FizZim – an open-source FSM design environment

Paul Zimmer
Zimmer Design Services

Michael Zimmer
Zimmer Design Services
(and University of California, Santa Barbara)

Brian Zimmer
Zimmer Design Services
Zimmertech
(and University of California, Davis)

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

Finite State Machine design is a common task for ASIC designer engineers. Many designers would prefer to design FSMs in a gui-based environment, but for various reasons no commercial tool for this task has really achieved wide-spread acceptance. The authors have written such a graphical FSM design tool, and offer it to the engineering community for free under the GNU public license. The gui is written in Java for portability, while the back-end code generation is written in Perl to allow for easy modification. The paper will describe the basic operation of the tool and the format of the Verilog it produces, then go on to describe some of the more advanced features and how they affect the Verilog output.

Table of contents

1 Introduction - What is fizzim?

Finite State Machines come up frequently in digital design. Sometimes designers code them directly in Verilog, but many designers prefer to design their FSMs as a state diagram ("bubbles and arrows"). This diagram must then be manually translated into Verilog.

For these designers, it would certainly be handy to design the FSM directly in a graphical tool and allow the software to generate the Verilog code. There have been several attempts by various EDA companies, large and small, to provide such a tool, but nothing has really gotten much traction.

This may be because the tool is in a strange niche. It is really too small to support business on an EDA scale, but it is too large for a "G-job". Also, the graphical part of the G-job is outside the usual experience of hardware designers.

So, it seems a good candidate for an open source project, provided *someone* is willing to tackle that nasty graphical part.

Someone has! Paul Zimmer and his young interns at Zimmer Design Services, Mike Zimmer and Brian Zimmer, are proud to present fizzim – an open-source, graphical FSM design environment.

Throughout this paper, it is assumed that the reader is familiar with FSMs and common FSM-related terms (such as Moore and Mealy). If the reader is unfamiliar with some of this material, just read through some of the papers in the "references" section.

2 Starting fizzim

The fizzim gui is written in java. It is distributed as a ".jar" (java archive) file. We run it using Sun Java Runtime Environment. Odds are that you already have this loaded for your browser, but if not you can download it from java.sun.com.

2.1 Windows

On most Windows machine, Java Runtime Environment will already be registered as the correct app for ".jar" files, so just double-clicking on the file should start it. If that doesn't work, you can start a terminal window and use the command-line approach as in Linux below.

2.2 Linux

On linux, try right-clicking the file and select "open using". If java runtime is listed, you're in business. You can also run from the command line using:

java –jar fizzim.jar

3 GUI basics

The gui is pretty intuitive. Right-click in open space gives you a menu to create new states and transitions. Right-click on an object gives you a menu to edit the object. Double (left) click on an object will bring up the properties menu for that object.

Left-click and drag to move an object. You can group objects for moves by either selecting them with a “box” or using click, ctrl-click.

Edit>undo or ctl-Z will undo, Edit>redo or ctl-Y will redo. Undo/redo is unlimited.

4 Attributes

It is our belief that few hardware engineers will want to touch the gui, but many will want to modify the Verilog output. In recognition of this, every attempt has been made to try to keep the gui as independent of the Verilog generation as possible.

To accomplish this, virtually everything is implemented as “attributes”. This should allow new backend (Verilog-generation) features to be added without touching the gui. Also, while the gui is written in Java, the backend is in the linga-franca of EDA – perl.

There are only 3 types of objects to the gui – the state machine itself, states, and transitions. Each of these can have attributes assigned to it. But state and transition object attributes have to be defined first in the global “states” and “transitions” attribute menus before they will be available in individual states and transitions. The gui knows about a few special attributes, but only those that require that the display be modified. Examples include transition equations (drop the “equation =” on the visible text) and output types (use “=” for combinational and “<=” for registered).

Inputs and outputs are just attributes. The name field is the name of the input or output signal.

Each attribute has 5 fields:

- Attribute Name – this is the name of the input or output, or the name of the special attribute.
- Default Value – Default value of the attribute. Will be used if no value is assigned in a state/transition.
- Visibility – Turns on/off visibility on the display. “Only non-default” means to only show the attribute if its value doesn’t match “Default Value”.
- Type – Information about the attribute. Inputs currently have no defined type, outputs can be “reg”, “regdp”, or “comb”. Others are attribute-specific.
- Comment – An optional comment that will show up on the diagram, in the Verilog, both, or neither, depending on the attribute.
- Color – Text color.

5 Encodings

There are two primary types of state encodings used for FSM design. Highly encoded FSMs use a dense binary code and few flops but can sometimes have very complex combinational logic. One-hot FSM encodings, on the other hand, use a sparse code and many flops, but usually have much simpler combinational logic. There are many papers on the advantages and disadvantages of each (reference [2] is one example).

The backend perl script (fizzim.pl) supports both of these encodings.

5.1 Highly Encoded with Registered Outputs as Statebits (HEROS)

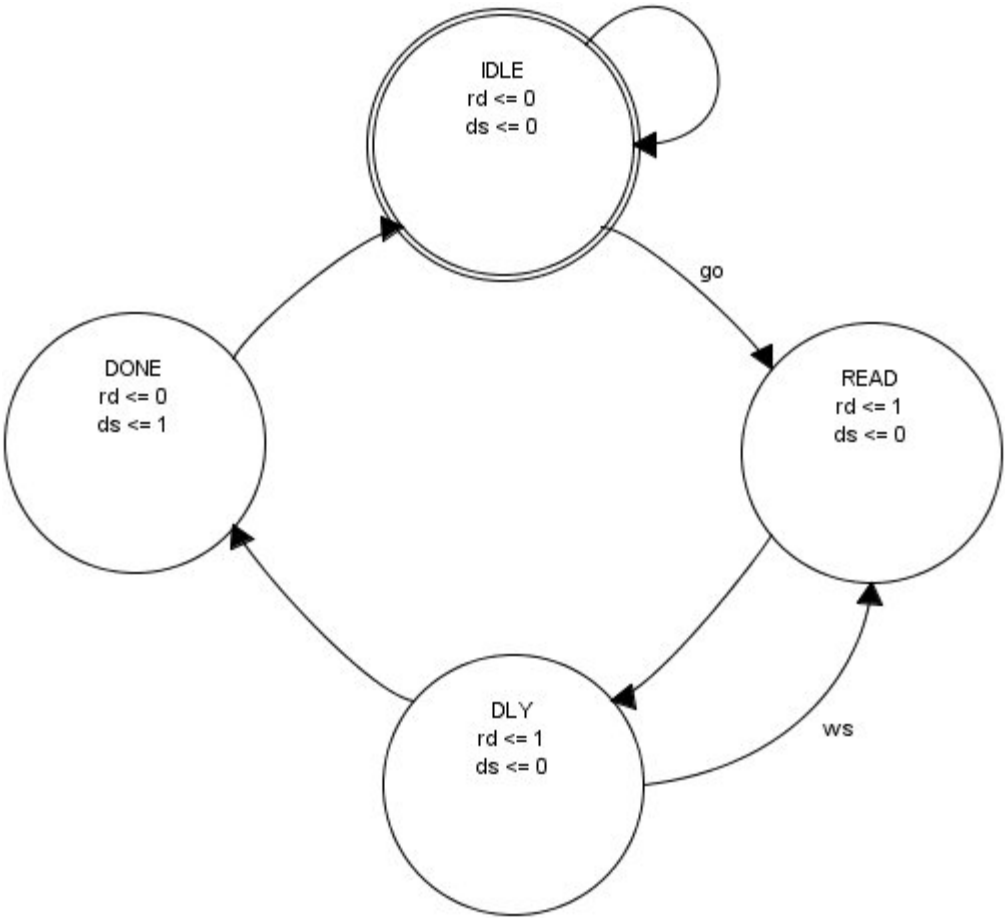
Heros is an encoding that uses a dense binary code. As the name implies, registered outputs will be encoded into the states to minimize flop count. There are mechanisms (discussed below) to allow particular outputs to be excluded from the state vector. The actual Verilog format is based on recommendations from Cliff Cummings' paper (reference [3]).

5.2 One Hot

One-hot encoding is also supported. The Verilog format is based on Steve Golson's paper (reference [2]). Some features, such as gray coded transitions, are not available with one-hot encoding.

6 Cliff's Classic

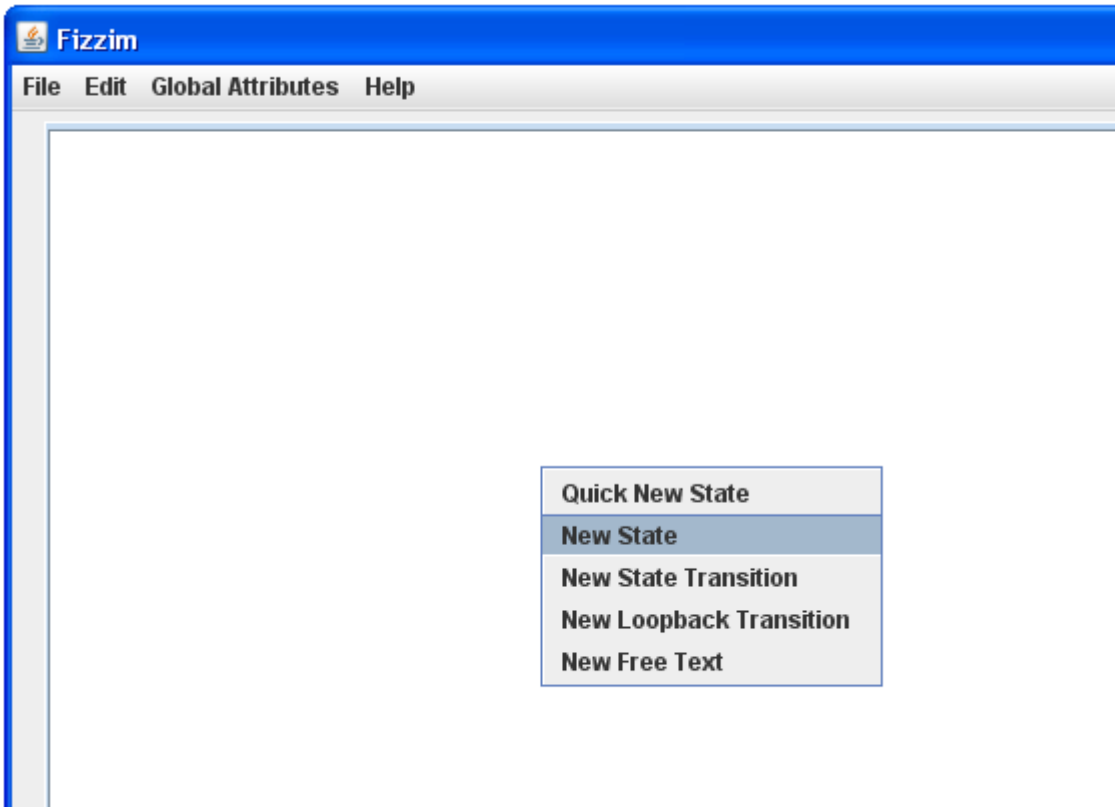
Let's jump right in with an example. In [3], Cliff Cummings introduced the following basic state machine:



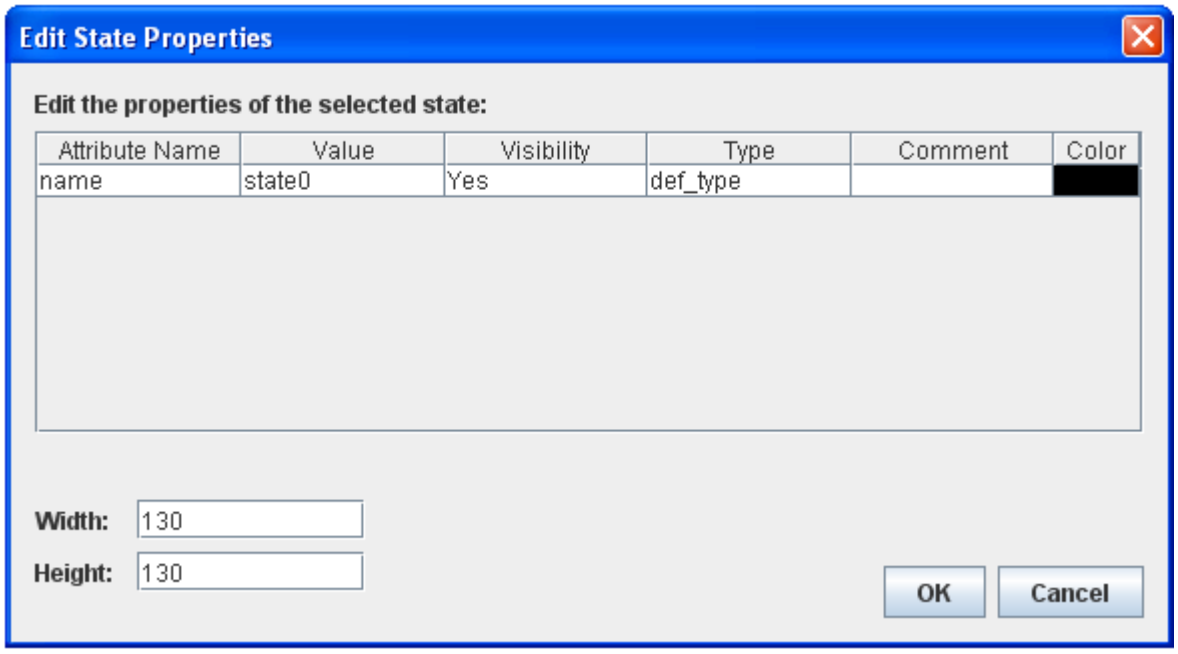
Here's how we would create this in fizzim.

6.1 Creating the states

Right-clicking in open space gives the following menu:

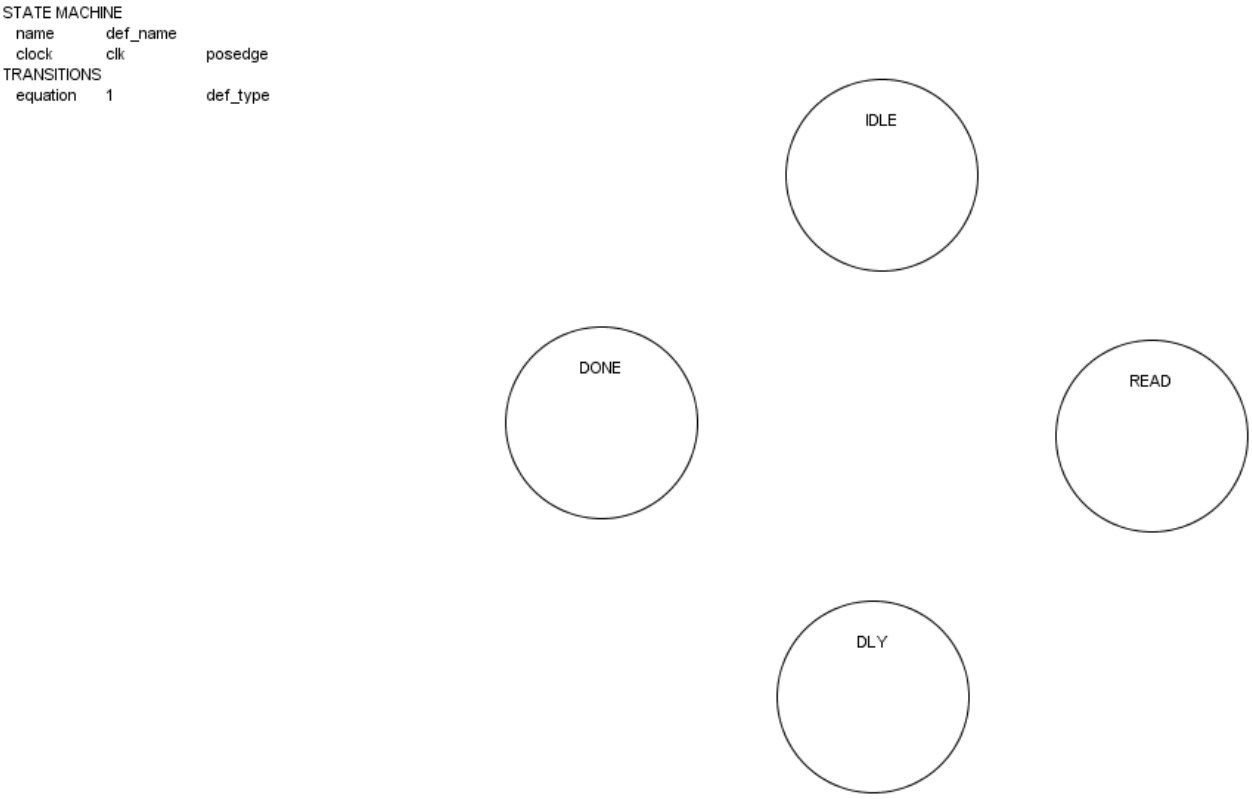


We select “New State” and get this:



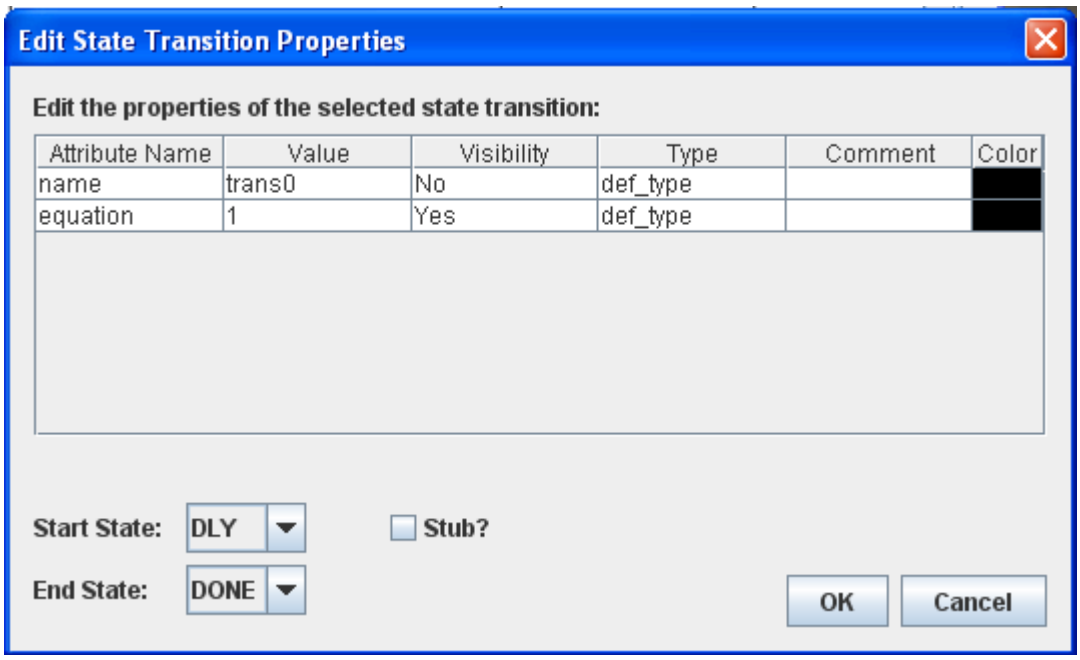
Change the state name to “IDLE” and hit “OK”.

Repeat this to add the other three states. Left-click and drag to move the states around.

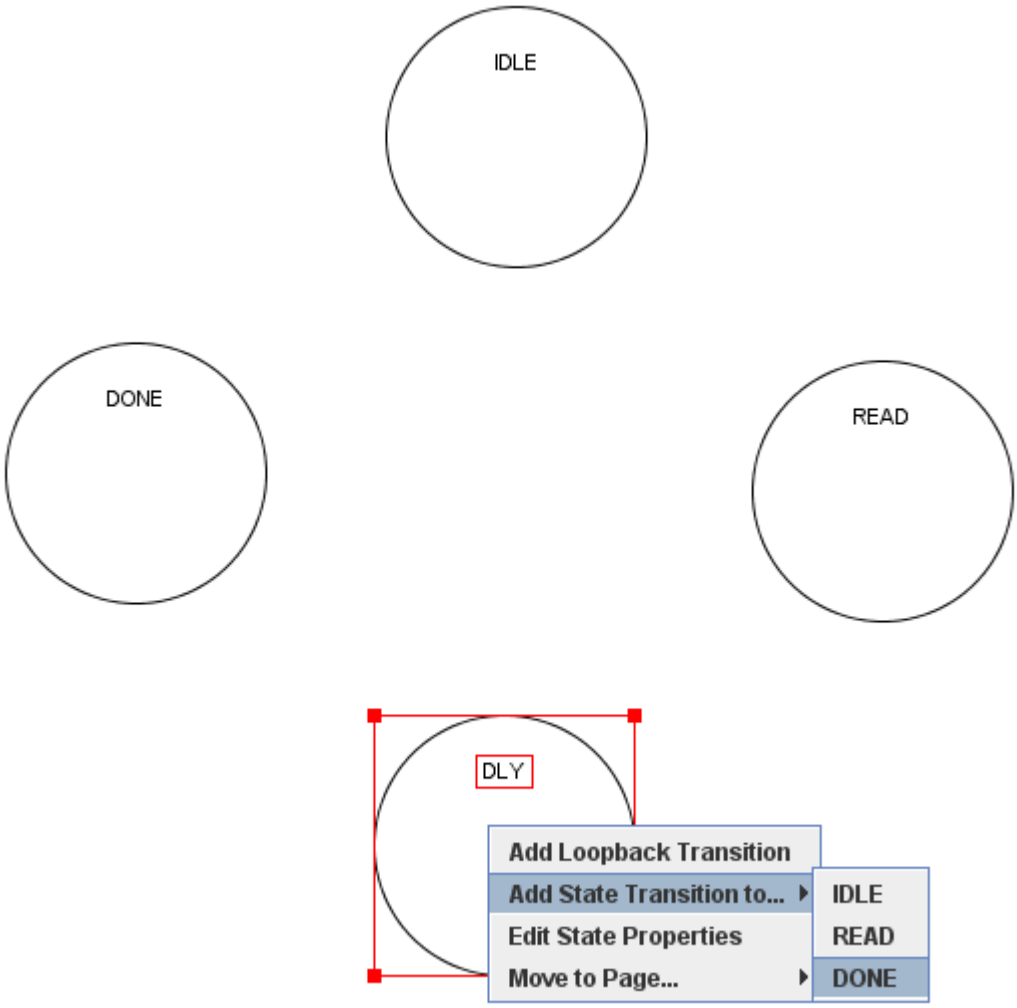


6.2 Creating the transitions

To create the state transitions, we can either right-click in open space and select “New State Transition” and get the full menu:



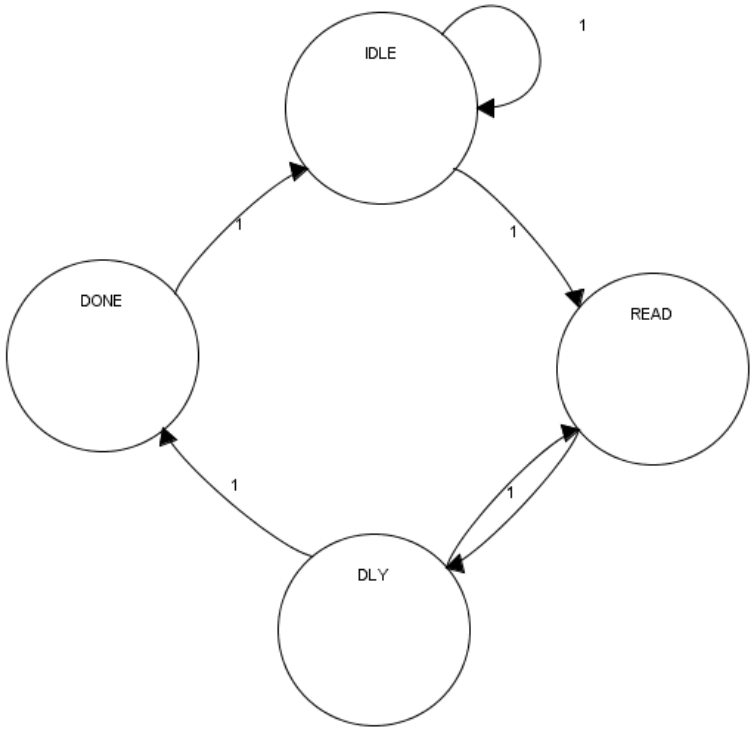
Or we can right-click on the start state and select “Add State Transition to”:



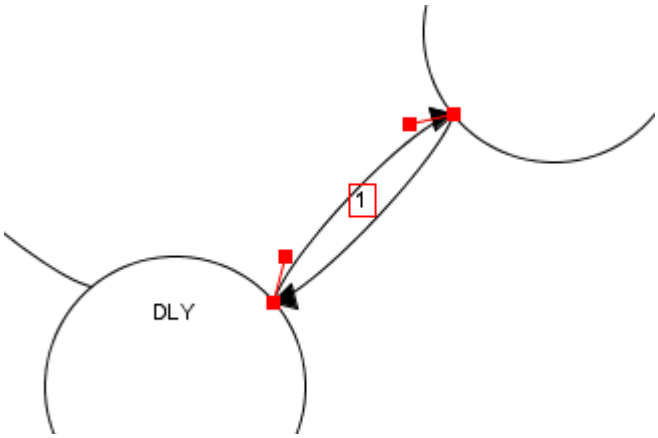
We repeat this to add all the transitions. Don’t forget to add the loopback transition. We’ll see why this matters in a moment.

Notice that when we add the transition from DLY back to READ, we get something like this:

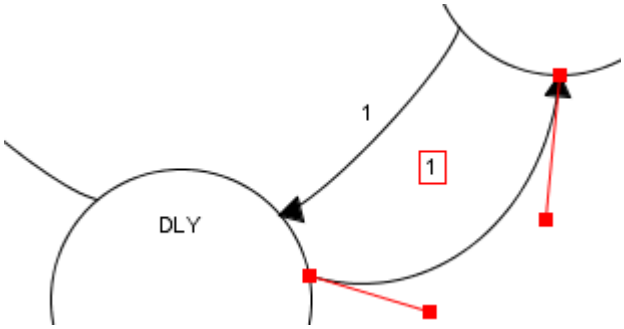
```
STATE MACHINE
  name        def_name
  clock       clk             posedge
TRANSITIONS
  equation    1               def_type
```



That doesn't look so great, so we need to move one of the transitions. To do this, left-click to select it. Endpoints and anchorpoints appear:



Drag the endpoints to a new location, then drag the anchorpoints to reshape the curve. The anchorpoints on the ends of the arc control where the arc intersects the state bubble. The other two control the shape of the curve.



If you move a state bubble, the attached arcs will move with it. As long as the move isn't too drastic, the anchorpoint modifications you made will be retained. If you move the state a lot, the anchorpoints may get reset. This works better than it sounds. Mostly your anchorpoints are retained when it makes sense.

All text, including the transition equation (the "1" above), output values in states, state names, and free text, can be moved by just selecting it and moving it.

Don't forget to add the loopback transition. We'll see why this matters in a moment.

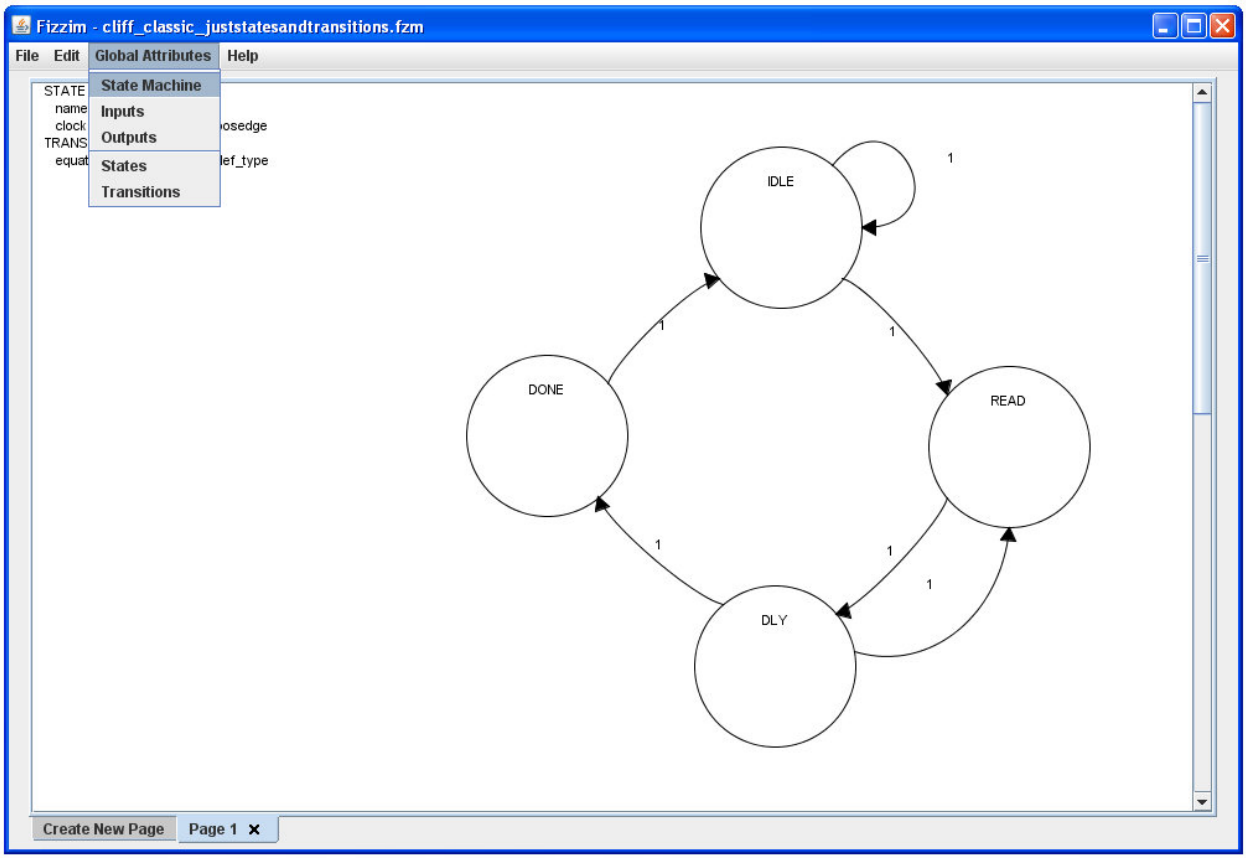
6.3 Filling in the details

6.3.1 Global Attributes

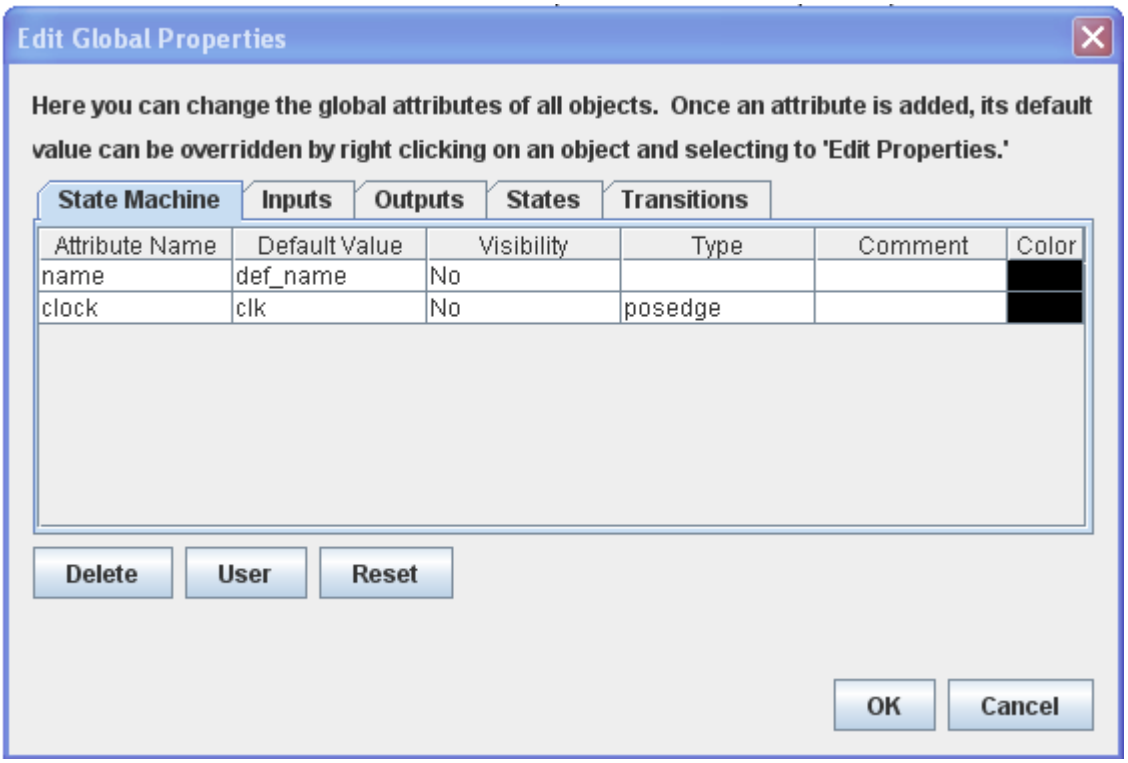
Recall that everything is stored as attributes – either attributes on the FSM itself or attributes on individual states and transitions. So, adding inputs, outputs, transition equations, etc is a matter of editing attributes.

