# 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
(and University of California, Davis)

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

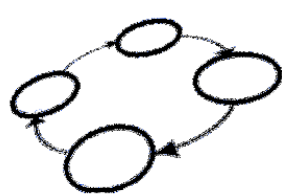
paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

29 August, 2011

**ABSTRACT**

Finite State Machine design is a common task for ASIC designer engineers. Many designers would prefer to design FSM's 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? ..... 5
2 Starting fizzim ..... 6
2.1 Windows ..... 6
2.2 Linux ..... 6
3 GUI basics ..... 7
4 Attributes ..... 8
5 Encodings ..... 9
5.1 Highly Encoded with Registered Outputs as Statebits (HEROS) ..... 9
5.2 One Hot ..... 9
6 Cliff's Classic ..... 10
6.1 Creating the states ..... 11
6.2 Creating the transitions ..... 13
6.3 Filling in the details ..... 16
6.3.1 Global Attributes ..... 16
6.3.2 Individual State Attributes ..... 23
6.3.3 Individual Transition Attributes ..... 25
6.4 Output using heros ..... 26
6.5 Output using onehot ..... 30
6.5.1 Output using onehot when "implied_loopback" is set. ..... 30
6.5.2 Onehot output when "default_state_is_x" is set ..... 33
6.6 Ascii state name ..... 35
6.7 (Un)Displaying the attributes table ..... 36
7 Mealy outputs ..... 37
7.1 Mealy outputs assigned in states ..... 37
7.2 Mealy outputs assigned on transitions ..... 40
7.3 Mixing the styles ..... 43
8 Datapath outputs ..... 46
9 Flags (new with version 4.0) ..... 53
9.1 Basic Example (flag set only on states) ..... 53
9.2 Flags set on transitions ..... 57
9.3 Capturing incoming data on an arc using flags ..... 61
10 Transition priority ..... 65
10.1 Basic Example ..... 65
10.2 The special case of equation equal to "1" ..... 67
11 Adding gray codes ..... 72
12 Mapping states to values in heros ..... 76
13 Stubs ..... 77
14 Bringing out internal signals ..... 80
14.1 Renaming internal signals ..... 80
14.2 Bringing out internal signals ..... 80
15 Using parameters ..... 84
16 Inserting random bits of code at strategic places ..... 86
17 Inserting comments ..... 87
18 Using multiple pages ..... 88
19 `include and `define ..... 93
20 Forcing the state vector ..... 96

20.1 With registered outputs as datapath bits ....... 96
20.2 With registered outputs assigned to state bits ....... 97
21 Suppressing outputs in the module portlist ....... 101
22 Splitting lines in free text and equations ....... 102
23 Unknown states ....... 103
23.1 Case 1 – sparse state space and unknowns go to an existing state ....... 103
23.2 Case 2 – full state space and unknowns go to an existing state ....... 106
23.3 Case 3 – sparse state space and unknowns go to a new state ....... 107
23.4 Case 4 – full state space and unknowns go to a new state ....... 109
24 Controlling and suppressing warning messages ....... 112
25 Printing and exporting the state diagram ....... 113
26 Specifying the fizzim.pl options ....... 114
27 Requiring a minimum revision of fizzim.pl ....... 115
28 Group select and move ....... 116
29 –terse (-sunburst) option ....... 119
30 SystemVerilog output ....... 123
31 Future directions / wishlist ....... 127
32 Acknowledgements ....... 128
33 References ....... 129

# 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") and then manually translate this diagram 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 tutorial, it is assumed that the reader is familiar with FSM's 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.

Note on the current state of the documentation:
The format of the pages changed a little bit with version 4.0. Older sections of the document have not been update yet. Usage is unchanged.

# 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_v10.02.26.jar

Starting with version 4.0, you can also add the fizzim file on the command line:

java -jar fizzim_v11.03.02.jar myfsm.fzm

# 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.

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 lingu-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 (see the section on comments).
- Color – Text color.
- (new with version 4.0) UserAtts - a per-item list of attributes for use by the backend processor.

# 5 Encodings

There are two primary types of state encodings used for FSM design. Highly encoded FSM's 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)

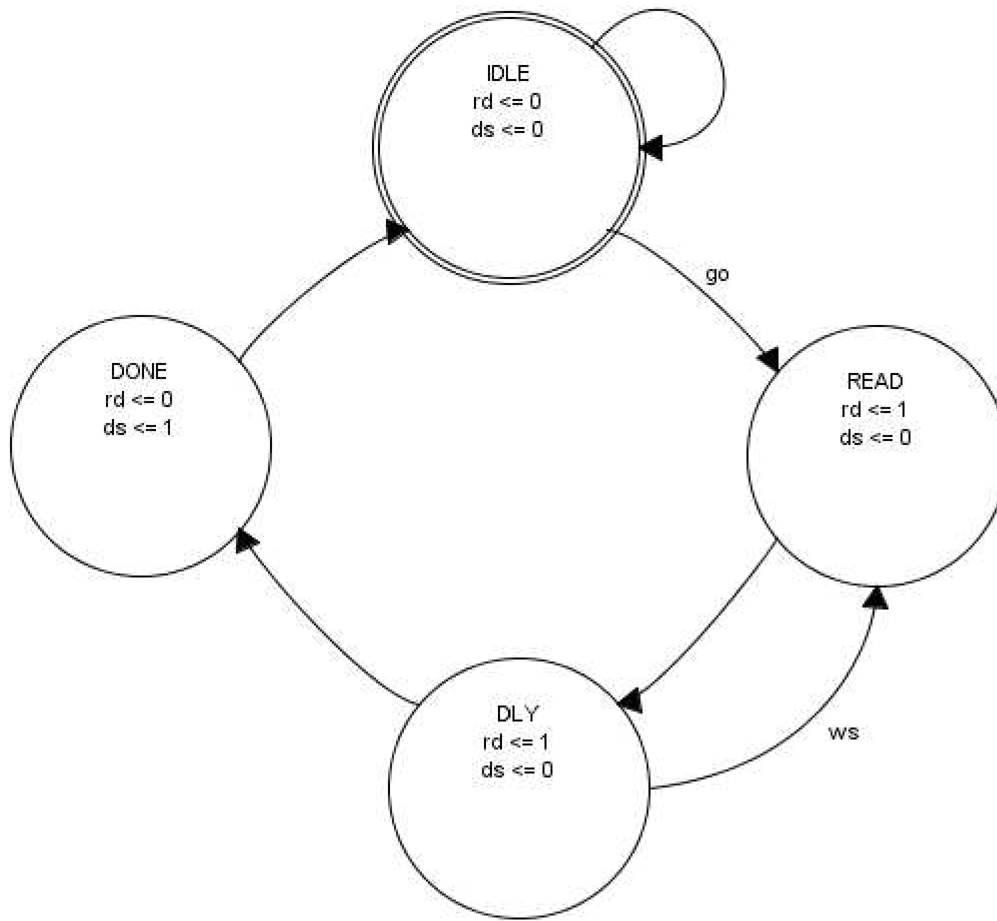
Heros is an encoding that uses a dense binary code. As the name implies, registered outputs will be encoded into the states to minimize flop count. There are mechanisms (discussed below) to allow particular outputs to be excluded from the state vector. The actual Verilog format is based on recommendations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

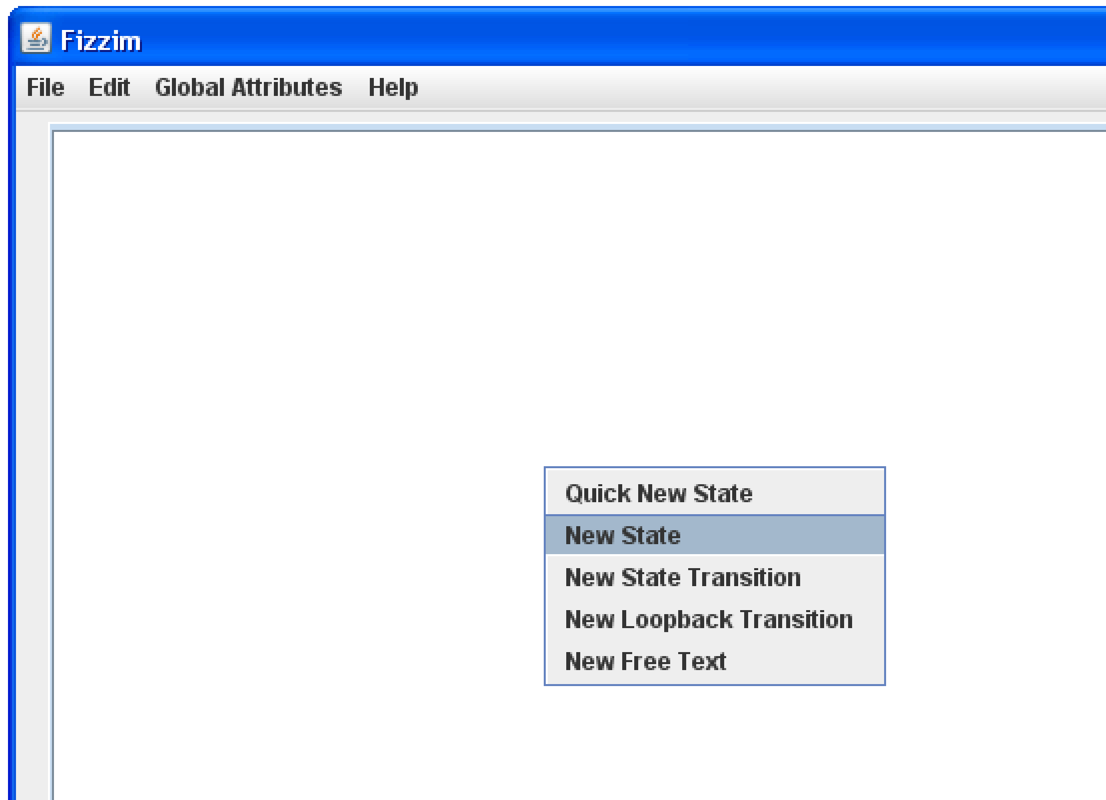
Let's jump right in with an example. In [3], Cliff Cummings introduced the following basic state machine:



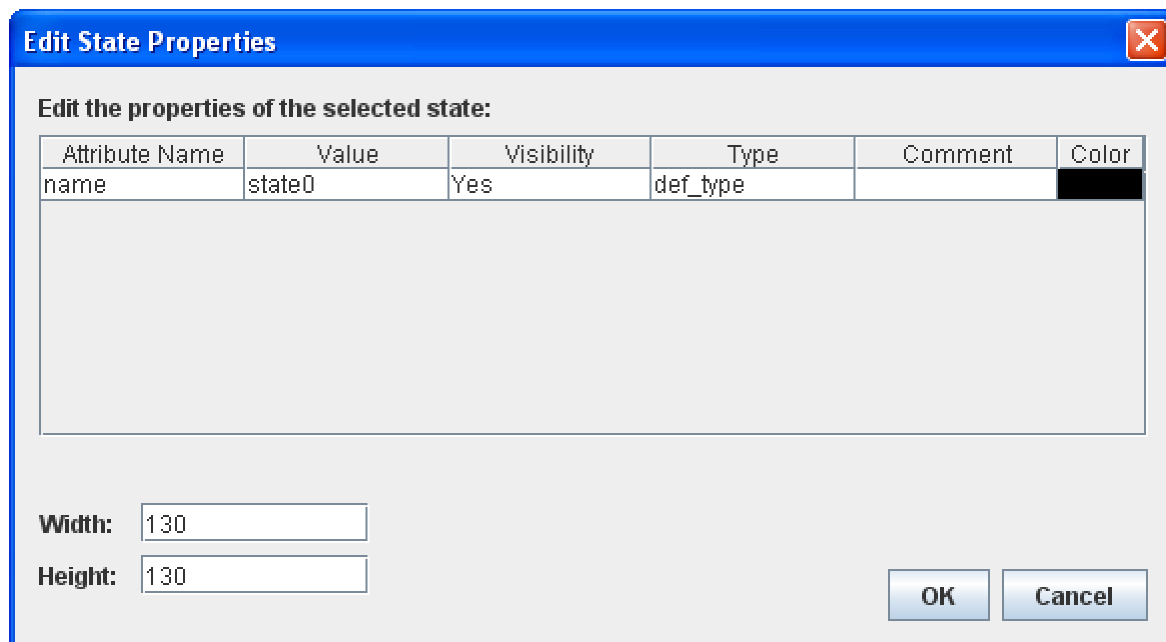

Here's how we would create this in fizzim (Example: cliff_classic.fzm).

## 6.1 Creating the states

Right-clicking in open space gives the following menu:



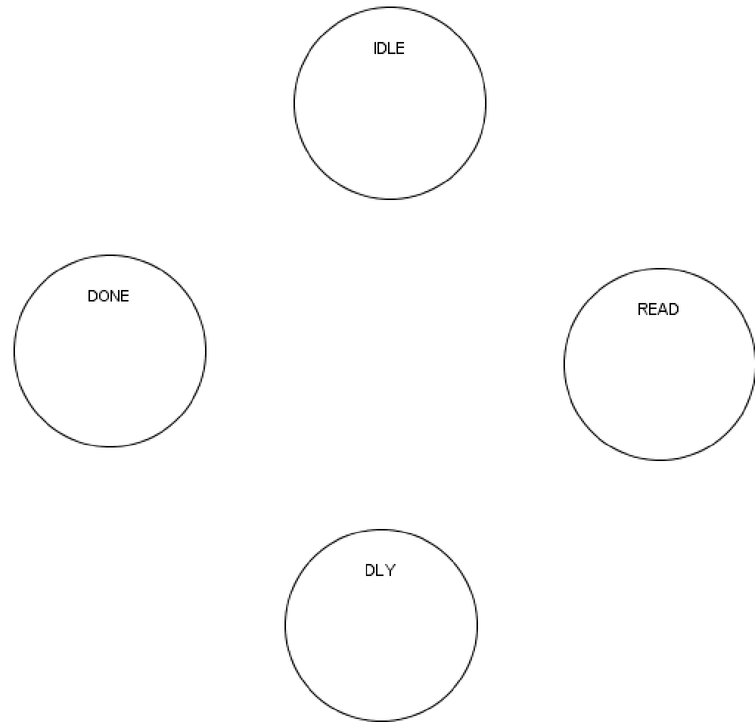

We select "New State" and get this:



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.

STATE MACHINE
name def_name
clock clk posedge
TRANSITIONS
equation 1 def_type



## 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:

Edit State Transition Properties

Edit the properties of the selected state transition:

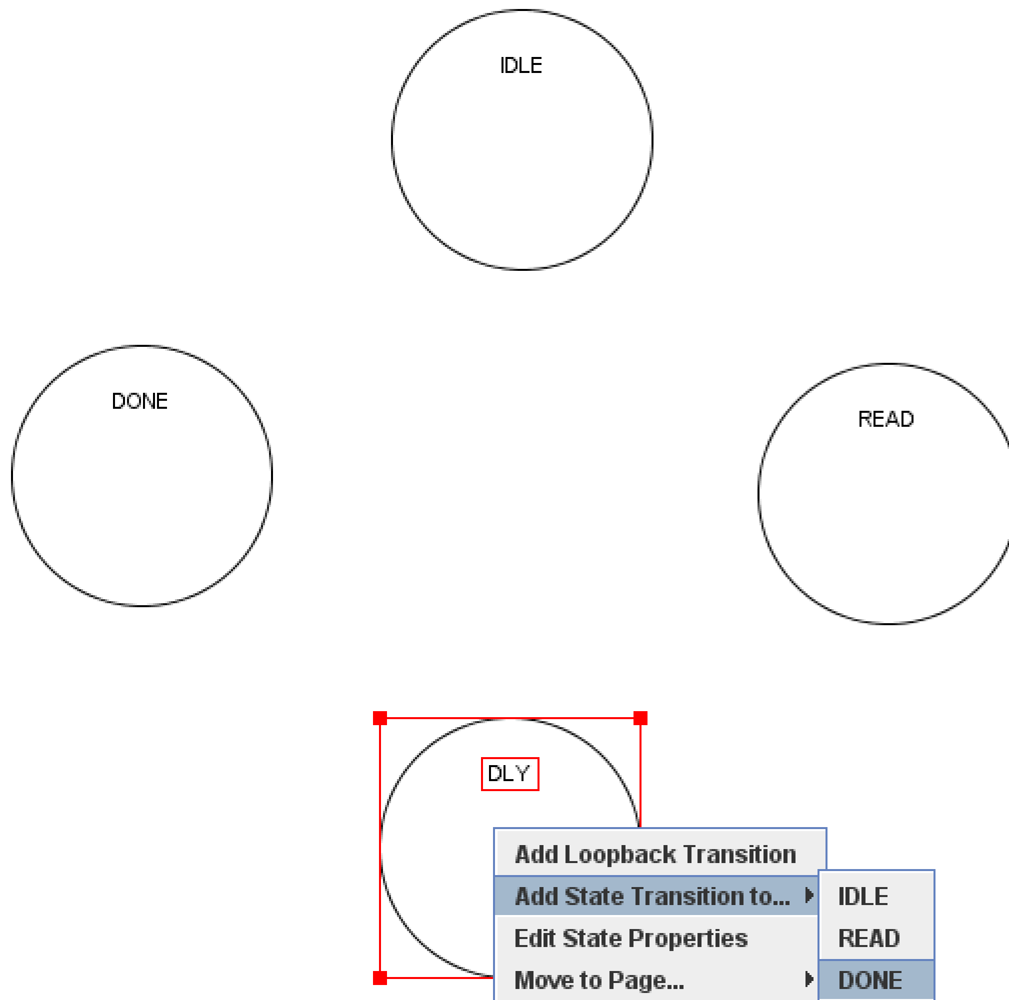
| Attribute Name | Value | Visibility | Type | Comment | Color |
|---|---|---|---|---|---|
| name | trans0 | No | def_type | | |
| equation | 1 | Yes | def_type | | |

Start State: DLY   Stub?

End State: DONE

OK   Cancel

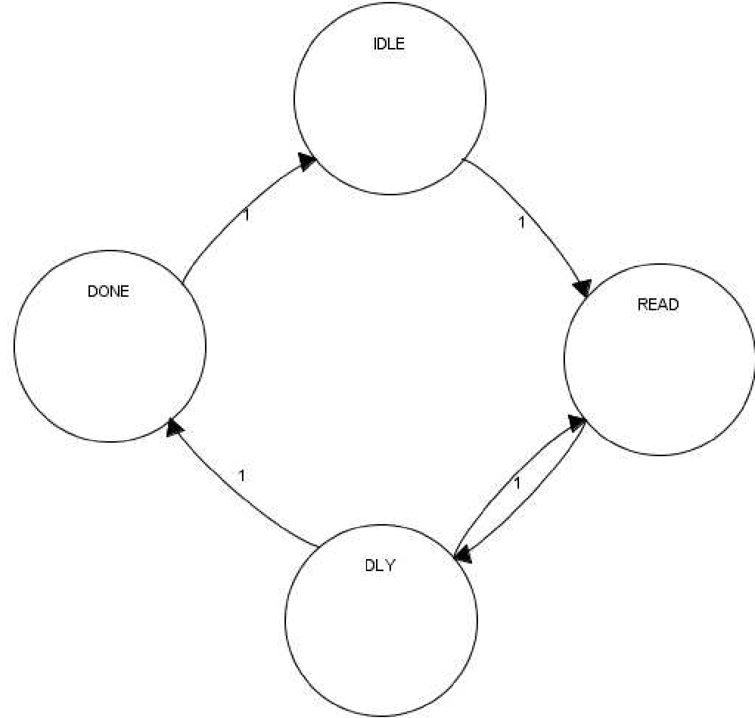
Or we can right-click on the start state and select “Add State Transition to”:



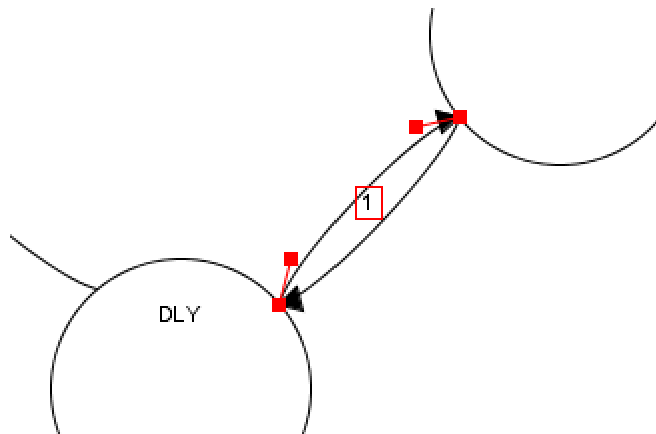

We repeat this to add all the transitions.  Don’t forget to add the loopback transition.  We’ll see why this matters in a moment.

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

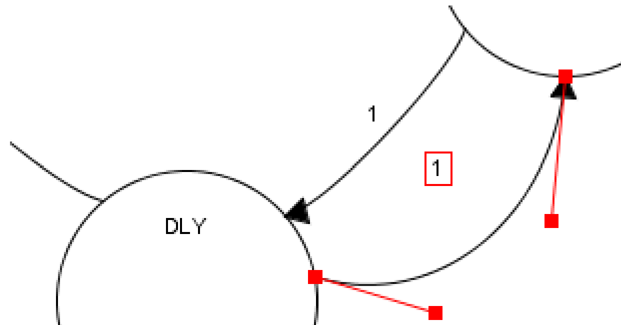
STATE MACHINE
  name  def_name
  clock  clk  posedge
TRANSITIONS
  equation  1  def_type




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.

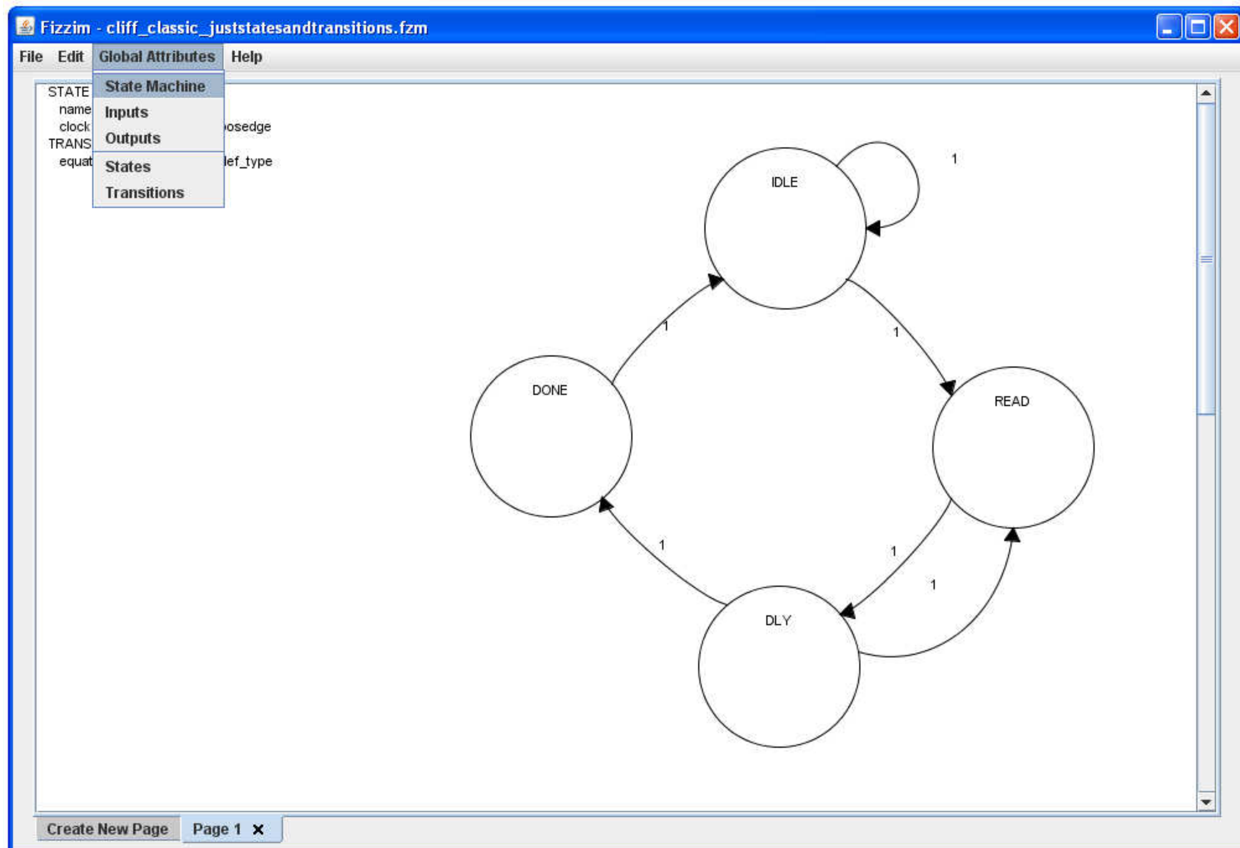
## 6.3 Filling in the details

### 6.3.1 Global Attributes

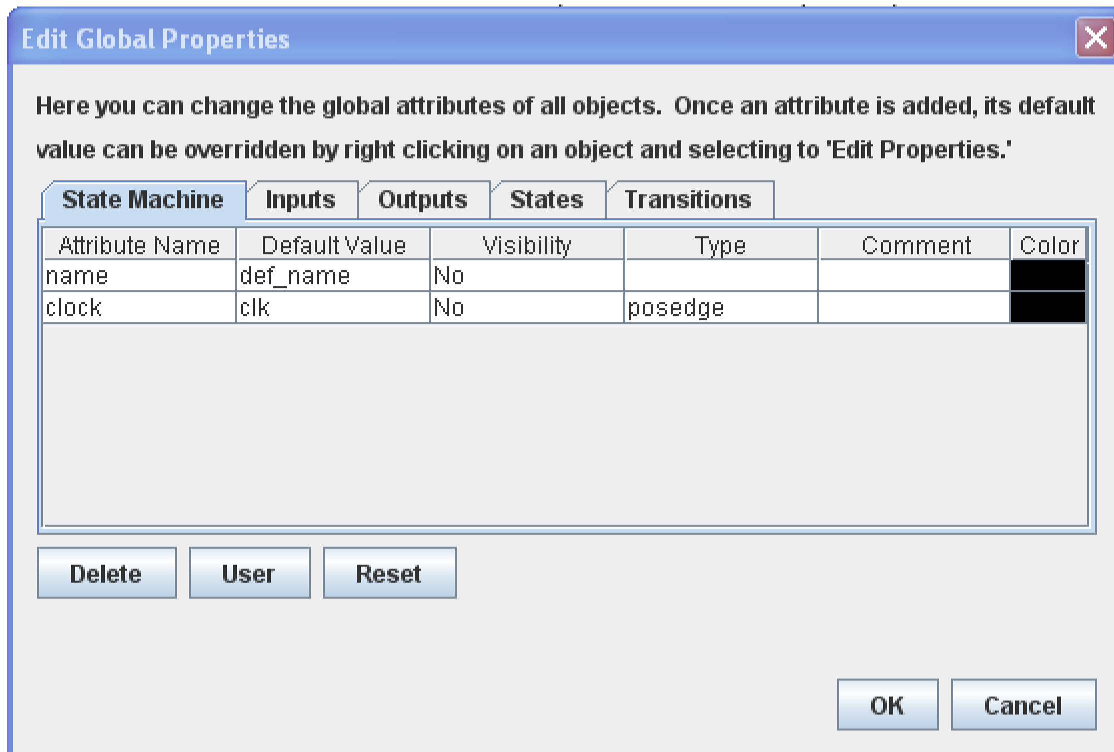
Recall that everything is stored as attributes – either attributes on the FSM itself or attributes on individual states and transitions. So, adding inputs, outputs, transition equations, etc is a matter of editing attributes.

