Verilog Wires and Registers

... are the basic components for

1. Connecting gates and modules
2. Creating signals
3. Keeping a status

4-valued Logic

Wires and registers carry the following values:

- **0** logic zero (FALSE)
- **1** logic one (TRUE)
- **z** high impedance output
- **x** unknown *(any or none of above)*

1. Unconnected inputs are 'z'
2. 'z' as input equals 'x'
3. initially, *everything* is 'x'

Example for NAND Gate

Truth table:

	0	**1**	**z**	**x**
0	**1**	**1**	**1**	**1**
1	**1**	**0**	**x**	**x**
z	**1**	**x**	**x**	**x**
x	**1**	**x**	**x**	**x**

Purpose (Wire)

- A *wire* is a structural element that connects a source of a signal with a number of sinks
- it does not have any status and must be continuously driven
- it provides the only way of connecting modules with each other

Defining a Wire

- A wire is declared as

  ```
  wire cable;
  ```

  and is referred to by its name "`cable`"
- a wire can be connected with *one* output, either from a gate or another module
- a wire can be connected with *any* number of inputs for gates or modules

Defining a Wire

- wires can be *assigned* to other wires:

  ```
  wire cable1, cable2;
  assign cable2 = cable1;
  ```

  which implies that `cable2` is branched from `cable1`
- `input` and `output` ports of a module are usually wires

Purpose (Register)

- A *register* generates a signal
- it can be set or changed by procedural assignments only
- it has a status at any time
- a register usually feeds a wire

Defining a Register

- A register is declared as

  ```
  reg bit;
  ```

  and is referred to by its name "`bit`"
- a register can be used to feed any input where a wire could be used as well, but a wire cannot feed a register!

Defining a Register

- a register has a status, which remains constant until it is changed by a procedural assignment
- the initial status is "`x`" (unknown)
- a register can never be an `input` port, but it can feed an `output` port

Initial and Always Constructs

Modules can perform one or more concurrent sequences of actions *(processes, threads)*

1. `initial`: a sequence is to be performed once only immediately after the instantiation of a module
2. `always`: a sequence is to be repeated as long as the module exists

Initial and Always Constructs

- In particular, the sequence contains
  1. *procedural assignments* to initialize or modify register values
  2. *delay* specifications to schedule actions for a specific time and order
- Sequences may contain *control statements*

Example:

A clock signal generator: after initialization, change the value every n time intervals:

```
module clockgen (clock);
   output clock;
   reg    clock;

   initial
      #5 clock = 1;

   always
      #50 clock = ~clock;

endmodule
```

Continuous Assignments

... either describe *fixed* connections between two wires, or how to create a new signal out of others:

- ```
  module everNAND (out, ina, inb);
    input ina, inb;
    output out;

    assign out = ~(ina&inb);

  endmodule
  ```

Note: Only *wires* can be used on the left hand side of a continuous assignment!

Procedural Assignments

... describe the assignment of a specific value to a register at a specific *time*:

- ```
  module idleNAND (out, ina, inb);
    input ina, inb;
    output out;
    reg out;

    always
      #20 out = ~(ina&inb);

  endmodule
  ```

Note: Only *registers* can be used for a procedural assignment!

Module Declarations

A module is declared following this scheme:

1. `module` *name (ports)*;
2. `input` *input-ports*;
3. `output` *output-ports*;
4. `reg` *output-registers*;
5. `wire` *local wires*;
6. `reg` *local registers*;
7. `assign` *wire* = *expression*;
8. *module* [*name*] *(ports)*;
9. `initial` *statement-list*
10. `always` *statement-list*
11. `endmodule`

Registers as Output-Ports

Ports are in general considered *wires*

- a register can never be an `input` port
- a register can feed an `output` port:

  ```
  output outreg;
  reg outreg;
  ```

  the output port `outreg` has always the value of the register `outreg`

$\Rightarrow$ A register cannot be altered by any other module than the one in which it is declared!

Vectors

... "bundle" a set of wires or registers

Example

a set of four registers can be defined as:

```
reg [3:0] regset;
```

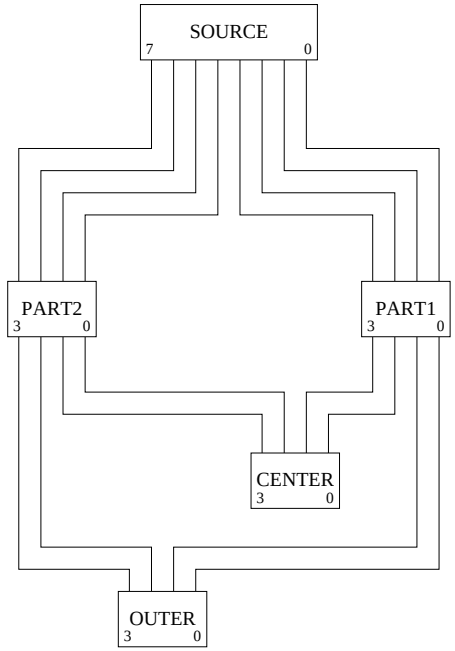
and accessed as `regset[0]` ... `regset[3]`

Vectors

- the numbering may be freely specified:
  `[3:0]` numbers $i \in \{3, 2, 1, 0\}$
  `[0:3]` numbers $i \in \{0, 1, 2, 3\}$
  `[6:9]` numbers $i \in \{6, 7, 8, 9\}$
  including negative indices
- subvectors can be specified on either side of an assignment, e.g. `regset[2:1]`
- scalars and vectors can be combined to new vectors with {`a, b, ...`}

Example:

```
module split_vector;
    reg  [7:0] source;
    wire [3:0] part1;
    wire [3:0] part2;
    wire [3:0] center;
    wire [3:0] outer;

    // connect parts with source
    assign part1 = source[3:0];
    assign part2 = source[7:4];

    // derive center and outer
    // from parts 1 and 2
    assign center = {part2[1:0],part1[3:2]};

    assign outer[3:2] = part2[3:2];
    assign outer[1:0] = part1[1:0];

    initial
    begin
        ...
    end

endmodule
```

Constant Signals

They may be specified in

- binary `1'b0`
- octal `3'o5`
- decimal `8'd135`
- hexadecimal `12'h3B7`

number preceding "'" is width of vector

Constant Signals

Constants may appear

1. on the right hand side of an assignment
2. as an input for a module port

Defaults:

- if no width → entire vector is used
- no base → decimal values are assumed