Verilog Hardware Description Language

Hardware design is complicated and exacting, which makes it difficult to get a design perfect. (Only "perfect" suffices.) Further, compared to software, hardware fabrication and replication are expensive, which increases the penalty for sloppy design. Accordingly, simulation is an essential tool in the creation of computer hardware, since it allows one to "test without fabrication" and thus gain confidence that a design is perfect.

Of the many hardware description languages and simulators available, we will use Verilog in CSE471.

Verilog Overview

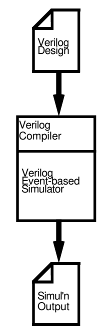
- Verilog is a hardware description language + an event-based simulator.
- Many aspects of Verilog are analogous to C, e.g. arithmetic operators, but beware of additional syntax and different concepts.
- Verilog has a wide range of specificational forms:

  -Low level concepts such as gates with timing, pullups, supply0 (ground), wired or, capacitive structures, drive strengths, high impedence, etc.

  -Designs defined using only low-level structures can be *synthesized* into VLSI circuit designs give the right tools.

  -*Behavioral specification* uses high level concepts, including events, normal arithmetic computation, etc., to express complex constructs without having to give a full design.

  -The two forms can be mixed, allowing for a top down design strategy: develop a working behavioral design, then iteratively replace components with synthesizeable equivalents until the whole design is complete.



Modules

Verilog is ...
- case sensitive
- indifferent to white space between lexemes

The basic structure in Verilog is a module, which is conceptually like a procedure in that it has a name, input/output parameters and a body.
Module Syntax is given in Appendix A, but example is better.

required keyword — module name — port list

```
module fulladder( CarryOut, Result,
                  a, b, c);
input a, b, c;
output CarryOut, Result;
reg CarryOut, Result;

    always @(a or b or c)
      {CarryOut, Result} = a + b + c;
endmodule
```

register declaration

input port specification

output port specification

behavioral block header

block

required terminator

Language Basics

Comments are of two forms

From `//` to the end of the line

All text within `/* */` pair including new lines.

Integer literals have the form <length>'<radix><value> as in

`3'b001` a three bit binary literal "001"

`6'hBAAB00` a six digit hex literal "BAAB00"

`2'o17` a two digit octal literal "17"

Decimal literals are given explicitly or as in 3'd999

Binary variables have 1 bit default width

Binary values are 0, 1, x (X) for uninitialized or unknown and z (Z) for high impedence

`reg` declares memory, `wire` declares interconnect

More Modules

Behavioral modules specify either "initial" or "always" execution conditions.

equivalent to --

```
always @(r[0] or r[1] or r[2] or r[3])
```

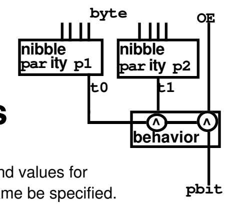
```
module par( p, r);
input [3:0] r;
output p;
reg p;
/*  Compute the parity of r, a nibble     */
        always @(r)
        p = (r[0] ^ r[1]) ^ (r[2] ^ r[3]);

endmodule
```

4 bit register specification

exclusive or

Invocation of Modules of Primitives



In addition to providing the name of the operation and values for parameters, invocation requires that an instance name be specified.

connectivity

```
module parity( pbit, byte, OE);
/* Computes the parity bit for a byte, according
to whether OE is 0 (even) or (odd)    */
input [7:0] byte;
input OE;
output pbit;
reg pbit;
wire t0, t1;

        par p1( t0, byte[3:0]);
        par p2( t1, byte[7:4]);

        always @(t0 or t1)
        pbit = (t0 ^ t1) ^ OE;
endmodule
```

instance name