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:

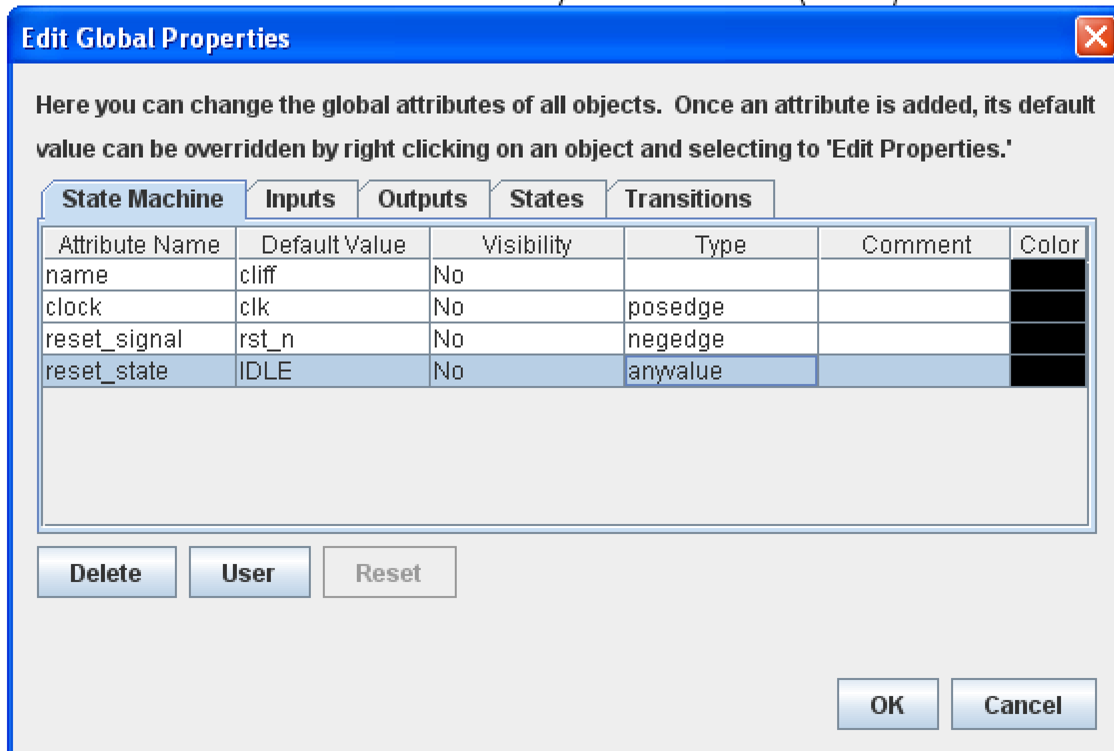


And you get this:




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

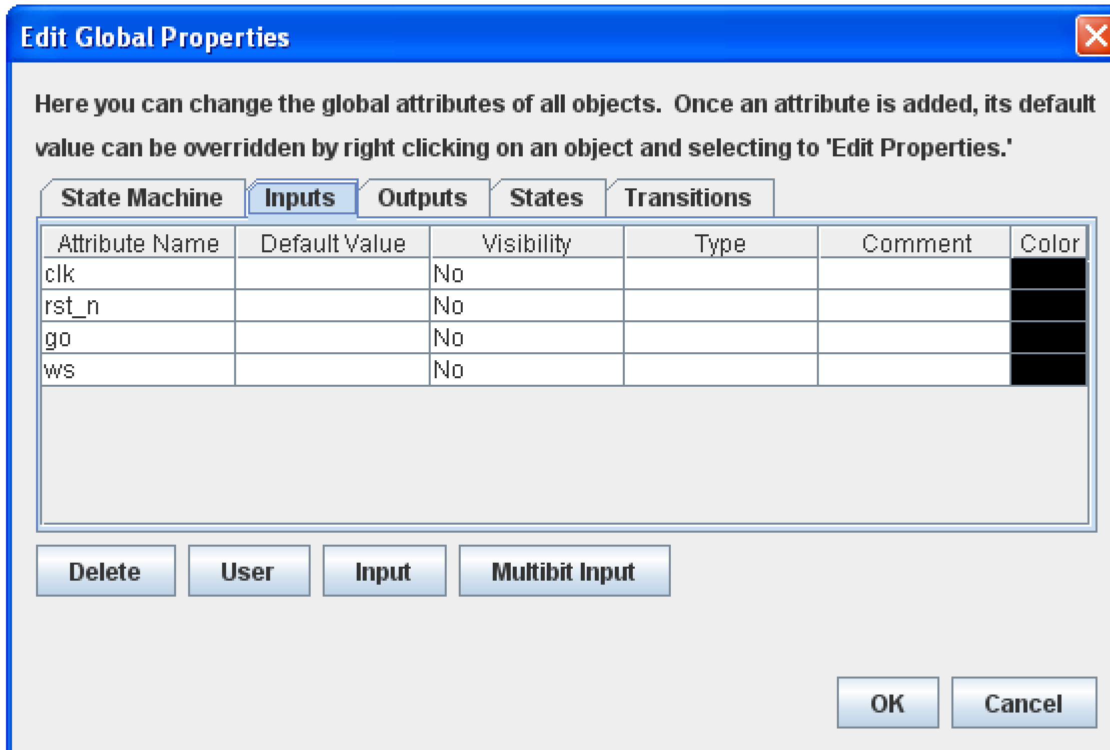
Click the "Reset" button, and two more attributes appear. One is "reset_signal". Change this to "rst_n", negedge. Set "reset_state" to IDLE via the pull-down menu and set its type to "anyvalue" ("allzeros" and "allones" will force the reset state to be all zeros or all ones, but this isn't compatible 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.

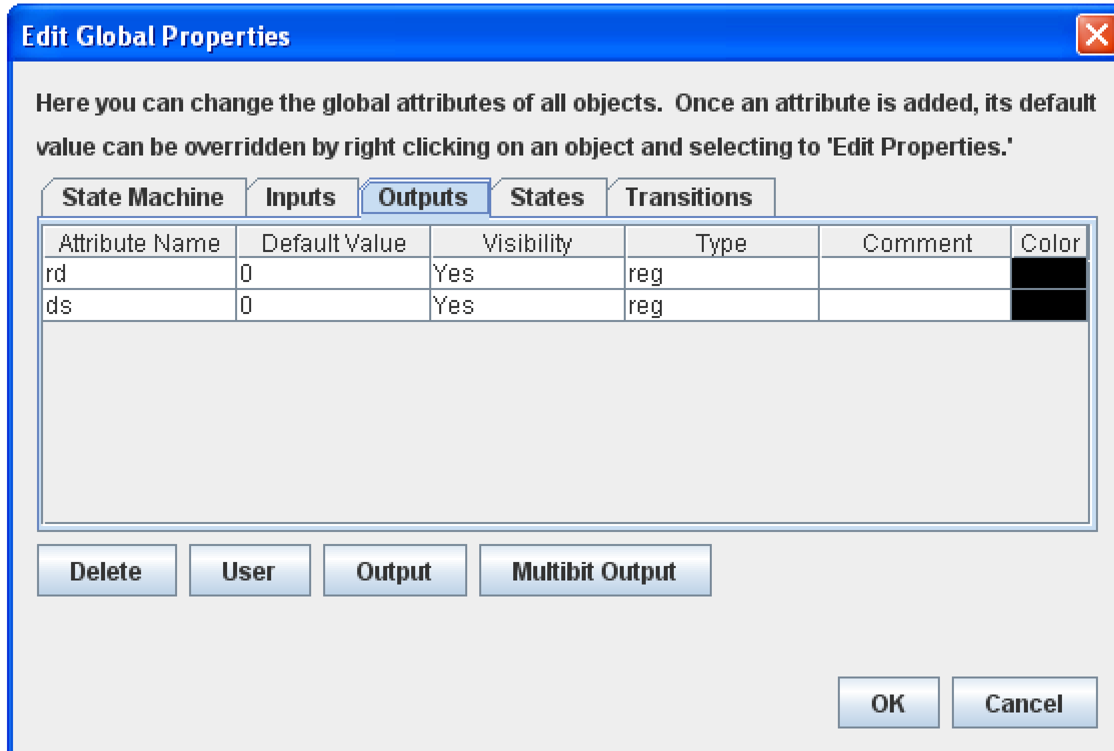
Now select “Global Attributes > Inputs” from the top menu.

Use the “Input” button to add the inputs:



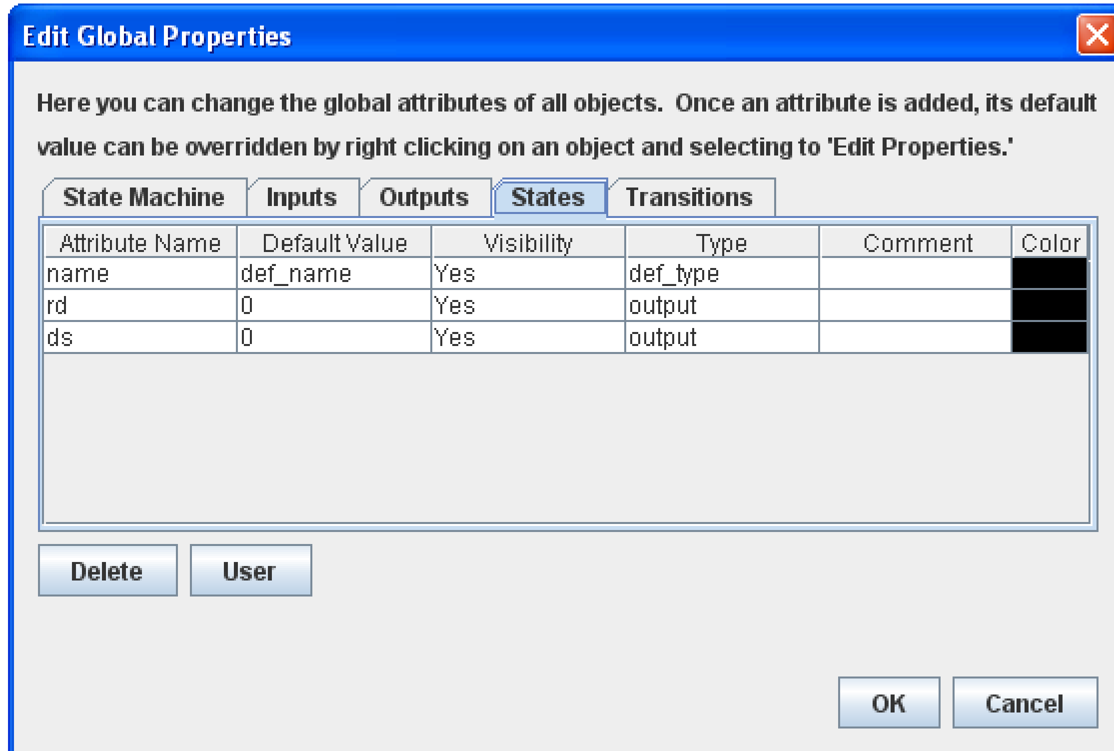

Note that “type” doesn’t matter for inputs. We could click OK, then reselect “Global Attributes > Outputs” from the top menu, or we can just switch to the “Outputs” tab without exiting the menu.

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”.

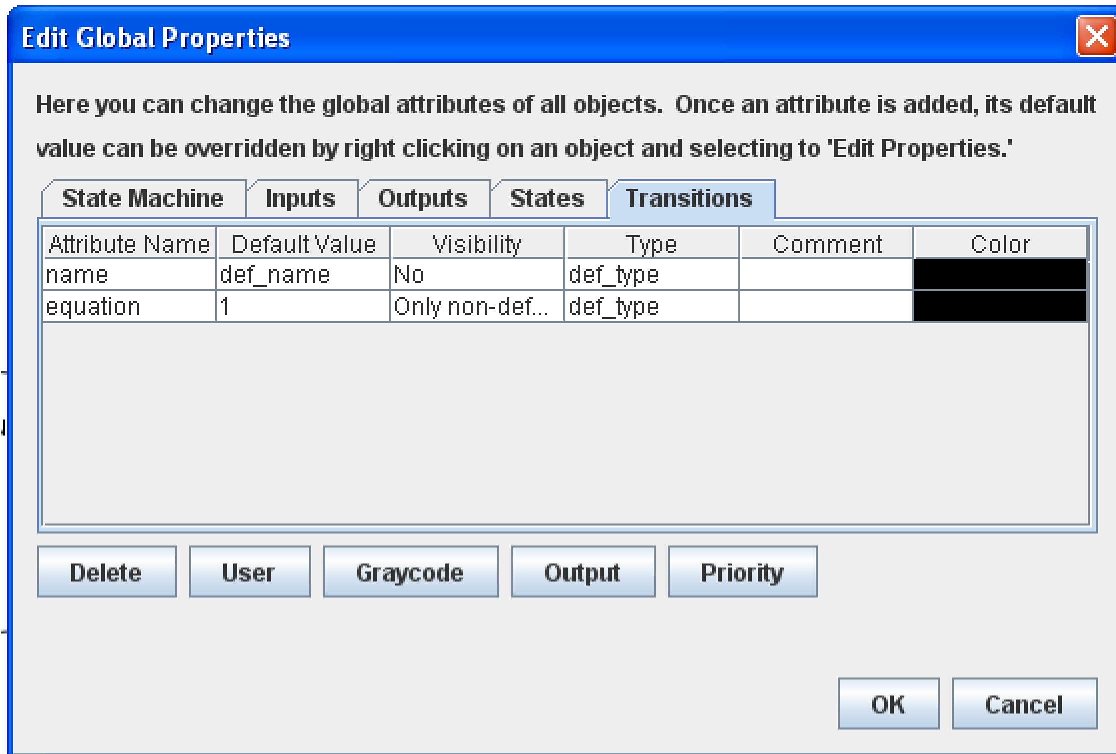



This will become clearer later, but type "reg" means that they are registered outputs (Moore) and that they should be encoded as state bits.

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.



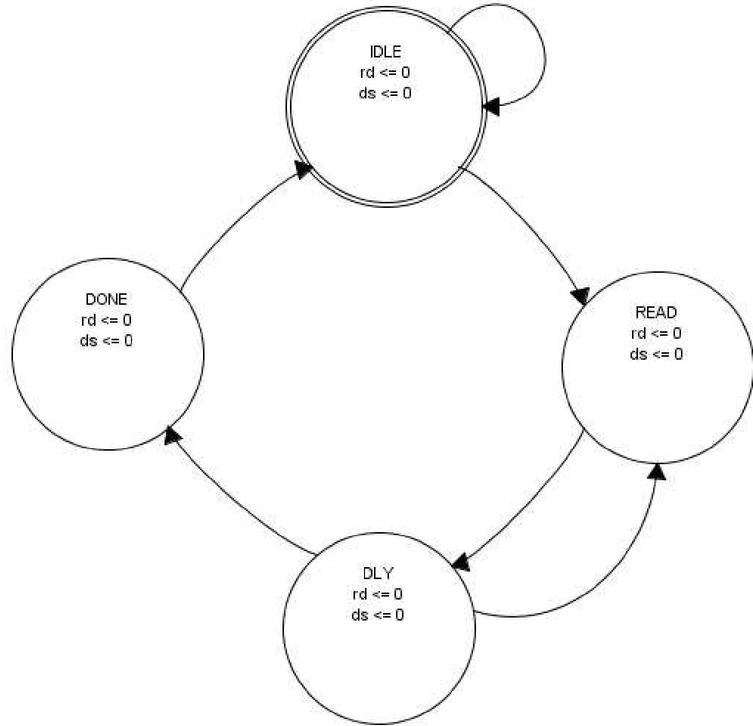

Flip over to the "Transitions" tab. "rd" and "ds" do NOT appear here, because it makes no sense to define registered outputs on a transition. The standard attribute "equation" DOES appear here, with the default value of "1". Leave it alone. But you can change the "Visibility" field to "Only non-default"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



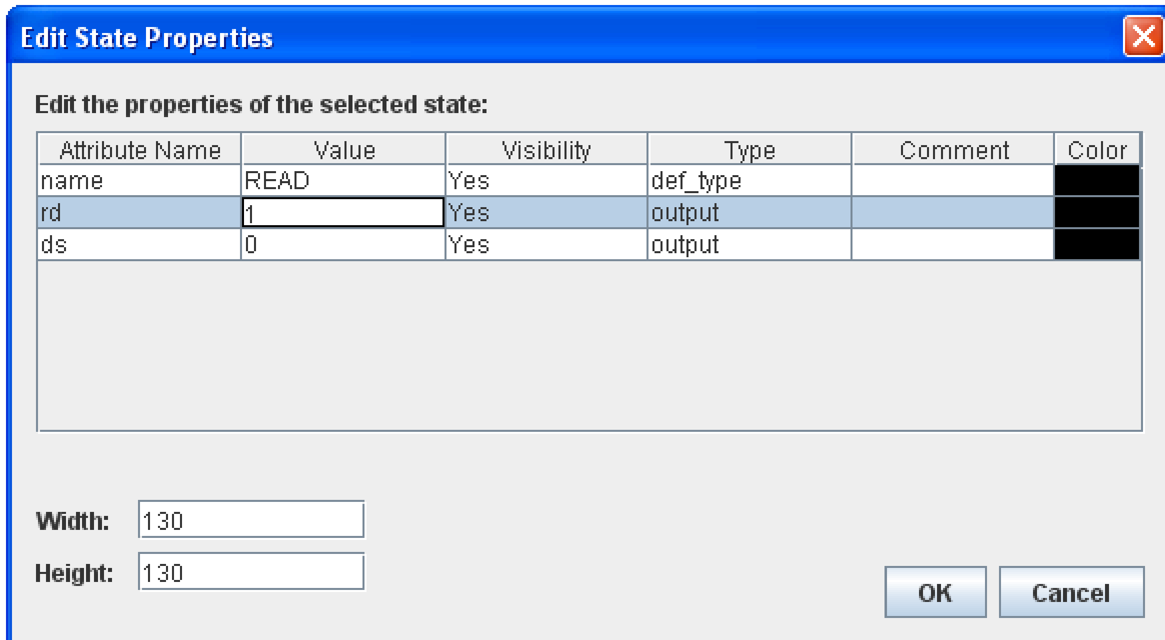

### 6.3.2 Individual State Attributes

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 |



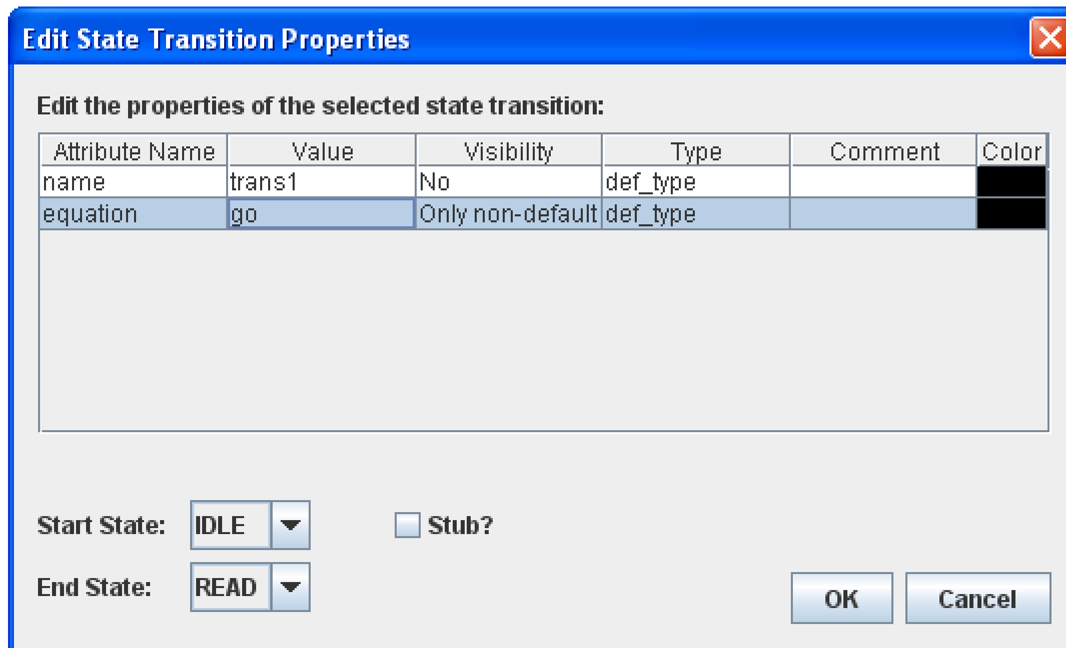

Now we need to enter the non-default values for rd and ds. Right-click on the READ state and select "Edit State Properties" to bring up the menu. Or just double-click the READ state bubble. Change the value of rd to "1".



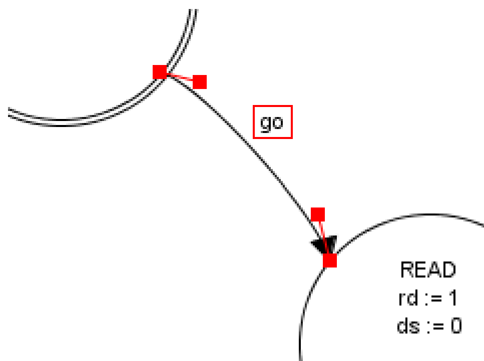
Do this for the other states to add appropriate output values (rd = 1 in DLY, ds = 1 in DONE).

### 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”.



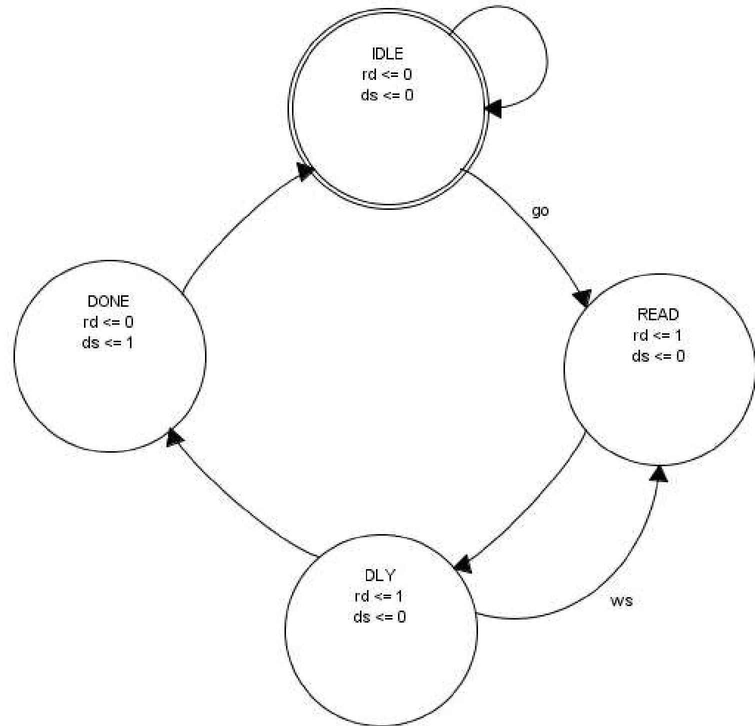

Hit “OK”. Now click on the “go” text and move it:




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 I2: Neither implied_loopback nor default_state_is_x attribute
is set on state machine - defaulting to implied_loopback to avoid latches
being inferred
    nextstate = state; // default to hold value because implied_loopback 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
```

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 I2: Neither implied_loopback nor default_state_is_x attribute is
set on state machine - defaulting to implied_loopback to avoid latches being
inferred
```

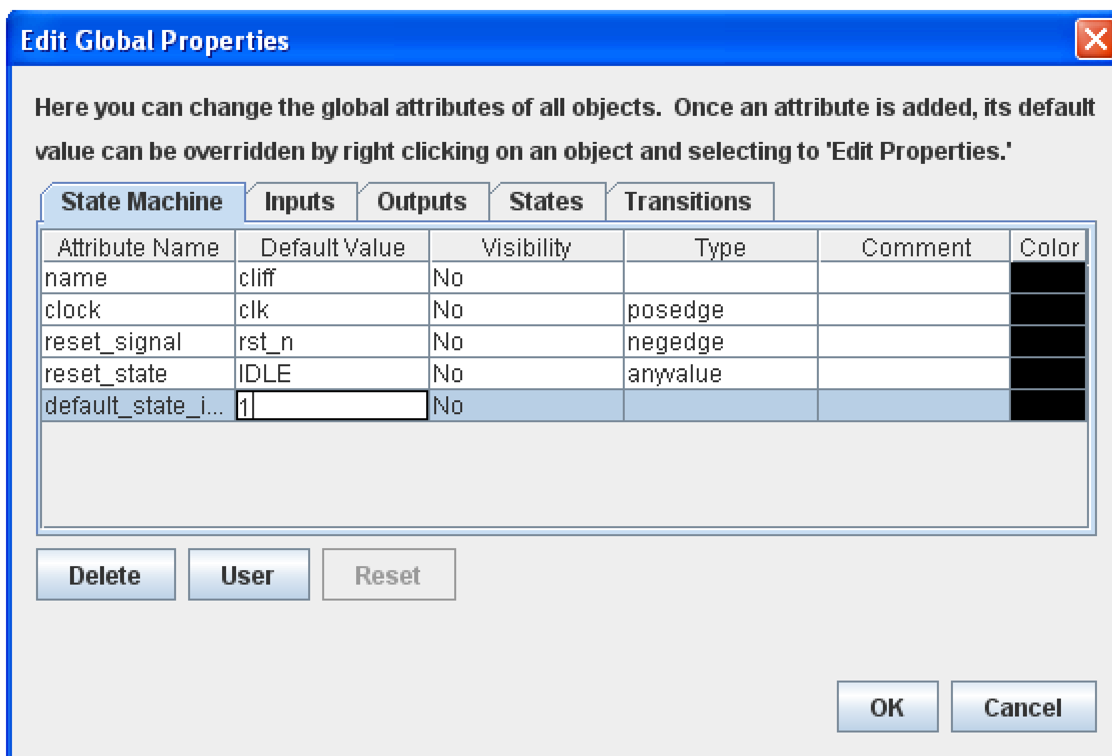
We have come to a philosophical fork in the road.

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". But to avoid problems for new users (who don't read the documentation first…), as of version 3.6 fizzim.pl will default to implied_loopback if neither attribute is set.

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'bxxx; // 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

### 6.5.1 Output using onehot when "implied_loopback" is set.

The default onehot encoding is based on Steve Golson's paper [2]. This technique doesn't really allow for the "default_state_is_x" behavior, so the output looks rather different when this attribute is set (see below). The following discussion assumes implied_loopback is set (setting neither flag is not recommended as it can result in inferred latches)

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; // Added because implied_loopback is true
      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)… state[IDLE]" gets translated to mean "when the IDLE bit of the state vector (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 not arise for onehot. If something goes wrong, you get an illegal all-zeros state which you never get out of. Since implied_loopback was set on this example, fizzim.pl added the "hold state" path (where the comment about implied_loopback is in the code above).

Note the use of "//synopsys parallel_case full_case". This tells DesignCompiler that it doesn't have to build logic to cover the illegal states (full_case), and it doesn't have to build priority into the case (parallel_case). This results in dramatically better synthesis results, but may require special handling in formal verification.

The use of "//synopsys parallel_case full_case" on this case statement (onehot combinational block) and in the regdp block described below is controlled by the state machine attribute "onehot_pragma". If this attribute is NOT set, you'll get the code shown. If it IS set, fizzim.pl will use the value string of this attribute in place of "synopsys parallel_case full_case". This can be used to add a pragma, delete one, or override this behavior entirely (by setting the attribute to a null string). If you set it to a null string, expect significantly worse synthesis results!

The use of onehot_pragma causes fizzim.pl to issue warning O12 (this can be suppressed as discussed later).

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
      ds <= 0; // default
      rd <= 0; // default
      case (1'b1) // synopsys parallel_case full_case
        nextstate[DLY] : begin
          rd <= 1;
        end
        nextstate[DONE]: begin
          ds <= 1;
        end
        nextstate[READ]: begin
          rd <= 1;
        end
      endcase
    end
  end
```

Note that this structure changed with fizzim.pl version 2.0. Older versions will look different from code show above.

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

### 6.5.2 Onehot output when "default_state_is_x" is set

Golson's code structure used above sets the nextstate vector to all zeros, then sets the single bit according to the nextstate logic. This technique cannot be used when the default_state_is_x behavior is required.

The handling of this case has changed with fizzim.pl revision 3.0. It now uses a format similar to that used for SystemVerilog (see the section on SystemVerilog output). The state bits block looks like this:

```
  // state bits
  parameter
  IDLE_BIT = 0,
  DLY_BIT = 1,
  DONE_BIT = 2,
  READ_BIT = 3;

  parameter
  IDLE = 4'b1<<IDLE_BIT,
  DLY  = 4'b1<<DLY_BIT,
  DONE = 4'b1<<DONE_BIT,
  READ = 4'b1<<READ_BIT,
  XXX  = 4'bx;

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

What's new here is the creation of parameter values for the various states, and for the all-ex state. These new parameter values are still based on the bit position parameters, but give a handy shorthand that makes the nextstate code a little cleaner:

```
// comb always block
always @* begin
  nextstate = XXX; // default to x because default_state_is_x is set
  case (1'b1) // synopsys parallel_case full_case
    state[IDLE_BIT]: begin
      if (go) begin
        nextstate = READ;
      end
      else begin
        nextstate = IDLE;
      end
    end
    state[DLY_BIT]: begin
      if (ws) begin
        nextstate = READ;
      end
      else begin
        nextstate = DONE;
      end
    end
    state[DONE_BIT]: begin
      begin
        nextstate = IDLE;
      end
    end
    state[READ_BIT]: begin
      begin
        nextstate = DLY;
      end
    end
  endcase
end
```

The sequential always block and the datapath sequential always block are unchanged from the implied_loopback case described above.