Let's start with the global FSM attributes. It is necessary to start here, because the individual state and transition attributes won't appear until they are entered as global attributes.

Select “Global Attributes > State Machine” from the top menu:

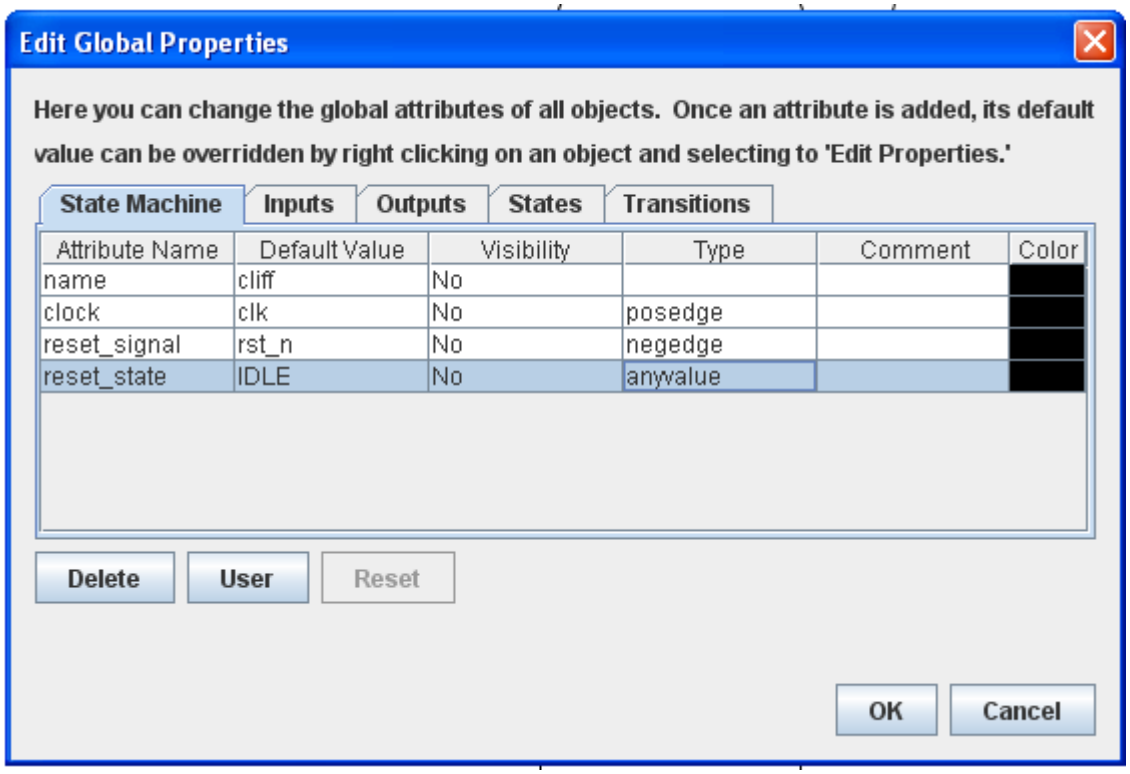


And you get this:



Edit the fields to fill in the module name “cliff”, the clock name “clk”, and make it a posedge clk.

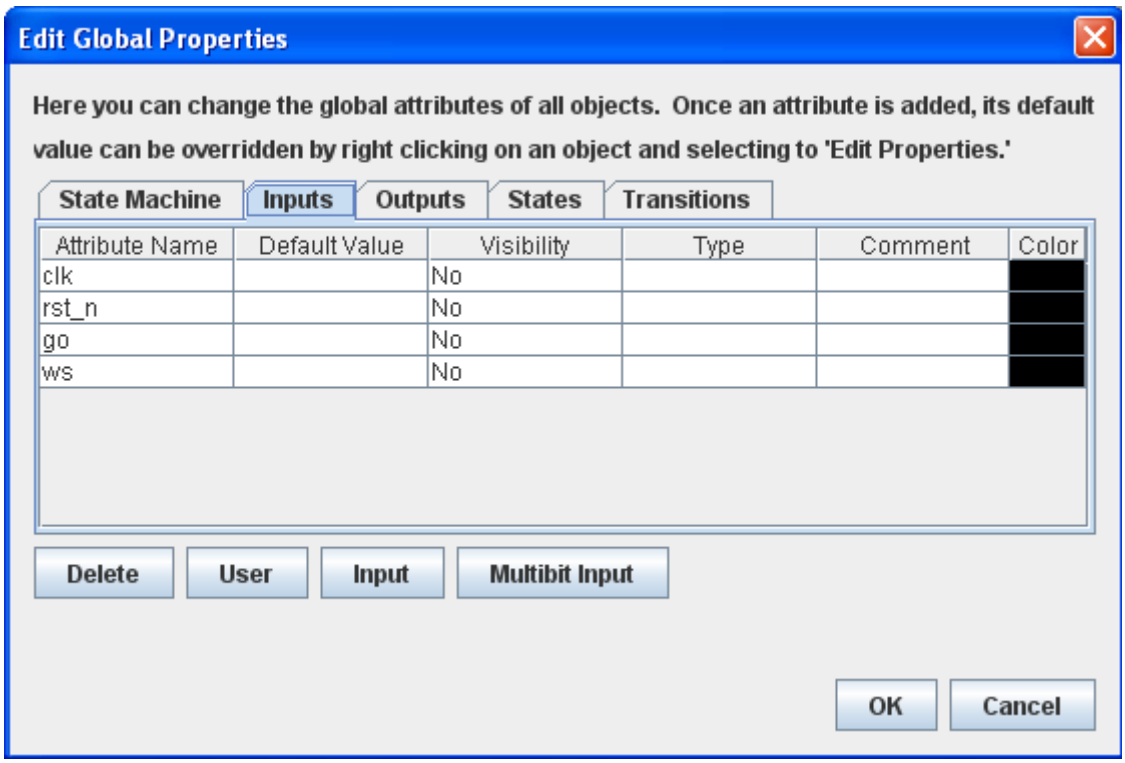
Click the “Reset” button, and two more attributes appear. One is “reset_signal”. Change this to “rst_n”, negedge. Set “reset_state” to IDLE via the pull-down menu and set its type to “anyvalue” (“allzeros” and “allones” will force the reset state to be all zeros or all ones, but this isn’t compatible with onehot encoding, so we won’t use it on this example).



Hit OK. Notice that IDLE now has a double ring to indicate it is the reset state.

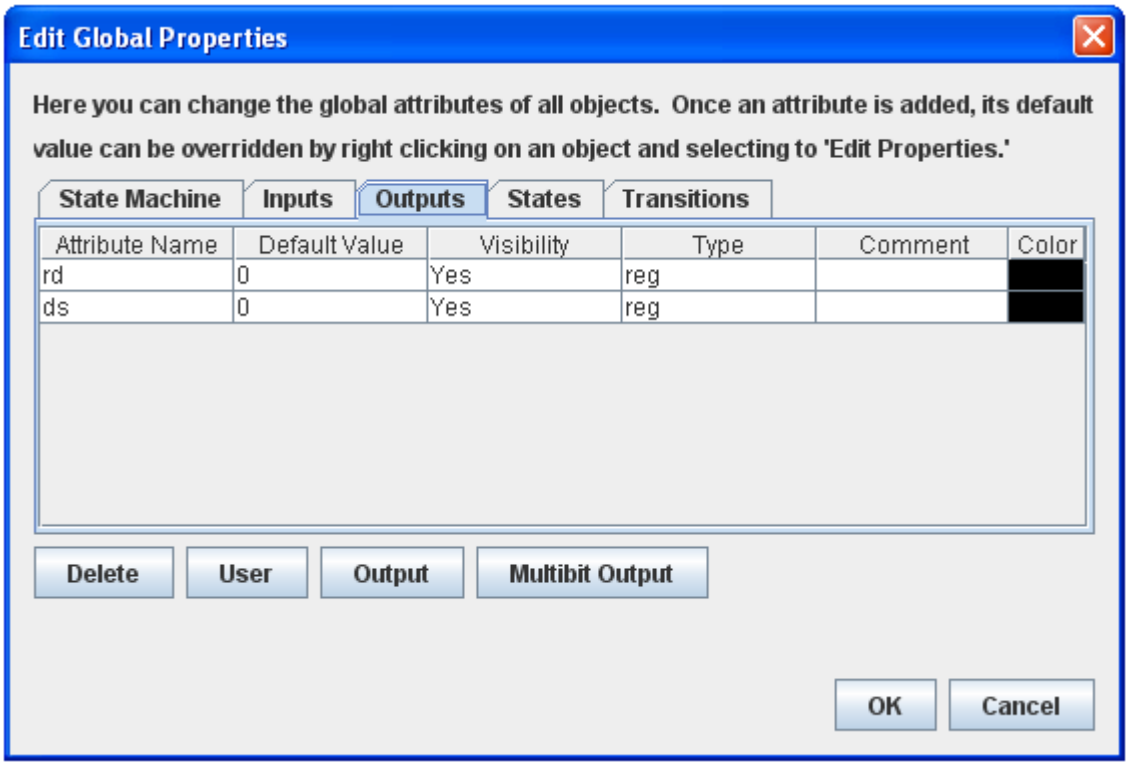
Now select “Global Attributes > Inputs” from the top menu.

Use the “Input” button to add the inputs:



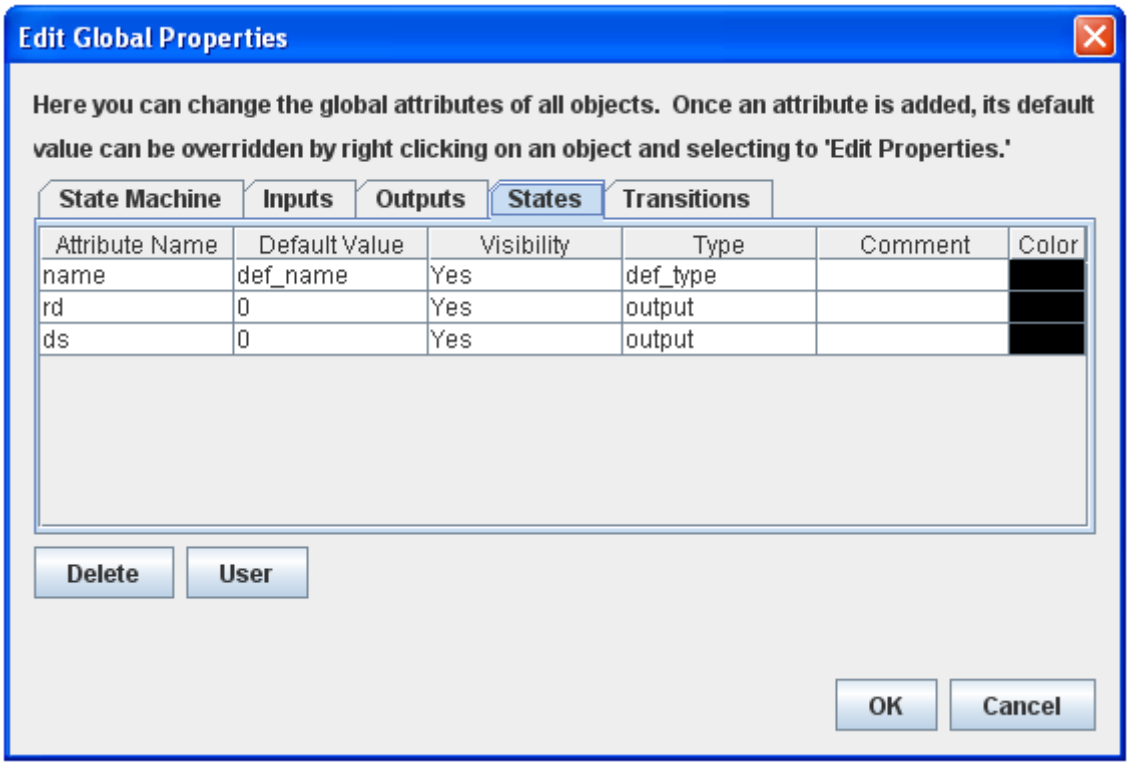
Note that “type” doesn’t matter for inputs. We could click OK, then reselect “Global Attributes > Outputs” from the top menu, or we can just switch to the “Outputs” tab without exiting the menu.

Click “Output” twice to add the two outputs, “rd” and “ds”. Their type field should be “reg”. Set “Default Value” to 0, and visibility “Yes”.

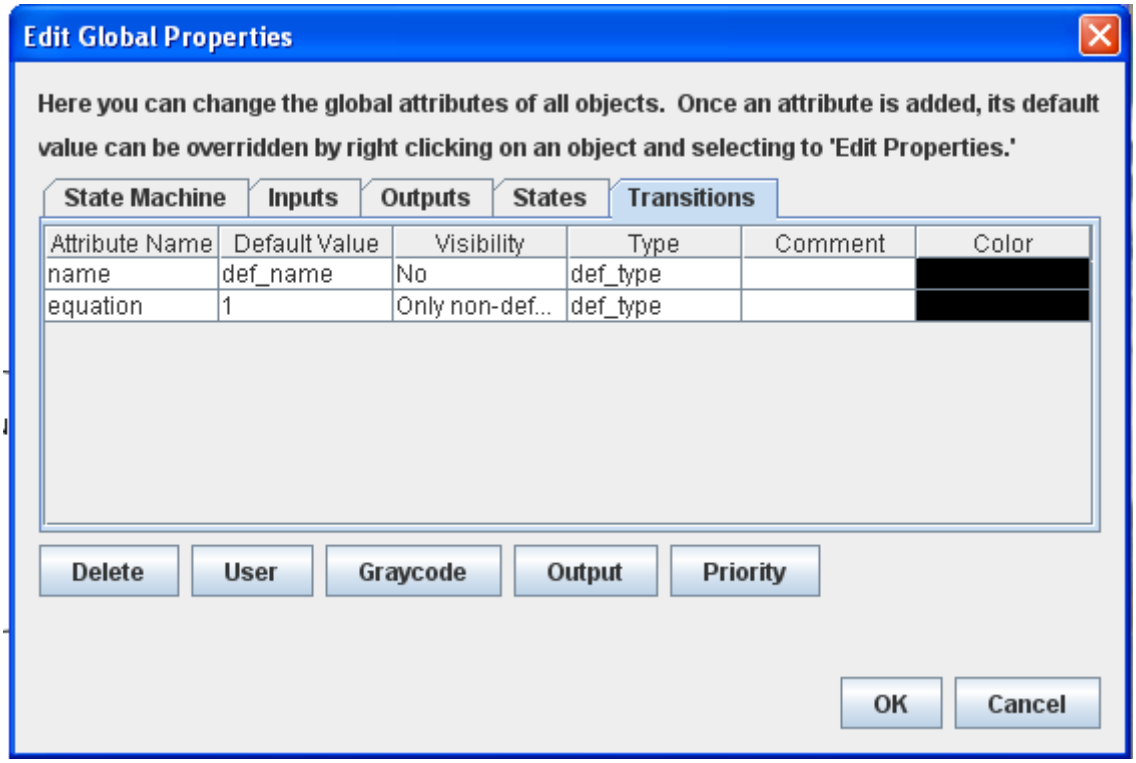


This will become clearer later, but type “reg” means that they are registered outputs (Moore) and that they should be encoded as state bits.

