C++HDL Specification

Mike Reznikov

2025-09-10

This document is currently in **draft** status.
Content may change significantly before final approval.

Contents

Introduction

C++HDL is an C++ Hardware Definition Language extension for digital Integrated Circuits development, designed for two purposes:

1. Building a full cycle of digital RTL development using C++ language
2. Allowing extra fast simulation of an blocking Verilog-style written RTL

In all operations C++HDL works as a reflection of SystemVerilog model, which means that, on all stages, 100% register-to-register copy of any C++HDL code exists in SystemVerilog domain. This allows C++HDL to SystemVerilog conversion for the purposes of

- Connection of C++HDL developer teams to classical verification and testing teams
- Final RTL delivery to fabrication processes and tools or third-party companies

The main idea of using C++ in RTL development is in replacing **simulation** with **execution** to provide ~20 times faster compilation (according to author's tests) and running of a model during code writing. The following properties of C++ language provide strong foundation for RTL development process:

- C++ provides ability to use any of plenty of IDEs/tools for development and debugging, including a lot of professional tools, supporting all necessary stages for huge projects management
- C++ is a one of the most popular and powerful programming languages in the World, having extra wide community and potentially all specific features of all compilers/tool described in minute details
- C++HDL is shifting hardware development to software development (at least as one of the stages of a process) which makes a lot of software developers potentially accessible on the labor market
- C++ is extra fast. There is no another language with same abilities which can run RTL faster

RTL modelling using C++HDL should be made together with verification and testing, using all power and speed of C++ language in modelling of digital signalling and digital systems interaction. It is also possible to involve third-party SystemVerilog code in C++HDL development project using Verilator tool. Therefore, C++HDL is intended to make Verification process much more simple, flexible, fast, and only containing programming tasks, avoiding complex and expensive proprietary simulators and allowing running of up to millions tests each day.

Same C++ testing and verification software then can be used to prove functionality in SV domain (despite it's 20-100 times slower than C++HDL verification, it can be run once a week or a month just to prove SV generated files are still fully qualified). Generated SystemVerilog files can be taken at any moment and used as the main source code for next stage of ASIC/FPGA/other production process. Conversion tool provides creation of readable and structured SystemVerilog files containing all source comments and data structures.

Since C++HDL goal is to precisely repeat behaviour of SystemVerilog in C++, it is recommended to understand SystemVerilog code and it's behaviour corresponding to each C++HDL line of code (each C++HDL line of code is translated directly into SystemVerilog line of code).

Limitations

- C++HDL supports only digital design components, written using blocking assignments
- Currently no multiclock or CDC is supported. Each clock domain should be developed separately
- Only clock-synchronous logic is supported currently

C++HDL is intended to bring ease and speed in development of various digital circuits like controllers, multiplexors, cache and memory functions, mathematics functions, digital data processing, transmitting circuits, etc.

Requirements

C++HDL is being delivered in two parts:

- C++ headers which contain some classes for C++HDL datatypes
- Conversion/checking tool which provides C++HDL to SystemVerilog conversion and syntax checking

C++HDL is actively using std::format during debug printing so GCC 14 or CLANG 18 is recommended, while it's still possible to achieve execution using older compilators.

Since C++HDL works as a reflection of SystemVerilog RTL model, strong understanding of SystemVerilog modelling techniques is required, in particular:

- Synchronous logic digital circuits
- Blocking and non-bloching assignments
- Combinational logic digital circuits
- Structures, packages, packed and non-packed arrays

C++HDL syntax

C++HDL source file can be written as .h C++ header or .cpp object file. Each header file should describe a module or auxiliary information like datatypes, constants, inline functions definitions, etc.

Auxiliary constructions

C++HDL provides ability of using of any C++ data structures and definition or own data types in order to build comfortable and harmonious ecosystem for the development of your project. All introduced types and constants will be translated into SystemVerilog datatypes during conversion process. (usage of enums and packages is under development)

An example is provided to show appropriate usage of C++ defines and structures in C++HDL.

```
#pragma once

#include "cpphdl.h"
#include "PrjConfig.h"

using namespace cpphdl;

#define    CMD_IDLE    0
#define    CMD_RESET   1
#define    CMD_CONFIG  2

struct CmdConfig
{
    uint8_t cmd_id:4;
    uint8_t cmd_id_rsv:4;
    uint8_t units:6;
    uint8_t flags:2;
    uint16_t address;

#ifdef USE_FORMAT_H
    std::string format()
    {
        return std::format("cmd: {}, units: {}, address: {}", (uint8_t)cmd_id,
         ↪  (uint8_t)units, address);
    }
#endif
}__PACKED;
static_assert (sizeof(CmdConfig) == 4, "struct CmdConfig size is not correct");
```