## 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:

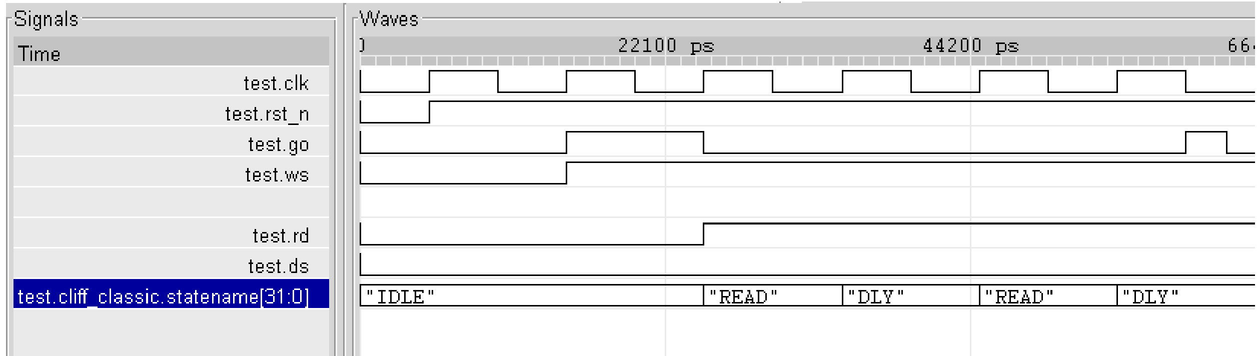
```
// This code allows you to see state names in simulation
`ifndef SYNTHESIS
reg [31:0] statename;
always @* begin
  case (1)
    state[IDLE]:
      statename = "IDLE";
    state[DLY]:
      statename = "DLY";
    state[DONE]:
      statename = "DONE";
    state[READ]:
      statename = "READ";
  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

This is automatically suppressed when SystemVerilog is selected, since the use of enumerated types in SystemVerilog output makes special code unnecessary. You can force it back on by using the “-simcode” option to 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.

# 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.

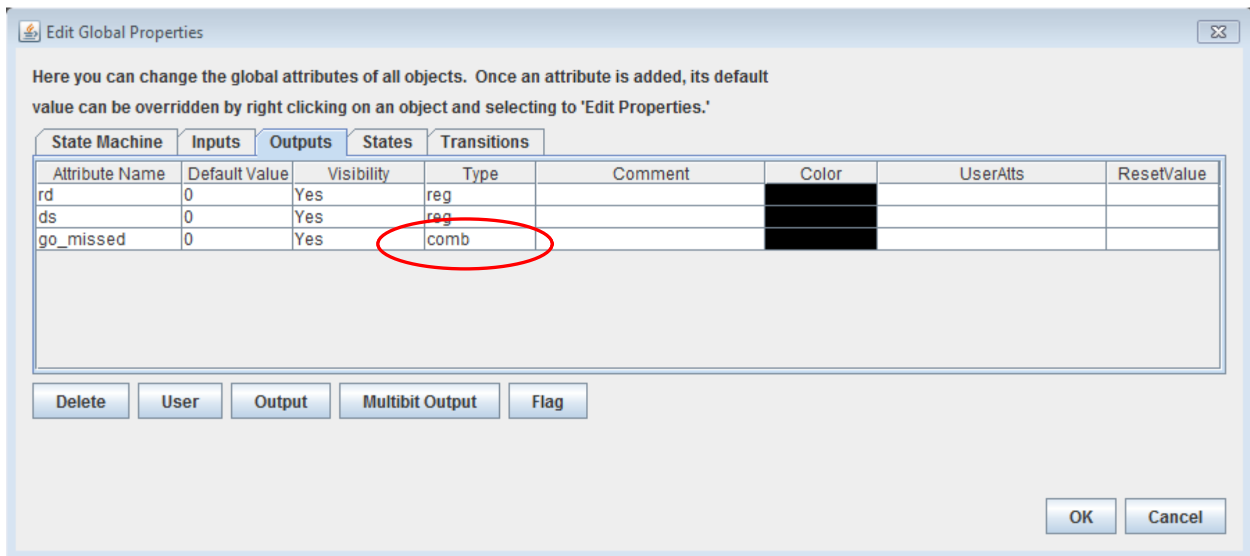
Let’s add a Mealy output to Cliff’s state machine using the on-states method.

## 7.1 Mealy outputs assigned in states

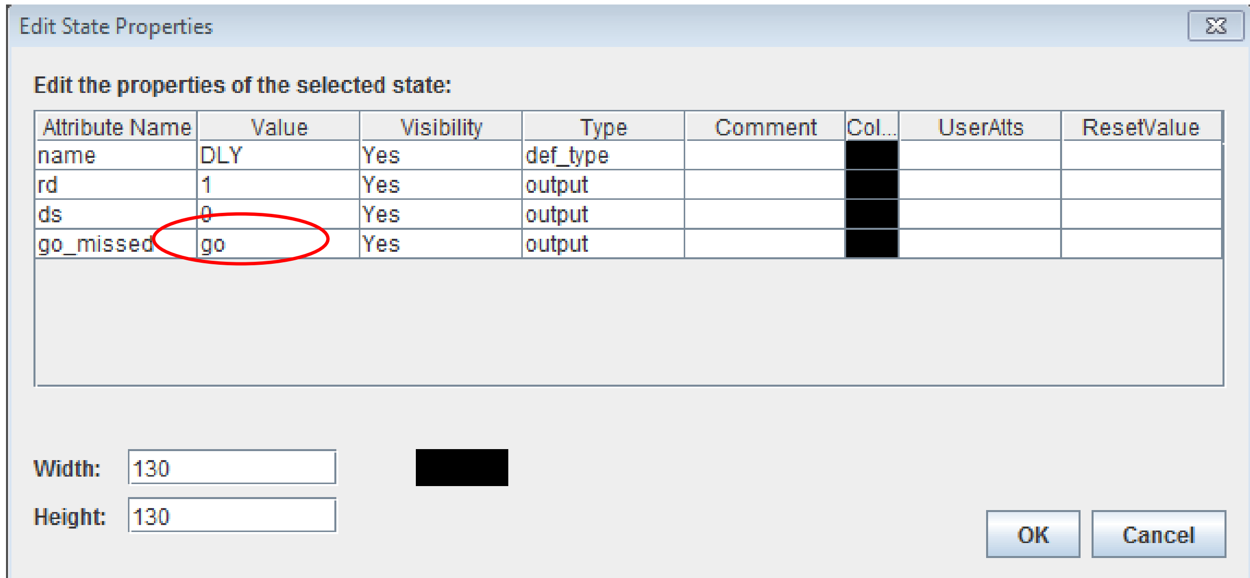
Supposed we wanted to create an output that would toggle if “go” was asserted during state “DLY”? This is just a comb output whose equation is “go” during the DLY state, and 0 at all other times.

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.

(Example: cliff_mealy_onstates.fzm)

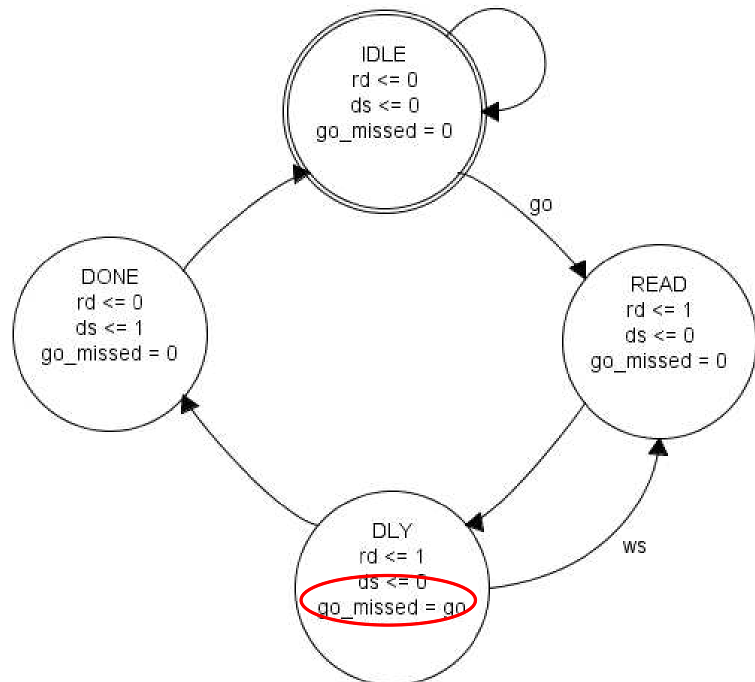


Now edit the DLY state to change the equation to “go”.




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 | | |
| OUTPUTS | | |
| rd | 0 | reg |
| ds | 0 | reg |
| go_missed | 0 | comb |
| STATES | | |
| rd | 0 | output |
| ds | 0 | output |
| go_missed | 0 | output |
| TRANSITIONS | | |
| equation | 1 | def_type |




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 = 0; // default
    case (state)
      IDLE: begin
        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
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        begin
          nextstate = DLY;
        end
      end
    endcase
  end
```

Note that this structure changed with fizzim.pl version 2.0. Older versions will look different from code show above.

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

Note the default value line (circled). To make the code easier to read, and to prevent latches, fizzim.pl will output the default value, then suppress any non-default values for the output in the case (state) block. If no default value is given, fizzim.pl will use "0". This is to provide better synthesis results out-of-the-box.

One side-effect of this may be zero-length transitions in some simulators. An alternative (used by fizzim.pl pre-version 2.0) is to set the default to the variable itself. This could be done in the example by setting the default for "go_missed" to "go_missed". This would reproduce the version 1.x behavior.

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.

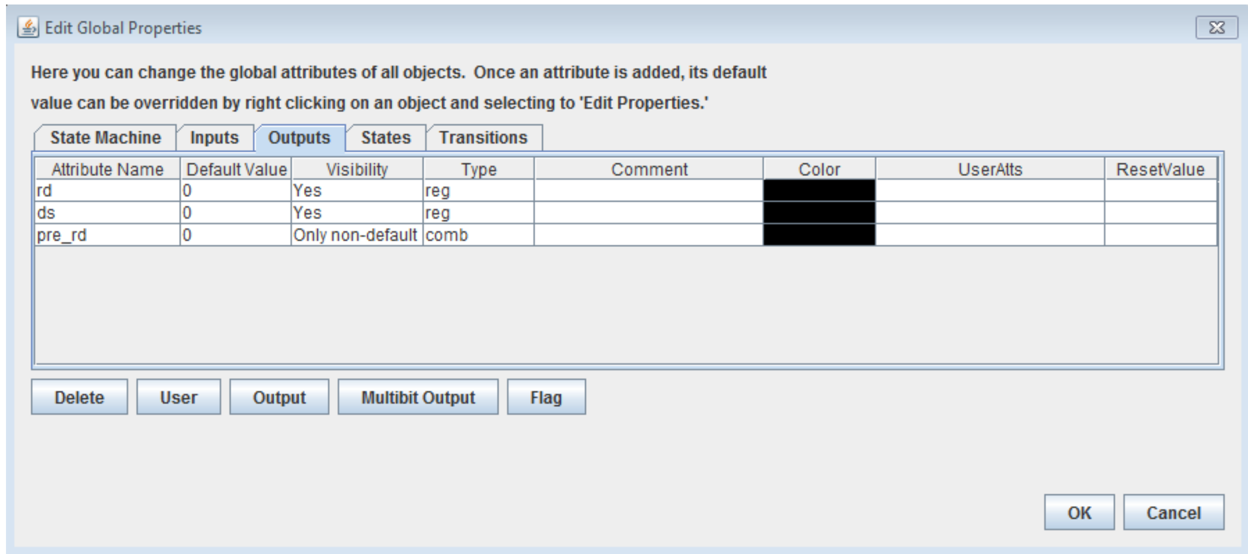
Suppose we wanted to send out an early copy of the "rd" output on the transition from IDLE to READ?

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.

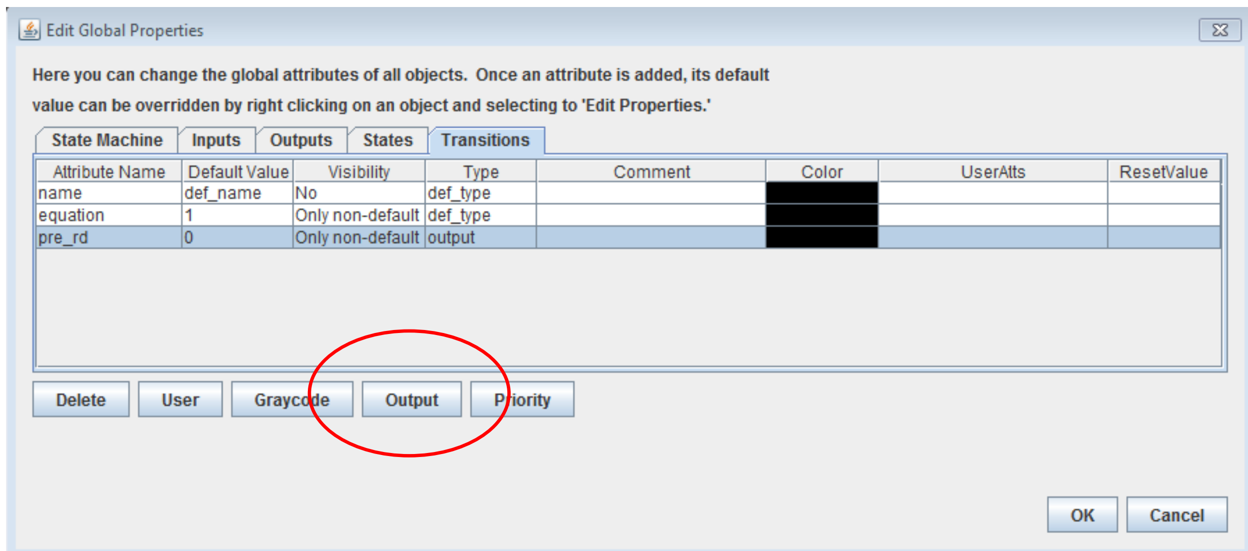
Let's take a closer look at this approach. First, we'll go back to cliff_classic and add the (comb) pre_rd output:

(Example: cliff_mealy_ontransition.fzm)




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:

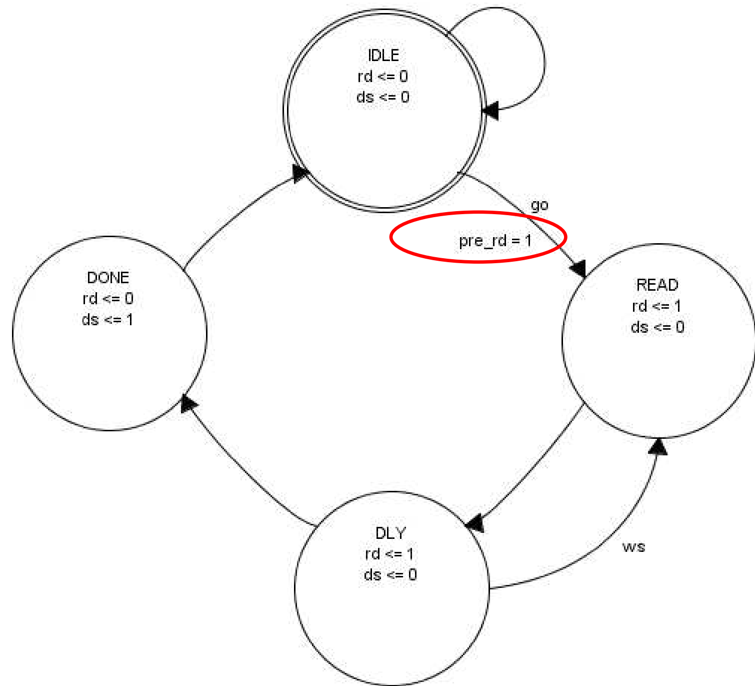
Go to the Global Attributes > Transitions tab, and use the “Output” button to add “pre_rd”. Set visibility to “Only non-default”.




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 = 0; // default
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
          pre_rd = 1;
        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
```

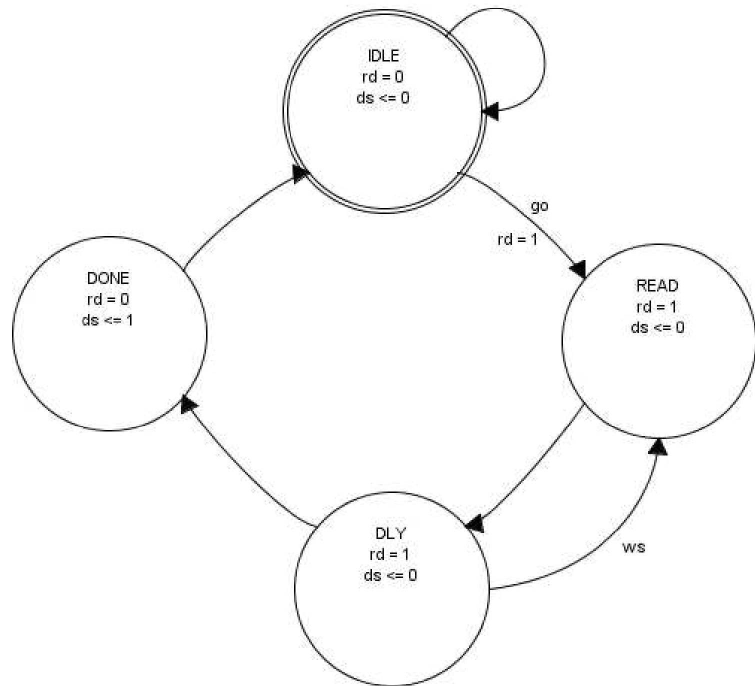
So, the output pre_rd does indeed change when the transition path is taken.

## 7.3 Mixing the styles

Also, note that you *can* mix the two styles. If the output has been created as a transition attribute, fizzim.pl will assume that you are going to use the "defined on transitions" approach, and the comb output value defined on the state will be suppressed *if it matches the default value*. If it *doesn't* match the default value, it will be output, you'll get a warning, and any non-default on-transition values for that combinational output from that state will be suppressed.

In this fsm, the output "rd" has been declared as comb, and has been added to the transition attributes table. So, fizzim.pl assumes that the definition will use the on-transitions style. The default value of "rd" is 0 for both states and transitions. "rd" has been given a value of 1 on the transition from IDLE to READ, and a value of 1 in states READ and DLY:

```
STATE MACHINE
  name                cliff_preread
  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                comb
  ds                  0                reg
STATES
  rd                  0                output
  ds                  0                output
TRANSITIONS
  equation            1                def_type
  rd                  0                output
```




(Example: cliff_preread.fzm)

The resulting output looks like this:

```
  // comb always block
  always @* begin
    nextstate = 3'bx; // default to x because default_state_is_x is set
    rd = 0; // default
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
          rd = 1;
        end
        else begin
          nextstate = IDLE;
        end
      end
      DLY : begin
        // Warning C7: Combinational output rd is assigned on transitions, but
has a non-default value "1" in state DLY
        rd = 1;
        if (ws) begin
          nextstate = READ;
        end
        else begin
          nextstate = DONE;
        end
      end
      DONE: begin
        begin
```

```
          nextstate = IDLE;
        end
      end
      READ: begin
        // Warning C7: Combinational output rd is assigned on transitions, but
has a non-default value "1" in state READ
        rd = 1;
        begin
          nextstate = DLY;
        end
      end
    endcase
  end
```

In state IDLE, the defined state value is the same as the default value, so transition values are used.

In states DLY and READ, however, rd has been assigned a non-default value of 1, so the line “rd = 1” is output, and no assignment values are used on the transitions (because all the transitions use the default value of 0). Warning C7 is issued to flag this issue.

Note that this structure effectively gives priority to non-default on-transition values, followed by non-default on-state values, followed by default transitions values.

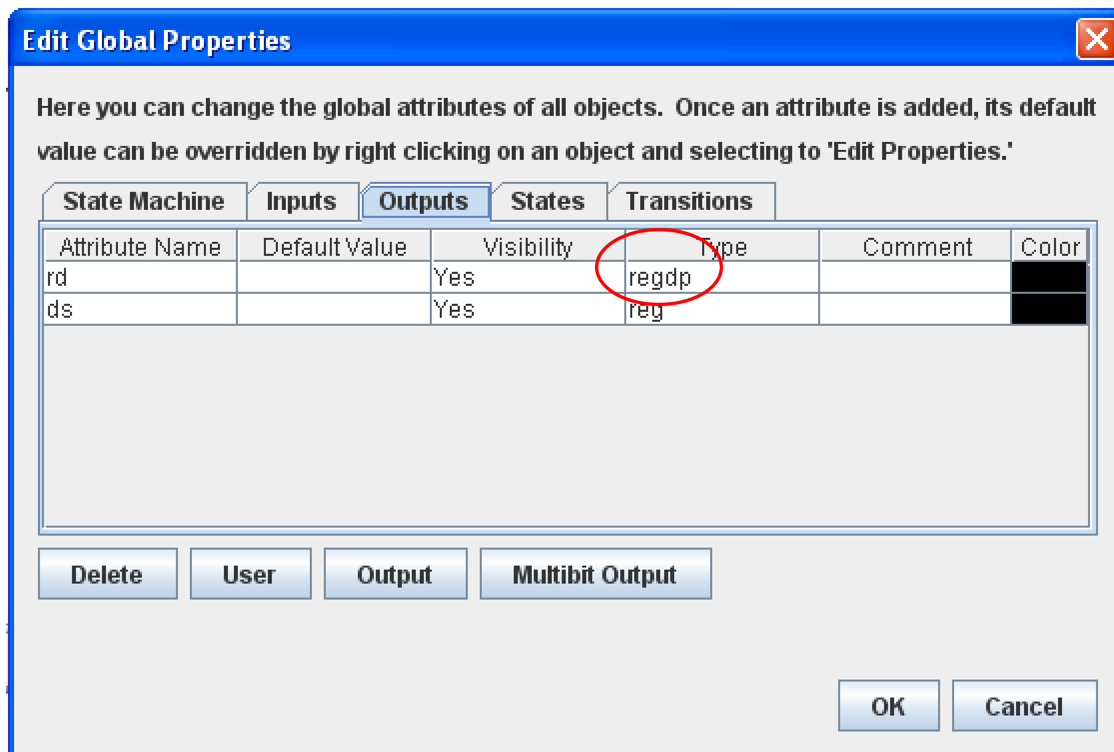
This is far from simple, so be very careful when mixing the two styles.

# 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:

(Example: cliff_rdregdp.fzm)




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
      // Warning D11: Datapath output rd has no default value - using 0
      rd <= 0; // default to zero for better synth results (no default set in
.fzm file)
      case (nextstate)
        DLY : begin
          rd <= 1;
        end
        READ: begin
          rd <= 1;
        end
      endcase
    end
  end
```

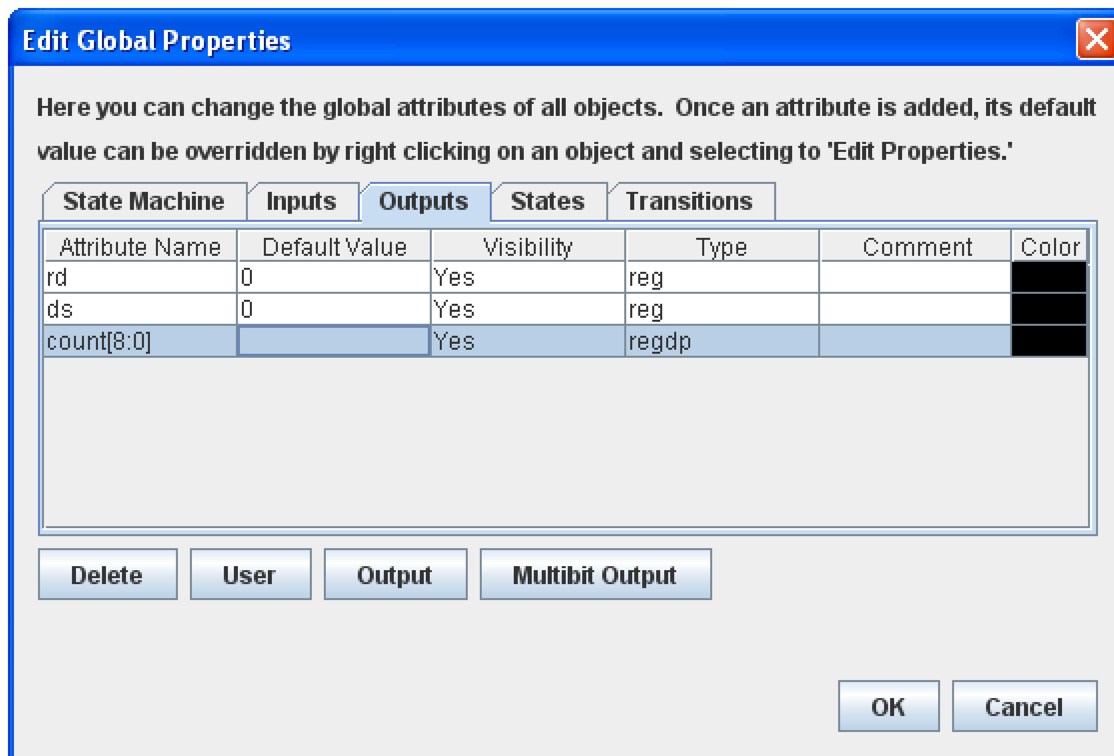
Notice that the signal rd is no longer included in the state vector, and that a third always block has been added. This third always block does a "case" on nextstate, and assigns rd on the clock edge – creating a registered rd output.

This is similar to the registered output format for onehot encoding discussed earlier. Note that this particular fsm did not have a default value assigned for rd. As mentioned earlier, fizzim.pl will default it to 0 for better synth results (and produce a D11 warning).

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. But 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.

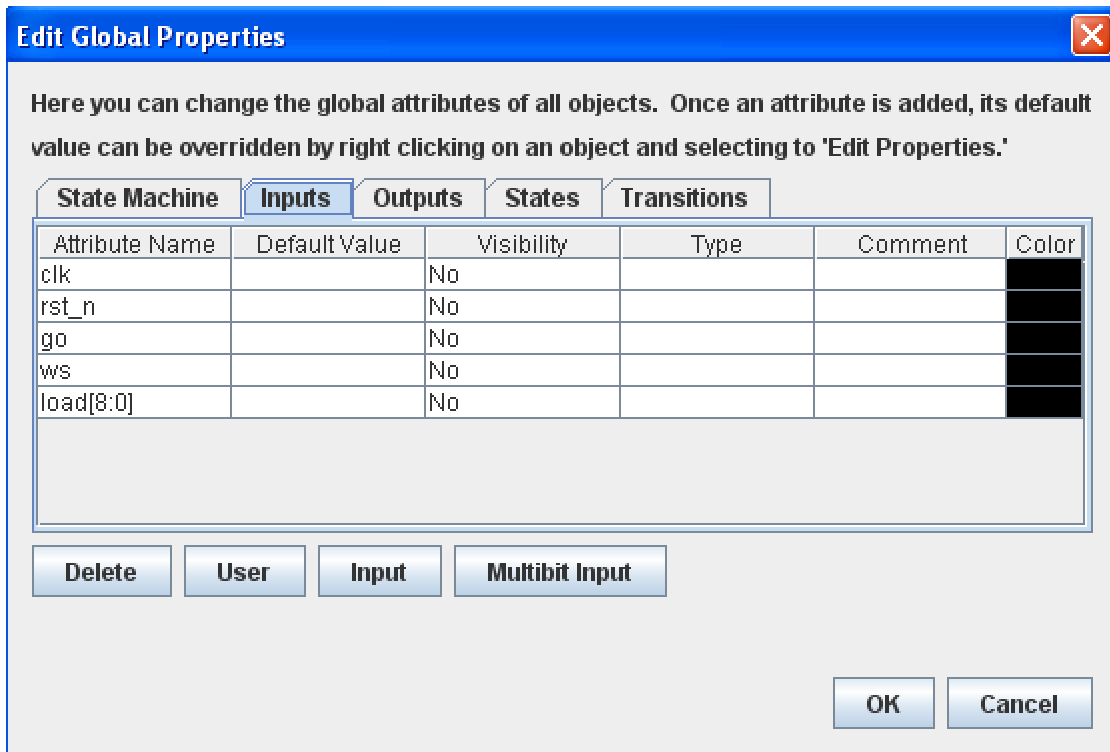
So, let's add a counter. First, we add a regdb output called count[8:0].