Now flip over to the “States” tab. “rd” and “ds” now appear as state attributes. This means you will be able to assign particular values to them in particular states.



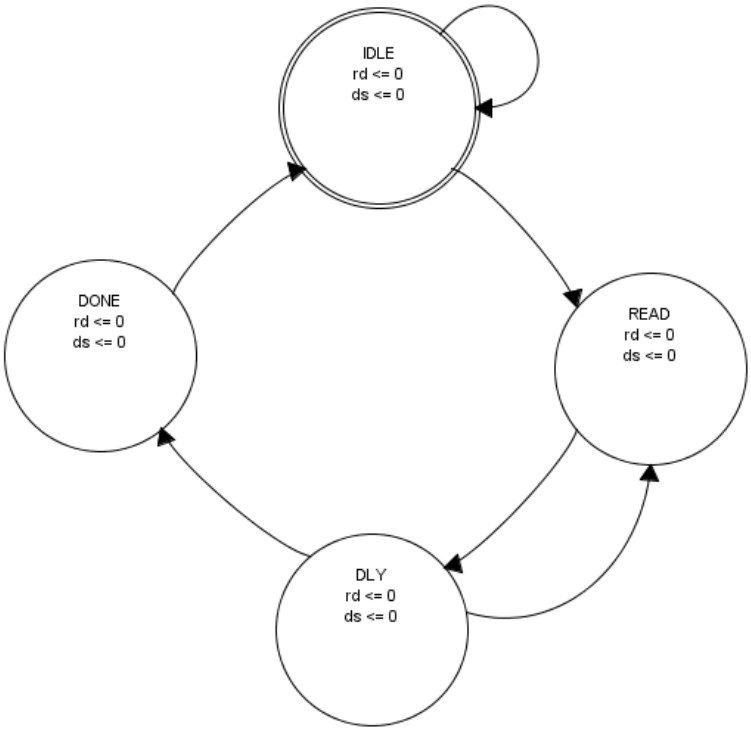
Flip over to the “Transitions” tab. “rd” and “ds” do NOT appear here, because it makes no sense to define registered outputs on a transition. The standard attribute “equation” DOES appear here, with the default value of “1”. Leave it alone. But you can change the “Visibility” field to “Only non-default” to make the “1” equations not show up on the diagram.



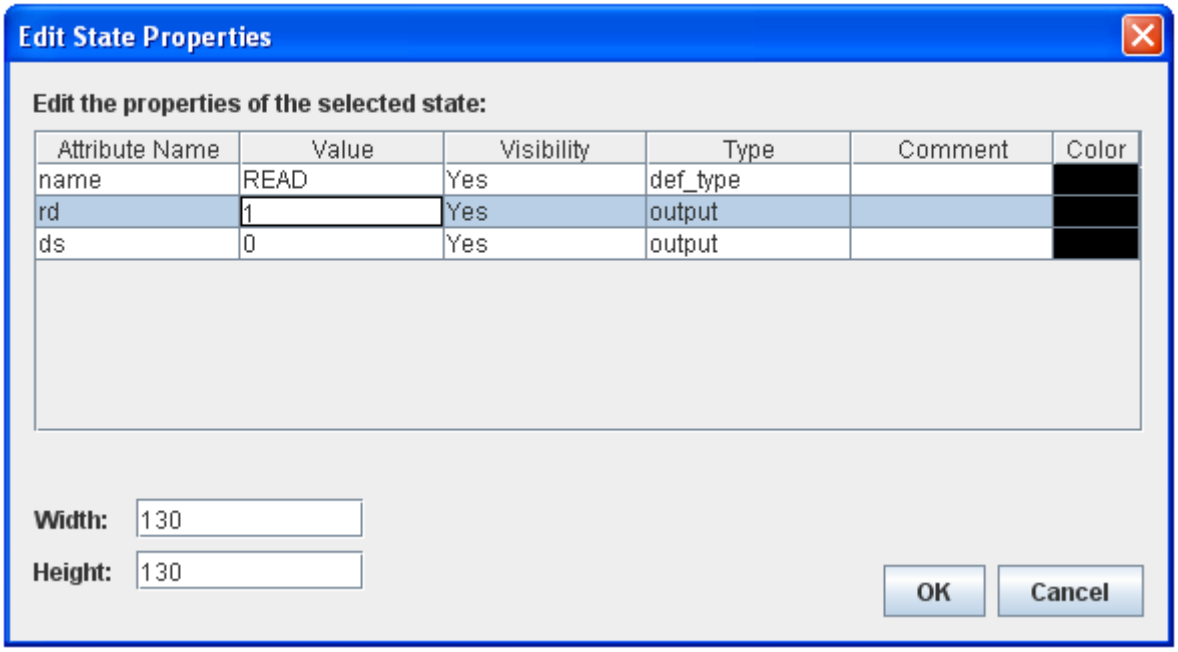
6.3.2 Individual State Attributes

Now we can enter the output values into the states. Notice that the outputs now appear on the states with a "<=" after them. This indicates registered outputs ("=" means combinational).

```
STATE MACHINE
  name          cliff
  clock         clk       posedge
  reset_signal  rst_n     negedge
  reset_state   IDLE      anyvalue
INPUTS
  clk
  rst_n
  go
  ws
OUTPUTS
  rd            0         reg
  ds            0         reg
STATES
  rd            0         output
  ds            0         output
TRANSITIONS
  equation      1         def_type
```



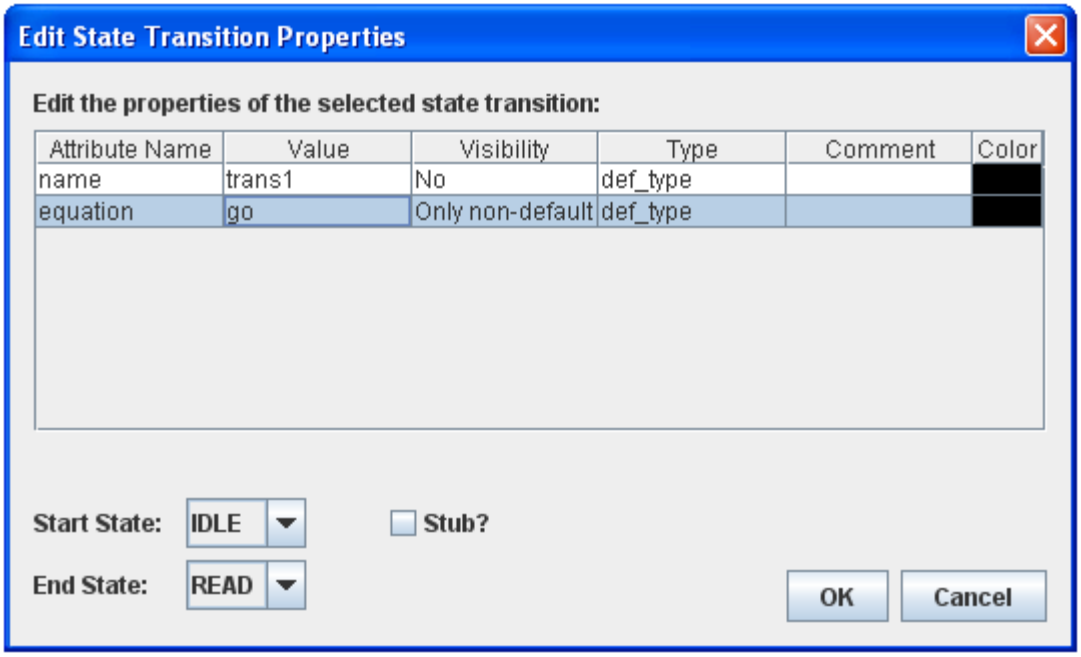
Now we need to enter the non-default values for rd and ds. Right-click on the READ state and select “Edit State Properties” to bring up the menu. Or just double-click the READ state bubble. Change the value of rd to “1”.



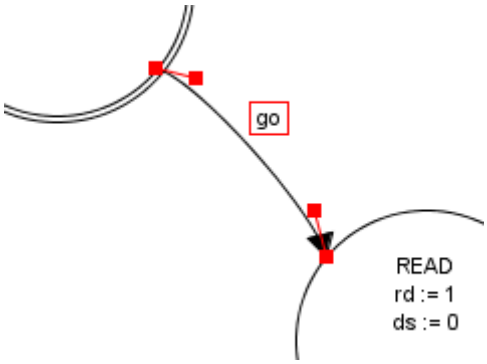
Do this for the other states to add appropriate output values (rd = 1 in DLY, ds = 1 in DONE).

6.3.3 Individual Transition Attributes

Double-click on the IDLE to READ transition to bring up the transition menu. Change the equation to “go”.



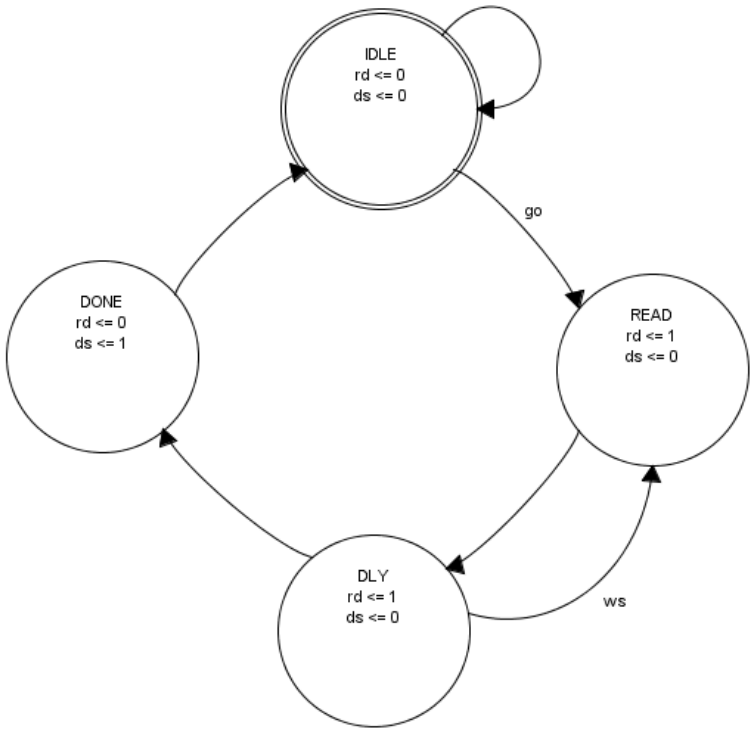
Hit “OK”. Now click on the “go” text and move it:



Repeat this for the state transition from DLY back to READ that has an equation of “ws”.

Our final state diagram looks like this.

```
STATE MACHINE
  name          cliff
  clock         clk       posedge
  reset_signal  rst_n     negedge
  reset_state   IDLE      anyvalue
INPUTS
  clk
  rst_n
  go
  ws
OUTPUTS
  rd            0         reg
  ds            0         reg
STATES
  rd            0         output
  ds            0         output
TRANSITIONS
  equation      1         def_type
```



You might have noticed that I did not put an explicit "!go" on the IDLE loopback transition, nor an explicit "!ws" on the DLY to DONE transition. That is because fizzim understands that a transition with an equation of "1" is the default, lowest priority, transition. This will be explained in the section on transition priorities. You *can* add the explicit equations, but you don't have to.

6.4 Output using heros

Now we can run the backend and generate code:

fizzim.pl < cliff.fzm > cliff.v

The default encoding is heros. Take a look at the output.

It is structured as two "always" blocks per [2]. The first one is combinational and does the next state determination, and the second is sequential and just infers the flops. See [2] for an explanation of why this is the preferred implementation.

Let's look at the output code in detail.

First, the module statement:

```
module cliff (
  output wire ds,
  output wire rd,
  input wire clk,
  input wire go,
  input wire rst_n,
  input wire ws );
```

Nothing special there, except that it uses the Verilog 2001 format.

Now look at the state encoding:

```
  // state bits
  parameter
  IDLE = 3'b000, // extra=0 rd=0 ds=0
  DLY  = 3'b010, // extra=0 rd=1 ds=0
  DONE = 3'b001, // extra=0 rd=0 ds=1
  READ = 3'b110; // extra=1 rd=1 ds=0

  reg [2:0] state;
  reg [2:0] nextstate;
```

Recall that the heros format uses registered outputs as state bits. Fizzim.pl has assigned state bit 0 to “ds”, and state bit 1 to ‘rd”. There are only four states, but DLY and READ both have state[1:0] equal to 01, because they have identical values of “ds” and “rd”. fizzim.pl recognizes this, and adds an “extra” bit to distinguish these states. Thus, we end up with 3 state bits to cover 4 states, but since the registered outputs are encoded in the states, we still have fewer flops overall. It is possible to force fizzim.pl to pull the output bits out of the state vector by changing their type to “regdp”. See the section on datapath outputs below.

Also note that the IDLE state ended up as all zeros. In the absence of a requirement that would prevent this, fizzim.pl heros encoding will favor the reset state as all zeros.

Next comes the combinational always block:

```
  // comb always block
  always @* begin
    // Warning: Neither implied_loopback nor default_state_is_x attribute is
set on state machine - this could result in latches being inferred
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
        end
        else begin
          nextstate = IDLE;
        end
      end
      DLY: begin
        if (ws) begin
          nextstate = READ;
        end
        else begin
          nextstate = DONE;
        end
      end
      DONE: begin
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        begin
          nextstate = DLY;
        end
      end
    endcase
  end
```

Pretty straightforward, and just what you would probably write if you were coding this by hand. There's a big case statement on "state", and the inputs (go and ws) determine "nextstate". But notice the warning message.

```
// Warning: Neither implied_loopback nor default_state_is_x attribute is set
on state machine - this could result in latches being inferred
```

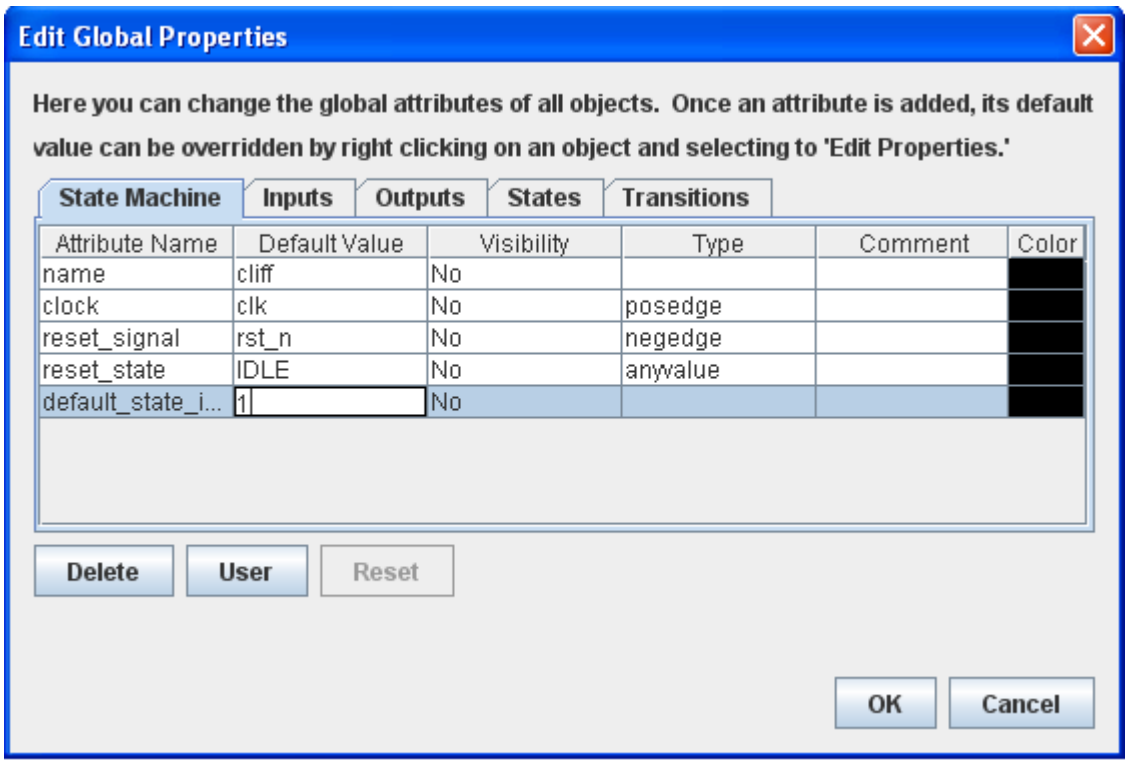
We have come to a philosophical fork in the road.

