# Introduction to CHDL 2
## Aggregates and Memory

Chad D. Kersey

# From Last Time

- CHDL is an open source C++ Hardware Design Library.
- The basic type is the `node`.
- Arrays of `nodes` are `bvecs`, which are `vecs` of `nodes`.
- Operator overloads of basic arithmetic are provided for `bvec`.
- Other combinational functions, including multiplexers, priority encoders, and numerical conversion are in the standard library.
- Full documentation of this library is in the file `CHDL` in the source directory.
- Sequential logic can be implemented using the `Reg` function, which creates a D Flip-Flop, providing most of the sequencing in CHDL.
- `LLRom` can be used to generate an FPGA-synthesizable ROM.

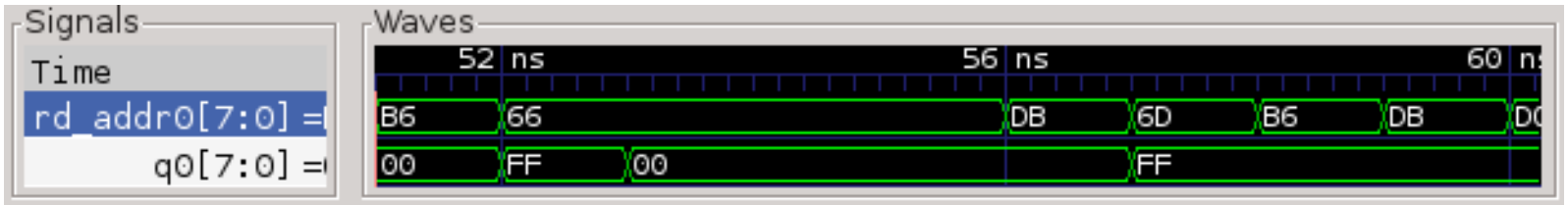
# Memory

A synchronous, FPGA-compatible SRAM can be produced using the `Syncmem` functions. Its size is determined by the widths of its data and address buses:

## Two Ways to Syncmem

```
q = Syncmem(addr, data_in, write_en);
q = Syncmem(rd_addr, data_in, wr_addr, write_en);
```

These are synchronous memories. The results of a read appear on their output in the cycle following a read:




**Exercise:** Build a simple state machine that loads the numbers 0 through 255 into a 256 byte SRAM, then reads the numbers back out and sums them up.

# Memory: Multiple Read Ports

There is also a version of `Syncmem` with multiple read ports, taking a `vec` of read addresses and returns a `vec` of results:

**Multi-port `Syncmem`**

```
vec<P, bvec<N>> Syncmem(
  vec<P, bvec<M>> qa, bvec<N> d, bvec<M> da, node wr
);
```

**Exercise:** Modify your state machine from the previous exercise to use a `Syncmem` with two read ports, summing values at even addresses on one of these ports and odd addresses on the other. It should finish summing all of them within 128 cycles of the end of initialization.

# The CHDL STL

In addition to CHDL proper, there is also a CHDL Standard Template Library, a set of useful functions 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, which we will discuss in the following slides.
- Network devices including routers and arbiters.
- Containers, including `Stack`, `Queue`, and `Map`.
- Bloom filters and hash functions.
- Pseudo-random number generation using `Lfsr`.

# Aggregate Types

If you need to arrange multiple signals into a collection, you could build a C++ `struct` or `class` of them.

- Could build a `struct` or `class`.
- This does not allow any kind of reflection.
- This does not work with the rest of the CHDL standard library.

We have solved this in CHDL using ag, the CHDL STL aggregate type. The following is a memory request port built using ag:

**MemReq Aggregate Type**

```
typedef ag<STP("ready"), node,
        ag<STP("valid"), node,
        ag<STP("addr"), bvec<N>,
        ag<STP("data"), bvec<N> > > > > memReq_t;
```

# Accessing Aggregate Members

Aggregates can be:

- `TAP`ped. The field names are underscore-separated.
- 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 aggregste, 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 sum example. Use a single CHDL ag to describe the signals contained in the pipeline register.