(Example: cliff_counter.fzm)




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.

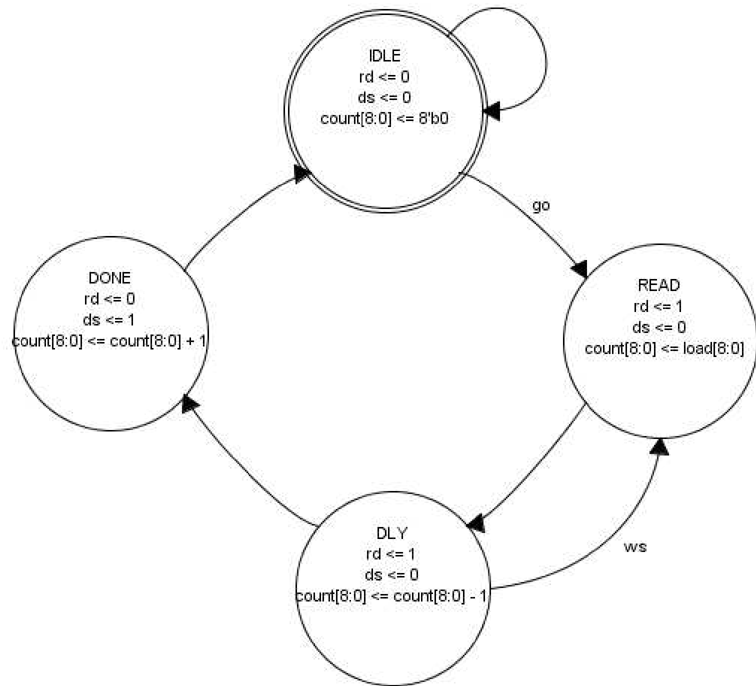



Now go around to the states and assign the counter like this:

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
      count[8:0] <= 8'b00000000; // default
      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.

Note also that fizzim.pl does a *string* compare to see if a default value matched the assigned value. That's why the IDLE case gets "count[8:0] <= 8'b0" – because the default value of "8'b00000000" doesn't match.

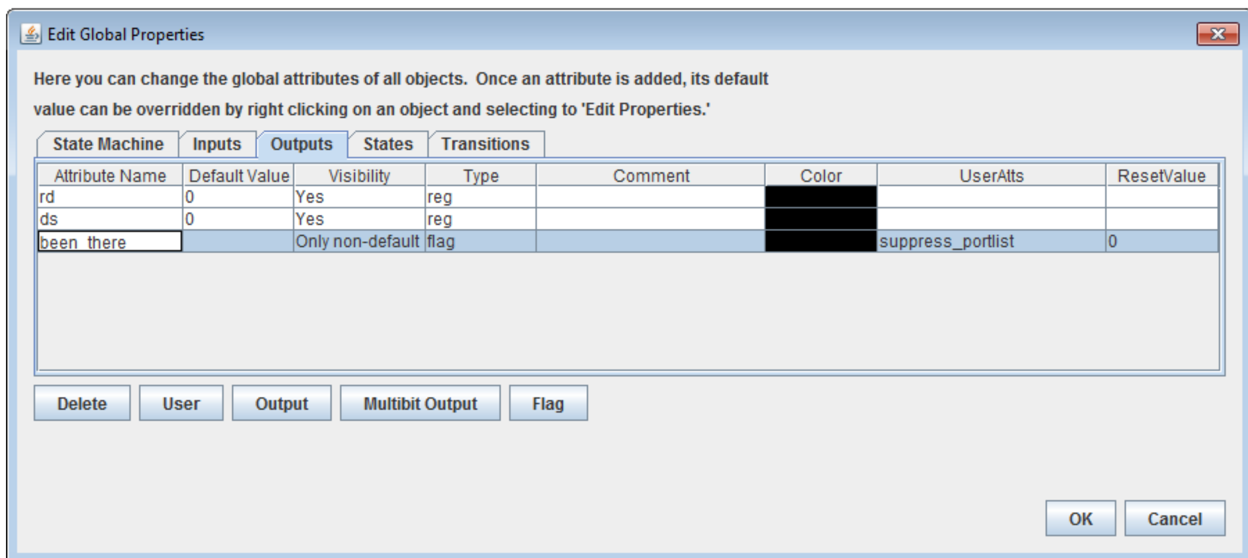
# 9 Flags (new with version 4.0)

Starting with version 4.0, there is a new output type - flags. Flags are like a cross between a comb and a regd. Like a comb, they can be assigned on both transitions and states. But, unlike comb outputs, they are registered - independent of the state vector like a regd.

## 9.1 Basic Example (flag set only on states)

One common use of flags is to keep track of where you've been. Let's look at an example. Starting with the cliff_classic fsm design again, suppose we wanted to skip the DLY state the very first time only, then run normally.

Go to Global Attributes > Outputs, and click the "Flag" button. This will give you a new entry with the Type set to "flag". We'll create one called "done_that". Set "ResetValue" to 0.




Note that you cannot set a default value on a flag (the gui won't let you). A flag is intended to hold it's state until it is explicitly changed, whereas other outputs take on their default value whenever not explicitly changed. So, it makes no sense to have a default value for a flag. But, since it is a register, it needs a reset value. That's what the new "ResetValue" column is used for.

Also note the UserAtts field has "suppress_portlist". This is included automatically when a new flag type output is created using the "flag" button. Since flags are normally only used internally, the "flag" button inserts this for you (you can delete it if you wish).

OK, now that we have our flag, we can start assigning values to it. We'll set it true in state DONE, and add a transition from READ to DONE with the equation "!been_there":

Note that the values use a "<=" to indicate that flags are registers.

For heros encoding, the resulting code looks like this:

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

  // 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;
  reg been_there;
  reg next_been_there;

  // comb always block
  always @* begin
    nextstate = 3'bxxx; // default to x because default_state_is_x is set
    next_been_there = been_there;
    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
        next_been_there = 1;
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        if (!been_there) begin
          nextstate = DONE;
        end
        else 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) begin
    state <= IDLE;
    been_there <= 0;
    end
  else begin
    state <= nextstate;
    been_there <= next_been_there;
    end
end
```

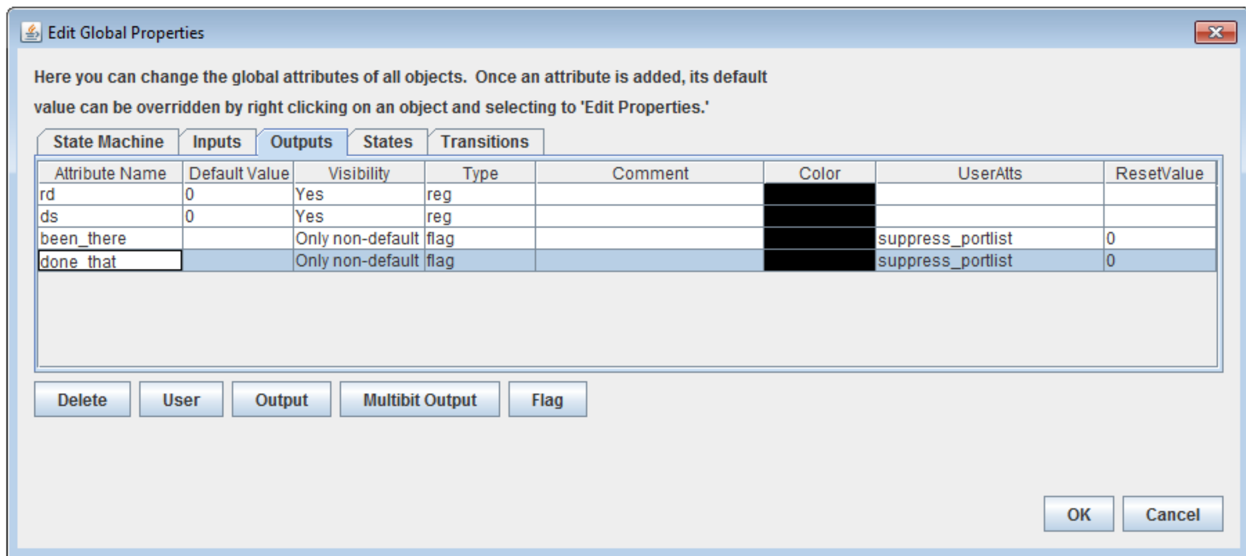
Notice all the code that got added!  "been_there" got created, along with "next_been_there", and the setting of "been_there" got added to the main sequential always block.

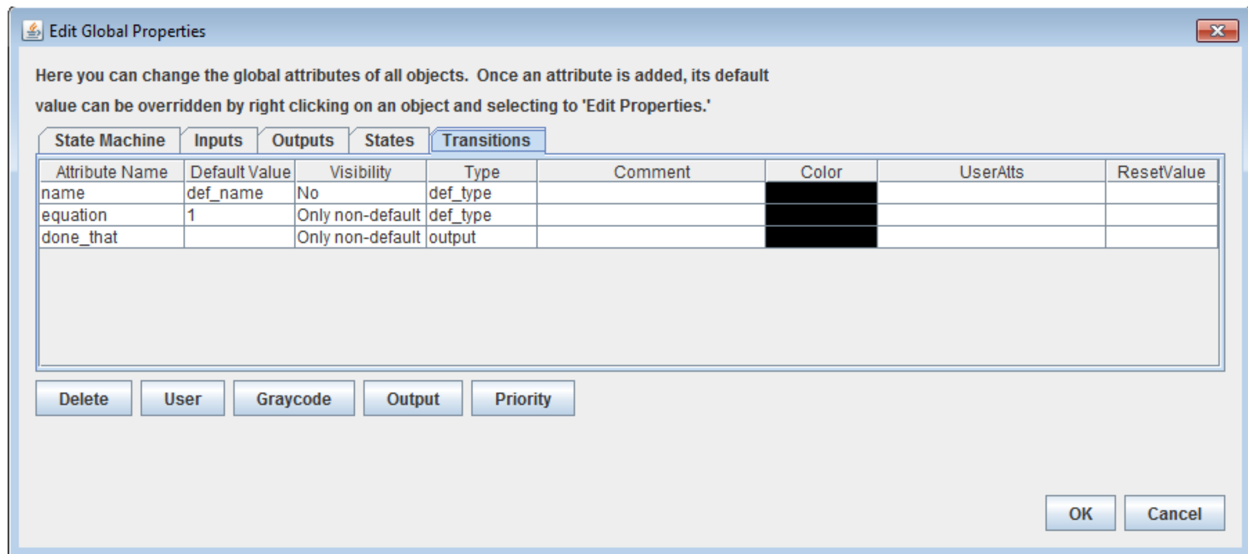
## 9.2 Flags set on transitions

OK, but we probably could have done that with a regdp.  The real power of flags is being able to set them on transitions.

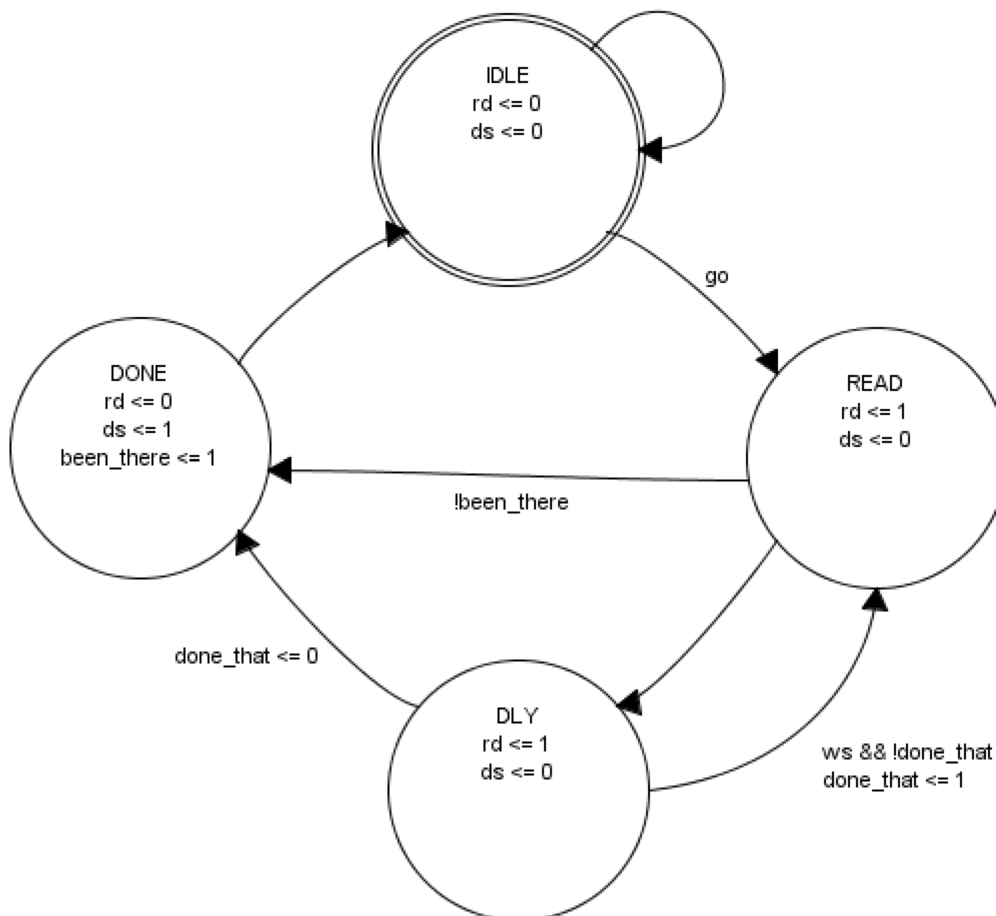
As an example, now assume that we want to change the fsm to only pay attention to "ws" once per transaction.  We can do this by setting a flag (done_that) on the way from DLY to READ, and clearing it again on the way from DLY to DONE.

Creating flags that change on transitions is like creating Mealy comb outputs.  You have to create the output, then add it to the transitions table:






Now edit the arcs from DLY to READ and from DLY to DONE as described:



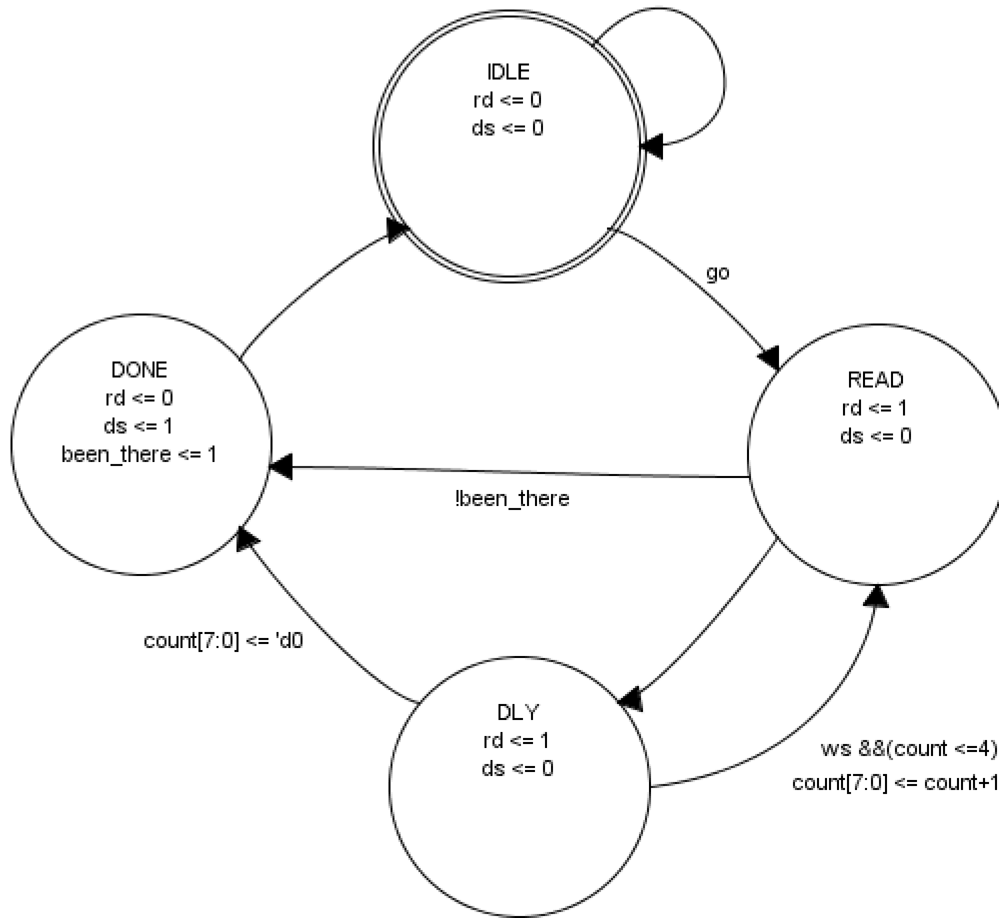

The resulting code looks like this:

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

  // 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;
  reg been_there;
  reg done_that;
  reg next_been_there;
  reg next_done_that;

  // comb always block
  always @* begin
    nextstate = 3'bxxx; // default to x because default_state_is_x is set
    next_been_there = been_there;
    next_done_that = done_that;
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
        end
        else begin
          nextstate = IDLE;
        end
      end
      DLY : begin
        if (ws && !done_that) begin
          nextstate = READ;
          next_done_that = 1;
        end
        else begin
          nextstate = DONE;
          next_done_that = 0;
        end
      end
      DONE: begin
        next_been_there = 1;
        begin
          nextstate = IDLE;
        end
      end
      READ: begin
        if (!been_there) begin
          nextstate = DONE;
        end
        else 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) begin
    state <= IDLE;
    been_there <= 0;
    done_that <= 0;
    end
  else begin
    state <= nextstate;
    been_there <= next_been_there;
    done_that <= next_done_that;
    end
end
```

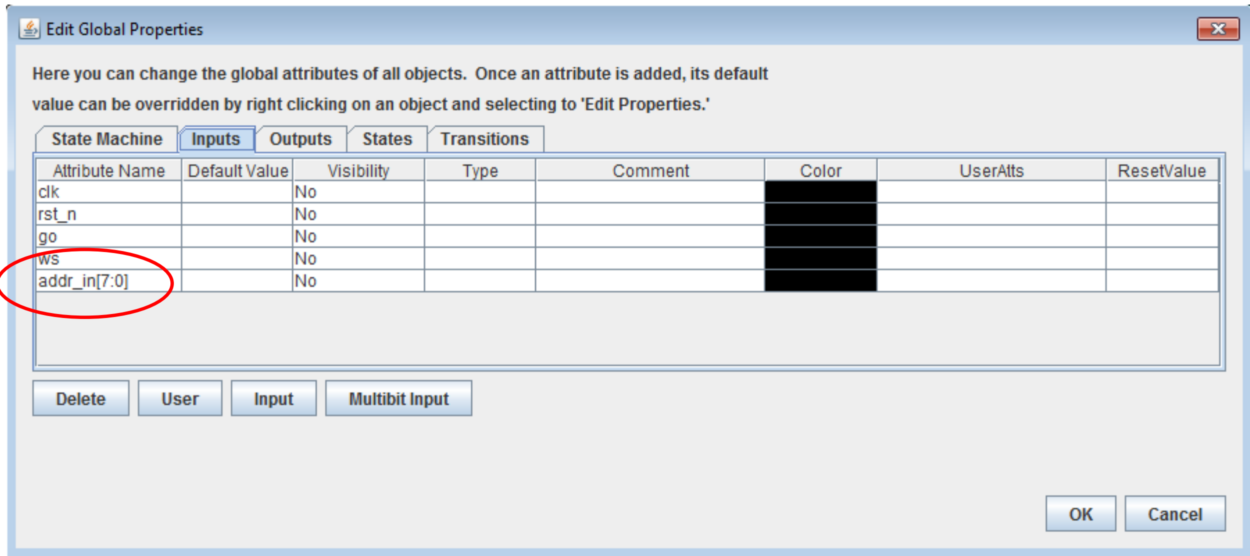
Notice the flag being set/cleared on the "if" code that corresponds to the transition arc.

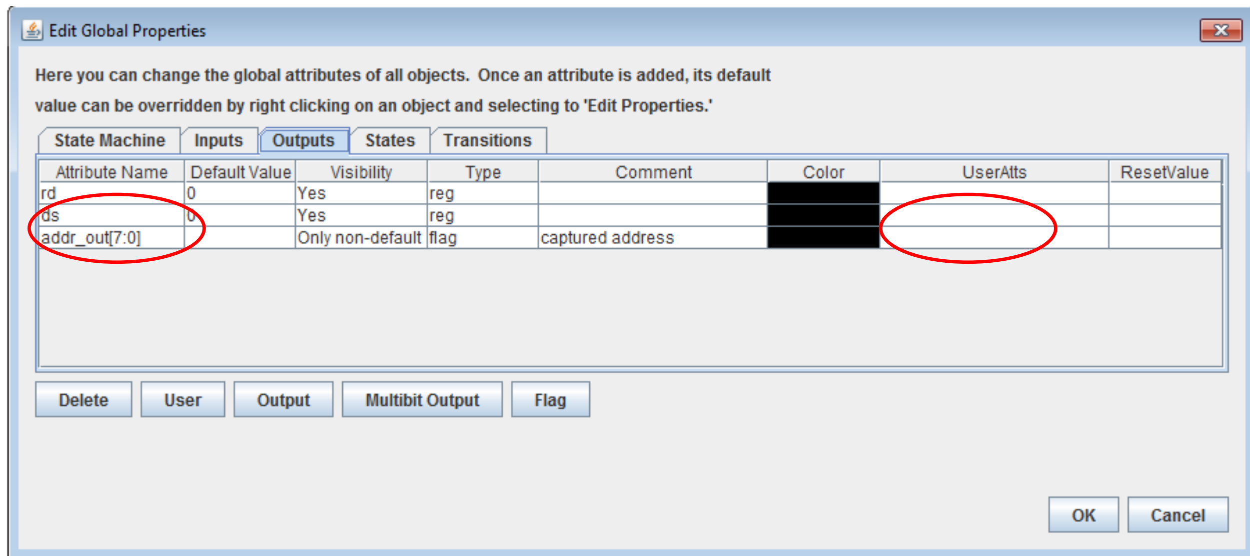
Instead of a simple flag, we could use a multibit variable, and look at a count. Change "done_that" to "count[7:0]", then change the equation to use "ws && (count <=4)", for example:



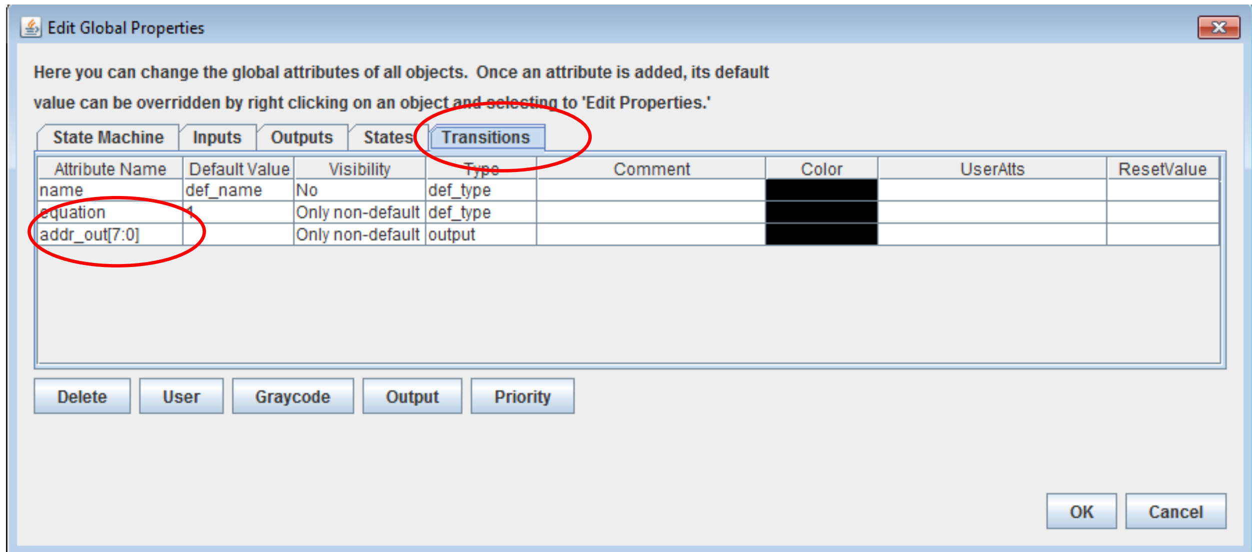

## 9.3 Capturing incoming data on an arc using flags

Flags can also be used to capture incoming data on an arc. In this case, we'll add an input addr_in[7:0] and a flag output addr_out[7:0]. But we're likely to want addr_out to be available in the portlist, so we'll delete the "suppress_portlist" from UserAtts:

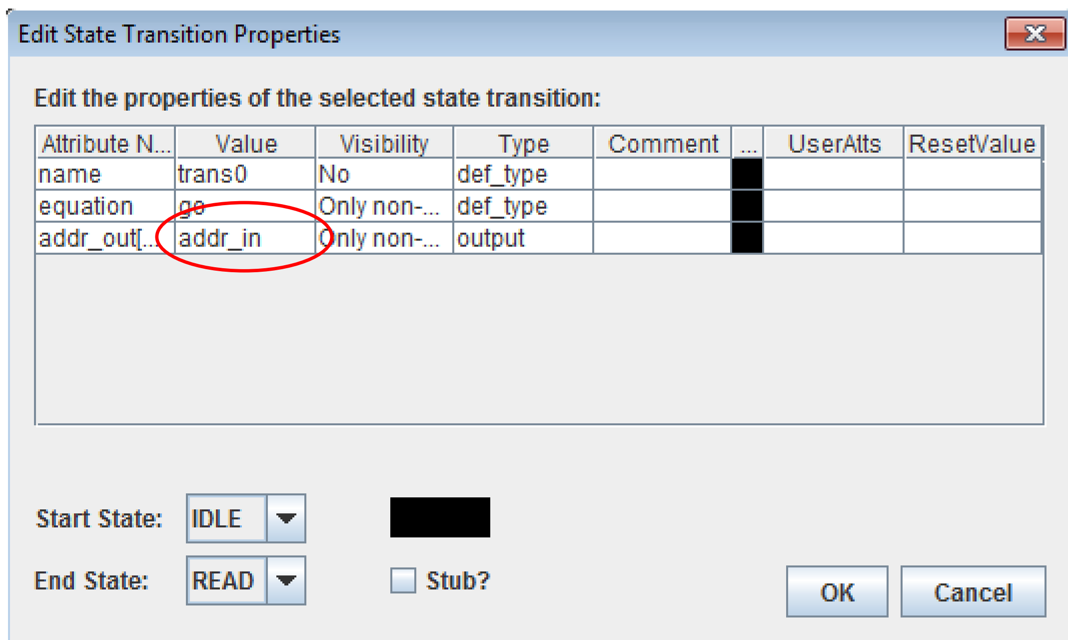


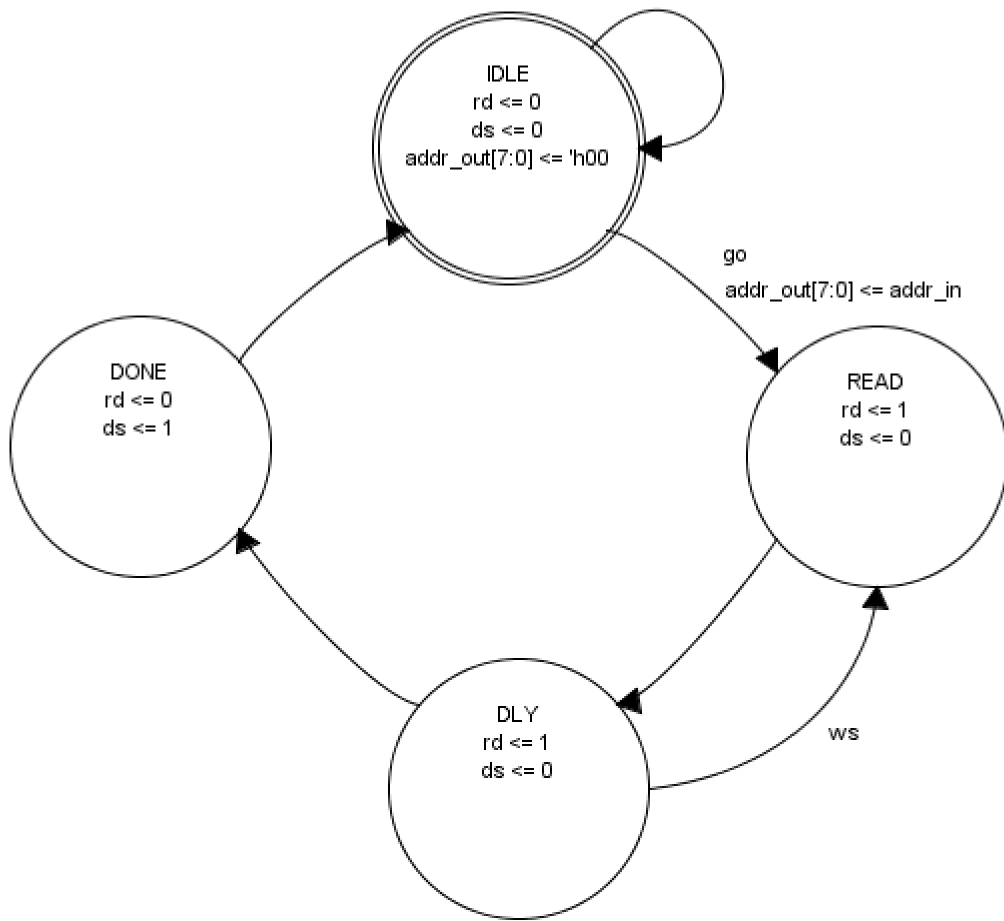

We also add addr_out[7:0] to the Transitions page so we can use it on transitions:




We double-click the transition from IDLE to READ, and enter "addr_in" as the value:




Now our state diagram looks like this:


IDLE
rd <= 0
ds <= 0
addr_out[7:0] <= 'h00
go
addr_out[7:0] <= addr_in
DONE
rd <= 0
ds <= 1
READ
rd <= 1
ds <= 0
DLY
rd <= 1
ds <= 0
ws

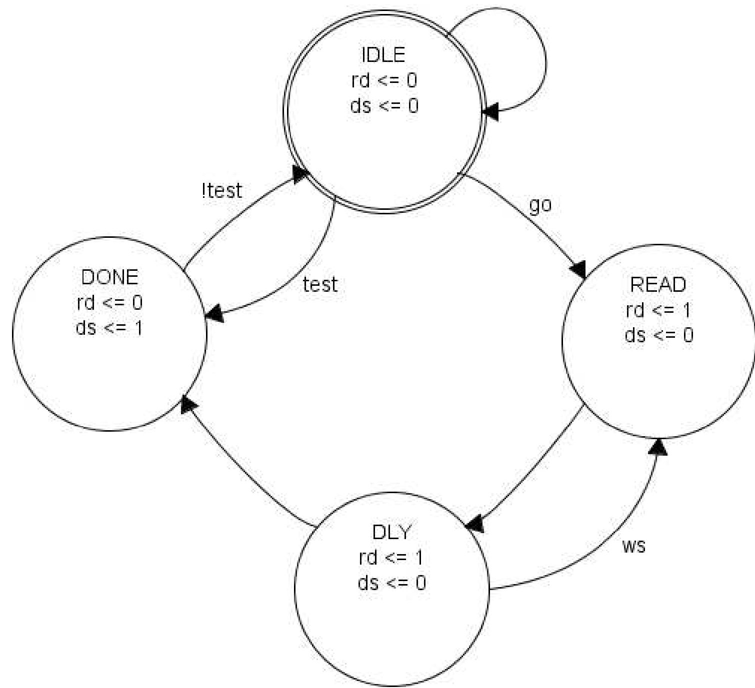
# 10 Transition priority

## 10.1 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?

(Example: cliff_priority.fzm)

| | | |
|---|---|---|
| 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.

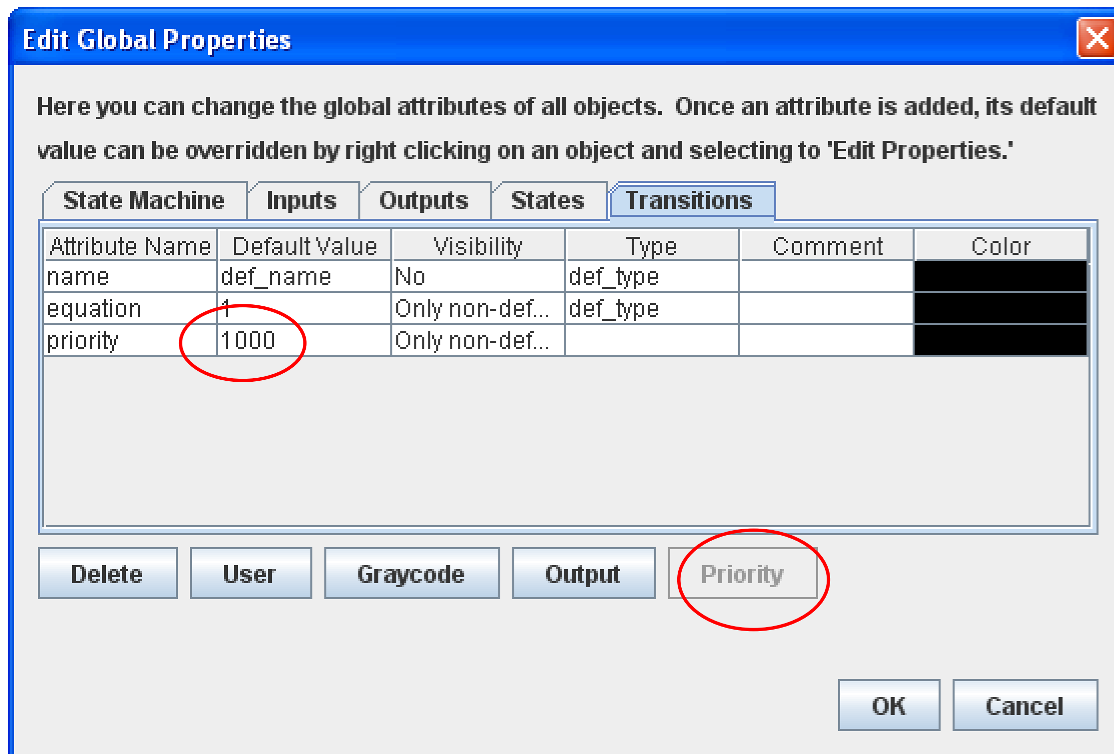
Assume that we give priority to test. We could change the equation for the IDLE->READ transition to be “!test && go”. But this gets really tedious when the transition equations get

complicated. If we were coding the FSM by hand, we would just encode the priority into the if/else stucture in Verilog by putting the "if (test)" first.

```
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.

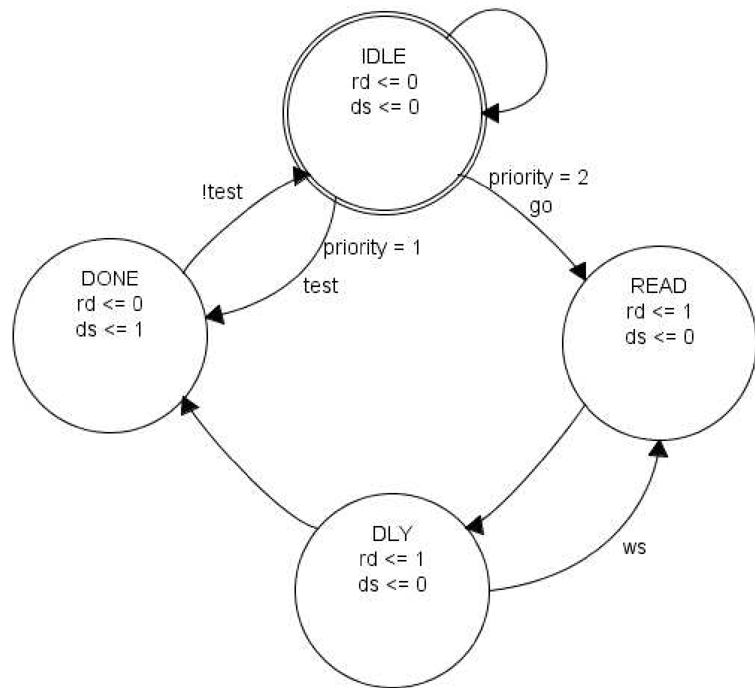
First we create a "priority" attribute for transitions in Global Attributes > Transitions. There's even a handy button to do it for you!




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
```

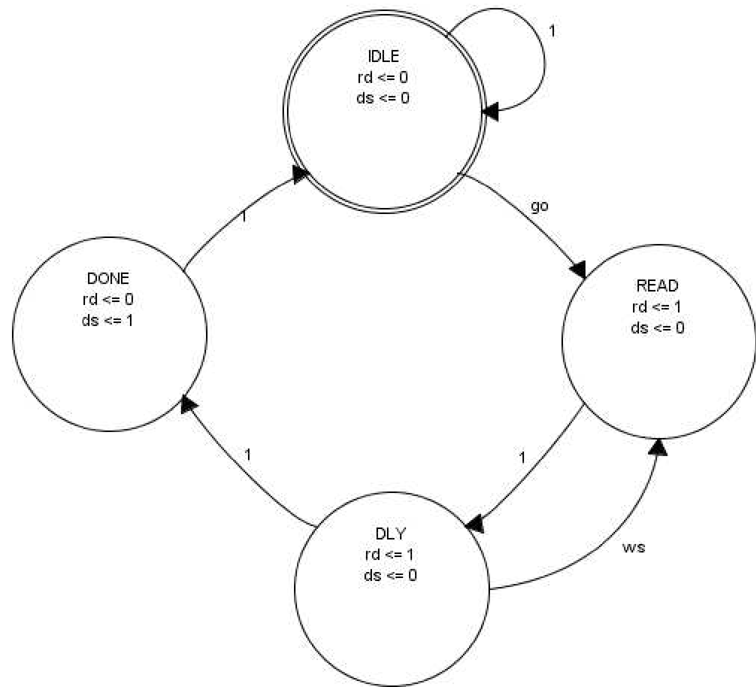
You might be wondering why fizzim.pl didn't complain about the loopback path on IDLE *before* we added the transition priorities. For that matter, why doesn't it complain about the exits from DLY? One is "ws" and the other is "1" (because this is the default value for the transition attribute "equation" 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.

## 10.2 The special case of equation equal to "1"

OK, let's go back to the original Cliff Classic state machine. We'll turn equation visiblity to YES so that all the transition equations are visible (they were 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.

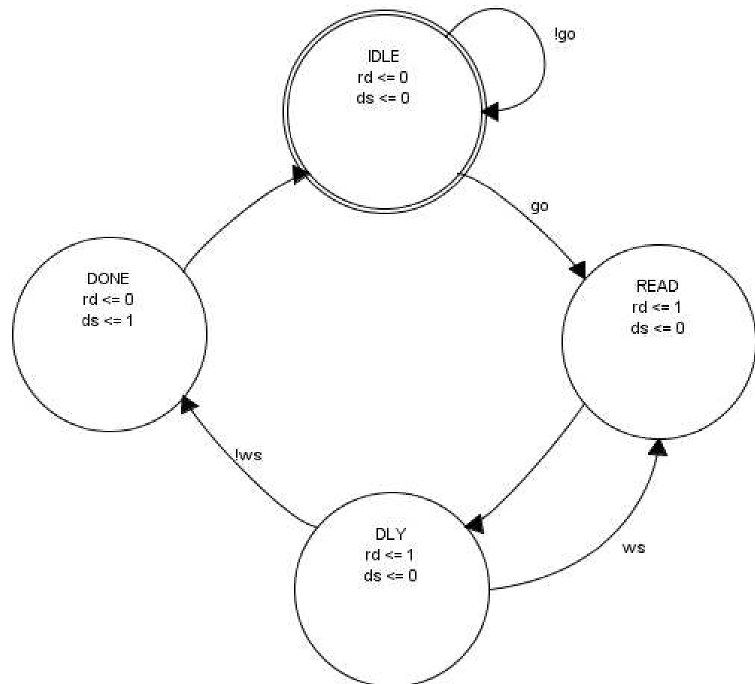
But fizzim has no easy way of inferring what is the default condition. So, you have to tell it. That's what priority is for – to tell fizzim.pl what the order of the "if" statement ought to be.

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:

(Example: cliff_classic_explicit_equations.fzm)

| 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'bxxx; // 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

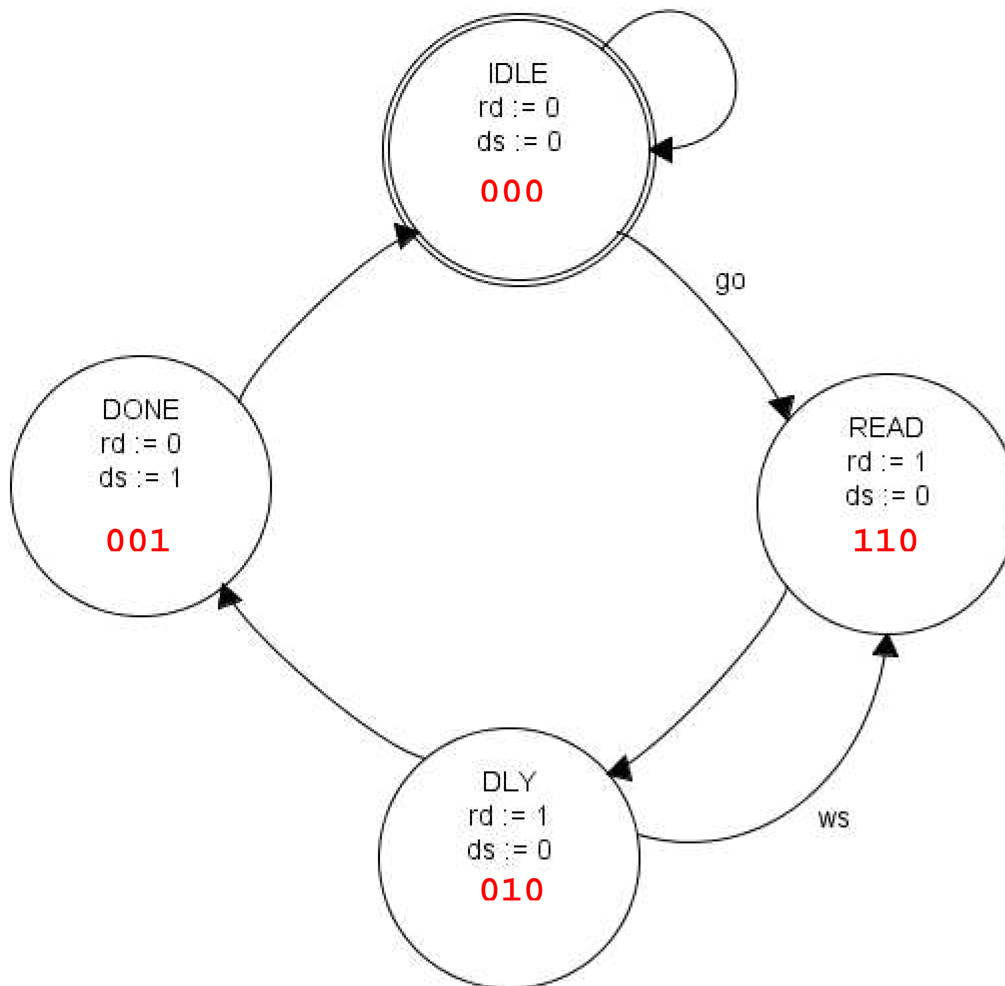
You can also turn off whole groups of warnings ("P" means priority warnings) by just using the letter:

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.

## 11 Adding gray codes

Back to Cliff Classic. Here's what heros came up with for the state encoding:




IDLE is 000, and READ is 110. Suppose we wanted the transition from IDLE to READ to be gray coded?

Easy – just add a "graycode" attribute the transition.

So, we double-click the transition, and…

(Example: cliff_graycode.fzm)

Edit State Transition Properties

Edit the properties of the selected state transition:

| Attribute Name | Value | Visibility | Type | Comment | Color |
|---|---|---|---|---|---|
| name | trans1 | No | def_type | | |
| equation | go | Only non-default | def_type | | |

Start State: IDLE    Stub?

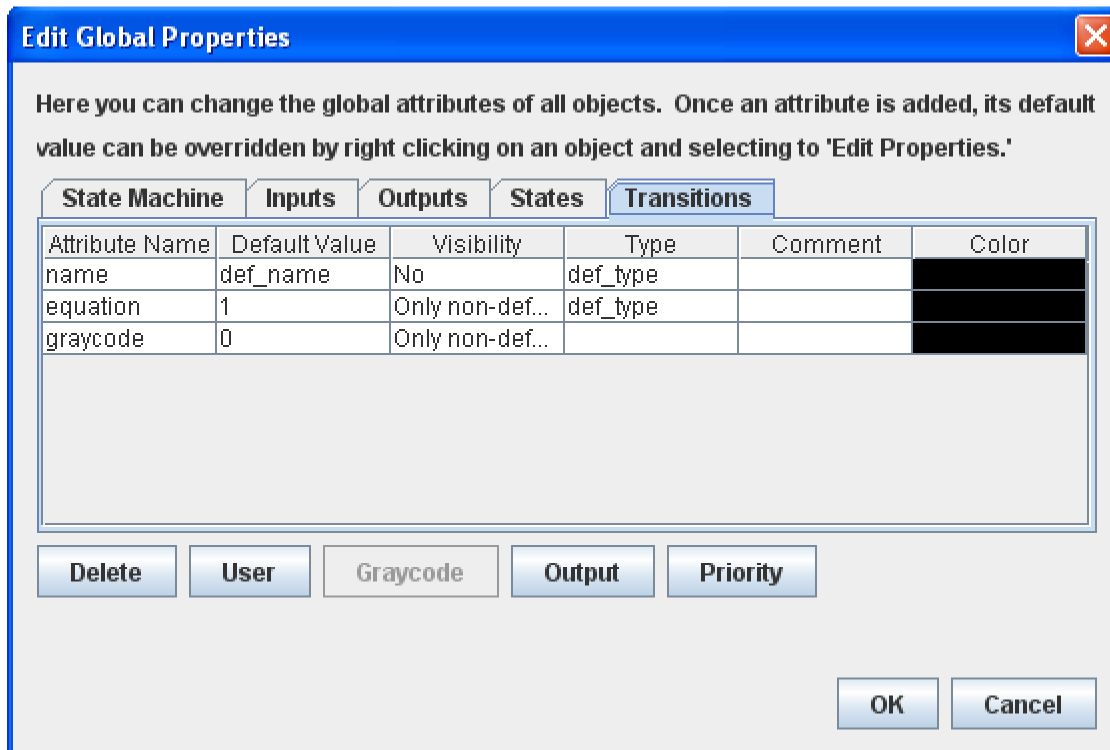
End State: READ

OK    Cancel

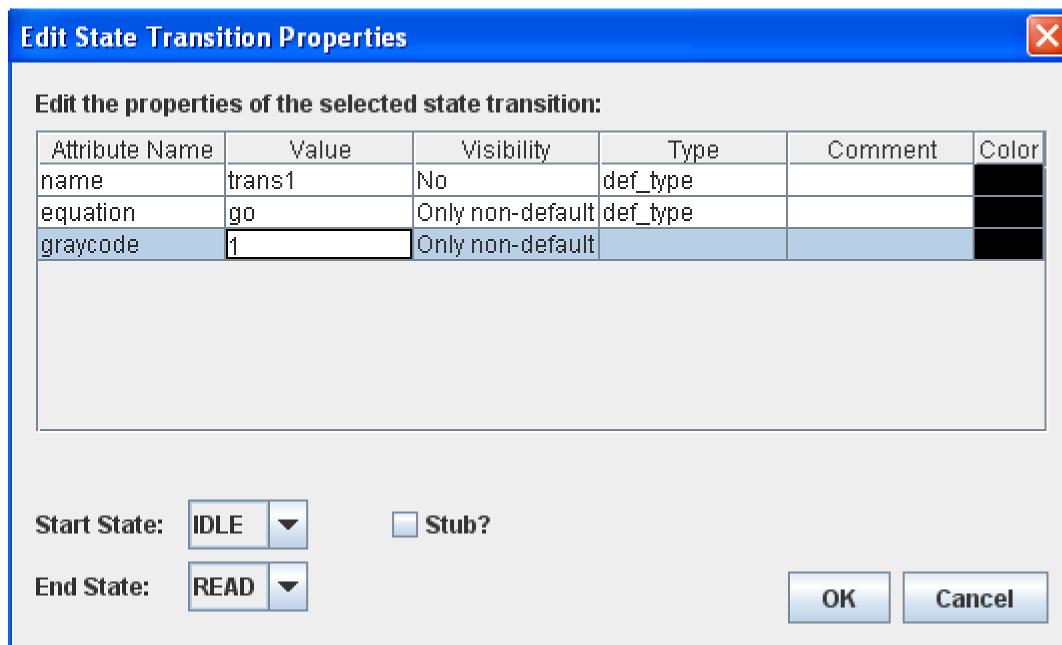
Wait, there's no "graycode" attribute, and no buttons to add one. How do we add a "graycode" attribute?

Recall that attributes on *individual* states and transitions are only available once they've been added in the global tabs.

So, select "Global Attributes > Transitions". Click the "Graycode" button. Select whatever visibility you want (we suggest "Only non-default") and click OK.




Now double-click the transition and change the value of the graycode field to “1”.



Save the file and re-run fizzim.pl, and the state encoding changes to this:

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

Note that the IDLE to READ transition is now graycoded (000 to 010). Also, a comment has been added on the transition itself:

```
      IDLE: begin
        if (go) begin
          nextstate = READ; // graycoded
        end
        else begin
          nextstate = IDLE;
        end
```

It is not always possible to make a transition gray coded. As an experiment, we'll try changing the value of "rd" in DONE to "1", then turning on gray code on the DONE to IDLE transition. The DONE to IDLE transition is a double-bit change in the registered outputs, so no gray code is possible. Save it and run fizzim.pl, and we get this:

```
Error: No valid state assignment found in range of 3 to 6 bits - try using -
minbits 7 -maxbits 7 on the command line or in be_cmd. - exiting
```

Note that it is *possible* to get this error even when gray coding is not strictly impossible. Fizzim.pl has certain rules for limiting the number of state bits to try. The error shows the range it tried. If you have a case where you think there really SHOULD be an encoding that meets all your requirements and fizzim.pl just isn't finding it, try using the "-maxbits" switch on fizzim.pl to widen the search space:

fizzim.pl -minbits 7 –maxbits 7 < cliff.fzm > cliff.v

In this case, it just isn't possible, so you still get the error:

```
Error: No valid state assignment found in range of 7 to 7 bits - try using -
minbits 8 -maxbits 8 on the command line or in be_cmd. - exiting
```

Notice that you *could* get around this by making one or more of your outputs type "regdp" (see below). This would allow the gray code, but whether this is *really* a solution is open to debate. Sure, the state machine is gray coded, but the outputs can now be out-of-sync with the state machine. Whether this meets the original need for gray coding is up to the designer.

Gray coding is, of course, not possible with onehot encoding.

## 12 Mapping states to values in heros

In addition to the impossible gray code example shown above, there are other cases where fizzim.pl may have trouble finding a mapping of states to codes that meets all the user requirements.

Starting with version 4.0, the algorithm got a little smarter, and it also got more controllability. To avoid long runtimes, fizzim.pl will only attempt a limited number of bit ranges. If it cannot find a correct mapping, it will error out with messages as shown above:

```
Error: No valid state assignment found in range of 3 to 6 bits - try using -
minbits 7 -maxbits 7 on the command line or in be_cmd. - exiting
```

At this point, you might want to examine your requirements and see that they really do make sense. If you still think fizzim.pl should be able to find a mapping, start bumping -minbits and -maxbits.

Fizzim.pl can also error out of this mapping code if it runs too many iterations:

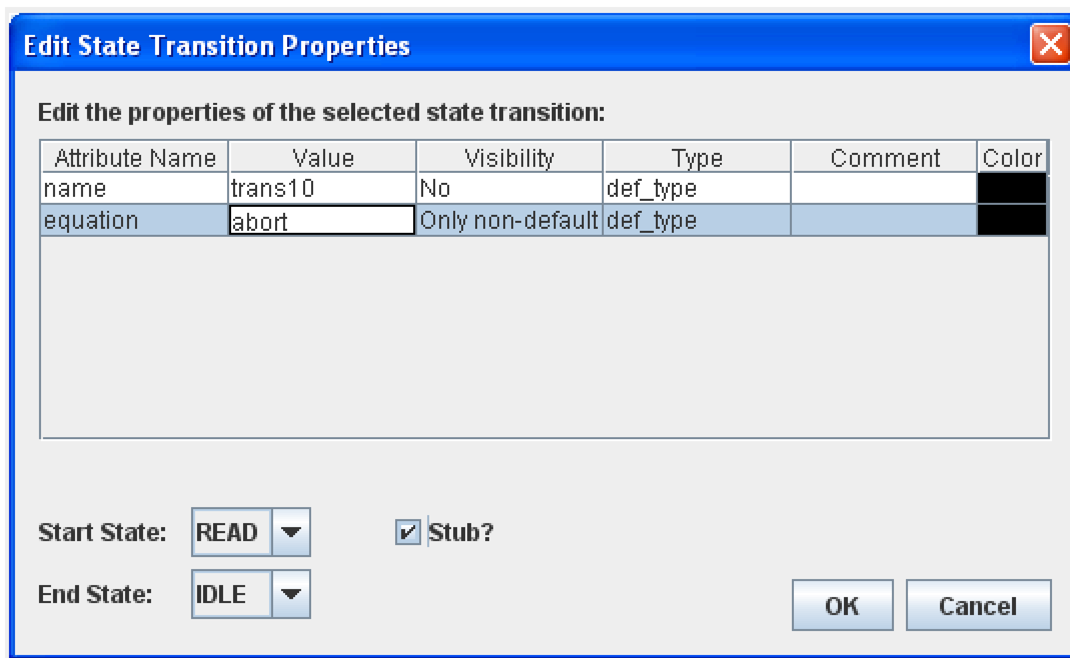
```
Error: No valid state assignment found after 10000000 iterations.  Try using -
minbits 8 or increase max iterations using -iterations - exiting
```

If you're still convinced that their should be a mapping, and you're willing to expend some more cpu time looking for it, you can increase the iterations limit by using the "-iterations" option on the command line or in be_cmd. Using the suggested -minbits value will skip all bit lengths that are known to fail, thus speeding up the search and not consuming those iterations.