Some people, including Cliff Cummings, like to make the default value of the nextstate vector equal to "X" before executing the "case" statement. This ensures that bad things will happen in simulation if the case statement is wrong, but it also means that all loopback conditions need to be entered explicitly.

Other people prefer to make nextstate equal to current state before executing the case statement. This means that the default action is loopback, so no explicit loopbacks are required.

Fizzim.pl is philosophically neutral on this (and most other such issues), so you can choose which way you want it. This is done by setting an attribute on the FSM – either "default_state_is_x" or "implied_loopback".

Since this is Cliff's state machine, we'll do it Cliff's way. Select "Global Attributes > State Machine" and click the "User" button. Enter the attribute name "default_state_is_x" and give it a value of "1":



Save the file and re-run fizzim.pl. The warning message goes away and the combinational block starts like this:

```
  // comb always block
  always @* begin
    nextstate = 3'bx; // default to x because default_state_is_x is set
    case (state)
      IDLE: begin
```

By the way, if we had used "implied_loopback" (create attribute "implied_loopback" and set it to 1), the output would have looked like this:

```
  // comb always block
  always @* begin
    nextstate = state; // default to hold value because implied_loopback is
set
    case (state)
      IDLE: begin
```

Continuing with our tour of the heros output, we next have the code that assigns the outputs to state bits:

```
// Assign reg'd outputs to state bits
assign ds = state[0];
assign rd = state[1];
```

Then the sequential always block. Recall that we set the “reset_signal” attribute to “rst_n” and it’s type as “negedge”. The “reset_state” was set to “IDLE”:

```
// sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    state <= IDLE;
  else
    state <= nextstate;
end
```

If we had instead chosen the type as “negative”, we would have gotten an active-low *synchronous* reset:

```
// sequential always block
always @(posedge clk) begin
  if (!rst_n)
    state <= IDLE;
  else
    state <= nextstate;
end
```

The final bit of code is for simulation purposes and will be explained in “Ascii state name” below.

6.5 Output using onehot

The onehot encoding is based on Steve Golson’s paper [2]. This technique doesn’t really allow for the “default_state_is_x” behavior, so this attribute is ignored.

fizzim.pl –enc onehot < cliff.fzm > cliff.v

Skipping over the module statement, here's what our "state encoding" looks like:

```
// state bits
parameter
IDLE = 0,
DLY  = 2,
DONE = 1,
READ = 3;

reg [3:0] state;
reg [3:0] nextstate;
```

Recall that onehot encoding uses one bit for each state. So, 4 states means 4 bits. The parameter refers to the bit position in the vector. So, when the FSM is in state DONE, for example, only bit 1 will be set (the state vector will be 0010).

The combinational always block looks equally bizarre:

```
// comb always block
always @* begin
  nextstate = 4'b0000;
  case (1'b1) // synopsys parallel_case full_case
    state[IDLE]: begin
      if (go) begin
        nextstate[READ] = 1'b1;
      end
      else begin
        nextstate[IDLE] = 1'b1;
      end
    end
    state[DLY]: begin
      if (ws) begin
        nextstate[READ] = 1'b1;
      end
      else begin
        nextstate[DONE] = 1'b1;
      end
    end
    state[DONE]: begin
      begin
        nextstate[IDLE] = 1'b1;
      end
    end
    state[READ]: begin
      begin
        nextstate[DLY] = 1'b1;
      end
    end
  endcase
end
```

The “case (1’b1)… state[IDLE]” gets translated to mean “when the IDLE bit of the state vector (bit 0) is a 1”. The nextstate is calculated by first setting it to all zeros, then turning on the bit that represents the next state.

Note that, because of the way it is coded (set to all zeros, then set the bit), the issue of defaulting the value doesn’t arise for onehot. If something goes wrong, you get an illegal all-zeros state which you never get out of.

The sequential always block looks like this:

```
// sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    state <= 4'b0001 << IDLE;
  else
    state <= nextstate;
end
```

It seems simpler to just set state to zero, then set state[IDLE] to one, but this format was used to stay as close as possible to Steve Golson’s code in [3]. His “1 << IDLE” got changed to have the full vector size to work around a bug in one of the Verilog simulators.

Note that there is now a *third* always block. It is a sequential always block, and creates the registered outputs. This is necessary because, unlike heros encoding, there is no way to use the state bits for registered outputs. The block looks at the value of “nextstate” and sets ds and rd accordingly:

```
// datapath sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    ds <= 0;
    rd <= 0;
  end
  else begin
    case (1'b1)
      nextstate[IDLE]: begin
        ds <= 0;
        rd <= 0;
      end
      nextstate[DLY]: begin
        ds <= 0;
        rd <= 1;
      end
      nextstate[DONE]: begin
        ds <= 1;
        rd <= 0;
      end
      nextstate[READ]: begin
        ds <= 0;
        rd <= 1;
      end
    endcase
  end
end
```

This structure is also used for registered datapath ("regdp") outputs (coming soon).

6.6 Ascii state name

Notice that both heros and onehot had some extra simulation code at the end. The code for onehot looks like this:

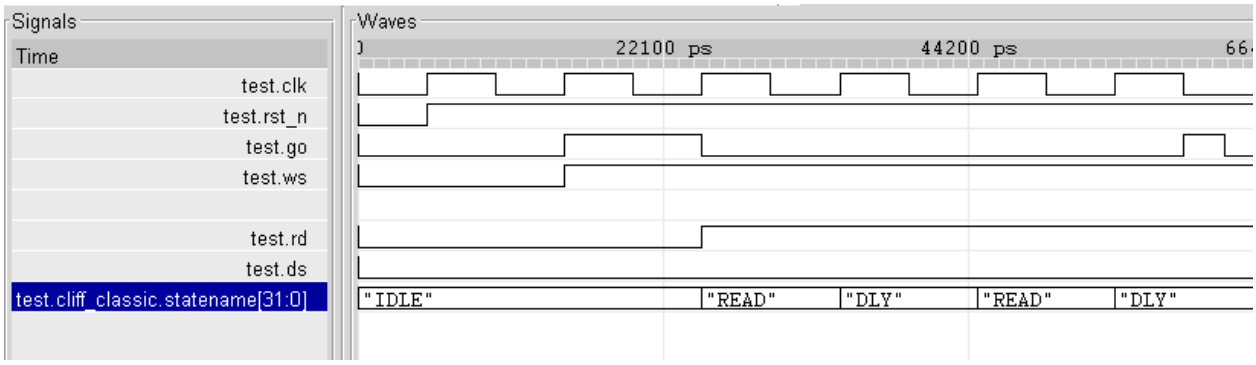
```
// This code allows you to see state names in simulation
`ifndef SYNTHESIS
reg [31:0] statename;
always @* begin
  case (1'b1)
    state[IDLE]:
      statename = "IDLE";
    state[DLY]:
      statename = "DLY";
    state[DONE]:
      statename = "DONE";
    state[READ]:
      statename = "READ";
    default:
      statename = "XXXX";
  endcase
end
`endif
```

This code allows the designer to see the ascii state name in simulation (set the data type to ascii in your waveform viewer), but does not affect synthesis. The "`ifndef SYNTHESIS/`endif" replaces the old "//synopsys translate on/off" syntax for making this simulation-specific (thanks to Cliff Cummings for pointing this out).

Equivalent code is generated for heros.

```
// This code allows you to see state names in simulation
`ifndef SYNTHESIS
reg [31:0] statename;
always @* begin
  case (state)
    IDLE:
      statename = "IDLE";
    DLY:
      statename = "DLY";
    DONE:
      statename = "DONE";
    READ:
      statename = "READ";
    default:
      statename = "XXXX";
  endcase
end
`endif
```

Here's an example of what this looks like:



This can be turned off by specifying the "-nosimcode" option on fizzim.pl.

6.7 (Un)Displaying the attributes table

Notice that most of the examples so far have had the attributes table to the left of the state machine. This is a handy feature, but you don't have to use it. To turn it off, do "File > Preferences" and uncheck the "Table Visible" box.

Alternatively, you can move the table to another (or its own) page. See the section on multiple pages in the tutorial.

7 Mealy outputs

Combinational outputs (Mealy outputs) are also supported. They are distinguished from sequential outputs by setting the type field to “comb”.

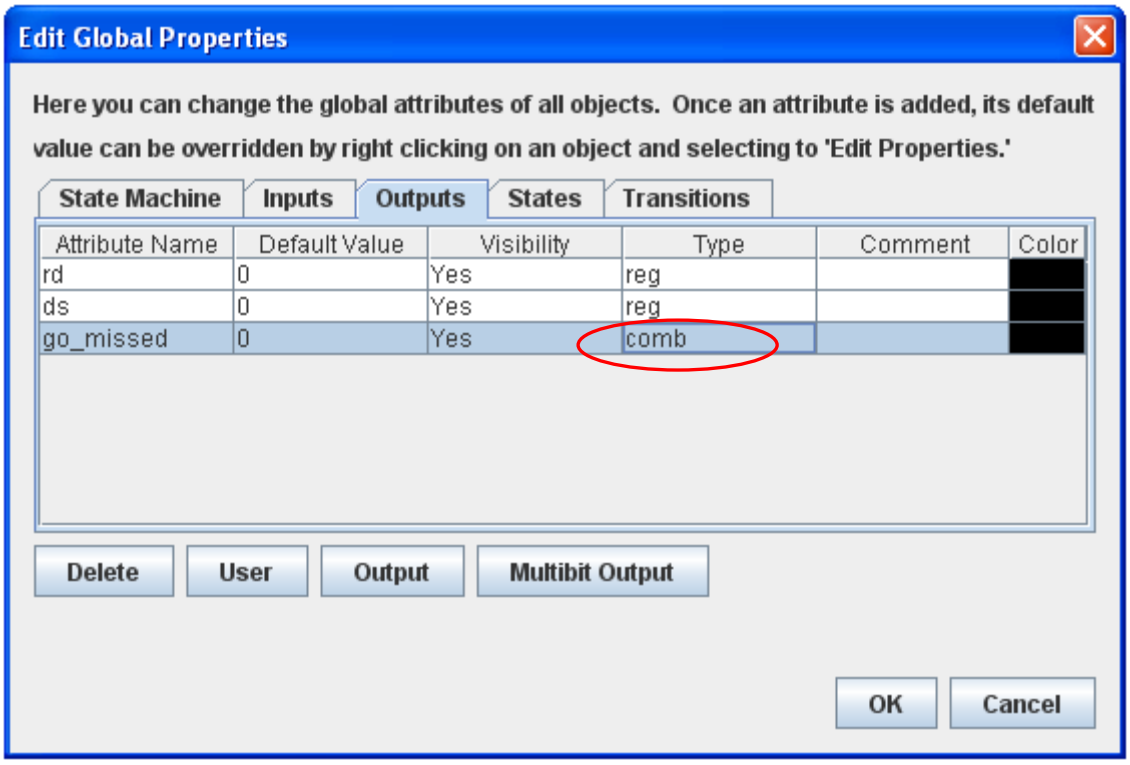
A Mealy output is defined as an output which is dependent on both the state and the inputs. There are two ways to describe a Mealy output. One way, which derives directly from the definition, is to specify the combinational equation that describes the output *for each state*. The other way is to specify the combinational equation that describes the output *on each transition*. Fizzim supports either style.

Let’s add a Mealy output to Cliff’s state machine using the on-states method.

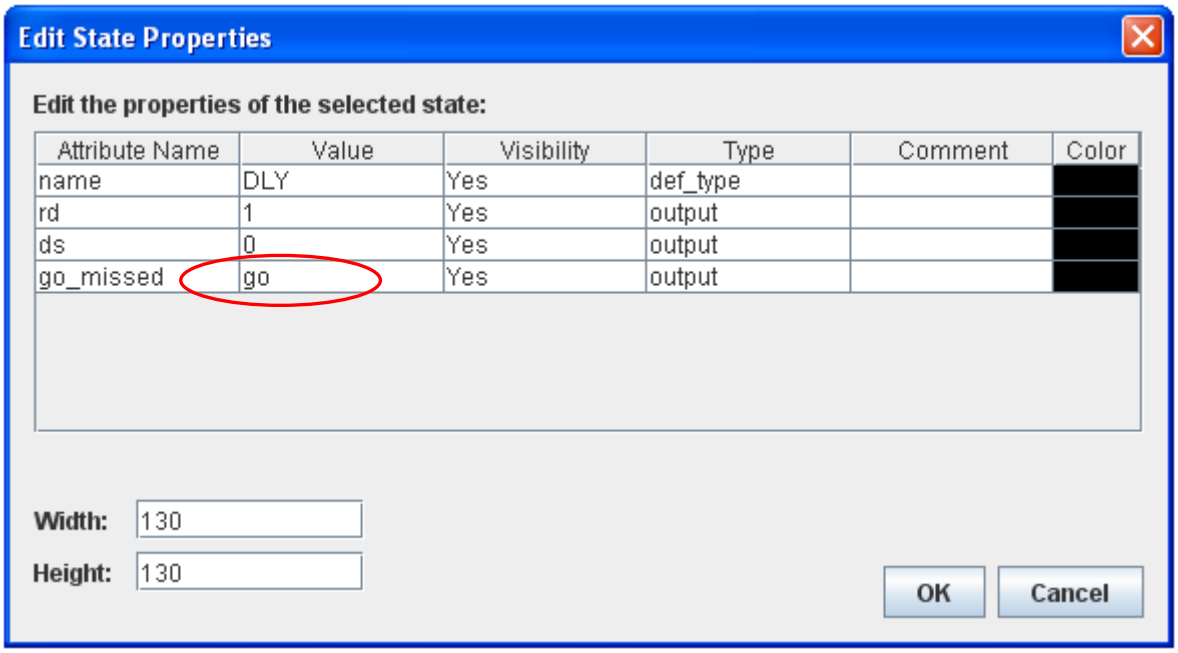
7.1 Mealy outputs assigned in states

Suppose we wanted to create an output that would toggle if “go” was asserted during state “DLY”? This is just a comb output whose equation is “go” during the DLY state, and 0 at all other times.

