# C++HDL Specification

Mike Reznikov

2026-01-10

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 and testing using C++ language
2. Allowing extra fast simulation of a blocking-assignments defined 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 live C++HDL to SystemVerilog conversion to

- Connect C++HDL teams to classical verification and testing teams
- Deliver SV RTL to fabrication processes and tools or third-party companies

The main benefits of using C++ for RTL development are in replacing of slow **simulation** with up to 100 times faster compilation and execution and in involvement of a modern language and new people in RTL modelling. Following properties of C++ language listed provide strong foundation for RTL development process:

- Ability to use any of plenty of professional IDEs and tools for development and debugging, supporting huge projects management
- C++ is a one of the most popular and powerful programming languages in the World, having extra wide community
- C++HDL makes many C++ developers accessible for chipmaking industry
- C++ is extrimely fast in compilation and in execution
- It's free and does not require paying for each instance

RTL modelling using C++HDL includes verification and testing, providing all power and speed of C++ language in modelling of digital signalling and digital systems interaction. It allows to make Verification process more simple, flexible, fast, and only containing programming.

C++HDL generates pure SystemVerilog output which mostly a reflection, providing line to line visibility. (it is even possible to apply patches synchronously to both representations of RTL during finalization stages) Generated SystemVerilog files can be frozen at any moment and used as the main source code for next stage of ASIC/FPGA production process.

## Who is C++HDL for

- Digital IC development teams (ASIC,IP,libraries,FPGA) - for faster development and testing of complex digital designs, free of charge
- Digital IC developers - to use modern C++ environment and smart IDEs, powerful C++ debug tools, static analysis and linting
- Software developers who want to deliver hardware and use powerful and modern language with OOP, templates, abstraction, recursion, etc
- CAD/Tool developers (especially AI-coding/training) - 100 times more work cycles per day + C++ is more AI-learned by GPTs
- Talent seekers - involving the most popular programming language speakers into variety of modern projects

## What is the difference from other C++ to Verilog products

- C++HDL is not HLS, it is a representation of SystemVerilog, register to register, clock to clock, completely repeating model behavior, line by line
- C++HDL module is a single-process activity without waits, notifies, streams and co-operative multitasking. Multi-threading is possible only for many modules/instances

## 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
- Timing or power critical sections should be isolated on architectural level

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 delivered in two parts:

- C++ headers which contain definitions of C++HDL datatypes
- Conversion/linter tool which provides C++HDL to SystemVerilog conversion

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 an .h header or a .cpp object file. Each header file should describe a module or auxiliary information like datatypes, constants, inline functions definitions, etc.

C++HDL provides ability of using of C++ structures and own data types in order to build comfortable and harmonious ecosystem for the project development. All types and constants will be translated into SystemVerilog datatypes during conversion process.

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

```cpp
#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;
    uint8_t units:6;
    uint8_t flags:2;
    uint16_t address;
}__PACKED;
static_assert (sizeof(CmdConfig) == 4, "struct CmdConfig size is not correct");
```

### Module description format

Module is defined by C++ class, based on cpphdl::Module, which includes:

1. Private zone with variables, registers and nested modules
2. Public zone with I/O ports definitions and *connect()* function body
3. *work()*, *strobe()* and combinational functions bodies