Module description format

Module definition is represented by C++ class and strictly splitted into 5 main sections and 1 optional:

1. Private: nested instances members list
2. Public: I/O ports definition and *connect()* function body
3. Private: full list of variables: registers, combinational values and all temporary variables
4. Public: combinational functions bodies
5. Public: *reset()*, *work()*, *strobe()* and *comb()* functions bodies (same order is strongly recommended)
6. Public: std::string name variable to handle instance name during hierarchy inferring

Simple FIFO RTL model description using C++HDL is provided as an example:

```
#pragma once

#include "cpphdl.h"
#include "PrjConfig.h"
#include "Memory.h"

using namespace cpphdl;

template<size_t FIFO_WIDTH_BYTES=BUS_WIDTH, size_t FIFO_DEPTH=FIFO_ROWS>
class Fifo
{
    Memory<FIFO_WIDTH_BYTES,FIFO_DEPTH> mem;
public:
    logic<FIFO_WIDTH_BYTES*8>* data_in   = nullptr;
    bool*                      write_in  = nullptr;
    logic<FIFO_WIDTH_BYTES*8>* data_out  = mem.data_out;
    bool*                      read_in   = nullptr;
    bool*                      empty_out = &empty_comb;
    bool*                      full_out  = &full_comb;
    bool*                      clear_in  = &ZERO;
    bool*                      afull_out = &afull_reg;

    void connect()
    {
        mem.data_in = data_in;
        mem.write_in = write_in;
        mem.read_in = read_in;
        mem.write_addr_in = &wp_reg;
        mem.read_addr_in = &rp_reg;
        mem.connect();
    }
private:

    u1 full_comb;
    u1 empty_comb;

    reg<u<clog2(FIFO_DEPTH)>> wp_reg;
    reg<u<clog2(FIFO_DEPTH)>> rp_reg;
    reg<u1> full_reg;

    reg<u1> afull_reg;

    size_t i;
public:

    bool full_comb_func()
    {
        full_comb = (wp_reg.next == rp_reg.next) && full_reg.next;
        return full_comb;
    }

    bool empty_comb_func()
    {
        empty_comb = (wp_reg.next == rp_reg.next) && !full_reg.next;
        return empty_comb;
    }

    void reset()
    {
        wp_reg.clr();
        rp_reg.clr();
        full_reg.clr();
        afull_reg.clr();

        mem.reset();
    }

    void work(int clk)
    {
        if (!clk) return;

        if (*read_in) {

            if (empty_comb_func()) {
                printf("%s: reading from an empty fifo\n", name.c_str());
                exit(1);
            }
            if (!empty_comb_func()) {
                rp_reg.next = rp_reg + 1;
            }
            if (!*write_in) {
                full_reg.next = 0;
            }
        }

        // rp_reg.next could be changed, lowering full_comb down

        if (*write_in) {

            if (full_comb_func()) {
                printf("%s: writing to a full fifo\n", name.c_str());
                exit(1);
            }
            if (!full_comb_func()) {
                wp_reg.next = wp_reg + 1;
            }
            if (wp_reg.next == rp_reg.next) {
                full_reg.next = 1;
            }
        }

        if (*clear_in) {
            wp_reg.next = 0;
            rp_reg.next = 0;
            full_reg.next = 0;
        }

        afull_reg.next = full_reg  (wp_reg >= rp_reg ? wp_reg - rp_reg : FIFO_DEPTH -
         ↪  rp_reg + wp_reg) >= FIFO_DEPTH/2;

        mem.work(clk);
    }

    void strobe()
    {
        wp_reg.strobe();
        rp_reg.strobe();
        full_reg.strobe();
        afull_reg.strobe();

        mem.strobe();
    }

    void comb()
    {
        mem.comb();
        mem.data_out_comb_func();
    }

    std::string name;
};
```

- Module class definition can use template parameters. (currently supported only default values)
- Only *work()* function body is recommended to be offloaded to .cpp object file, while all other bodies should be filled in .h file.
- Only positive edge clocking is supported now, but supporting of both edges requires minor changes.

- It is allowed to use compatible native C++ types like bool, unsigned, unsigned long, etc in all places except reg<>
- Each of *connect()*, *reset()*, *work()*, *strobe()* and *comb()* functions should call corresponding functions of module's nested instances
- Only reg_name.next value should be changed in *work()* function. Both reg and reg.next values can be used on right side of expressions
- In *reset()* function methods only *.clr()* and *.set(val)* are used to set up both current and next register's values
- *[f]printf()* functions are converted to *$write()* during SV conversion, *exit()* becomes *$finish()*, DEBUG_NAME macroses are converted to *`DEBUG_NAME* and *{}* are replaced with *%x* in debug format string

Input/output ports (bidir?)