# 13 Stubs

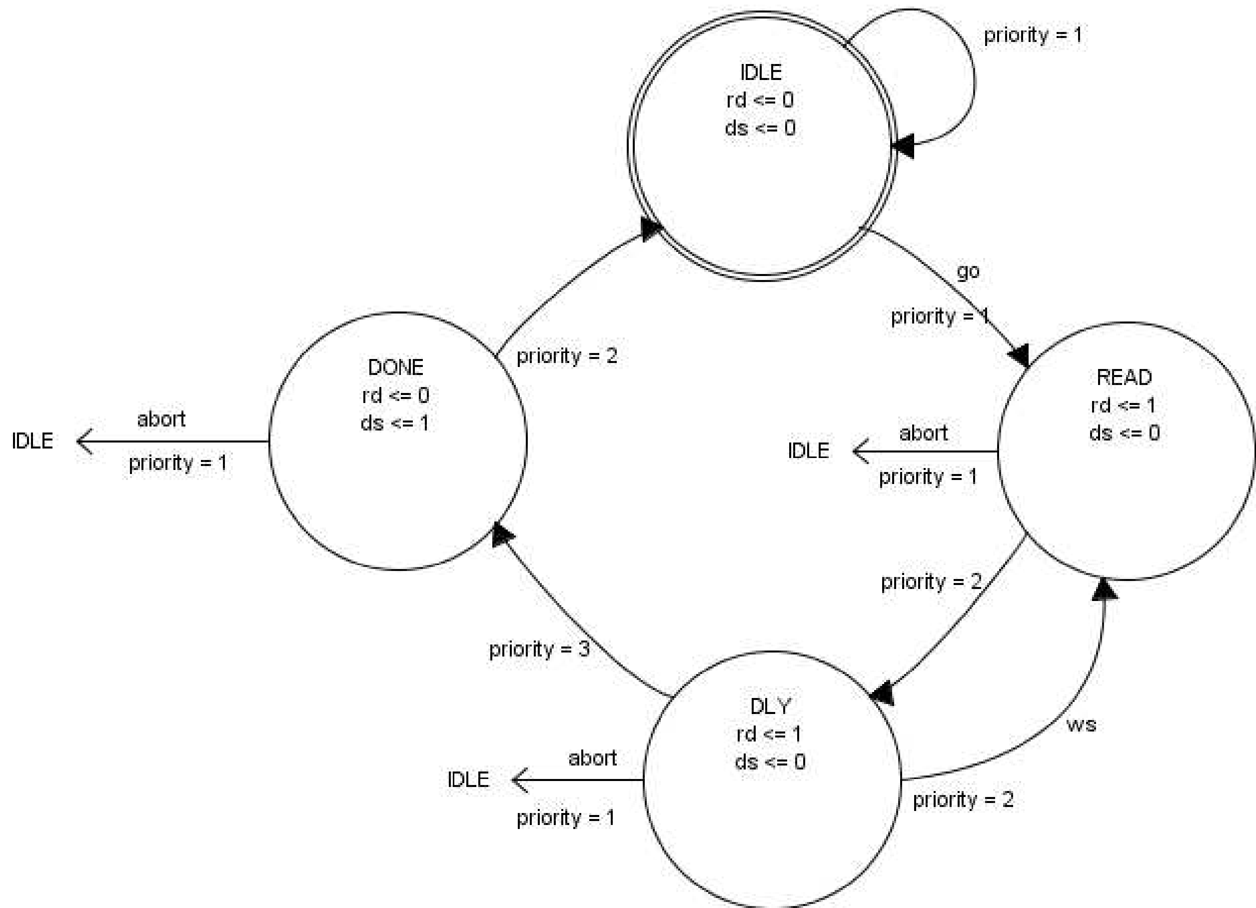
Suppose we wanted to add an "abort" input to Cliff Classic that would cause the FSM to go back to idle, no matter what state it happened to be in? It's easy enough to add the transitions, but the resulting FSM has so many arcs that it becomes very difficult to read.

To avoid this problem, transitions can be designated as "stubs". Stubs are just like regular transitions, except the arc only goes to a stub symbol with the name of the destination state. Here's how we would create the stub back to idle on abort for Cliff Classic. After adding "abort" as an input, we create new transition arcs back to IDLE for each state by right-clicking in open space and selecting "New State Transition". This brings up a box where we can select the states and set the equation. To make it a stub, check the "Stub?" box.




We'll also have to add priorities to the transition attributes and assign the DLY->READ transition on "ws" a lower priority than the "abort" transition.

(Example: cliff_abort_stub.fzm)




The resulting Verilog has all the expected transitions:

```
always @* begin
  nextstate = 3'bxxx; // 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 (abort) begin
        nextstate = IDLE;
      end
      else if (ws) begin
        nextstate = READ;
      end
      else begin
        nextstate = DONE;
      end
    end
    DONE: begin
      if (abort) begin
```

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

# 14 Bringing out internal signals

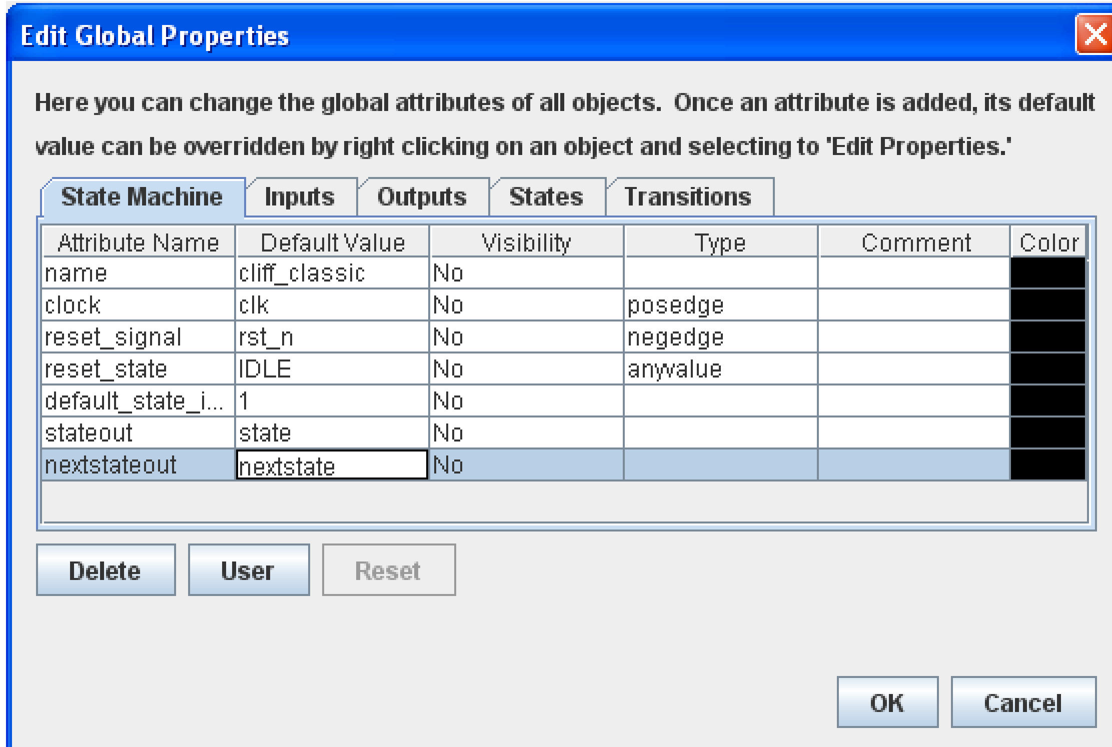
## 14.1 Renaming internal signals

The default values of the state vector, nextstate vector, and ascii statename are “state”, “nextstate”, and “statename”, respectively. You can change this on the command line using the switches “-statevar”, “-nextstatevar”, and “-statenamevar”.

## 14.2 Bringing out internal signals

Sometimes the designer wants to bring the internal state vectors (state and/or nextstate) out as ports on the module. This is *not* done by adding them to the output list (fizzim.pl will error out if you do this). Instead, there are special FSM global attributes that you can set:

- “stateout” – value field is the name of the signal to use. Do not use [m:n] – size will be determined automatically by fizzim.pl.
- “nextstateout” – value field is the name of the signal to use. Again, do not use [m:n] – size will be determined automatically by fizzim.pl.

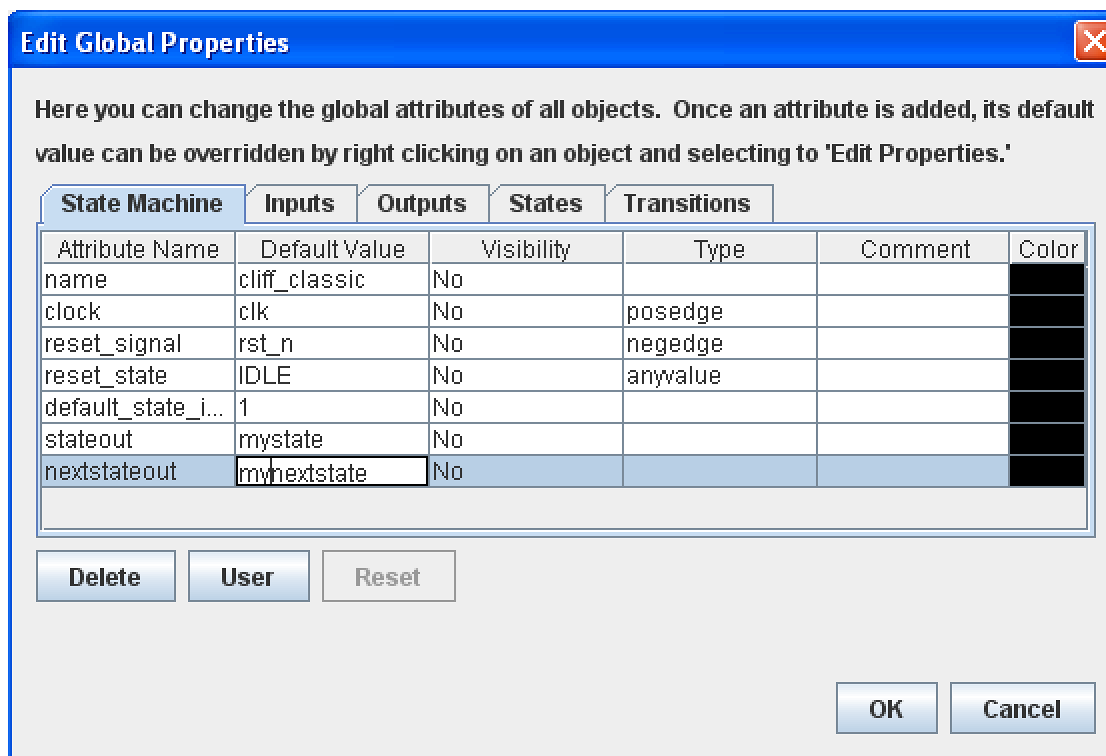


If the signal name matches the internal signal name ("state" and "nextstate" by default – see "renaming internal signals" below), fizzim.pl will output these directly.

```
module cliff_classic (
  output wire ds,
  output wire rd,
  output reg [2:0] state,
  output reg [2:0] nextstate,
  input wire clk,
  input wire go,
  input wire rst_n,
  input wire ws );
```

(Example: cliff_stateout.fzm)
If not, it will create a new wire with the correct width for the output and assign this wire to the internal signal. Suppose we change the names to "mystate" and "mynextstate".

(Example: cliff_mystateout.fzm)




```
module cliff_classic (
  output wire ds,
  output wire rd,
  output wire [2:0] mystate,
  output wire [2:0] mynextstate,
  input wire clk,
  input wire go,
```

```
  input wire rst_n,
  input wire ws );



  // 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;
  assign mystate = state;
  reg [2:0] nextstate;
  assign mynextstate = nextstate;
```

In other words, whatever you name it, fizzim.pl will do the right thing.

Note that SystemVerilog does not support outputting the state variables as module ports. This is because the state variables are enumerated types and not available outside the module (technically, the declaration could be moved outside the module, but then there is no way of knowing if this conflicts with something else in the design).

So, in SystemVerilog, the internal state/nextstate variables *must* be different from the port names. This can be accomplished in one of two ways. First, you can simple give the output a different name, like the "mystate/mynextstate" example above. In this case, the SystemVerilog output would look like this:

```
module cliff_classic (
  output logic ds,
  output logic rd,
  output logic [2:0] mystate,
  output logic [2:0] mynextstate,
  input logic clk,
  input logic go,
  input logic rst_n,
  input logic ws
);


  // state bits
  enum logic [2:0] {
    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
    XXX = 'x
  } state, nextstate;

  assign mystate = state;
  assign mynextstate = nextstate;
```

The other approach is to use the –statevar/-nextstatevar options to rename the internal names. The example file cliff_stateout.fzm normally produces an error when run with –lang SystemVerilog:

```
module cliff_classic (
  output logic ds,
  output logic rd,
```

```
Error: Cannot use state or nextstate variables as module ports in
SystemVerilog - you must rename them.  See documentation for details. -
 exiting
```

But when run with "-statevar stateI –nextstatevar nextstateI", it produces this:

```
module cliff_classic (
  output logic ds,
  output logic rd,
  output logic [2:0] state,
  output logic [2:0] nextstate,
  input logic clk,
  input logic go,
  input logic rst_n,
  input logic ws
);


  // state bits
  enum logic [2:0] {
    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
    XXX = 'x
  } stateI, nextstateI;

  assign state = stateI;
  assign nextstate = nextstateI;
```

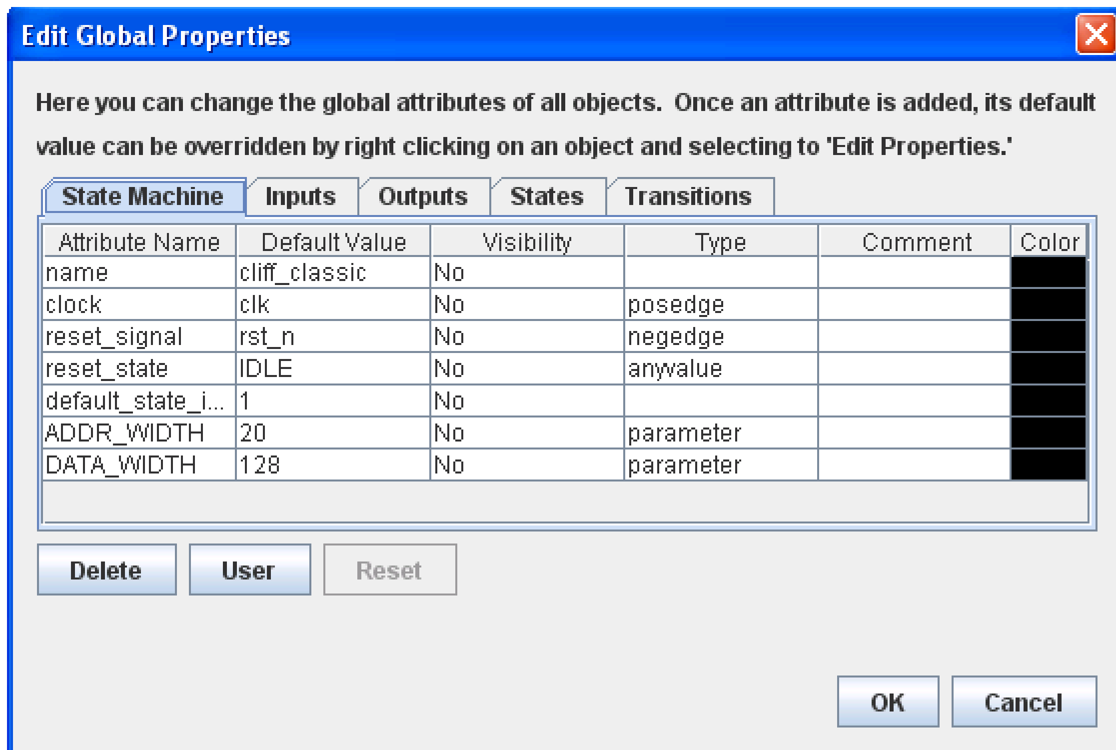
Either way, the result is the same - different names for the ports and the internal signals.

## 15 Using parameters

Parameters are a very handy feature of the verilog language. They allow code to be written once and used in a variety of contexts with different widths, for example. They also provide a mechanism for applying meanful names to values – fizzim.pl uses parameter statements to assign names to the state values, for example.

Parameters are often preferable to `define values because they are more tightly bound to their module, instead of being global. If fizzim.pl used `define to specify the statenames, for example, this might accidentally redefine this value elsewhere in the design.

Stating with version 3.6, fizzim supports parameters. They are entered in the gui as state machine attributes of type "parameter":

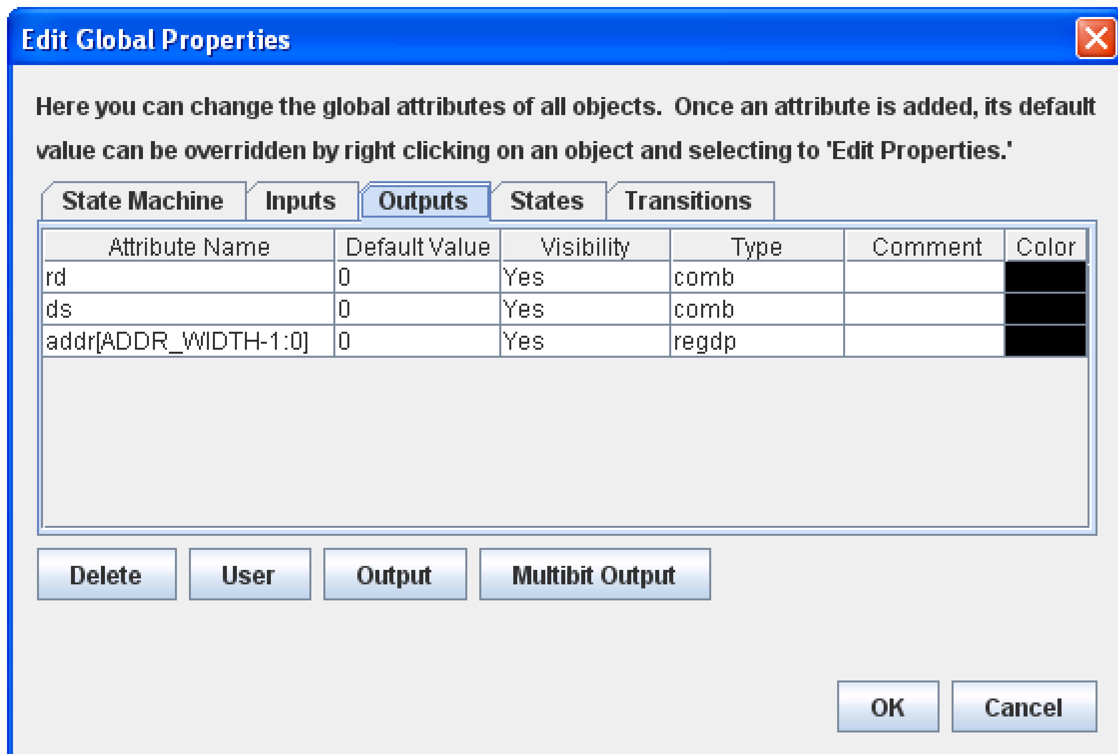


This results in a "parameter block" begin added to the module statement:

```
module cliff_classic
  #(
    parameter ADDR_WIDTH = 20,
    parameter DATA_WIDTH = 128
  )(
  output reg [ADDR_WIDTH-1:0] addr,
  output reg ds,
  output reg rd,
  input wire clk,
  input wire go,
  input wire rst_n,
  input wire ws
);
```

The default values specified in the gui will be used as the defaults.

These parameter values can then be used to specify things inside the fsm.  In the example above, ADDR_WIDTH was used as part of the declaration of "addr":




Note that "addr" is of type "regdp".  *Parameters cannot be used to size type "reg" outputs!*  This is because fizzim.pl needs to know the size of type "reg" outputs at compile time in order to create the state vector assignments.

## 16 Inserting random bits of code at strategic places

Fizzim.pl has the following attributes that allow you to insert random bits of code at strategic locations:

- insert_at_top_of_file – string from value field will be inserted at the top of the file, before the "module" statement.
- insert_in_module_declaration – string from value field will be inserted into the module declaration.
- insert_at_top_of_module – string from value field will be inserted after the module statement, but before anything else.
- insert_at_bottom_of_module - string from value field will be inserted just before the endmodule statement.
- insert_at_bottom_of_file - string from value field will be inserted after the endmodule statement.

Using these "hooks", it should be possible to insert about anything you want into the Verilog code.

Since it is common to insert a large chunk of code at the top of the file (copyright statement), there is a special attribute that will read from a file an put whatever it finds at the top of the output file:

- include_at_top_of_file – pointer to file whose contents should be inserted at the top of the file.

Currently, the other insert_at attributes have no similar file provision, although it would be easy to add. There just doesn't seem to be any great need for it.

# 17 Inserting comments

All of the attribute forms have a comment field.  Some of these comments are intended for the visible table in the gui, some are for the Verilog code, some show up in both, and some are utterly useless.  Here's a basic guide:

| Comment Field | Attribute | On Diagram? | In Verilog? |
|---|---|---|---|
| Globals > State Machine | name | Yes | Yes – on "module" line |
| | clock | Yes | No |
| | reset_signal | Yes | No |
| | reset_state | Yes | No |
| | <user atts> | Yes | No |
| Globals > Inputs | <all> | Yes | Yes – on input declaration of module statement |
| Globals > Outputs | <all> | Yes | Yes – on output declaration of module statement |
| Globals > States | name | No | No |
| | <outputs> | (outputs tab) | No |
| Globals > Transitions | name | No | No |
| | equation | Yes | No |
| | <user atts> | Yes | No |
| State Properties | name | No | Yes – on STATE: line in comb block case statement |
| | <outputs> | No | No |
| Transition Properties | name | No | Yes – on transitions "if" statement in comb block |
| | <user atts> | No | No |

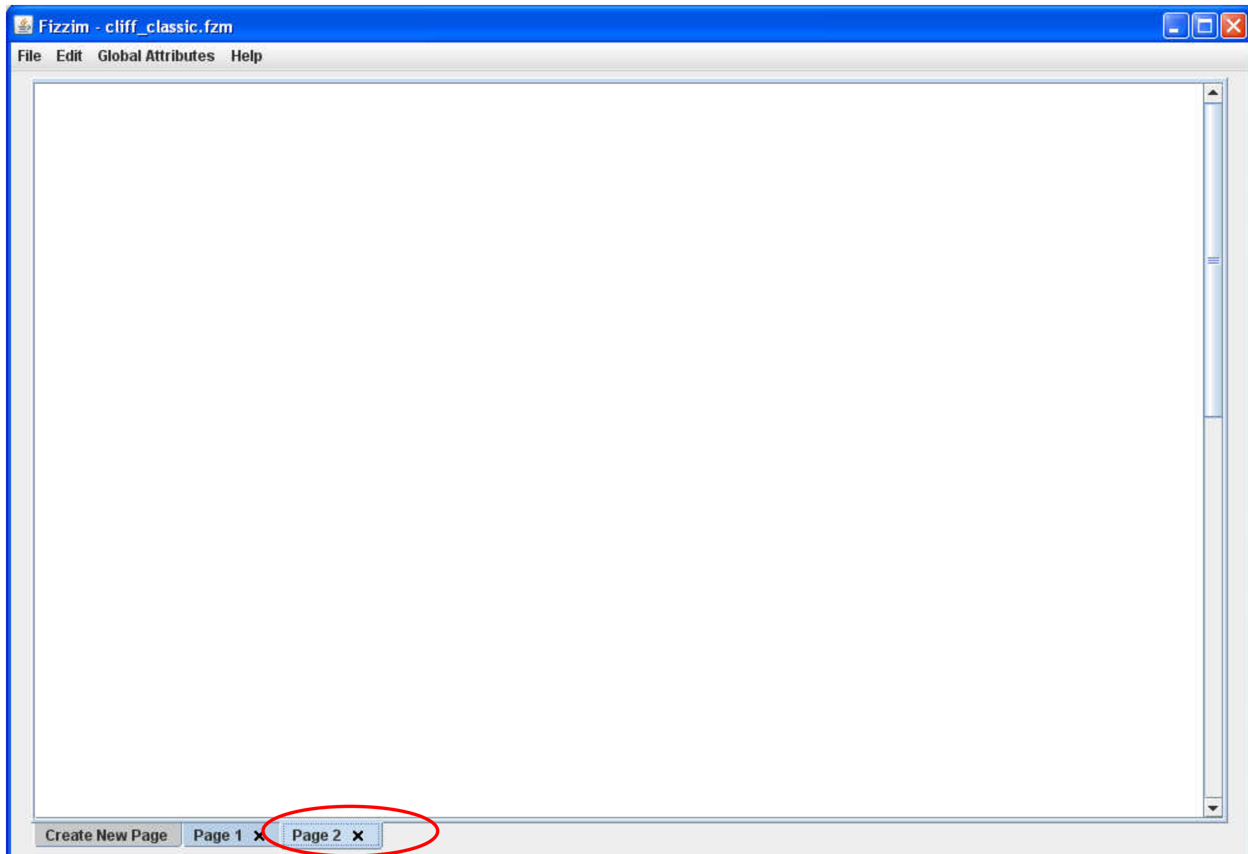
# 18 Using multiple pages

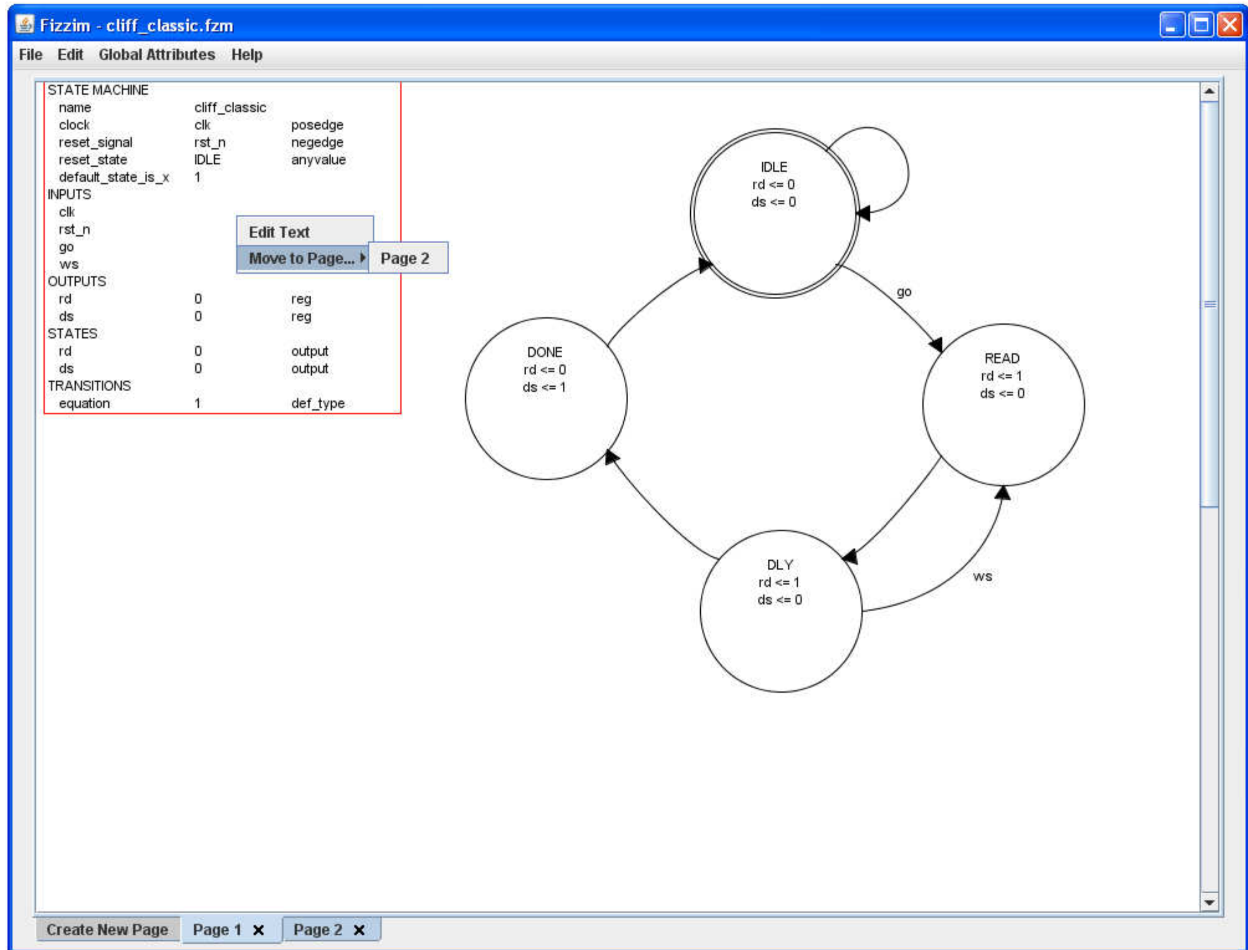
Fizzim will also let you split the FSM across multiple pages.

We'll start with a simple example. Back to Cliff Classic. Let's move the READ state to its own page.