In the following block of code the simple FIFO RTL model description using C++HDL is provided as an example:

```cpp
#pragma once

#include "cpphdl.h"
#include "Memory.h"
#include <print>

using namespace cpphdl;

template<size_t FIFO_WIDTH_BYTES, size_t FIFO_DEPTH, bool SHOWAHEAD = true>
class Fifo : public Module
{
    Memory<FIFO_WIDTH_BYTES,FIFO_DEPTH,SHOWAHEAD> mem;

    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;

public:
    __PORT(bool)                            write_in;
    __PORT(logic<FIFO_WIDTH_BYTES*8>)    write_data_in;

    __PORT(bool)                            read_in;
    __PORT(logic<FIFO_WIDTH_BYTES*8>)    read_data_out   = mem.read_data_out;

    __PORT(bool)                            empty_out      = __VAL( empty_comb_func() );
    __PORT(bool)                            full_out       = __VAL( full_comb_func() );
    __PORT(bool)                            clear_in       = __VAL( false );
    __PORT(bool)                            afull_out      = __VAL( afull_reg );

    bool                         debugen_in;

    void connect()
    {
        mem.write_data_in = write_data_in;
        mem.write_data_in = write_data_in;
        mem.write_in      = write_in;
        mem.write_mask_in = __VAL( 0xFFFFFFFFFFFFFFFFULL );
        mem.write_addr_in = __VAL( wp_reg );
        mem.read_in       = read_in;
        mem.read_addr_in  = __VAL( rp_reg );
        mem.__inst_name = __inst_name + "/mem";
        mem.debugen_in  = debugen_in;
        mem.connect();
    }

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

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

    void work(bool clk, bool reset)
    {
        if (!clk) return;
        mem.work(clk, reset);

        if (debugen_in) {
            std::print("{:s}: input: ({}){}, output: ({}){}, wp_reg: {}, rp_reg: {}, full: {}, empty: {}, reset: {}\n", __inst_name,
                (int)write_in(), write_data_in(), (int)read_in(), read_data_out(), wp_reg, rp_reg, (int)full_out(), (int)empty_out(), reset);
        }

        if (reset) {
            wp_reg.clr();
            rp_reg.clr();
            full_reg.clr();
            afull_reg.clr();
            return;
        }

        if (write_in()) {

            if (full_comb_func() && !read_in()) {
                std::print("{:s}: writing to a full fifo\n", __inst_name);
                exit(1);
            }
            if (!full_comb_func()  read_in()) {
                wp_reg.next = wp_reg + 1;
            }
            if (wp_reg.next == rp_reg) {
                full_reg.next = 1;
            }
        }

        if (read_in()) {

            if (empty_comb_func()) {
                std::print("{:s}: reading from an empty fifo\n", __inst_name);
                exit(1);
            }
            if (!empty_comb_func()) {
                rp_reg.next = rp_reg + 1;
            }
            if (!write_in()) {
                full_reg.next = 0;
            }
        }

        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;

    }

    void strobe()
    {
        mem.strobe();
        wp_reg.strobe();
        rp_reg.strobe();
        full_reg.strobe();
        afull_reg.strobe();
    }
};
```

- Module class definition can use template parameters
- It is allowed to use integrated C++ types like bool, unsigned, unsigned long, etc in all places except reg<>
- Each of *connect()*, *work()*, *strobe()* functions should call corresponding functions of nested modules
- Only *reg_name.next* value can be changed in all places except reset. Both reg and *reg.next* values can be used on right side of expressions
- During reset *.clr()* and *.set(val)* methods are used to set up both current and next register's values
- *[f]printf()*, *std::print*, *$write()*, *exit()* functions are converted to their SV synonyms and parameters are changed appropriatly

## Input/output ports

All ports are of type *std::function<`data_type()`>* in C++HDL. It allows instant recalculation of all combinational functions chain.

- Macros *___PORT(* `data_type` *)* allows port simplier declaration.
- Macros *___VAL(* `member_or_expression` *)* represents lambda function `[&](){ return member_or_expression; }` used as replacement for Verilog assign expression. Return type of port lambda function represents type of the port.

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

- `port_in` or `obj_port_in` - generic input port name
- `port_out` or `obj_port_out` - generic output port name
- or longer name, ending with *_in* or *_out*

It is recommended to initialize all output ports right away in class description. In more complex situations it's possible to initialize output ports in a special *connect()* function. Input ports can be assigned *___VAL(0)* value to emulate Verilog unassigned inputs behavior.

**NOTE!** To build a complex bus interface between C++HDL module with third-party SV module, use packed *structs* to achieve proper <8bit fields packing.

## Clock and reset

The only one clock is used currently, named *clk*. Reset is the main *reset* parameter to work function. It is possible to define any synchronous reset sequence. Asynchronous reset it not supported yet.

## Variables list

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

- **registers**
- **combinational**
- **temporary**

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

- Registers can carry structs, arrays, and single values.

- Combinational variable can be of any type.

- Declaration of variables inside methods is prohibited.

## Connect method

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

## Work method

Work method can make any changes to registers and temporary variables. Only *.next* value of registers should be changed directly.

- Work method can call other methods to make code well-structured.
- Methods with return value become functions SystemVerilog, methods with void return become tasks in Verilog.
- It is possible to change registers only from void methods.
- Methods can take references to registers as parameters.

## Strobe method

Strobe function should contain all registers of the module and call .strobe() functions for them. Also strobe() should be called for each of nested instances of the class. Forgotten registers will be reported by *cpphdl* tool.