Input and output ports are always pointers in C++HDL. It allows instant value update on change. For combinational variables pointer to value is used, but value change requires *name_comb_func()* call. This question will be discussed in corresponding chapter.

The strict naming convention is used for input/output ports:

- property_in or obj_property_in - generic input port name
- property_out or obj_property_out - generic output port name

It is strongly recommended to initialize all ports pointers right away in class description. All output ports should take address of particular variables of the module or it's nested instances. All input ports should be assigned *nullptr* value to make uninitialization visible.

Since pointers are used to connect ports, any pair of connected ports must have similar variable types or variable sizes in bytes.

Any port size is multiple of 8 bit. There is no way to use less size of a variable in C++. Size reduction happens after SystemVerilog conversion and uses one of the following ways:

- Standalone types of `<8bit` size (like reg`<u1>`) are translated to corresponding SystemVerilog types (like logic[0:0])
- Structs should be packed and can have integral-type fields of any bit size (not more than maximum possible size in C++)
- Composite types (structs with non-integral types) always align their subtypes size to 8 bit. Unused bits should be removed after SV generation, during optimization.

NOTE! If you build a complex bus joint point between C++HDL/SV module with third-party SV module, you should use packed *structs* to achieve proper `<8bit` fields placing.

Clock (and reset)

The only one clock is used currently, always named clk. Each *work()* function currently checks that only positive edge condition is called. (more abbilities for custom clock and reset should be delivered later)

Variables list

Variables are module class members and can be of 3 types:

- **registers**
- **combinational values**
- **temporary variables**

Registers are of type reg`<TYPE>` and always contain value, updated on last clock edge. To access next value of register *reg_name.next* property is used. It is recommended to give register variable names a *reg* suffix in case when register is used as output port or in parent modules.

Combinational variable can be of any type.

In cases when you need temporary variables inside *work()* or another function they should be declared in variables list as class member.

Connect function

Function *connect()* is used to assign nested instance's inputs to data sources in the module.

Work function

Work function can make any changes to variables. Definition of stack variables is prohibited (same as in Verilog). Only *.next* value of registers should be changed. Work function can call other functions to make code well-structured. Function with return value becomes function SystemVerilog, function with void return becomes task in Verilog. It is mandatory to give all changable member values by reference to a void function. It is prohibited to change member values in non-void functions.

Strobe function

Strobe function should contain all registers of the module calling, .strobe() functions for them. Also strobe() should be called for each of nested instances of the class. The full list of registers and nested instances can be also generated and checked with C++HDL conversion tool.

Comb function

Combinational functions declaration is the most complicated part of the C++HDL-driven development process (as well as SystemVerilog-driven development process) because of lots of ambiguity in behaviour of generic combinational logic circuits. Combinational functions are used implicitly in any mathematical or logical expression in blocking or non-blocking Verilog code line. Their evaluation happens directly during the line execution in simulation, and after synthesis RTL should repeat same well-defined behaviour. At the same time, combinational functions can be defined stanalone and being connected to each other. This possibility makes a variety of complex logic circuits achievable, including loops and oscillators. Since SystemVerilog RTL development process has a number of rules to avoid loops and overcomplicated combinational circuits (google it), C++HDL inherits same rule set. With this, C++HDL uses only blocking assignments as default coding technique (as well as other programming languages).

Since SystemVerilog simulation (and synthesis) refreshes all combinational function values after each line of blocking assignments, to achieve same behaviour, all C++HDL combinational functions should operate register's *.next* values and be called each time when they are used inside an owner module. Third module function insertion in the middle of module's combinational functions chain is prohibited.

From the fact that C++HDL builds a reflection of SystemVerilog RTL model, the obvious conclusion follows, that C++HDL similarly allows forming of loops and oscillators, using standalone combinational functions definition. With this, C++HDL simulation is only capable of runninng a limited number of combinational functions from input A to output B (not backwards), making loops and oscillators impossible.

Basically, in C++HDL each standalone combinational function should be represented by a C++ method with a *comb_func()* suffix and an output variable with a *comb* suffix, always accesible by an address. All module's combinational functions should be executed by a main *comb()* function, which is always called after full system strobing or reset. All nested instance's *comb()* functions should be called at the same place.

More combinational functions complexity follows from the two additional circumstances:

1. Often the only one specific order of their execution provides right calculation of an output during simulation (bacause of cross-dependencies)
2. Several combinational functions from different modules can be connected together and make combinational circuit to be intermodular

This two possible complexities are handled in C++HDL with the following two rules of combinational logic development:

1. At the beginning of design iteration, developer is responsible for minimization of combinational logic complexity, especially cross modules borders
2. After code is designed, C++HDL SV conversion tool builds combinational function's dependencies tree, checks the order of functions calling and gives the feedback on how to fix the problems found.

