Writing parameterized modules

Oprițoiu Flavius
flavius.opritoiu@cs.upt.ro

October 30, 2025

Introduction

Objectives:

- Datapath and control path separation
- Understand how to write parameterized modules

Datapath

- Consists of elements processing the data: no decisions are taken
- Typical components: multiplexers, registers, Arithmetic and Logic Units (ALUs), counters
- Construct shared buses using three-state drivers

Control path

- Concerned with decision making
- Described in terms of state machines

Note: State-enabled components (such as registers, counters) can be part of the datapath, as well.

Reusable modules

Reusable modules are defined in terms of *parameters*, which can be redefined. In Verilog 2001 the module's parameters are specified in a dedicated section, marked by #(and) symbols.

The code bellow describes a parallel load register with parameterized width (no. of bits) and initialization value (register content after reset or clear).

```
module rgst #(
        parameter w = 8,        //register's width parameter; default of 8
        parameter iv = {w{1'b0}}//initialization value parameter
)(
        input clk,
        input rst_b,            //asynchronous reset; active low
        input [w-1:0] d,        //input data, on w bits
        input ld,               //synchronous load; active high
        input clr,              //synchronous clear; active high
        output reg [w-1:0] q    //register's content, on w bits
);

        always @ (posedge clk, negedge rst_b)
                if (!rst_b)
                        q <= iv;//set content to initialization value
                else if (clr)
                        q <= iv;//set content to initialization value
                else if (ld)
                        q <= d;
endmodule
```

Reusable modules (contd.)

In-line, explicit redefinition of module's parameters, in Verilog 2001, uses the following format:

```
module-name #(.parameter-name(value), ...)
    instance_name (.port-name(signal), ...)
```

Code bellow instantiate a 16-bit register, with an initialization value of 0

```
rgst #(
    .w(16)
) registru1 (
    .clk(clk), ...
);
```

Code bellow instantiate a 4-bit register, initialized 15

```
rgst #(
    .w(4),
    .iv(4'd15)
) registru2 (
    .clk(clk), ...
);
```

Solved problem

Constructing a register file using discrete registers

Exercise: Construct a 4x8 register file.

Solution: An *MxN* *register file* is a storage element organized as an array of **M** registers, each register having **N** bits. It permits simultaneous reading one internal register and writing one internal register (possibly the same).

A register file's interface includes the following connections:

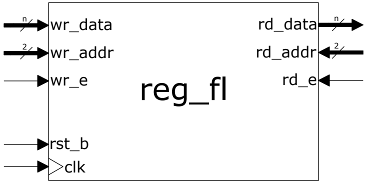
- an *N*-bit data input, for writing internal registers (*wr_data*)
- an *N*-bit data output, for reading internal registers (*rd_data*)
- an input address, selecting the register to be written (*wr_addr*)
- an output address, selecting the register to be read (*rd_addr*)
- data write enable signal (*wr_e*)
- data read enable signal (*rd_e*)

The enable lines of the writing/reading port are optional. **M** is, typically, of form 2^k: the input/output addresses use k bits.

Solved problem (contd.)

Constructing a register file using discrete registers

The interface of a *4xn* register file is depicted bellow



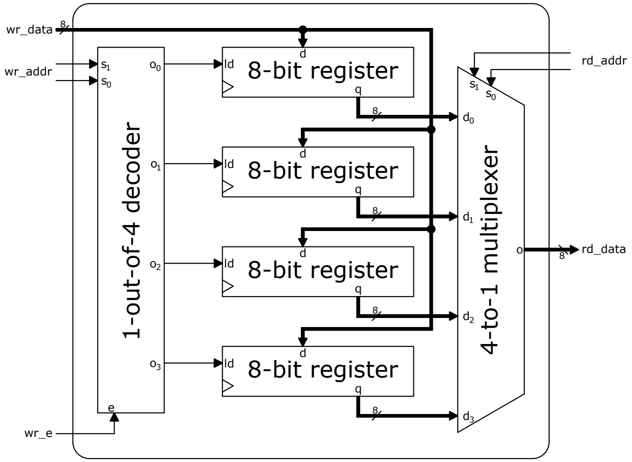
For this case, the interface is made up of:

- the writing port (*wr_data*, *wr_addr*, *wr_e*)
- the reading port (*rd_data*, *rd_addr*, *rd_e*)
- clock signal (*clk*)
- reset signal (*rst_b*)

Solved problem (contd.)

Constructing a register file using discrete registers

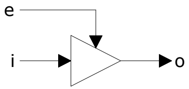
An *4x8* register file with no output enable line:



Note: The clock and reset lines were omitted for brevity.

Three-state driver

Used for connecting several components on a shared line or bus.



Output *o* is set to *i* when enable line, *e*, is active, and to *high impedance* otherwise. An output set to high impedance (symbolized by z, in Verilog) allows other component to drive the logic level of the shared line or bus it connects to.

The code fragment bellow demonstrates commanding a signal into high impedance by a control line, *e*:

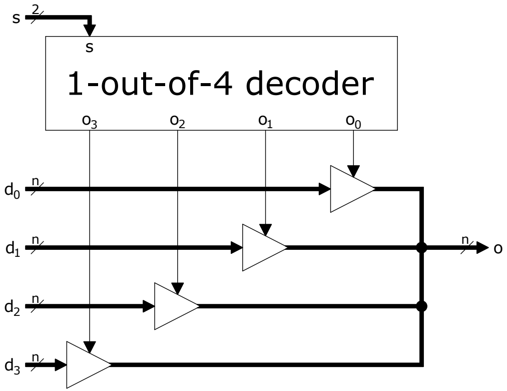
```
1    wire [15:0] data, data_hiz;
2    assign data_hiz = (e) ? data : 16'bz;
```

Because the high impedance symbol, z, is the most significant bit of the constant in line 2, it is extended to 16 high impedance bits.

Solved problem

Constructing a multiplexer using three-state drivers

An *n*-bit 4-to-1 multiplexer, implemented with three-state drivers:



Multiplexer's select input, *s*, drives a 1-out-of-4 decoder. The final stage connects all three-state drivers' outputs together.