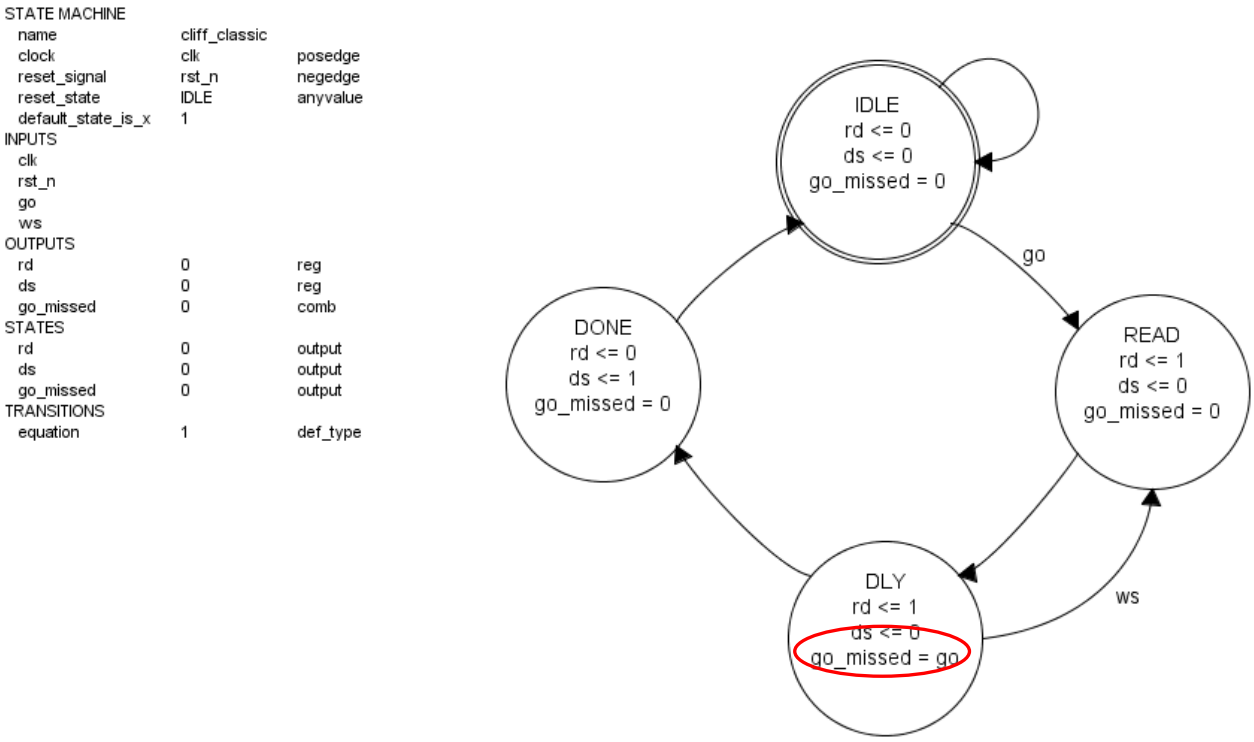
Back to Cliff Classic. Start by creating the new output “go_missed”. Go to the Global Attributes > Outputs tab and add “go_missed” . Set the type to “comb” and the default value to 0.



Now edit the DLY state to change the equation to “go”.



The result looks like this:



Notice the go_missed output shows up on each state bubble with an “=” instead of a “<=”, because it is of type “comb”.

Re-run the backend, and the new output is added as type “reg”:

```
module cliff_classic (
  output wire ds,
  output reg go_missed,
  output wire rd,
  input wire clk,
  input wire go,
  input wire rst_n,
  input wire ws
);
```

That seems a bit counter-intuitive for a comb output, but recall that “reg” in Verilog doesn’t necessarily imply a physical register. It’s type reg because it will be assigned in the combinational always block, which now looks like this:

```
// comb always block
always @* begin
  nextstate = 3'bx; // default to x because default_state_is_x is set
  go_missed = go_missed; // default to hold value to avoid latch inference
  case (state)
    IDLE: begin
      go_missed = 0;
      if (go) begin
        nextstate = READ;
      end
      else begin
        nextstate = IDLE;
      end
    end
    DLY: begin
      go_missed = go;
      if (ws) begin
        nextstate = READ;
      end
      else begin
        nextstate = DONE;
      end
    end
    DONE: begin
      go_missed = 0;
      begin
        nextstate = IDLE;
      end
    end
    READ: begin
      go_missed = 0;
      begin
        nextstate = DLY;
      end
    end
  endcase
end
```

Notice the new lines have been added to each state's case entry that assign values to go_missed.

Also, a new line has been added to default "go_missed" to hold its value at each pass through the loop. Without this, design_compiler might add latches because the "case" may not be full. This line will not appear in the onehot output, since that "case" is guaranteed to be full.

Note that output equations for comb outputs (in this case, just "go") are NOT parsed by fizzim. They are just strings to fizzim.

7.2 Mealy outputs assigned on transitions

Although this behavior could also be described by putting the equation "go" on the transition from READ to DLY, and creating a loopback transition and putting the same equation on it, it is probably most naturally described using the "on states" method above.

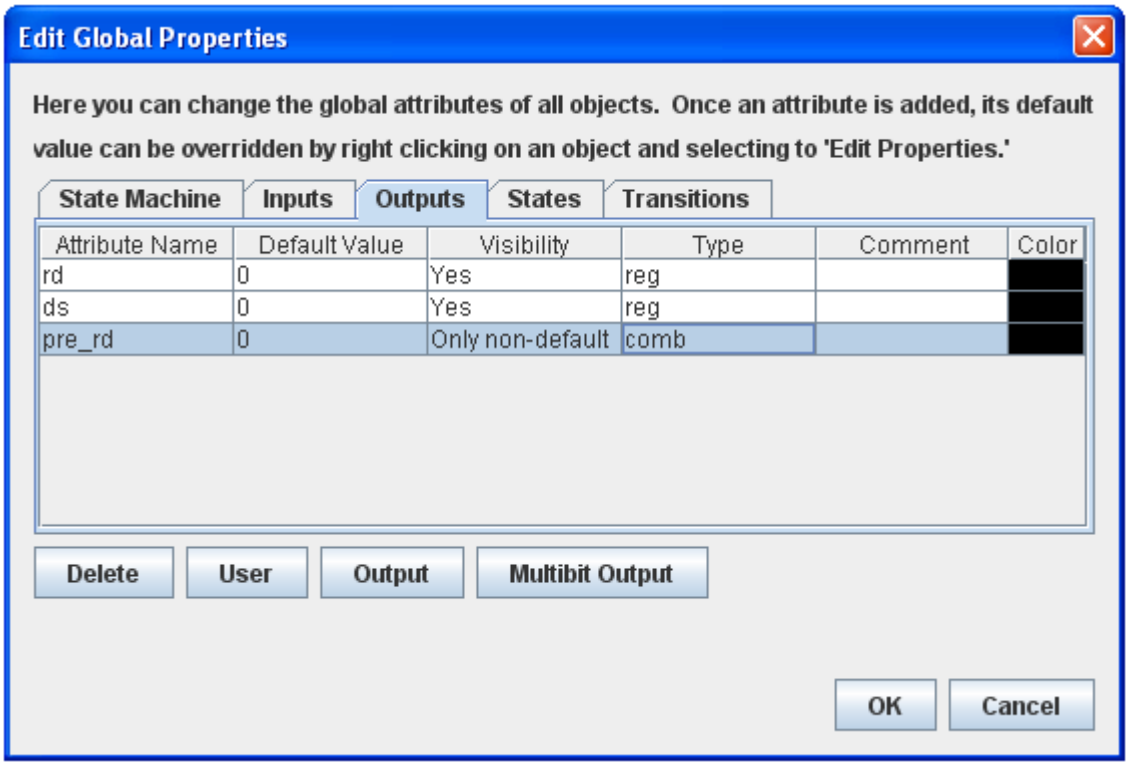
But there is a case where assigning the Mealy output on transitions might make more sense than assigning it on states – when the Mealy output equation matches the transition equation.

Suppose we wanted to send out an early copy of the "rd" output on the transition from IDLE to READ?

This is the same as saying that the new pre_rd output is equal to "go" in state IDLE. So, one way to implement this is by setting the pre_rd output to "go" in the IDLE state, similar to the example above.

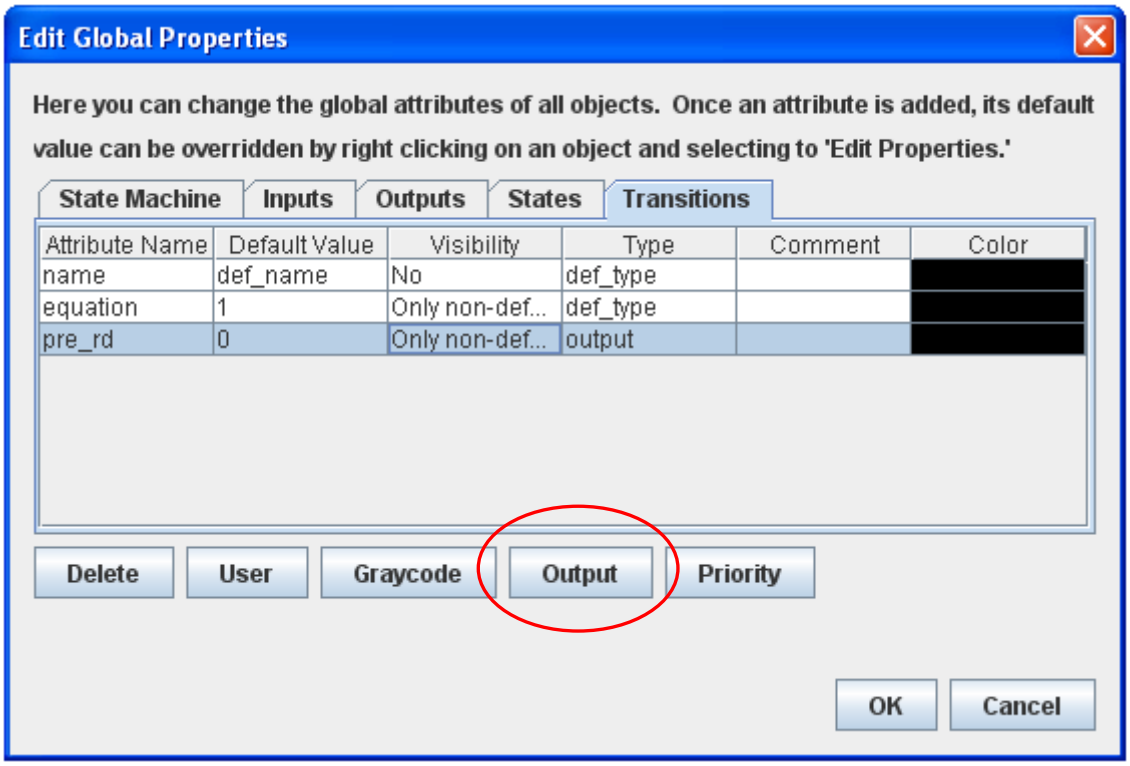
But since the equation is the same as for the transition from IDLE to READ, another way is to make the pre_rd output equal to 1 on the *transition* from IDLE to READ.

Let's take a closer look at this approach. First, we'll go back to cliff_classic and add the (comb) pre_rd output:



Fizzim will automatically transfer your new comb output to the states attributes list (as in the previous example), as it does for registered outputs. If you want to specify a comb output changing on a transition, you have to add it to the Transitions attribute list yourself:

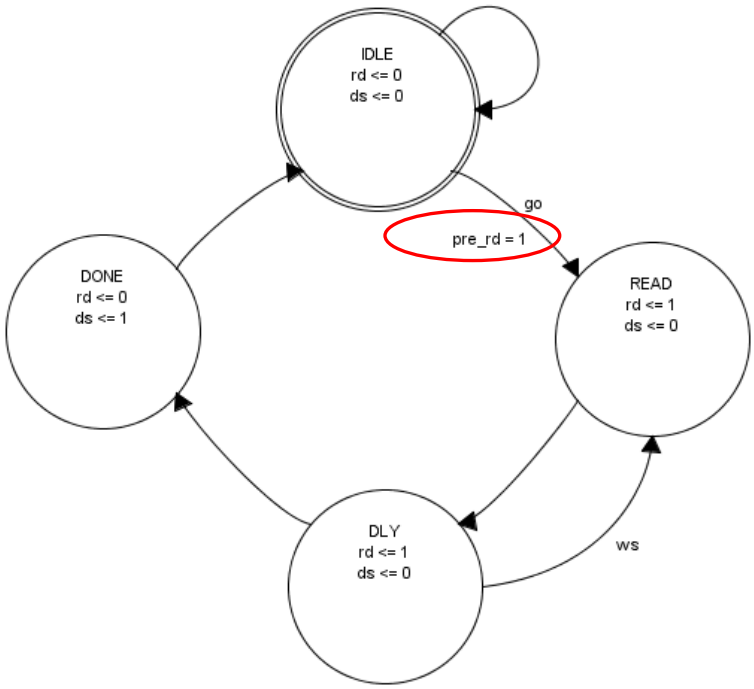
Go to the Global Attributes > Transitions tab, and use the "Output" button to add "pre_rd". Set the default value to 0, and Visibility to "Only non-default".



Now double-click the IDLE to READ transition. It now has “pre_rd” as an attribute (of type output). Change the value to 1.

Since we set the visibility to only non-default, the value will only show up on this transition, and we get the following state diagram:

```
STATE MACHINE
  name                cliff_classic
  clock               clk             posedge
  reset_signal        rst_n           negedge
  reset_state         IDLE            anyvalue
  default_state_is_x  1
INPUTS
  clk
  rst_n
  go
  ws
OUTPUTS
  rd                  0               reg
  ds                  0               reg
  pre_rd              0               comb
STATES
  rd                  0               output
  ds                  0               output
  pre_rd              0               output
TRANSITIONS
  equation            1               def_type
  pre_rd              0               output
```



The Verilog output looks like this:

```
// comb always block
always @* begin
  nextstate = 3'bx; // default to x because default_state_is_x is set
  pre_rd = pre_rd; // default to hold value to avoid latch inference
  case (state)
    IDLE: begin
      if (go) begin
        nextstate = READ;
        pre_rd = 1;
      end
      else begin
        nextstate = IDLE;
        pre_rd = 0;
      end
    end
    DLY: begin
      if (ws) begin
        nextstate = READ;
        pre_rd = 0;
      end
      else begin
        nextstate = DONE;
        pre_rd = 0;
      end
    end
    DONE: begin
      begin
        nextstate = IDLE;
        pre_rd = 0;
      end
    end
    READ: begin
      begin
        nextstate = DLY;
        pre_rd = 0;
      end
    end
  endcase
end
```

So, the output pre_rd does indeed change when the transition path is taken.

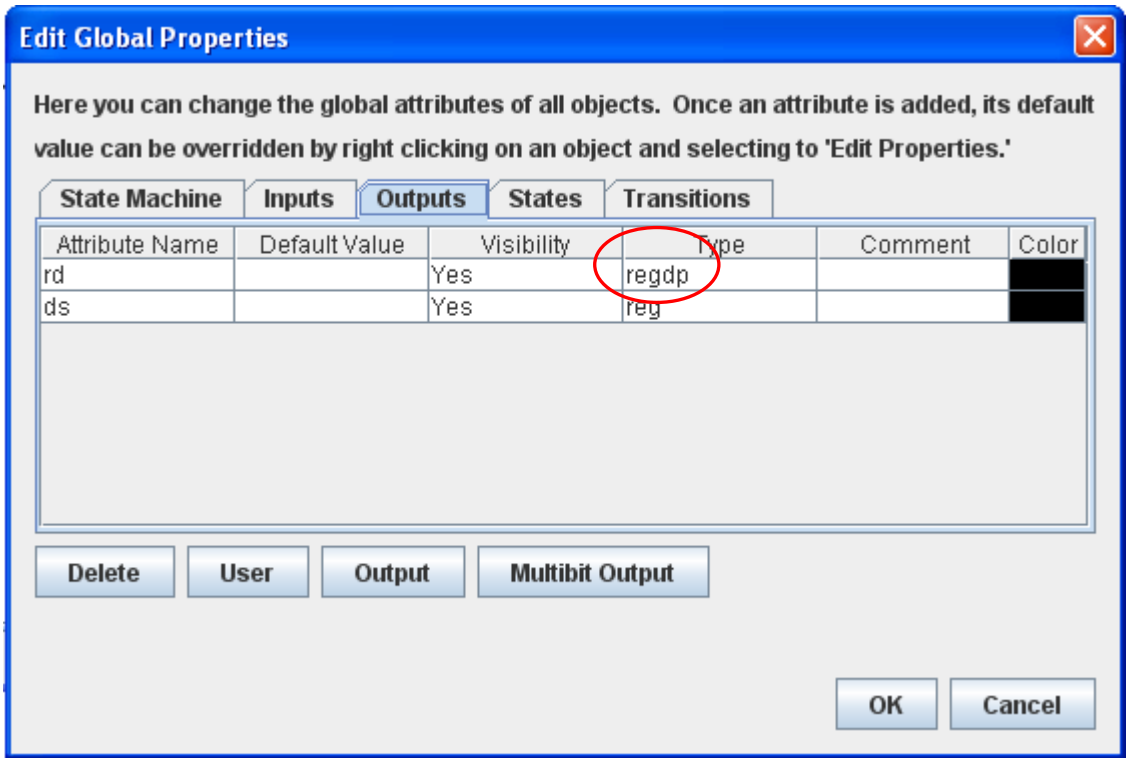
7.3 Mixing the styles

Also, note that you *can* mix the two styles, but the rules about what has priority over what are somewhat complicated. See the tutorial and the fizzim documentation for details.