In conclusion of the most important chapter of this specification, the note should be made, that everything said above is also a permament headache of the most of all RTL developers. Such signalling techniques as “valid/ready” require a lot of intermodule combinational signalling, which lead to a necessity of combinational function definition in SystemVerilog, even if the author applied a lot of effords into avoiding of it. Therefore, C++HDL role here is to compensate complexities of combinational functions designing process with automatic tools possibilities, which help in building of a correct calculation order and avoiding loops.

Data types

To repeat SystemVerilog behaviour C++HDL implements a number of basic data types, responding to a specific SystemVerilog datatype each. Currently the list of C++HDL datatypes includes:

- *logic*`<WIDTH>` - any width variable, optimized for bit-access
- *u*`<WIDTH>`, u1, u8, u16, u32, u64 - unsigned variables
- *s*`<WIDTH>`, s1, s8, s16, s32, s64 - signed variables (reserved but not implemented because of lack of demand and examples)
- *reg*`<TYPE>` - register definition, works only with C++HDL types or any data structures
- *array*`<TYPE,SIZE>` - variable, optimized for large arrays and access and changing by elements
- *memory*`<TYPE,SIZE>` - special registered container implementing optimal access with strobing changing of one element

logic<WIDTH>

logic<> is the basic type of C++HDL toolchain, representing SystemVerilog type logic. It is universal and can be of any width and can be used as standalone variable or inside reg<> construction.

Example of usage as variable:

```
logic<MEM_WIDTH_BYTES*8> data_out_comb;
```

Example of usage as port:

```
logic<MEM_WIDTH_BYTES*8>* data_in = nullptr;
```

logic<> type provides read/write access to particular bits using operator[] and to partial bitmap using method .bits(hi,lo):

```
buffer1_byteenable.next[addr_sub+i] = 1;
host_addr.bits(39,32) = *s_writedata_in >> 32;
```

u<WIDTH>

u<> is a basic unsigned value of variable size which supports all math operators and castable to a logic<> variable. Despite *u*<> can be of any size, tit supports maximum 64-bit math. Example of *u*<> usage:

```
u<STEPS_SIZE> cmd_steps;
```

u1, u8, u16, u32, u64

u1, *u8*, *u16*, *u32*, *u64* are aliaces for *u<1>*, *u<8>*, *u<16>*, *u<32>* and *u<64>* respectively.

reg<TYPE>

reg<> template is intended to make variable to be a register. It adds *.next* property to be changed in a *work()* function as well as .strobe() method which synchronize current value with next. Despite it should never be used as port definition, it is allowed to take address of register to assign it to a port. Examples of reg<> usage are provided further:

```
reg<State> state;
reg<u16> size;
reg<array<u8,WIDTH/8>> buffer1;
reg<logic<WIDTH/8>> buffer1_byteenable;
```

```
state.next.steps = state.steps - 1;
if (state.steps == 0) {
    state.next.steps = 255;
}

size.next = 0xFFFF;

buffer1.next |= buffer1_precalc;

buffer1_byteenable.next[i] = buffer2_byteenable[i];
```

array<SIZE>

array<> type is used for storing a vector of similar types. It should be used with reg<> template in case if registered access is required.

```
array<u8,WIDTH/8>* avmm_writedata_out = &buffer1;
```

memory<TYPE,SIZE>

memory<> type is developed for optimal performance of access to registered memory with ability to change one word at a clock cycle. It is special container which is always registered and cant be used as a port. It uses apply() method for strobing data. The following example shows how memory<> type should be used to organize simple one r/w port memory.

```
#pragma once

#include "cpphdl.h"
#include "PrjConfig.h"

using namespace cpphdl;

template<size_t MEM_WIDTH_BYTES, size_t MEM_DEPTH>
class Memory
{
public:
    u<clog2(MEM_DEPTH)>*      write_addr_in = nullptr;
    logic<MEM_WIDTH_BYTES*8>* data_in       = nullptr;
    bool*                     write_in      = nullptr;
    u<clog2(MEM_DEPTH)>*      read_addr_in  = nullptr;
    logic<MEM_WIDTH_BYTES*8>* data_out      = &data_out_comb;
    bool*                     read_in       = nullptr;

    void connect() {}
private:
    logic<MEM_WIDTH_BYTES*8> data_out_comb;
    memory<u8,MEM_WIDTH_BYTES,MEM_DEPTH> buffer;

    size_t i;
public:

    void data_out_comb_func()
    {
        data_out_comb = buffer[*read_addr_in];
    }

    void reset()
    {
    }

    void work(int clk)
    {
        if (!clk) return;

        if (*write_in) {
            buffer[*write_addr_in] = *data_in;
        }
    }

    void strobe()
    {
        buffer.apply();
    }

    void comb()
    {
        data_out_comb_func();
    }
};
```

C++HDL SV Conversion tool User Guide

To be continued...