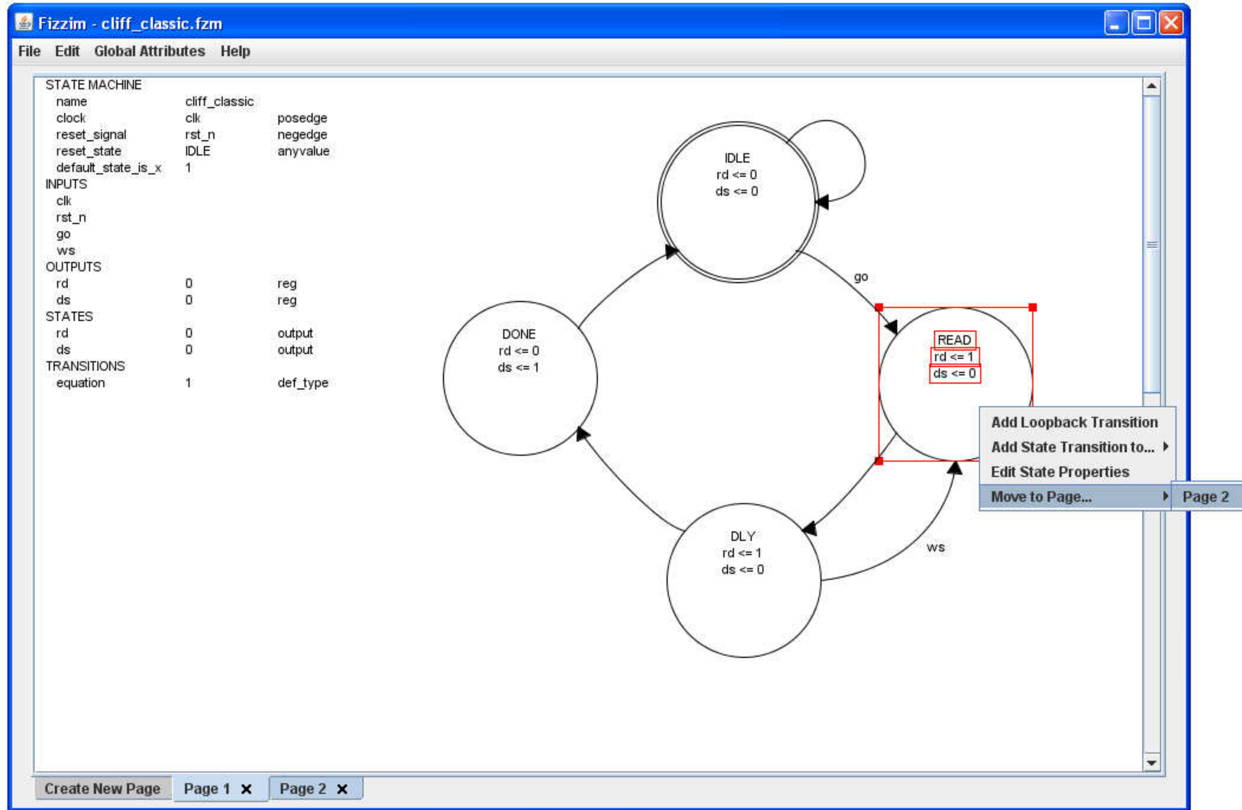
(Example: cliff_classic_multipage.fzm)

Click the "Create New Page" tab at the bottom left. We now have 2 page tabs:

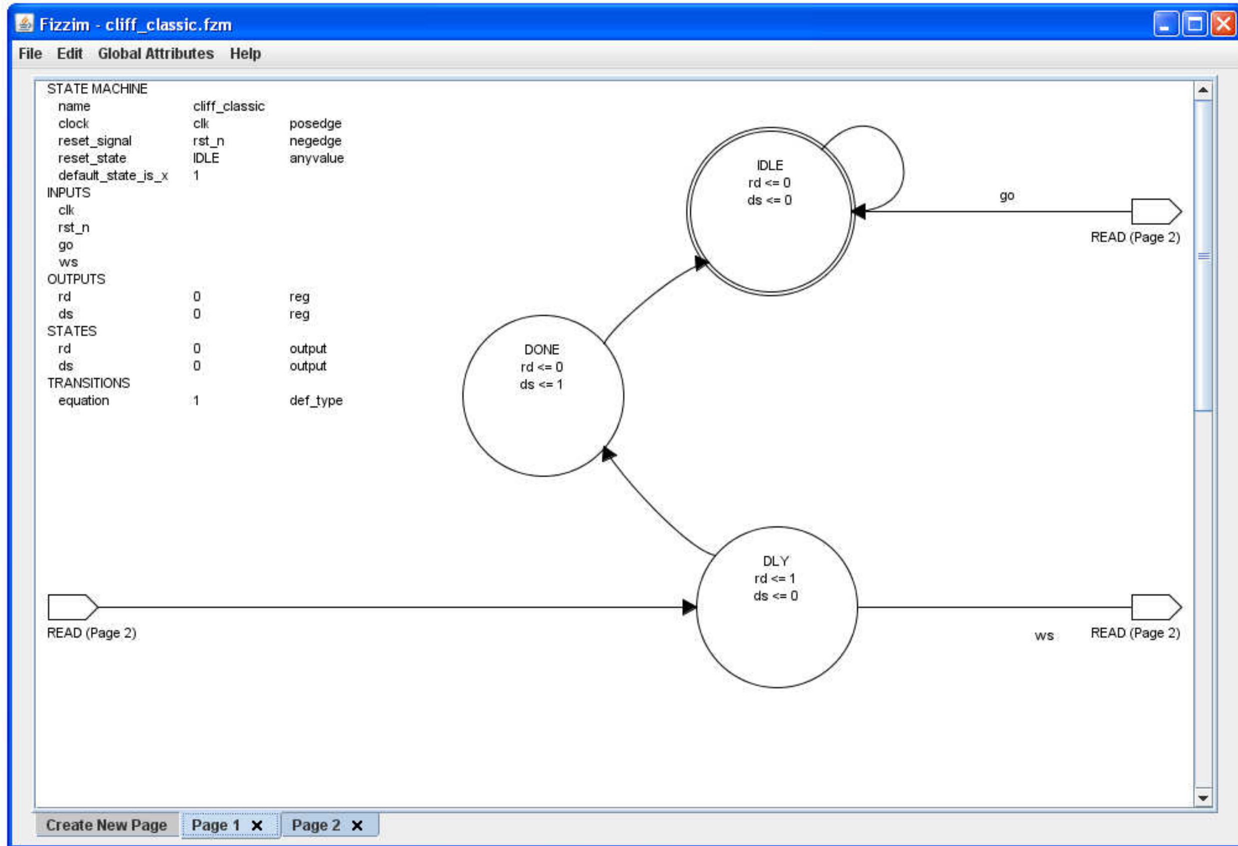





Pop back to Page 1, select the READ state by clicking on it, then right-click to select Move to Page > Page 2:

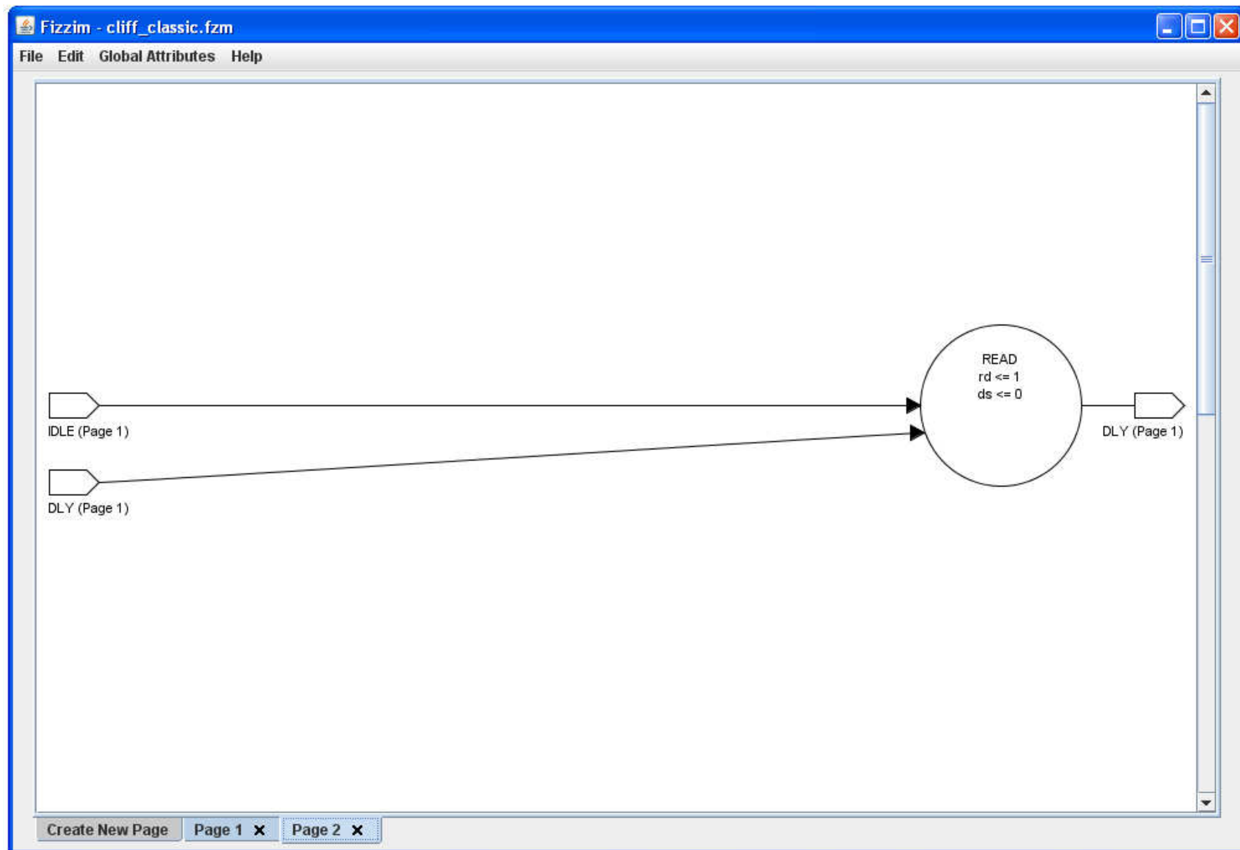



Page 1 now looks like this:




The arcs leading to/from state READ now terminate on page connectors. Input arcs come in from the left, output arcs go out on the right.

Page 2 looks like this:




The usual editing rules apply. You can select the page connectors, state, etc and move them around to clean up the diagram.

One handy use of multiple pages is to move the attributes table to its own page. You can select the attributes table just like a state and move it.

# 19 `include and `define

Many designers prefer to assign constants by using names set by `define:

```
`define OPCODE_READ 4'b0110
```

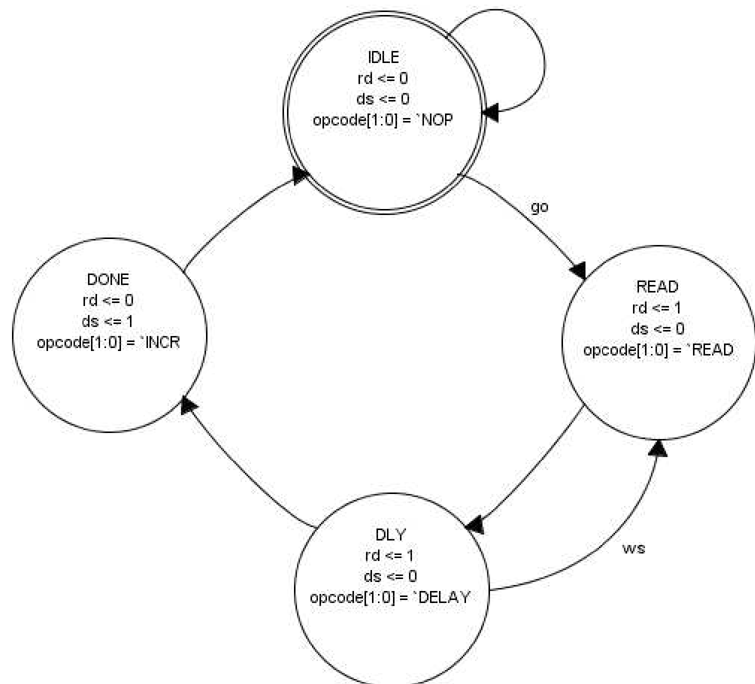
Often these `define statements will be put into a single shared file, which is then read into Verilog using the `include directive.

This is easy enough to do in fizzim, *provided that the values are not being assigned to a output of type "reg"*. This restriction will be explained in a moment. For now, let's look at how you can do it for type comb or regdp.

(Example: cliff_ticdefine.fzm)

Since values of comb and regdb are not parsed by fizzim.pl, there's no problem using a `define value. Here, I've added a multibit output called "opcode[1:0]" and given it values of `NOP, `READ, `DELAY, and `INCR.

| | | |
|---|---|---|
| STATE MACHINE | | |
| name | cliff_ticdefines | |
| clock | clk | posedge |
| reset_signal | rst_n | negedge |
| reset_state | IDLE | anyvalue |
| default_state_is_x | 1 | |
| insert_at_top_of_file | `include "defines.v" \n\n | |
| INPUTS | | |
| clk | | |
| rst_n | | |
| go | | |
| ws | | |
| OUTPUTS | | |
| rd | 0 | reg |
| ds | 0 | reg |
| opcode[1:0] | `NOP | comb |
| STATES | | |
| rd | 0 | output |
| ds | 0 | output |
| opcode[1:0] | `NOP | output |
| TRANSITIONS | | |
| equation | 1 | def_type |




Now I create my "defines.v" file:

```
`define NOP 2'b00
`define READ 2'b01
```

```
`define DELAY 2'b10
`define INCR 2'b11
```

To get it read in, we use the state machine attribute “insert_at_top_of_file” (see “inserting random bits of code a strategic places above), and set it to:

`include "defines.v" \n\n

The result looks like this:

```
`include "defines.v"


module cliff_ticdefines (
  output wire ds,
  output reg [1:0] opcode,
  output wire rd,
  input wire clk,
  input wire go,
  input wire rst_n,
  input wire ws
);



  // 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
    opcode[1:0] = `NOP; // default
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
        end
        else begin
          nextstate = IDLE;
        end
      end
      DLY : begin
        opcode[1:0] = `DELAY;
…
```

So, why not allow type reg? Well, the problem is that fizzim.pl must *know* the values for type reg outputs so that it can encode the state machine properly (well, not for onehot, but the idea is to have a single source able to produce both heros and onehot).

Fine, so parse the Verilog, right? Well, it's not quite that simple. First, you'd have to FIND the include file(s). Does that mean parsing the ".vc" file and reproducing Verilog's directory searchpath algorithm? Hmmm. And what if the `define statements are in among other compiler directives? Now you have to parse most or all of the compiler directives as well.

Worse, the code generation happens in a different step than the simulation or synthesis. What happens if someone edits the defines file after the FSM code is generated? Ouch. To get around this, you'd probably want to add some sort of sim-only code that verifies that the required values didn't change. But that only works for simulation, what about synthesis? Ideally, you'd like to do this with compiler directives, but I don't see how to do that.

So, it might be feasible, but allowing `define values for reg outputs raises a lot of thorny issues, as well as being a fair amount of work. So, for now, it remains on the "maybe, but probably not" list.

# 20 Forcing the state vector

Despite the heros encoding's ability to do all that whizzy stuff, some control-freaks (or speed-freaks!) will *still* insist on forcing particular values onto the state bits.

Fizzim.pl doesn't support this directly (in part because we think it's generally a bad idea), but it's easy enough to fake it. How you fake it depends on whether you want to just force the assignments (making the registered outputs datapath bits), or you want to force the assignments, then *use* the values as your registered outputs.

## 20.1 With registered outputs as datapath bits

To force the state assignment without trying to use the values as registered outputs, here's what you do:

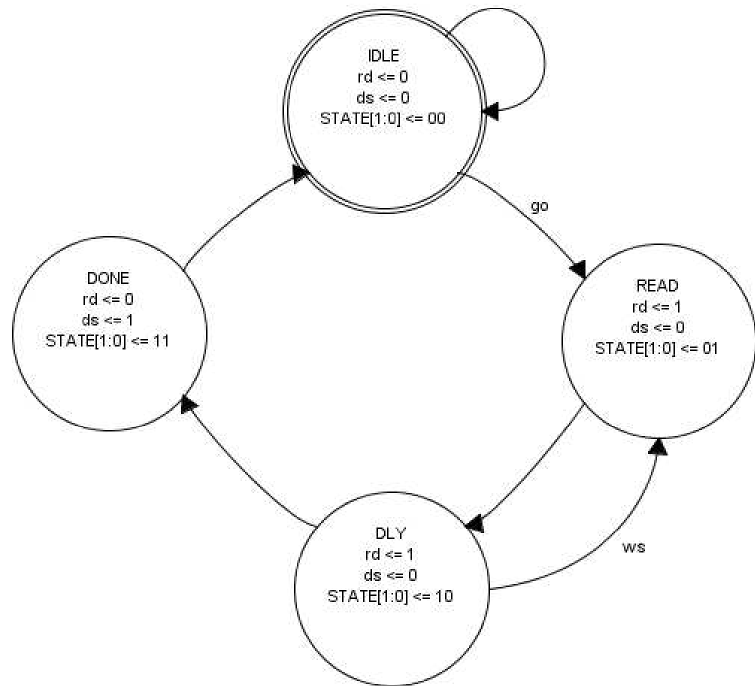
First, create your registered outputs as type regdp.

Now, add an output called, for example, "STATE" with the width of your state vector. Edit each state to assign this to your target value.

Here's what Cliff Classic looks like with this done:

(Example: cliff_forcestate_regdp.fzm)

| 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 | | regdp |
| ds | | regdp |
| STATE[1:0] | | reg |
| STATES | | |
| rd | | output |
| ds | | output |
| STATE[1:0] | | output |
| TRANSITIONS | | |
| equation | 1 | def_type |




If you've encoded the state bits correctly, heros will find your encoding to be just exactly what it needs, and you get output like this:

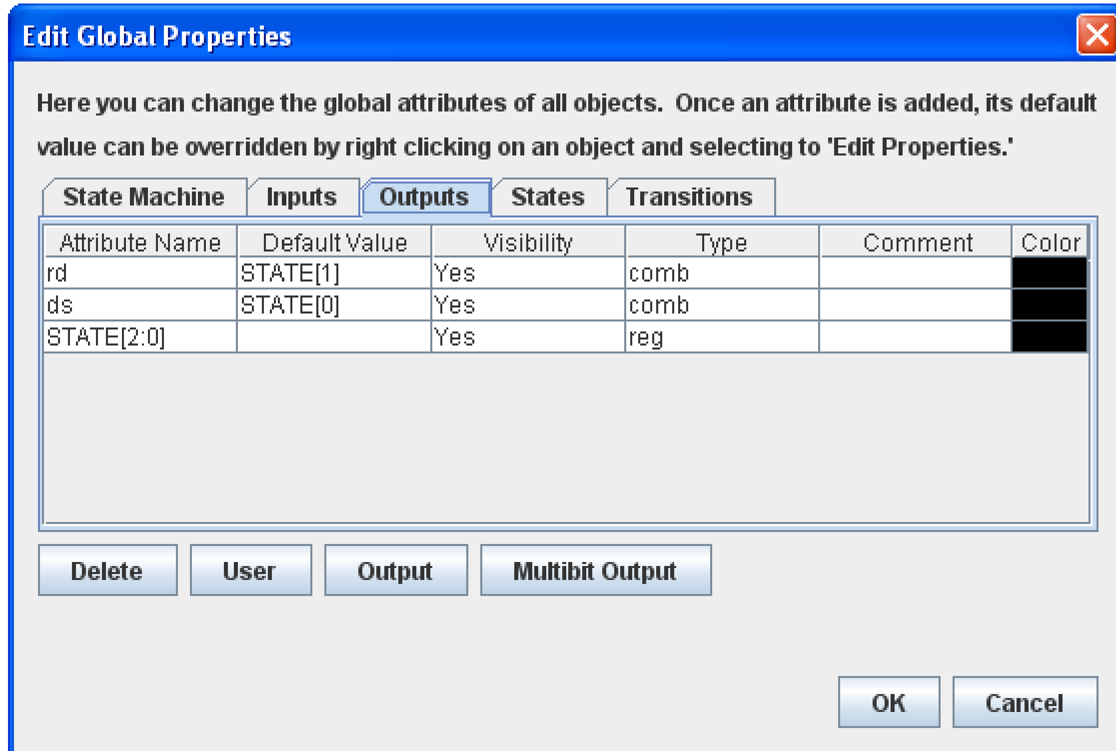
```
  . . .
// state bits
  parameter
  IDLE = 2'b00, // STATE[1:0]=00
  DLY  = 2'b10, // STATE[1:0]=10
  DONE = 2'b11, // STATE[1:0]=11
  READ = 2'b01; // STATE[1:0]=01
. . .
  // Assign reg'd outputs to state bits
  assign STATE[1:0] = state[1:0];
```

## 20.2 With registered outputs assigned to state bits

If you want to assign your registered outputs to bits from your forced state vector, do this:

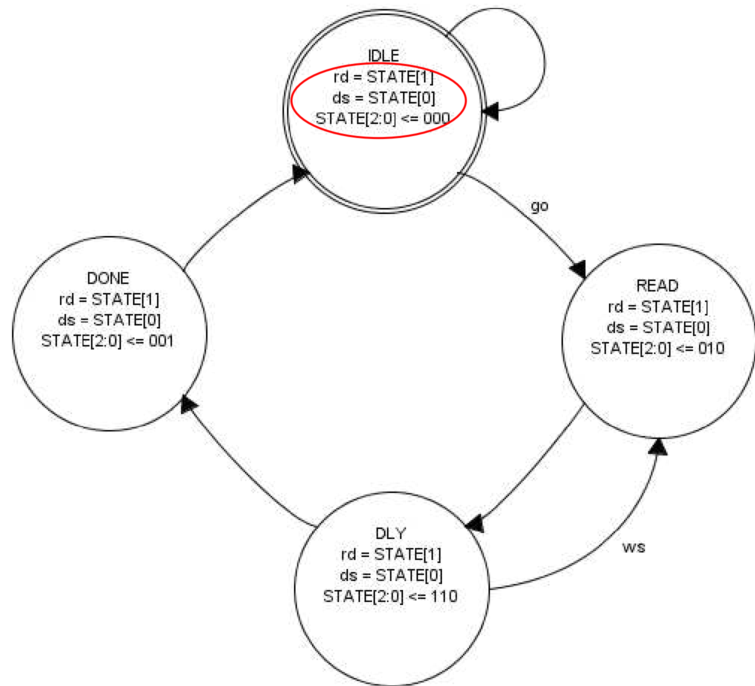
(Example: cliff_forcestate_regout.fzm)

Change their type to "comb" and set their default values to assign each to a state bit (ex: name=ds, Default value=STATE[0]). Add the STATE vector as described above.



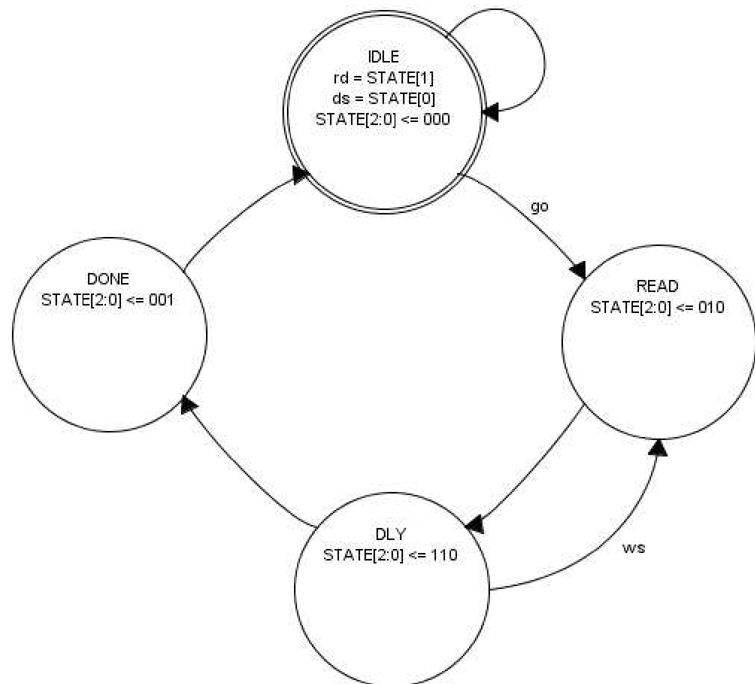

The result would look something like this:

```
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
OUTPUTS
  rd                    STATE[1]      comb
  ds                    STATE[0]      comb
  STATE[2:0]                          reg
STATES
  rd                    STATE[1]      output
  ds                    STATE[0]      output
  STATE[2:0]                          output
TRANSITIONS
  equation              1             def_type
```




To make it look even prettier, you could turn the default visibility on rd and ds to “NO”, then go to one state (IDLE) and turn it on:

```
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                    STATE[1]      comb
  ds                    STATE[0]      comb
  STATE[2:0]                          reg
STATES
  rd                    STATE[1]      output
  ds                    STATE[0]      output
  STATE[2:0]                          output
TRANSITIONS
  equation              1             def_type
```




Or you could turn visibility off complete, and add the mapping as free text. You get the idea.

However you choose to do it, the comb block will now look like this:

```
// comb always block
always @* begin
  nextstate = 3'bx; // default to x because default_state_is_x is set
  ds = STATE[0]; // default
  rd = STATE[1]; // default
  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
```

Now your outputs are forced to the state bit values.

## 21 Suppressing outputs in the module portlist

Starting with version 4.0, there is a new "UserAtt" called "suppress_portlist" that will remove any output from the module portlist. It defaults true for flags. For other outputs, you have to set it manually. Just edit the "UserAtts" column on the "Outputs" page and add "suppress_portlist".

## 22 Splitting lines in free text and equations

Beginning with gui version v110824 and fizzim.pl version 4.01 (package release 4.01), you can split lines in free text and transition equations by embedding newline characters in the text. Just insert the string "\n". This will cause a line break when the text (free text or transition equation) is displayed in the gui. Free text is never part of the fizzim.pl verilog/systemverilog output, but the newline will be stripped from transition equations before the output is generated.

## 23 Unknown states

Most state machines have more possible combinations of the state bits than they have states. Cliff_classic is like this. Due to the fact that two of the states have identical outputs, the heros encoding will use 3 bits for the states – one for rd, one for ds, and one "extra". This means that only 4 of the 8 possible values of the 3-bit state vector correspond to states of the state machine.

There's no inherent problem in this. The coding of the fsm guarantees that it will not be possible to get into any of these "unknown" states. The logic created by synthesis will only ever go to legal states. The only way the fsm can get into one of these states is if the gates or flipflops malfunction. This is distinct from a "bug in the fsm" which would mean the fsm didn't do what the designer intended. Getting into one of these states requires a circuit problem, not a design flaw.

Still, some designers like to design their fsm such that these unknown states go to a known state – so the fsm doesn't "hang" if the circuit malfunctions (but it had better be a one-time malfunction or all bets are off). Fizzim supports this through an attribute called "unknown_states_go_here".

### 23.1 Case 1 – sparse state space and unknowns go to an existing state

Here's a simple example. We'll add the unknown_states_go_here attribute to cliff_classic, and send the unknown states to IDLE.

