7 Hardware Description Languages

Hardware Description Languages

Instead of having to write out the individual gates for a 64 bit adder, we can just use behavioral languages and let the computer translate it into gates.

We can translate the structural Verilog
and a1(n1,a,b);
or o1(y,n1,c);

Into behavioral Verilog
assign y = a&b | c;

The fancier we get, it'll be harder to tell exactly what hardware were getting. The biggest danger with HDL is to think of it like a computer program though - everything ties down to hardware. Don't write code that'll give you really weird stuff.

We think of the hardware that we want, and find the most appropriate way to build it, either structurally or behaviorally.

Lets imagine we want to build a 64 bit adder. We have inputs $a[0]b[0] \text{ to } a[63]b[63]$. In structural verilog, we would build this out of full adders, and internally write out each of those gates.

Rules of Behavioral Verilog

In behavioral, we have a set of logic equations

and	&
or	|
xor	^
not	~
add	+

We could write, for example,
assign y=a&b&c | a& ~b & ~c | ~a&~b&c | ~a & b & ~c ;
We could easily write this out in a series of gates, or this is a 3 input xor gate. The beauty of logic synthesis is that you give it what you want, and it will make the cheapest/fastest (or whatever other metric) circuit possible. We usually tell the tool what it should do and the timing constraint, and it makes the cheapest way to satisfy those.

Behavioral verilog is case sensitive ($A \neq a$).
You can use your own gates:

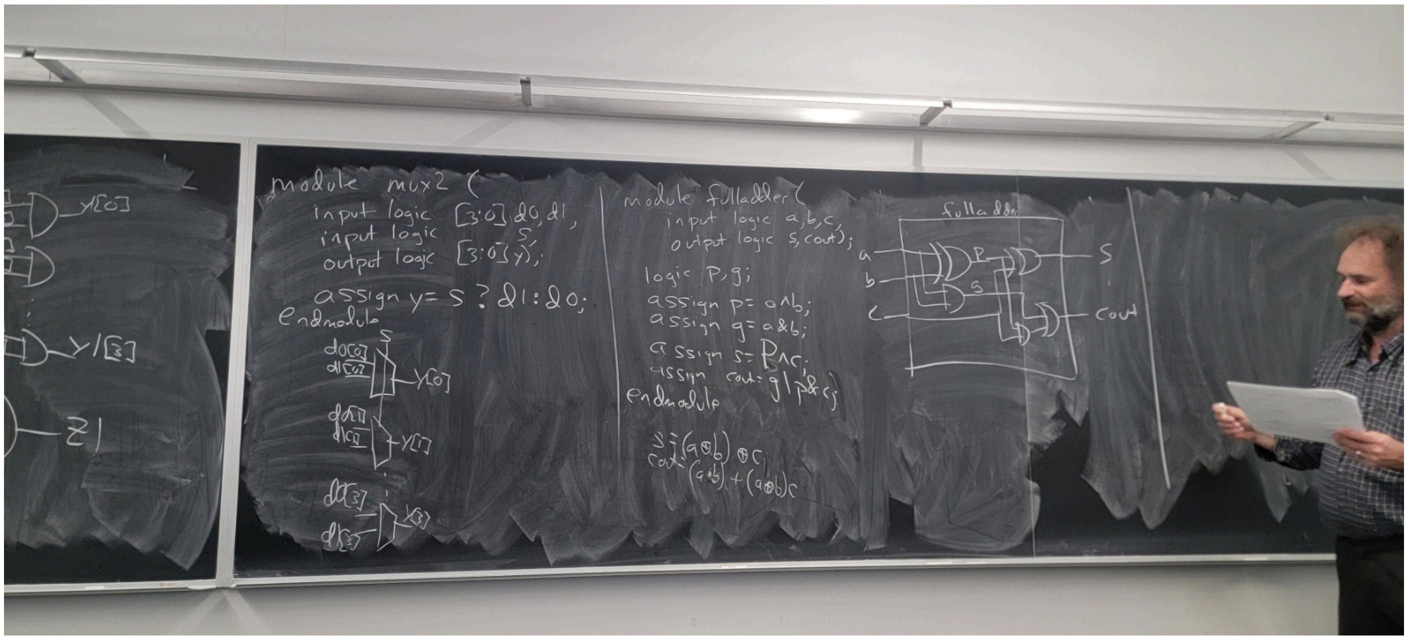
```
module and3(
input logic a,b,c,
output logic y);
assign y = a&b&c
endmodule
```

