What is CAST?

- A Hardware description language
  - Not to be confused with a programming language!
- Describe logic circuits at the gate level
  - Build up from primitives (Nand, Nor, Inverter)
- Define circuit blocks
- Simulate the functionality of circuits
  - Show signals as (bundled) waveforms
Data Types

- We give you three logic “blocks” with the following interfaces:
  - Nand2(a,b,out)
  - Nor2(a,b,out)
  - Inv(a,_a)
- You can find the definitions of these blocks in `/usr/local/cad/cast/314/parts.cast`

- Primitive type is a node
  - Think of this as a wire in a logic circuit
Defining Functional Blocks

- Let's take a look the definition of Nand2() in parts.cast...

```c
define Nand2()(node a,b; node out) {
    prs{
        ~a | ~b -> out+
        a & b   -> out-
    }
}
```
Function Headers

- The header of a block has a standard format
  - define BlockName([parameter list])(inputs;outputs)
- The parameters are useful for generalizing gates, more on this later
- Inputs and output lists follow the convention that a type is followed by a comma separated list of node names, and types are separated by semicolons
- Header defines the *interface* of your block
Function Body

- You should not have to write a body that looks anything like the body of Nand2().

- Function bodies that you write will only instantiate other blocks (yours or the primitive ones we give you) and wire the gates by specifying which nodes are connected.
Blocks can be instantiated and wired in several ways

- node a, b, c; Nand2 g1; g1.a = a; g1.b = b; g1.out = c;
- node a, b, c; Nand2() (a, b, c); (anonymous gate)
- node a, b, c; Nand2() g1(a, b); (named gate)

Note the equal sign means connection not assignment – remember it’s not programming, its circuit description…

- What you are really doing is aliasing the names
A Simple Example

- Want an AND gate... draw a picture... then *describe it* with CAST

```cpp
define And2()(node a, b; node out)
{
    Nand2() g1(a, b);
    Inv()(g1.out, out);
}
```
Arrays

- CAST also allows you to declare an indexed array of nodes as follows
  - `node[10] b; // declares b[0]..b[9]`

- You can also make arrays of blocks you define
  - `And2[10] b; // declares 10 AND gates`

- Nice feature because most logically constructed circuits exploit repetition
Ranges

- CAST supports the ability to pick the index range when you declare an array as well
  - node[6..10] x; //declares nodes x[6]..x[10];

- You can also specify a subset of an array using similar notation (useful for connection)
Connecting Arrays

- Arrays can be connected to one another using the "=" operator
  - Only restriction is the arrays (or ranges) being connected must be the same size (obviously)
  - x[3..5] = y[8..10];
CAST provides *syntactic constructs* to make the wiring more “elegant”

- **Caution!!** This is not a way of specifying circuit behavior… it’s just a way of being concise in your description of the circuit!

- Loops have the following structure
  - `<i: range: (some CAST statements) >`

- Conditionals have the following structure
  - `[ condition -> (some CAST statements) ]`
Parameterized Types

- Sometimes you may want to make a block more general

  - Instead of making a 3-bit adder, a 4-bit adder, etc., you could make one adder definition and parameterize it by how many bits you’d like it to be.

    define adder(int N)(node[N] a,b,sum; node cout)

- Can use parameters in things like loop bounds, conditionals and such... exploit circuit structure
Parameterized Example

- Bitwise AND of two N-bit variables...
  - This is easy, just N AND gates, right?

```plaintext
define BitAnd(int N)(node[N] a, b; node[N] out) {
    <i:N: And2()(a[i], b[i], out[i]);>
}
```

- Isn’t that pretty 😊
**Miscellaneous Tips**

- Don’t start coding CAST until you’ve drawn yourself a circuit diagram.

- CAST also allows you to define your circuit recursively... this is actually *really* useful for generalizing certain circuit topologies like trees.
  - Important from an efficiency standpoint!

- You should have a file called myparts.cast that you include in each cast definition file you make.
  - Myparts.cast should have as its first line
  - `import "314/parts.cast";`
Simulating

- Once you have your definitions all set you **must** instantiate the definition you want to test.

- You can then run the following command on the file which contains your instantiation:
  
  ```
  prs2sim filename.cast
  ```

- This creates two new files:
  
  - `filename.sim` and `filename.al`
Simulating

You are now ready to simulate your circuit by typing the following!

- `irsim.sh filename.sim filename.al`

You can type `help` to see a list of all available commands `irsim` offers, and `help command` to get help on a specific command.
Simulation

- The basic thing you do in IRSIM is set input nodes high or low, take a step forward in time, and observe the changes (if any) in the output nodes.
  - To set node A high you say: h A <enter>
  - To set node A low you say: l A <enter>
  - To take a step you say: s <enter>
  - Usually don’t simulate “interactively”…
Instead you can type your simulation into a separate file and then just type the filename in after launching irsim to run your script...

Lets say I defined some function FOO that takes inputs: node[8] a, b; and produces outputs node[3] c;
A typical command file might look like this:

```
vectore A a[{7:0}]
vectore B b[{7:0}]
vectore C c[{2:0}]
ana -b A B C |graphical analyzer, show vectors in binary
set A 01001011 |set the value using a binary number
set B %xf4 |set the value using a hex number
s |take a step (you can set duration with stepsize)
set A %x11
s
...
```