

CHDL: From node to Load()

A Tutorial

Chad D. Kersey
School of ECE
Georgia Institute of Technology



Outline

CHDL is a simple open source library for designing digital hardware; a nucleus around which an entire prototyping infrastructure has condensed. In this tutorial, we will go from an empty directory to a simple computer system ready to run on an FPGA evaluation board.

Outline

- Prerequisites
- CHDL Basics: `node` and `vec`
- CHDL STL: `ag`
- CHDL STL: `mem_req`, `mem_resp`, and `mem_port`
- The CHDL Module Library
- FPGA Demo

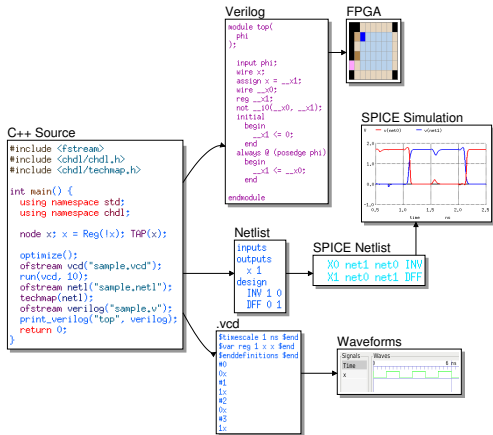
Prerequisites

The following are needed to get and use CHDL:

- C++ compiler supporting C++11 (GCC 4.7 or higher)
- Unix-like OS (Linux, Mac OS with Homebrew, Windows with Cygwin)
- Understanding how libraries work on your platform of choice.
- GNU Make
- Optional: Git client (.zip also available)
- Optional: Waveform viewer (GTKWave)

Exercise: Write a library in C++ that contains a single function say `hi()` that prints “Hello, world!” to the screen. Compile and link it as a shared object and write a program, in a separate directory, that calls this function once and exits.

CHDL: A C++ Hardware Design Library



- Designs in CHDL are netlist generators.
- Ultimately generating basic gates and memory arrays.
- Extreme parameterizability with C++ templates.

Netlist Introspection

CHDL allows programs to modify their own netlist as they run, providing a convenient mechanism for optimization.

CHDL output can be:

- Synthesized & run in SPICE.
- Simulated at logic level.
- Run on FPGA w/ vendor tools.

Getting and Building CHDL

The source can be obtained with the git client:

Downloading the Source

```
$ git clone https://github.com/cdkersey/chdl.git
```

Or from GitHub as a .zip archive:

```
https://github.com/cdkersey/chdl/archive/master.zip
```

Once the code is cloned or unpacked, you can build it using Make.

Compiling the Source

```
$ cd chdl  
$ make -j 8  
$ sudo make install
```

The installation phase is optional and installs by default into `/usr/local` or whatever is set in the environment variable `PREFIX`. If this produces any unexpected errors, please report them to `cdkersey@gatech.edu`.

The blinkenlights Example

Full Listing

```
#include <iostream>
#include <fstream>

#include <chdl/chdl.h>

using namespace std;
using namespace chdl;

int main() {
    node x;
    x = Reg(!x);

    TAP(x);

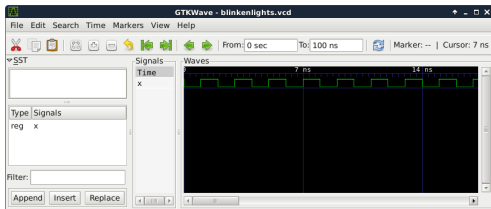
    optimize();

    ofstream vcd("dump.vcd");
    run(vcd, 100);

    return 0;
}
```

This basic example:

- Instantiates a single node, x.
- Instantiates Reg() and Not().
- Makes the value of x depend on its own past value.



Waveform for blinkenlights example.

The Nature of nodes

The heart of our first design is the following pair of lines:

```
node x;  
x = Reg(!x);
```

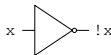
Note that:

- The node's value is used before it is set.
- The assignment does not redefine x .
- All users of x remain so.
- Assignment only changes a node's *source*.
- Cycles must pass through `Reg()`.