## Comb methods

Combinational methods represent Verilog combinational logic (functions). All combinational methods should comply with the following requiremets:

- The name of the function should contain *_comb_func()* suffix
- Corresponding variable should be defined in module class: *var_name_comb*
- Combinational function should calculate, assign a value to *var_name_comb* variable and return it

It will be converted to corresponding Verilog variable and `always (*)` block during conversion.

It is important to avoid loops in combinational functions call chains.

## Data types

To repeat SystemVerilog behavior 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 structs
- *array*`<TYPE,SIZE>` - variable, optimized for large arrays access and changing by elements
- *memory*`<TYPE,SIZE>` - special registered container implementing optimal memory access with strobing

### 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, it 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 synchronizes current value with next. It should not be used as port definition, but it can provide data to a port. Examples of reg<> usage are provided below:

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

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

size.next = 0xFFFF;

buffer1.next |= buffer1_precalc;

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

mask.next.bits((i+1)*32-1,i*32) = 0;
```

### array<SIZE>

array<> type is used for storing a vector of similar types. It can be used with reg<> template.

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

### memory<TYPE,SIZE>

The *memory*<> type is developed for optimal performance of access to registered memory with ability to change one word at a clock cycle. It 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 memory with one read and one write port.

```
#pragma once

#include "cpphdl.h"
#include <print>
```

```cpp

using namespace cpphdl;

template<size_t MEM_WIDTH_BYTES, size_t MEM_DEPTH, bool SHOWAHEAD = true>
class Memory : public Module
{
    logic<MEM_WIDTH_BYTES*8> data_out_comb;
    reg<logic<MEM_WIDTH_BYTES*8>> data_out_reg;
    memory<u8,MEM_WIDTH_BYTES,MEM_DEPTH> buffer;

    size_t i;

public:
    __PORT(u<clog2(MEM_DEPTH)>)       write_addr_in;
    __PORT(bool)                      write_in;
    __PORT(logic<MEM_WIDTH_BYTES*8>)  write_data_in;
    __PORT(logic<MEM_WIDTH_BYTES>)    write_mask_in;

    __PORT(u<clog2(MEM_DEPTH)>)       read_addr_in;
    __PORT(bool)                      read_in;
    __PORT(logic<MEM_WIDTH_BYTES*8>)  read_data_out = __VAL( data_out_comb_func() );

    bool                      debugen_in;

    void connect() {}

    logic<MEM_WIDTH_BYTES*8>& data_out_comb_func()
    {
        if (SHOWAHEAD) {
            data_out_comb = buffer[read_addr_in()];
        }
        else {
            data_out_comb = data_out_reg;
        }
        return data_out_comb;
    }

    logic<MEM_WIDTH_BYTES*8> mask;

    void work(bool clk, bool reset)
    {
        if (!clk) return;

        if (write_in()) {
            mask = 0;
            for (i=0; i < MEM_WIDTH_BYTES; ++i) {
                mask.bits((i+1)*8-1,i*8) = write_mask_in()[i] ? 0xFF : 0 ;
            }
            buffer[write_addr_in()] = (buffer[write_addr_in()]&~mask) |
             ↪  (write_data_in()&mask);
        }

        if (!SHOWAHEAD) {
            data_out_reg.next = buffer[read_addr_in()];
        }

        if (debugen_in) {
            std::print("{:s}: input: ({}){}@{}({}), output: ({}){}@{}\n", __inst_name,
                (int)write_in(), write_data_in(), write_addr_in(), write_mask_in(),
                (int)read_in(), read_data_out(), read_addr_in());
        }
    }

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

## C++HDL SV Conversion tool

The main purposes of *cpphdl* tool is to

- Provide conversion of C++HDL code to SystemVerilog models
- Check dependencies in combinational chains and forgotten strobe calls in C++HDL source code

### Structures

- Each structure or union is converted into SystemVerilog package in a separate file
- The order of fields is reversed due to differences between C++ and SystemVerilog
- Anonymous structures and unions declared inside other structures get 'anon' name substitution

### Templates

- During conversion cpphdl uses Module class template parameters as SystemVerilog module parameters, if they are numerical
- cpphdl creates separate SV module per each instantiated combination of data types used as template parameters

### References

- All references are removed during SV conversion. Reference should be applied in C++ when it's necessary and possible to improve performance.

## Syntax

The *generated* folder is created after *cpphdl* call, containing .sv files. The syntax of *cpphdl* tool is the following:

```
cpphdl <source.h> <source.cpp> ... [compilation parameters]
```

The cpphdl tool is based on llvm clang and supports all usual C++ command line parameters.