the hardware description, even at a later stage of the development process while always having a rather streamlined behavioural model due to TMP and the associated compiler optimizations at all times.

In the future, we will use our approach in conjunction with the System Simulator *gem5* [2]. We expect to make the choices of the arithmetic units in our accelerator depending on the implementation of the memory interface for an optimal balance between processing speed and memory bandwith. Using [7], gem5 and SystemC can be co-simulated. This enables more opportunities for the fine-tuning of co-processor and memory interface implementations.

One point which has not been covered by our work but begs further investigation is the compatibility with the *SystemC Synthesis Subset*. Metaprogramming has static properties which predestine it for hardware description. This could be the angle from which synthesis of metadescribed hardware could be approached, if the metainformation are used for an actual implementation and not only for the behavioural description as in our `fu_model` example.

References

[1] Abrahams, David and Aleksey Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] Binkert, Nathan et al.: *The gem5 simulator*. ACM SIGARCH Computer Architecture News, 39(2):1, 2011.

[3] Borkar, Shekhar: *Design challenges of technology scaling*. IEEE Micro, 19(4):23–29, 1999.

[4] Damasevicius, Robertas and Vytautas Stuikys: *Application of UML for hardware design based on design process model*. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004*, pages 244–249, Yokohama, Japan, 2004. IEEE.

[5] Design Automation, Standards Committee: *IEEE Std 1666-2011, IEEE Standard for Standard SystemC® Language Reference Manual*, volume 2011. 2012.

[6] Görgen, Ralph, Philipp A. Hartmann, and Wolfgang Nebel: *Automated SystemC Model Instantiation with Modern C++ Features and sc_vector*. In *DVCon Europe*, Munich, Germany, 2015.

[7] Menard, Christian, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn: *System Simulation with gem5 and SystemC*. In *IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*, 2017.

[8] Rincón, Fernando et al.: *Model reuse through hardware design patterns*. In *Proceedings Design, Automation and Test in Europe, DATE '05*, volume I, pages 324–329, 2005.

[9] Zhou, Meng, Minglun Gao, and Xiao Song Huo: *Design method for parameterized ip generator using structural and creational design patterns*. In *2nd International Conference on Anti-counterfeiting, Security and Identification, ASID 2008*, pages 378–381, 2008.