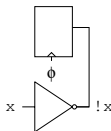
The first line declares x :

x

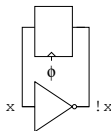
“! x ” declares new node containing logical not of x :



`Reg()` creates a D flip-flop:



The `=` operator assigns the value to x , completing the cycle:



Working with nodes

There are only a few basic synthesizable node types (and a few more for simulation):

Basic node types

Reg(), Inv(), Nand(), Lit(), tristatenode

Many structures are built from this simple set of operations:

Library Functions Using nodes

And(), Or(), Xor(), Mux(), Wreg(), Latch(), CAssign(), &&, ||, !=, ==

Let's build an example using these features:

Differential Manchester Coding

```
// Alternate clocking, data periods
node Encode(node &ready, node data) {
  // Assume data clock 1/2 reg clock
  node out, sync, sw(sync || data);
  sync = Reg(!sync, 1); // Init 1
  out = Reg(Xor(out, sw));
  ready = !sync;
  return out;
}

node Decode(node &valid, node in) {
  node out, sync;
  sync = Reg(!sync, 1);
  out = !Reg(in) == in;
  valid = !sync;
  return x;
}
```


vec: Arrays of CHDL Objects

Declaring a vec

```
vec<4, vec<4, node>> mat1;  
vec<4, bvec<4>> mat2;  
\\ Equivalent; bvec<N> = vec<N, node>
```

A full library of functions operating on vecs has been provided as part of stock CHDL:

Functions Using vecs

`[]`, `[range<A,B>()]`, `Reg()`, `Lit<N>()`, `Flatten()`, `Wreg()`, `Mux()`, `And()`, `AndN()`, `Or()`, `OrN()`, `Xor()`, `XorN()`, `Add()`, `Sub()`, `Mul()`, `Div()`, `Log2()`, `Decoder()`, `Zext<N>()`, `+`, `-`, `*`, `/`, `&`, `|`, `^`, `~`, `>`, `<`, `>=`, `<=`, `==`, `!=`

All arithmetic involving bvecs (vecs of nodes) assumes unsigned binary integers. Special numeric features are provided by other libraries (discussed later).

4-bit Counter Example

```
bvec<4> Count() {  
    bvec<4> x;  
    x = Reg(x + Lit<4>(1));  
    return x;  
}
```

The sz Template

Use `sz<T>::value` to determine the flattened size, in bits, of a CHDL type, including nodes and vecs. (At compile time.)

Memory: LLRom() and Syncmem()

Individual SRAM banks can be instantiated with Syncmem(). These are synchronous SRAMs representable by FPGA block RAM:

Instantiating a 1-Port Syncmem()

```
const unsigned SZ(10); // 1kB SRAM
bvec<32> addr;
node wr; // Write signal
bvec<8> d, q = Syncmem(Zext<SZ>(addr), d, wr);
```

Additionally, low-level asynchronous ROM devices (LLRom()) are available, initialized with a flat, 1-word-per-line hex file:

Instantiating a 1-Port Syncmem()

```
bvec<10> addr;
bvec<8> q = LLRom<10, 8>(addr, "contents.hex");
```

Exercise: Build a simple general-purpose computer using the CHDL facilities presented so far.

The CHDL STL

In addition to CHDL proper, there is also a CHDL Standard Template Library, a set of useful functions and types that work with the CHDL main library.

Getting and Testing CHDL STL

```
$ git clone https://github.com/cdkersey/chdl-stl.git
$ cd chdl-stl
chdl-stl$ make test
```

Features include:

- Aggregate types using `ag`, `un`, and `dir` which we will discuss in the following slides.
- `mem_port`, a parameterized memory interface.
- Containers, including `Stack`, `Queue`, and `Map`.
- Pseudo-random number generation using `Lfsr`.
- Documentation in `README`.

Aggregate Types

Why not use C++ struct or class?

- Does not allow reflection.
- Does not work with sz or Flatten().

We have solved this in CHDL using ag, the STL aggregate type seen here:

Sign-Magnitude Aggregate Type

```
template <unsigned N> using smag =  
    ag<STP("neg"), node,  
    ag<STP("val"), bvec<N>  
> >;
```

- Note use of C++ *alias templates*.
- Note Lisp-like nesting.

A few of the types provided by CHDL STL:

CHDL STL Types

```
ag<NAME, T, NEXT>,  
un<NAME, T, NEXT>,  
flit<T>,  
mem_port<B, N, A, I>,  
in<T>, out<T>,  
inout<T>,  
reverse<T>::type,  
undir<T>::type,  
in_mem_port<B, N, A, I>,  
out_mem_port<B, N, A, I>,  
fp<E, M>, fxp<E, M>,  
si<N>, ui<N>
```

Accessing Aggregate Members

Aggregates can be:

- TAP()ped. Field names underscore-separated in waveform file.
- Used as inputs and outputs to a Reg.
- Flattened into a bvec.
- Measured. Their size is `ag<...>::sz::val`.