8 Datapath outputs

Recall that fizzim has two types of registered outputs – reg and regdp. The "dp" in regdp stands for "datapath". When the type is regdp, fizzim will not attempt to encode the bits in the state vector.

As a simple example, we'll go back to Cliff Classic and change the type of output rd to regdp:



Re-run fizzim.pl, and the output looks like this:

```
// state bits
parameter
IDLE = 3'b000, // extra=00 ds=0
DLY  = 3'b010, // extra=10 ds=0
DONE = 3'b001, // extra=01 ds=1
READ = 3'b100; // extra=00 ds=0

reg [2:0] state;
reg [2:0] nextstate;

// comb always block
always @* begin
  nextstate = 3'bx; // default to x because default_state_is_x is set
  case (state)
    IDLE: begin
      if (go) begin
        nextstate = READ;
```

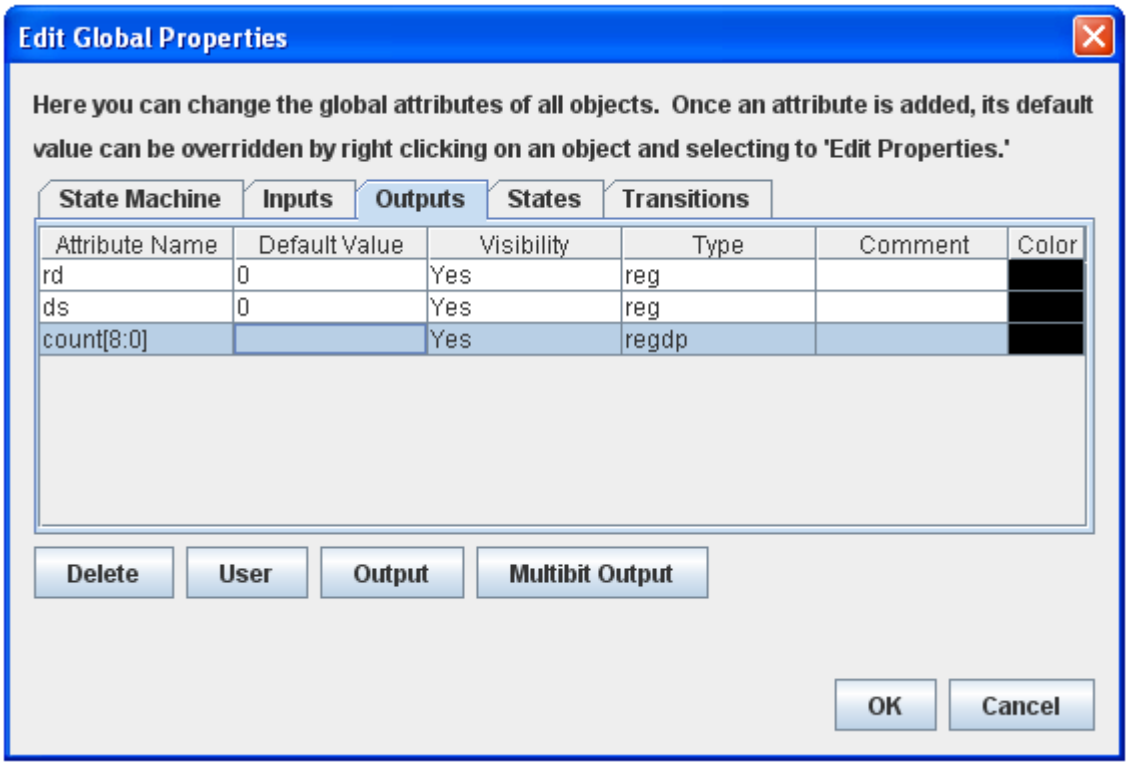
```
      end
      else begin
        nextstate = IDLE;
      end
    end
    DLY: begin
      if (ws) begin
        nextstate = READ;
      end
      else begin
        nextstate = DONE;
      end
    end
    DONE: begin
      begin
        nextstate = IDLE;
      end
    end
    READ: begin
      begin
        nextstate = DLY;
      end
    end
  endcase
end

// Assign reg'd outputs to state bits
assign ds = state[0];

// sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    state <= IDLE;
  else
    state <= nextstate;
end
```

```
// datapath sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    rd <= 0;
  end
  else begin
    case (nextstate)
      IDLE: begin
        rd <= 0;
      end
      DLY: begin
        rd <= 1;
      end
      DONE: begin
        rd <= 0;
      end
      READ: begin
        rd <= 1;
      end
    endcase
  end
end
```

Notice that the signal rd is no longer included in the state vector, and that a third always block has been added. This third always block does a "case" on nextstate, and assigns rd on the clock edge – creating a registered rd output.

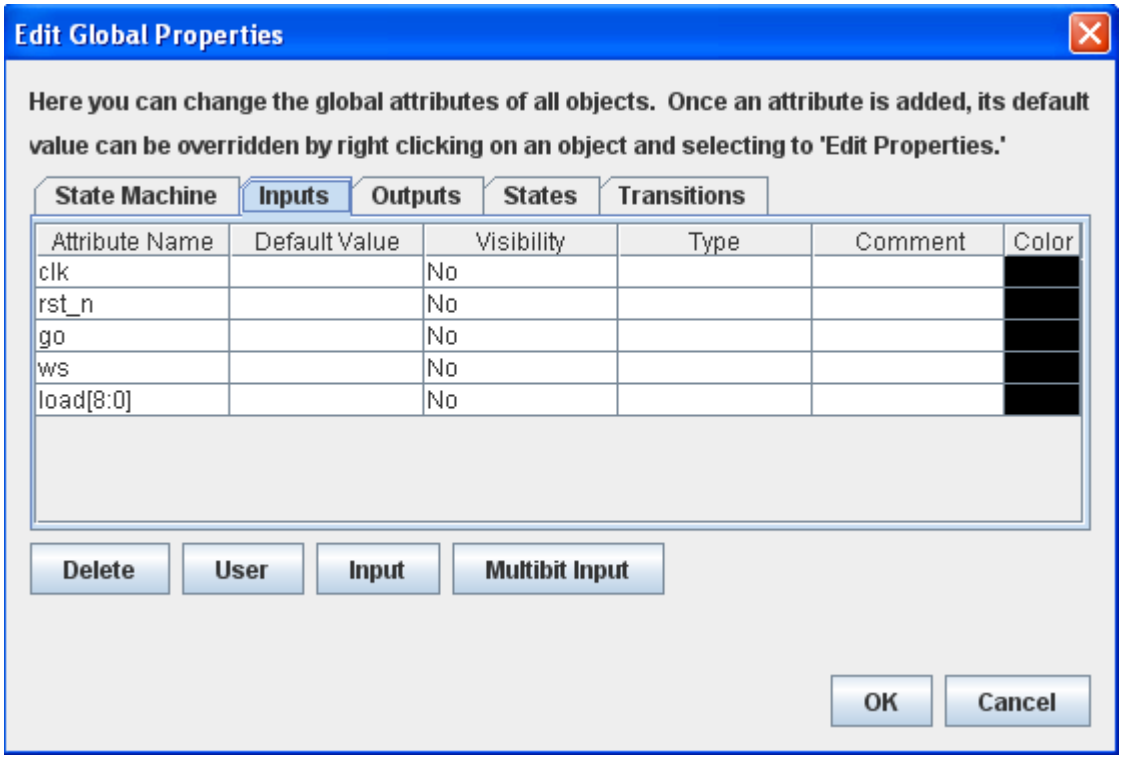
Well, that's fine if all you want to do is pull bits out of the state vector. But the real value of regdp is true datapath outputs. Suppose we wanted a counter to be controlled by the state machine? You can't very well embed *that* in the state bits! Some tools require you to push out a control signal (usually a Mealy output) and implement the counter externally. Fizzim will let you bury the counter right in with the state machine.

So, let's add a counter. First, we add a regdp output called count[8:0].



The “Multibit Output” button creates an example with the correct syntax (bit field after the name).

Add an input of “load[8:0]” so we can load the counter.

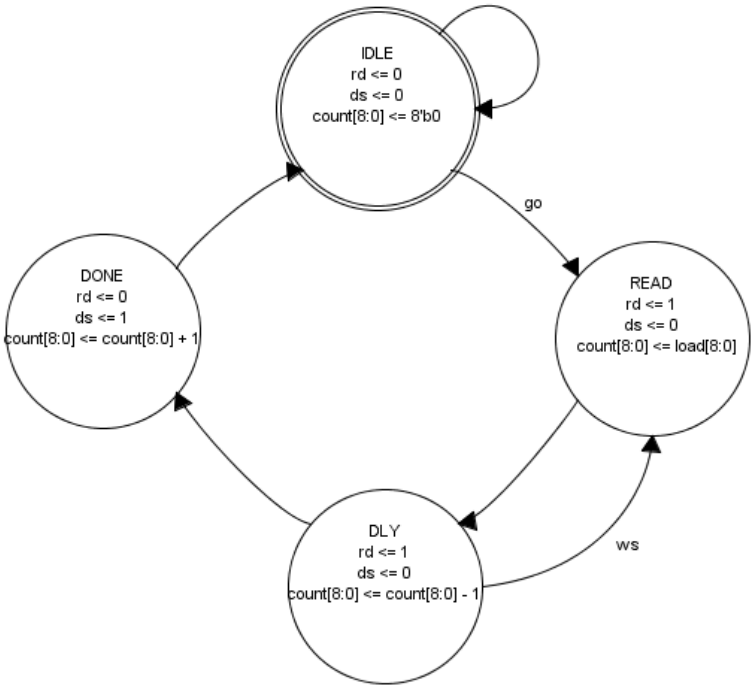


Now go around to the states and assign the counter like this:

IDLE: 8'b0
READ: load[8:0]
DLY: count[8:0] - 1
DONE: count[8:0] + 1

The result looks like this:

STATE MACHINE		
name	cliff_classic	
clock	clk	posedge
reset_signal	rst_n	negedge
reset_state	IDLE	anyvalue
default_state_is_x	1	
INPUTS		
clk		
rst_n		
go		
ws		
load[8:0]		
OUTPUTS		
rd	0	reg
ds	0	reg
count[8:0]	8'b00000000	regdp
STATES		
rd	0	output
ds	0	output
count[8:0]	8'b00000000	output
TRANSITIONS		
equation	1	def_type



Save it away and re-run fizzim.pl, and here's what you get:

```
// state bits
parameter
IDLE = 3'b000, // extra=0 rd=0 ds=0
DLY  = 3'b010, // extra=0 rd=1 ds=0
DONE = 3'b001, // extra=0 rd=0 ds=1
READ = 3'b110; // extra=1 rd=1 ds=0

reg [2:0] state;
reg [2:0] nextstate;

// comb always block
always @* begin
  nextstate = 3'bx; // default to x because default_state_is_x is set
  case (state)
    IDLE: begin
      if (go) begin
```

```
          nextstate = READ;
        end
        else begin
          nextstate = IDLE;
        end
      end
      DLY : begin
        if (ws) begin
          nextstate = READ;
        end
        else begin
          nextstate = DONE;
        end
      end
      DONE: begin
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        begin
          nextstate = DLY;
        end
      end
    endcase
end

// Assign reg'd outputs to state bits
assign ds = state[0];
assign rd = state[1];

// sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    state <= IDLE;
  else
    state <= nextstate;
end

// datapath sequential always block
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    count[8:0] <= 8'b0;
  end
  else begin
    case (nextstate)
      IDLE: begin
        count[8:0] <= 8'b0;
      end
      DLY : begin
        count[8:0] <= count[8:0] - 1;
      end
      DONE: begin
        count[8:0] <= count[8:0] + 1;
      end
      READ: begin
```

```
          count[8:0] <= load[8:0];
        end
      endcase
    end
  end
```

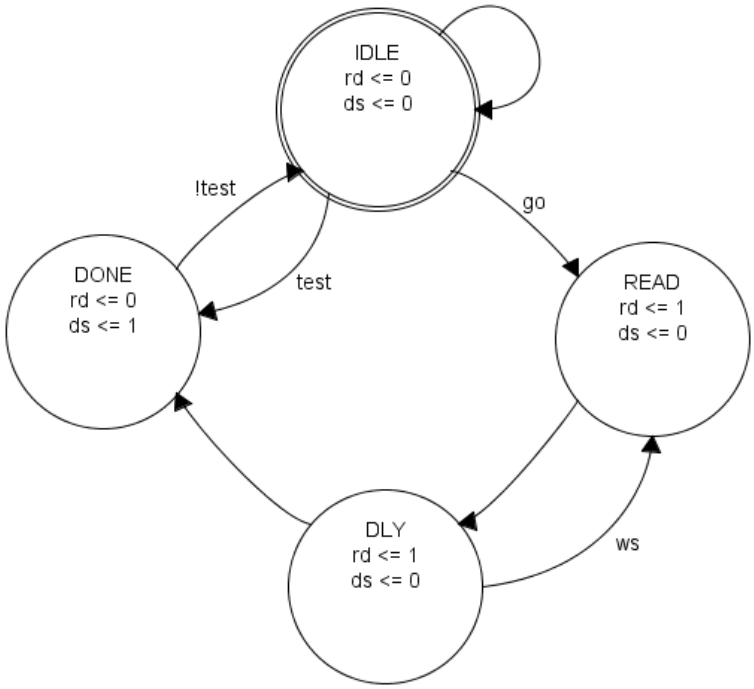
Note that, as with comb outputs, the values for regdp outputs are *not parsed* by fizzim. They're just strings. Outputs of type reg must be parsed so that they can be included in the state assignments. Currently, only constants are allows as values in reg outputs (no macros, parameters, etc) because fizzim.pl must parse them.

9 Transition priority

9.1 Basic Example

Suppose we add an input to Cliff Classic called “test” that will cause the FSM to pop over to DONE, wait for test to go away, then pop back to IDLE?

STATE MACHINE		
name	cliff_classic	
clock	clk	posedge
reset_signal	rst_n	negedge
reset_state	IDLE	anyvalue
default_state_is_x	1	
INPUTS		
clk		
rst_n		
go		
ws		
test		
OUTPUTS		
rd		reg
ds		reg
STATES		
rd		output
ds		output
TRANSITIONS		
equation	1	def_type



Since we expect test to be false during normal operation, we can just change the DONE->IDLE equation to “!test”.