This could be a transistor level, gates implementation, whatever. Doesn't matter.
If we want to use this, we would write
and3(a,b,c,y), since we have to keep the order of the inputs from the module definition.

We have bitwise and reduction methods.
You can take two inputs $[3:0]a, b;$
and do $\text{assign } y_1 = \text{ a \& b}$ (bitwise)
or $\text{assign } y_2 = \text{ \&a}$ which performs $\&$ on every bit of a.

We also have conditionals, i.e.

assign y = s ? d1:d2 would have y = d1 if s is true, or y = d2 if s is false.



```
module mux2 (
   input logic [3:0] d0, d1,
   input logic       s,
   output logic [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

```
module fulladder (
   input logic a, b, c,
   output logic s, cout);

   logic p, g;
   assign p = a^b;
   assign g = a&b;
   assign s = p^c;
   assign cout = g | p&c;
endmodule
```

$s = (a \oplus b) \oplus c$

$cout = (a \cdot b) + (a \oplus b)c$

We have more operators that follow an order of operations

~ Not

∗,/,%

+,-

<<,>> shift

<<< , >>> arithmetic shift

<,<=,>,>= comparison

==, !=, equality comparison

&

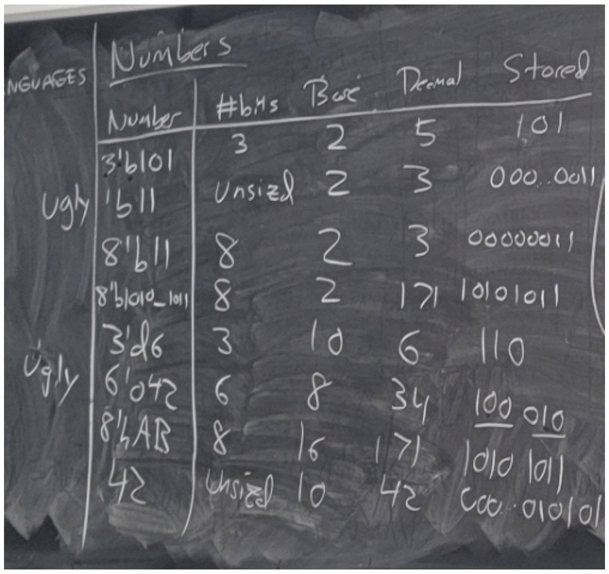
^

|

?:

The important thing to note is that ~ comes before & which is before ^.

We have ways of dealing with numbers where you define the number of bits, the base, and the value:



NGUAGES

Numbers

	Number	#bits	Base	Decimal	Stored
	3'b101	3	2	5	101
Ugly	'b11	Unsized	2	3	000...0011
	8'b11	8	2	3	00000011
	8'b1010_1011	8	2	171	10101011
	3'd6	3	10	6	110
Ugly	6'o42	6	8	34	100 010
	8'hAB	8	16	171	1010 1011
	42	Unsized	10	42	000...010101

We have bit swizzling, where you can set a value equal to a set of appended bits
assign y = {$a[2:1]$,{3{$b[0]$}}, $a[0]$, 6'b100_010}
This would make y

$$a[2], a[1], a[0], b[0], b[0], b[0], a[0], 1, 0, 0, 0, 1, 0$$

Combinational Logic

Delays

Sequential Logic