To access individual members of an aggregate, a special `_` macro is used. This leads to code that looks like:

Accessing ag Members

```
stall = !_(req, "ready");
_(req, "valid") = req_pending;
_(req, "addr") = rsrc_val_0;
_(req, "data") = rsrc_val_1;
```

Exercise: Build a 2-stage pipelined version of the previous example. Use a single CHDL ag to describe the signals contained in the pipeline register.

CHDL Module Writer/Loader

- Large designs take a long time to elaborate (generate netlist).
- Save to netlist file instead; replace elaboration operation with loading.

Getting CHDL Module

```
$ git clone https://github.com/cdkersey/chdl-module
$ cd chdl-module
chdl-module$ make
chdl-module$ sudo make install
```

- Produces `libchdl-module.so`. Remember in `LDLIBS`!
- Primary functionality added (in `chdl/loader.h`) through:
 - `Expose()/EXPOSE()`: export directed aggregate.
 - `Load()`: load module form disk.
- Use CHDL built-in `print_netlist()` after `optimize()` to save modules.

Building a Module

The module source (using CHDL STL floating point):

Floating Point MAC Module

```
#include <chdl/chdl.h>
#include <chdl/numeric.h>
#include <chdl/loader.h>
#include <fstream>

using namespace std;
using namespace chdl;

int main() {
    fp32 out, in_a, in_b;
    out = Reg(out + in_a * in_b);

    Expose("out", chdl::out<fp32>(out));
    Expose("in_a", in<fp32>(in_a));
    Expose("in_b", in<fp32>(in_b));

    optimize();

    ofstream nand("module.nand");
    print_netlist(nand);

    return 0;
}
```

Module User

```
#include <chdl/numeric.h>
#include <chdl/loader.h>
#include <chdl/lfsr.h>
#include <fstream>

using namespace std;
using namespace chdl;

int main() {
    fp32 a, b, x;

    Load("module.nand")
        ("in_a", in<fp32>(a))
        ("in_b", in<fp32>(b))
        ("out", out<fp32>(x));

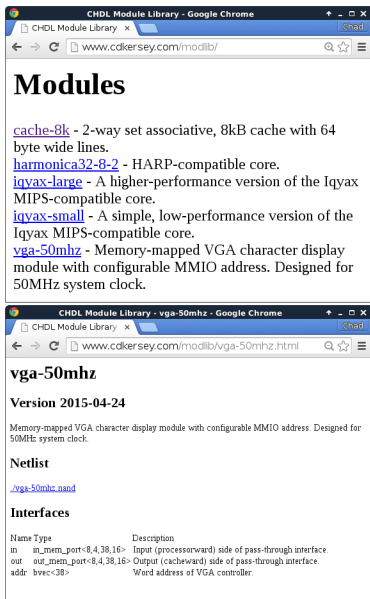
    a = IToF<8,23>(Lfsr<4,31,3,0x1234>());
    b = IToF<8,23>(Lfsr<4,31,1,0xabcd>());

    TAP(a); TAP(b); TAP(x);

    optimize();

    ofstream vcd("dump.vcd");
    run(vcd, 1000);
    return 0;
}
```

CHDL: Module Library



The image shows two screenshots of a web browser displaying the CHDL Module Library website. The top screenshot shows the main 'Modules' page with a list of modules: [cache-8k](#), [harmonica32-8-2](#), [iqyax-large](#), [iqyax-small](#), and [vga-50mhz](#). The bottom screenshot shows the detailed page for the **vga-50mhz** module, including its version (2015-04-24), description, netlist file (`./vga-50mhz.nand`), and a table of interfaces.

Name	Type	Description
in	in_mem_port<8,4,38,16>	Input (processorward) side of pass-through interface.
out	out_mem_port<8,4,38,16>	Output (cacheward) side of pass-through interface.
addr	bvec<38>	Word address of VGA controller.

Modules

Modules are pieces of pre-elaborated CHDL designs distributed as netlists with complex interfaces. These netlists can be loaded using a simple `Load()` function interface.

- The CHDL module library stores sources for these modules alongside metadata.
- Building the CHDL module library updates a set of simple HTML documentation.
- This can then be uploaded to the Web for distribution.

Demo

The design we will be demonstrating:

- Uses resources from both the CHDL template library and the online module library (highlighted in figure).
- Produces console output both on FPGA and in simulation.
- Uses CHDL memory interface and memory system components from CHDL template library.