If we run fizzim.pl, the following warnings appears:

```
      IDLE: begin
        // Warning P3: State IDLE has multiple exit transitions, and
transition trans0 has no defined priority
        // Warning P3: State IDLE has multiple exit transitions, and
transition trans6 has no defined priority
```

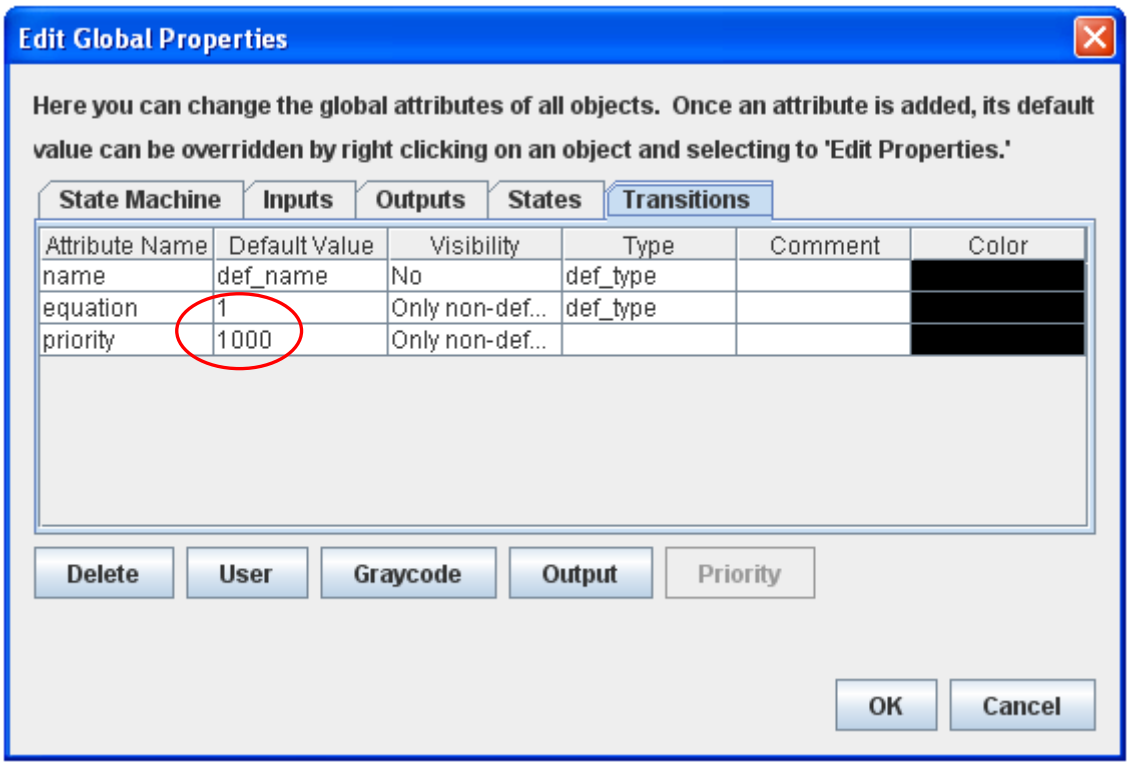
This is telling us that we haven’t defined what the FSM should do when both test and go are true.

Assume that we give priority to test. We could change the equation for the IDLE->READ transition to be “!test && go”. But this gets really tedious when the transition equations get complicated. If we were coding the FSM by hand, we would just encode the priority into the if/else structure in Verilog by putting the “if (test)” first, like this:

```
if (test) begin
  nextstate = DONE;
end
else if (go) begin
  nextstate = READ;
end
else begin
  nextstate = IDLE;
end
```

You can do this in fizzim by assigning a "priority" attribute to the transitions. This will tell fizzim.pl what order to use in the if/else block in Verilog.

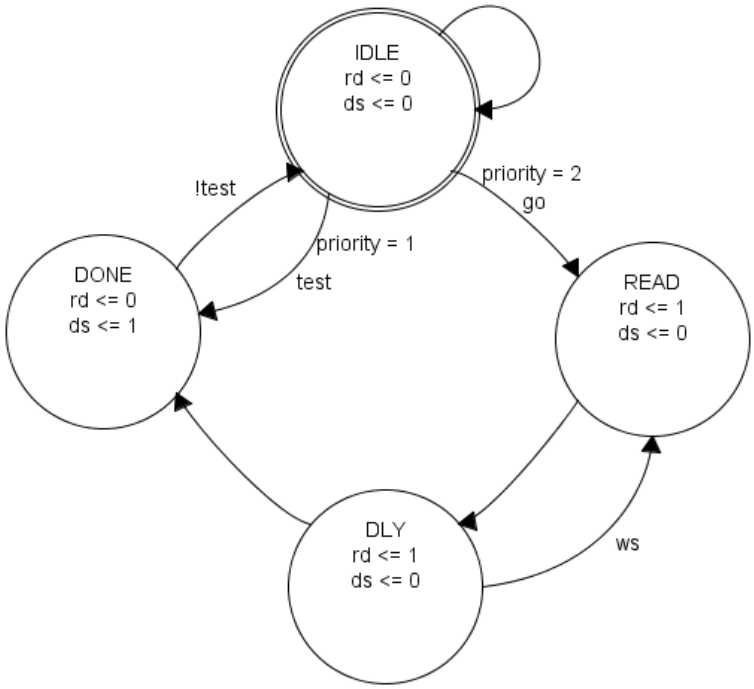
First we create a "priority" attribute for transitions in Global Attributes > Transitions. There's even a handy button to do it for you!



Note that I set the default priority to 1000 – a number larger than I expect to ever use. That means that any transition whose priority is *not* defined explicitly will have low priority. More on this in a moment.

Now we can set priority 1 on the test transition out of idle, and priority 2 on the go transition (double-click each transition and edit the value of priority).

STATE MACHINE		
name	cliff_classic	
clock	clk	posedge
reset_signal	rst_n	negedge
reset_state	IDLE	anyvalue
default_state_is_x	1	
INPUTS		
clk		
rst_n		
go		
ws		
test		
OUTPUTS		
rd	0	reg
ds	0	reg
STATES		
rd	0	output
ds	0	output
TRANSITIONS		
equation	1	def_type
priority	1	



Now when we run fizzim.pl, and the IDLE transition block looks like this:

```
IDLE: begin
  if (test) begin
    nextstate = DONE;
  end
  else if (go) begin
    nextstate = READ;
  end
  else begin
    nextstate = IDLE;
  end
end
```

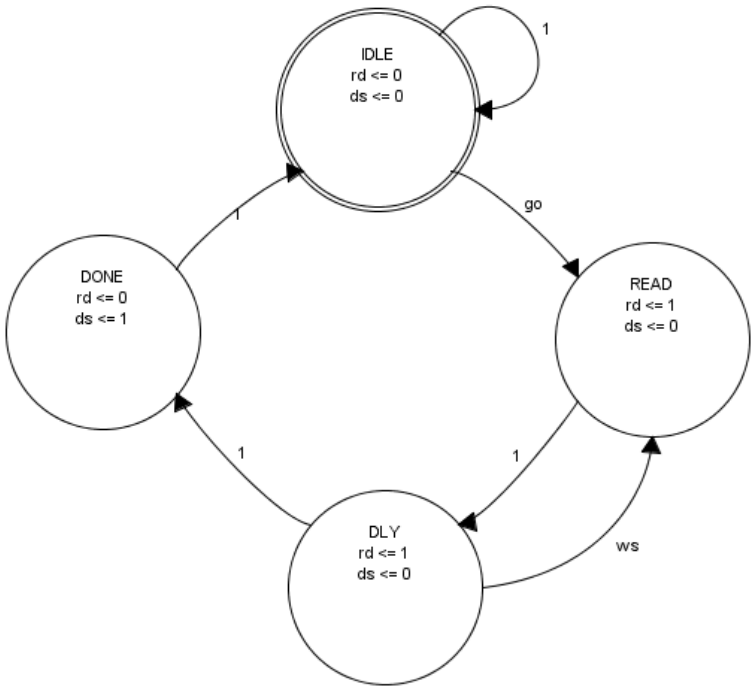
You might be wondering why fizzim.pl didn't complain about the loopback path on IDLE *before* we added the transition priorities. For that matter, why doesn't it complain about the exits from DLY? One is "ws" and the other is "1" (because this is the default value for the transition attribute "equation" that was set in the Global Attributes – fizzim sets it this way by default), and they both have the default priority of 1000.

The answer is that the equation value of "1" gets special handling by fizzim.pl.

9.2 The special case of equation equal to “1”

OK, let’s go back to the original Cliff Classic state machine. We’ll turn equation visibility to YES so that all the transition equations are visible (they were previously set to “Only non-default” to suppress all the “1” equations):

```
STATE MACHINE
  name                cliff_classic
  clock               clk             posedge
  reset_signal        rst_n           negedge
  reset_state         IDLE            anyvalue
  default_state_is_x  1
INPUTS
  clk
  rst_n
  go
  ws
OUTPUTS
  rd                  0               reg
  ds                  0               reg
STATES
  rd                  0               output
  ds                  0               output
TRANSITIONS
  equation            1               def_type
```



Why don’t I need a “!go” equation on the IDLE loopback (and “!ws” on the DLY to DONE transition)?

The answer is that fizzim.pl has some special rules regarding transition priority and equations equal to “1”. First, if two exit transitions have the same (or no) priority set, the one with the always-true equation (“1”) is assumed to have lower priority, and no warning is issued. Similarly, if there are only two exit conditions and the always-true one is the lower priority (either due the rule above or because it has explicitly been set), no warning is issued.

So, fizzim.pl sees the transition equations from IDLE as “go” and “1”, and assumes that “1” is the default (lower-priority) transition.

But there’s a little more to this than just saving some typing. It allows fizzim.pl to output Verilog code that matches what most designers would have written had they coded this by hand. You wouldn’t write:

```
case (state)
   IDLE: begin
     if (go) begin
       nextstate = READ;
     end
     else if (!go) begin
       nextstate = IDLE;
     end
```

You'd write this:

```
 case (state)
   IDLE: begin
     if (go) begin
       nextstate = READ;
     end
     else begin
       nextstate = IDLE;
     end
```

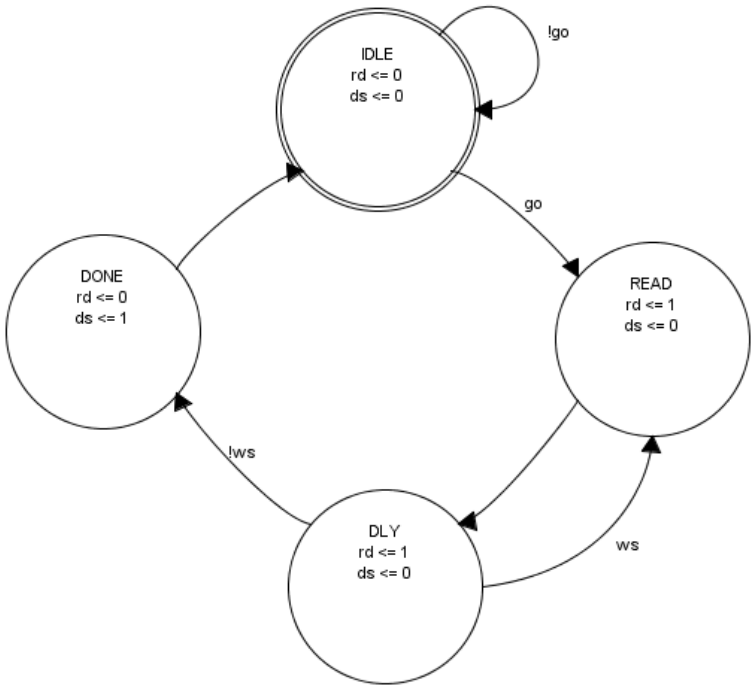
You'd look at the state diagram, recognize that the loopback was the default, and make it the "else" condition.

But fizzim has no easy way of inferring what is the default condition. So, you have to tell it – either by leaving the equation as "1", or using explicit priorities.

That's what priority is for – to tell fizzim.pl what the order of the "if" statement ought to be.

If you don't like this feature, you don't have to use it. Let's add the "missing" equations:

STATE MACHINE		
name	cliff_classic	
clock	clk	posedge
reset_signal	rst_n	negedge
reset_state	IDLE	anyvalue
default_state_is_x	1	
INPUTS		
clk		
rst_n		
go		
ws		
OUTPUTS		
rd	0	reg
ds	0	reg
STATES		
rd	0	output
ds	0	output
TRANSITIONS		
equation	1	def_type



The Verilog output now looks like this:

```
  // comb always block
  always @* begin
    nextstate = 3'bx; // default to x because default_state_is_x is set
    case (state)
      IDLE: begin
        // Warning P3: State IDLE has multiple exit transitions, and
transition trans0 has no defined priority
        // Warning P3: State IDLE has multiple exit transitions, and
transition trans5 has no defined priority
        if (go) begin
          nextstate = READ;
        end
        else if (!go) begin
          nextstate = IDLE;
        end
      end
      DLY: begin
        // Warning P3: State DLY has multiple exit transitions, and transition
trans2 has no defined priority
        // Warning P3: State DLY has multiple exit transitions, and transition
trans3 has no defined priority
        if (ws) begin
          nextstate = READ;
        end
        else if (!ws) begin
          nextstate = DONE;
        end
      end
      DONE: begin
```

```
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        begin
          nextstate = DLY;
        end
      end
    endcase
  end
```

Except for the warnings, this is what you would expect.

The warnings are telling you that you have two non-1 transition equations and haven't defined their priorities. *You and I* know that they are mutually exclusive, but fizzim.pl doesn't parse the equations, so it doesn't know. So, it warns you.

But you can easily turn the warnings off. To turn off this specific warning, use the –nowarn switch:

fizzim.pl –nowarn P3 < cliff.fzm > cliff.v

You can also turn off whole groups of warnings ("P" means priority warnings) by just using the letter:

fizzim.pl –nowarn P < cliff.fzm > cliff.v

So, if you prefer to always use explicit equations, and never use priorities, just use "-nowarn P" when you invoke fizzim.pl.

For more on suppressing warnings, see the tutorial.

10 Brief overview of advanced features

Fizzim has a number of other features not described here. For a complete tutorial and documentation, please visit my web page www.zimmerdesignservices.com.

Here is a short list of the more advanced features:

10.1 Gray codes

Individual transitions can be marked for gray coding, and fizzim.pl will choose an appropriate state encoding (if one exists).

10.2 Stubs

Rather than have every transition arc between states shown the diagram, it is possible to have transition arcs "stub out", meaning they go to (and come from) stub connectors labeled with the destination (and source) state.

10.3 Controlling internal signals

Fizzim's internal signals (like state and nextstate) can be renamed using command line switches on fizzim.pl. They can also be brought out as ports, using either the internal name or a different name.

10.4 Inserting code

You can use attributes to insert arbitrary pieces of code at strategic places in the Verilog output (such as before the module statement, after the module statement, etc).

In particular, this can be used to insert a line to "`include" a file.

Also, there is an attribute to insert code from another file at the top of the Verilog output, specifically for reading in the copyright statement.

This is described in the tutorial.

10.5 Comments

You can use the comment field in the attributes table to comment both the diagram and the code. Details of which comments carry over into the Verilog code are described in the tutorial.

10.6 Multiple pages

Fizzim has multiple pages. You can easily move states and the attributes table between pages. The transitions are handled via interpage connectors. This is all transparent to fizzim.pl and thus to the Verilog output.

10.7 Forcing the state vector

Although not directly supported by fizzim (because the state vector width is determined on the fly), it is possible to force the state vector values. The techniques are described in the tutorial.

10.8 Controlling and suppressing warning messages

Warning messages can be suppressed individually and in groups. Output can be directed to the Verilog output (as comments), to STDERR, or both. This is described in the tutorial.

10.9 Printing and exporting the state diagram

Fizzim gives you several options for printing and exporting the state diagram. This is described in the tutorial.

10.10 Specifying the backend command and options

There is a special "state machine" attribute called "be_cmd" that can be used to specify what the backend code generation command should be. Some day, this will be used to allow users to run the backend directly from the gui. For now, fizzim.pl will parse the command for its own options and configure itself accordingly. So, if you always want "-nowarn P3", you can set be_cmd to:

fizzim.pl –nowarn P3

And you'll never get P3 warnings.

Options given directly on the command line override conflicting options from be_cmd.

11 Future directions / wish list

- Multi-page print
- Better support for pages sizes other than 8-1/2 by 11.
- (Limited?) parsing of `include files for `defines and/or parameters to allow their use as values for reg outputs.
- Add a “-terse” (or “-cliff”?) option to output the minimum code necessary (suppress unnecessary wire/reg statements, begin/end, etc)

12 Conclusion

Fizzim is a freely available, open source fsm design tool. We hope that fizzim will provide ASIC designers with a valuable new tool for designing their state machines and that others will make use of the open source nature of the tool to add new features and make these available to all.

13 Acknowledgements

The authors would like to acknowledge the following individuals for their assistance:

Bruce Lavigne – Hewlett Packard
Mark Gooch – Hewlett Packard
Jon Watts – Hewlett Packard

Cliff Cummings – Sunburst Design

14 References

(1) Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements
Cliff Cummings
Synopsys Users Group 2003 San Jose
(available at www.sunburst-design.com)

(2) State machine design techniques for Verilog and VHDL
Steve Golson
Synopsys Users Group 1994 San Jose
(available at www.trilobyte.com)

(3) Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs
Cliff Cummings
Synopsys Users Group 2000 Boston
(available at www.sunburst-design.com)