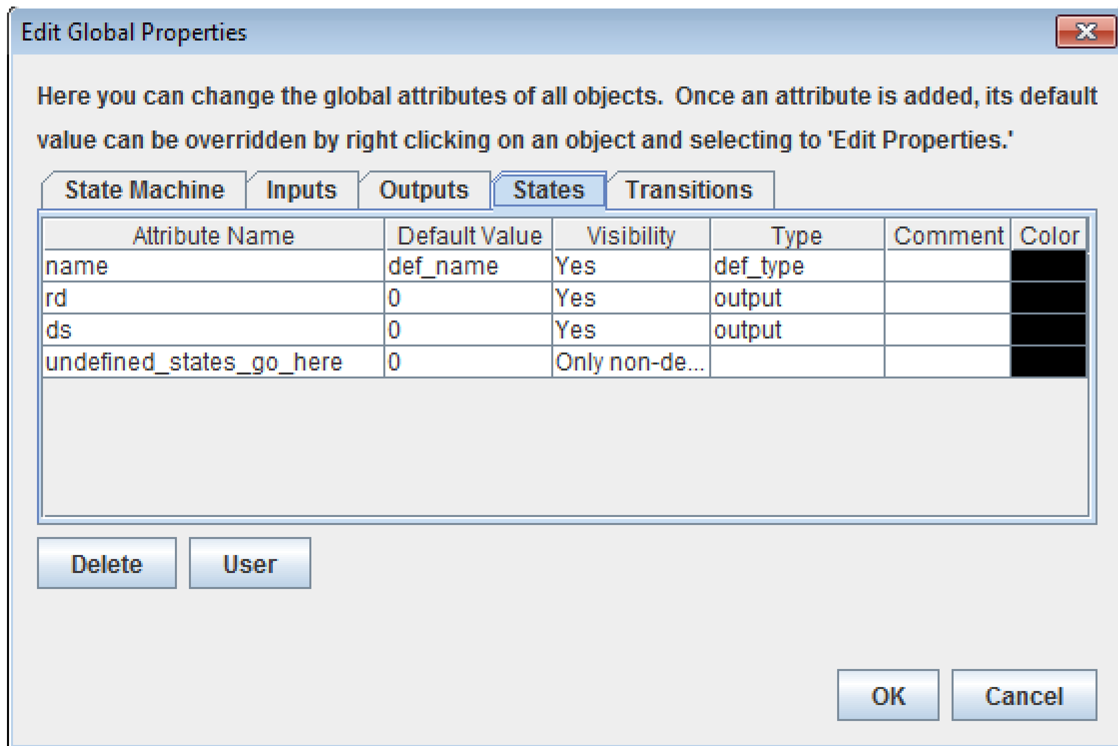
Recall that the nature of the output values in cliff_classic forces fizzim to generate a 3-bit vector for this 4-bit state machine:

```
// 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
```

So, there are lots of undefined state values (100, for example). Using the unknown_states_go_here attribute, we can have fizzim create code that will send the fsm to IDLE if it ever lands in one of these states.

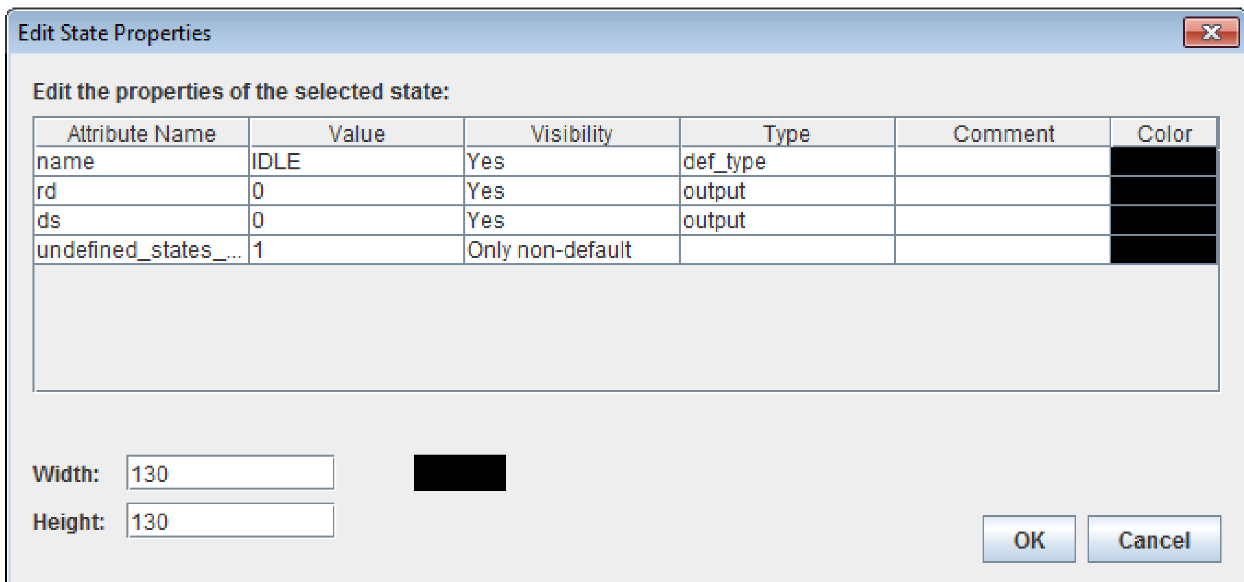
"unknown_states_go_here" is a *state* attribute, and it is not predefined in the gui (it is a "user" attribute). So, as with other such special attributes, we have to create it first, then set it:

To create a new user attribute, use Global Attributes > States

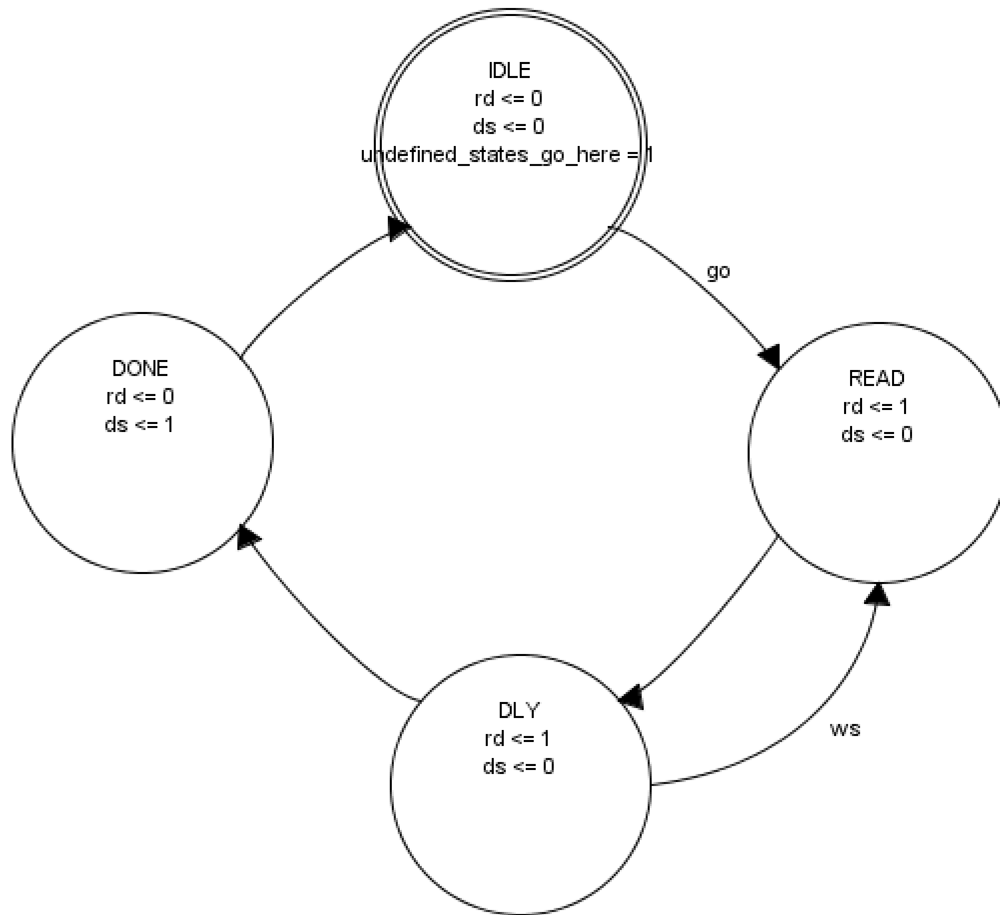



Click "User" and type in the name. Set the default value to 0 and set visibility however you like.

Now, double click on IDLE and set the value of this attribute to 1 in this state:




The state diagram should now look like this:




Now, the case statement will look like this:

```
  // comb always block
  always @* begin
    nextstate = state; // default to hold value because implied_loopback is
set
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
        end
      end
      DLY : begin
        if (ws) begin
          nextstate = READ;
        end
        else begin
          nextstate = DONE;
        end
      end
      DONE: begin
        begin
          nextstate = IDLE;
```

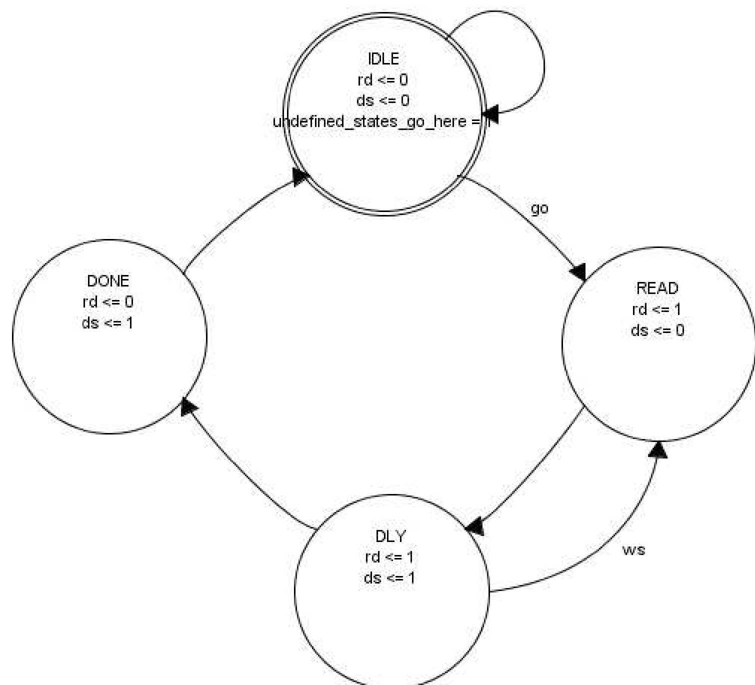
```
      end
    end
    READ: begin
      begin
        nextstate = DLY;
      end
    end
    default : begin
      nextstate = IDLE; // Added because undefined_states_go_here is set
    end
  endcase
end
```

Note the addition of the "default : " statement. Any states that don't match the known ones fall through to this statement and the next transition will be to IDLE.

## 23.2 Case 2 – full state space and unknowns go to an existing state

OK, so what happens if we change the encoding so that this 4-bit state machine actually fits into a 2 bit state vector?

| STATE MACHINE | | |
|---|---|---|
| name | unknown_4state | |
| 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 |
| undefined_states_go_here | 0 | |
| TRANSITIONS | | |
| equation | 1 | def_type |




Notice that the state vector is now only 2 bits:

```
// state bits
parameter
```

```
  IDLE = 2'b00, // rd=0 ds=0
  DLY  = 2'b11, // rd=1 ds=1
  DONE = 2'b01, // rd=0 ds=1
  READ = 2'b10; // rd=1 ds=0
```

Well, you get the "default :" statement anyway:

```
  reg [1:0] state;
  reg [1:0] nextstate;

  // comb always block
  always @* begin
    nextstate = 2'bxx; // 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
      default : begin
        nextstate = IDLE; // Added because undefined_states_go_here is set
      end
    endcase
  end
```

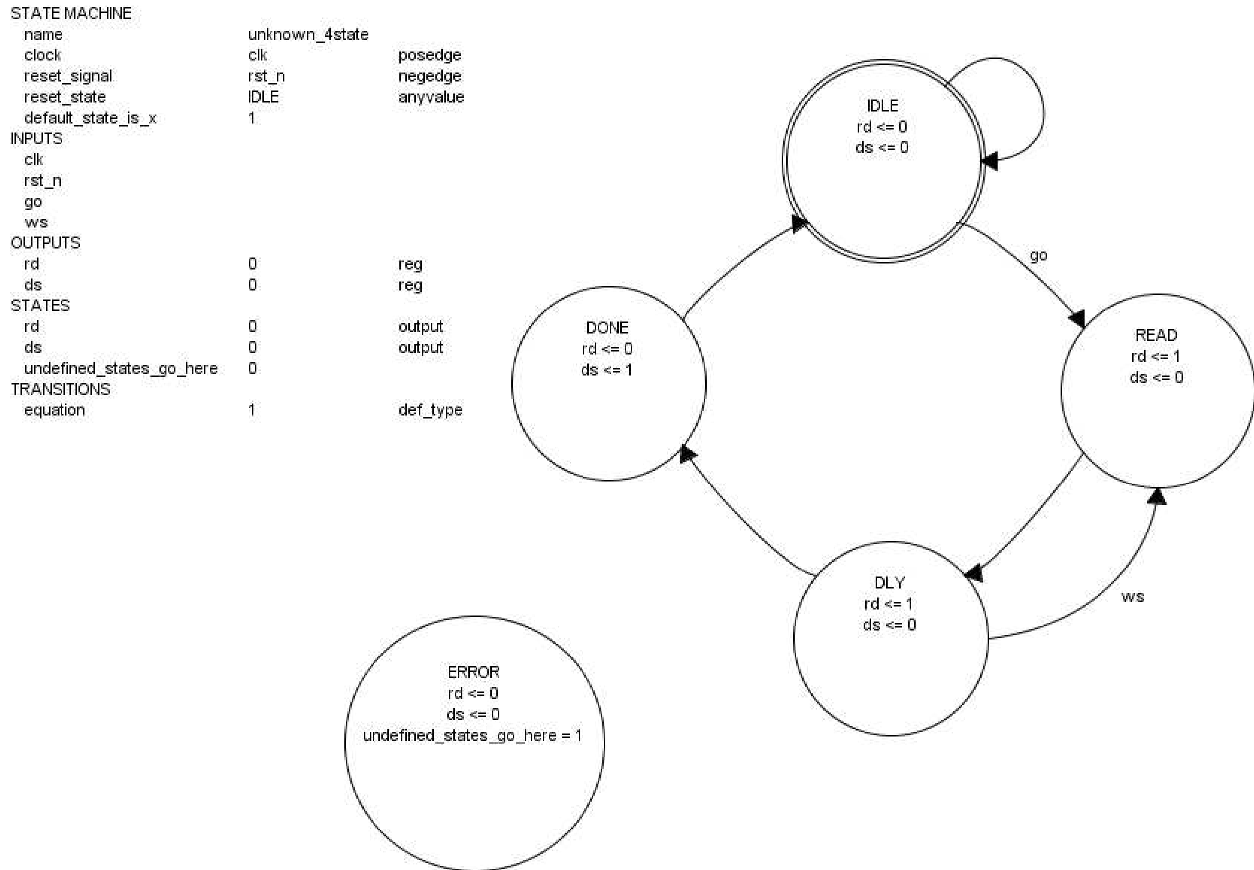
This is harmless functionally, but it might be useful for linting tools that insist on seeing the "default :".

## 23.3 Case 3 – sparse state space and unknowns go to a new state

OK, back to the original cliff_classic. Suppose, rather than have unknown states go to IDLE, we want them to go to a new ERROR state (note: I don't recommend actually calling the state

"ERROR", since I always avoid using the string in verilog names as it makes grepping more difficult)?

Well, just add the new state as usual, create the undefined_states_go_here attribute as above, and set it in the new state:




The resulting code looks like this:

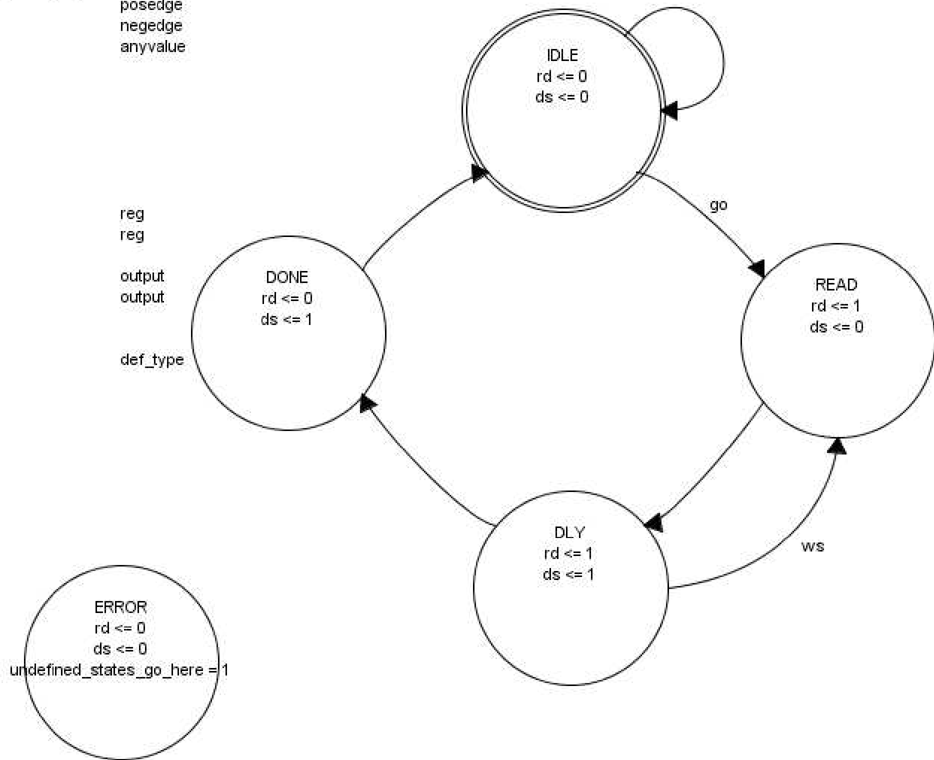
```
// state bits
parameter
IDLE  = 3'b000, // extra=0 rd=0 ds=0
DLY   = 3'b010, // extra=0 rd=1 ds=0
DONE  = 3'b001, // extra=0 rd=0 ds=1
ERROR = 3'b100, // extra=1 rd=0 ds=0
READ  = 3'b110; // extra=1 rd=1 ds=0

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

// comb always block
always @* begin
  nextstate = 3'bxxx; // 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
      ERROR: begin
      end
      READ  : begin
        begin
          nextstate = DLY;
        end
      end
      default : begin
        nextstate = ERROR; // Added because undefined_states_go_here is set
      end
    endcase
  end
```

## 23.4 Case 4 – full state space and unknowns go to a new state

Ahah! What if the space state was already full and we added the ERROR state? Are we going to end up with a 3-bit vector instead of a 2-bit vector *just* to have a now-useless ERROR state?

NO! Fizzim is smart enough to detect this. If fizzim sees a state with unknown_states_go_here set, and there are no transitions into this state, and it is not the reset staet, it will first try the state encoding without this state. If this encoding ends up full, the special state is suppressed.




```
  // state bits
  parameter
  IDLE = 2'b00, // rd=0 ds=0
  DLY  = 2'b11, // rd=1 ds=1
  DONE = 2'b01, // rd=0 ds=1
  READ = 2'b10; // rd=1 ds=0
  // Note: State ERROR (with undefined_states_go_here attribute) dropped
because it had no transitions into it and the state map was full without it.

  reg [1:0] state;
  reg [1:0] nextstate;

  // comb always block
  always @* begin
    nextstate = state; // default to hold value because implied_loopback 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
```

Notice the "Note: " showing what fizzim has done.

What if this special state has outputs? Well, if the outputs are comb or regd, nothing changes. The default of the output will always be asserted. For a comb output, you get something like this:

For a regdp output, you'll get this:

```
  // datapath sequential always block
  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      err <= 0;
    end
    else begin
      err <= 0; // default
        // Warning D9: Did not find any non-default values for any datapath
outputs - suppressing case statement
    end
  end
```

This ends up as a wire tied to ground...

If, however, you declare this output as type "reg", you won't get the special state suppression functionality. Type "reg" means the bit is supposed to be part of the state vector. Figuring out which outputs could be suppressed (and tying them off) is just too hard - use a regd or comb.

Note that this suppression behavior an be overidden using the "-force_keep_undefined_goto_state" switch.

# 24 Controlling and suppressing warning messages

Fizzim.pl has a couple of command-line switches that allow you to control what warning messages are generated, and where they go.

Currently, warning messages are placed in the following groups:

1. "R" messages – Reset-related warnings.
2. "I" messages – Implied loopback warnings.
3. "P" messages – Priority-related warnings.
4. "C" messages – Combinational output-related warnings.

Each individual message also has a number. Note that the numbers within the group are not necessarily continguous, since each number is itself unique across all groups.

So, for example, warning message "R1" is "No reset specified" and warning message "R5" is "No reset value for datapath output <output> set in reset state <state> - Assigning a reset value of <value> based on default".

You can use the –nowarn switch to suppress these warnings. Using the full group+number will suppress just that message:

-nowarn R1

Using *just the group* will suppress all messages in the group, so:

-nowarn R

suppresses all reset messages.

You can also control where the messages are sent using the –warnout switch. This switch has 3 possible values:

- stdout – place the messages in the Verilog as comments.
- stderr – just send the messages to unix stderr.
- both – send the messages to both places.

The default value for warnout is "both".

## 25 Printing and exporting the state diagram

One of the nice things about using a gui-based FSM design tool is the ability to use the state diagram in your documentation. In addition to printing the state diagram, you can export it in three ways:

1. As a .png file
2. As a .jpg file
3. Directly to the clipboard

Having the attributes table on the diagram allows you to put ALL the information into your documentation quickly and easily.

All the state diagrams in this paper were inserted using the export to clipboard feature.

Note that currently fizzim only prints/exports one page at a time.

## 26 Specifying the fizzim.pl options

There is a special state machine attribute called “be_cmd” that is used to specify the backend command to run. Some day, this will be used to run the backend from within the gui. That’s still on the todo list (because it has some platform dependencies), but the be_cmd attribute is fully supported in fizzim.pl. The attribute is parsed to obtain the *options* and those options are treated exactly as if they had been specified on the command line – ahead of the actual command line options. Since they come first, they can be overridden by the options specified on the command line, giving priority to these.

# 27 Requiring a minimum revision of fizzim.pl

Beginning with revision 3.6, fizzim.pl has a command line option (more often used in be_cmd) “-minversion” which will cause it to error out if it’s version is less than the version specified:

```
$ cat cliff.fzm | fizzim.pl -minversion 6.1

Error: Version 3.55 is less than required minversion 6.1 - exiting
```

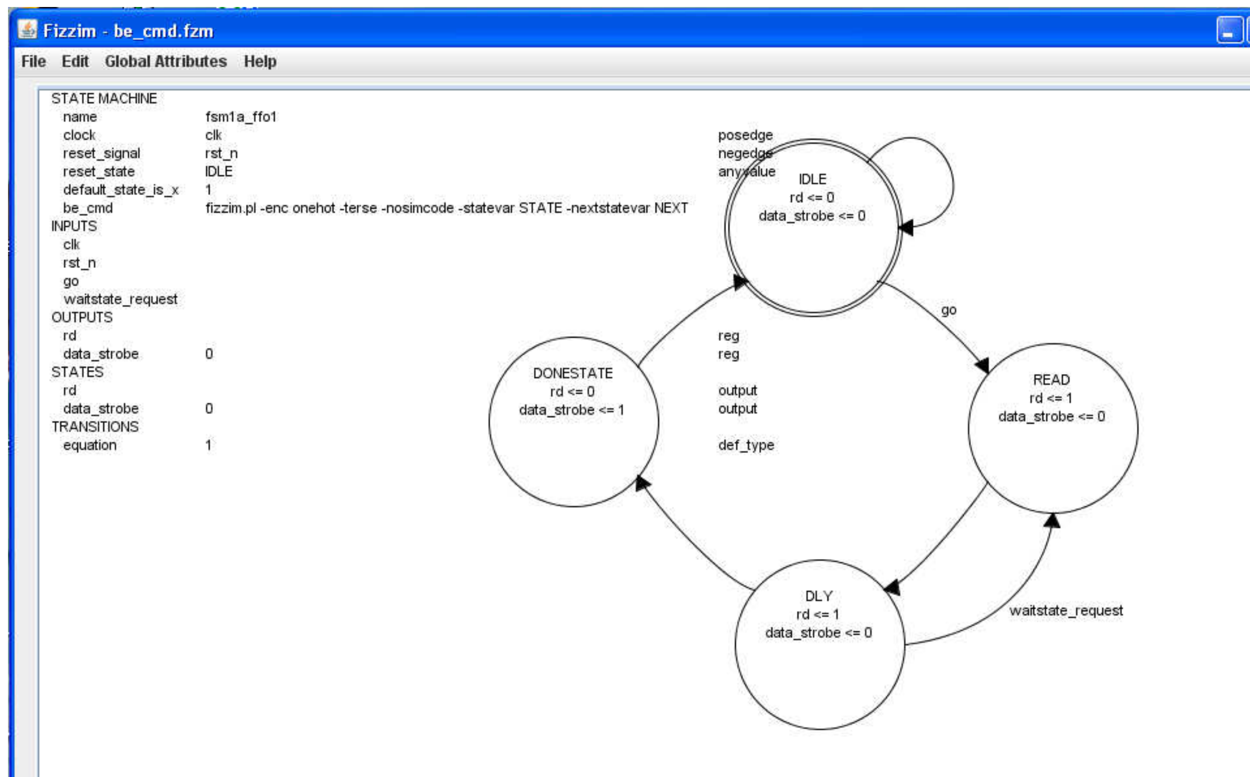
This is to cover the situation where your fsm requires a specific feature or bug fix.

# 28 Group select and move

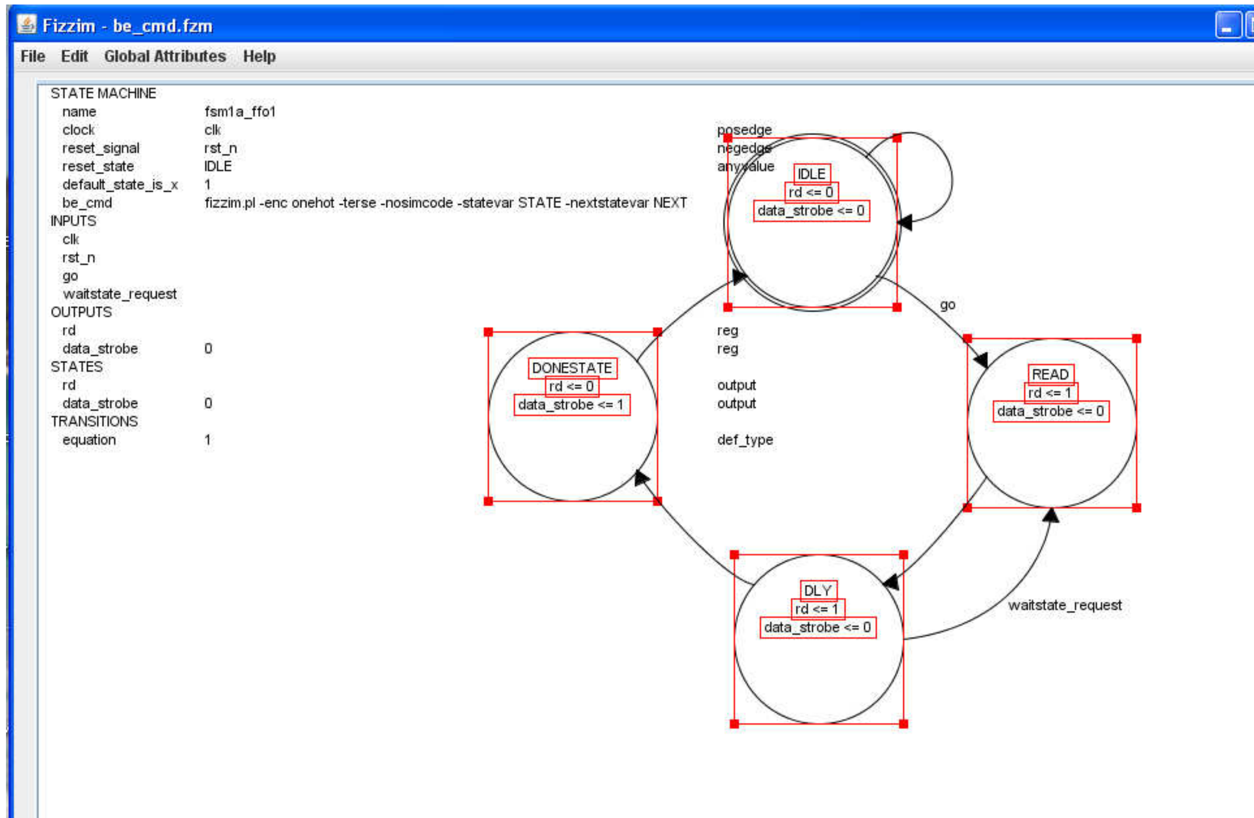
The fizzim gui also supports multiple item select and move.  In this case, the “items” are states and the attribute table – transitions are only moved by moving the attached state(s).

Any modifications made to attached transitions are generally retained if both of the attached states are moved (in fact, whenever possible, they are retained when a single attached state is moved).

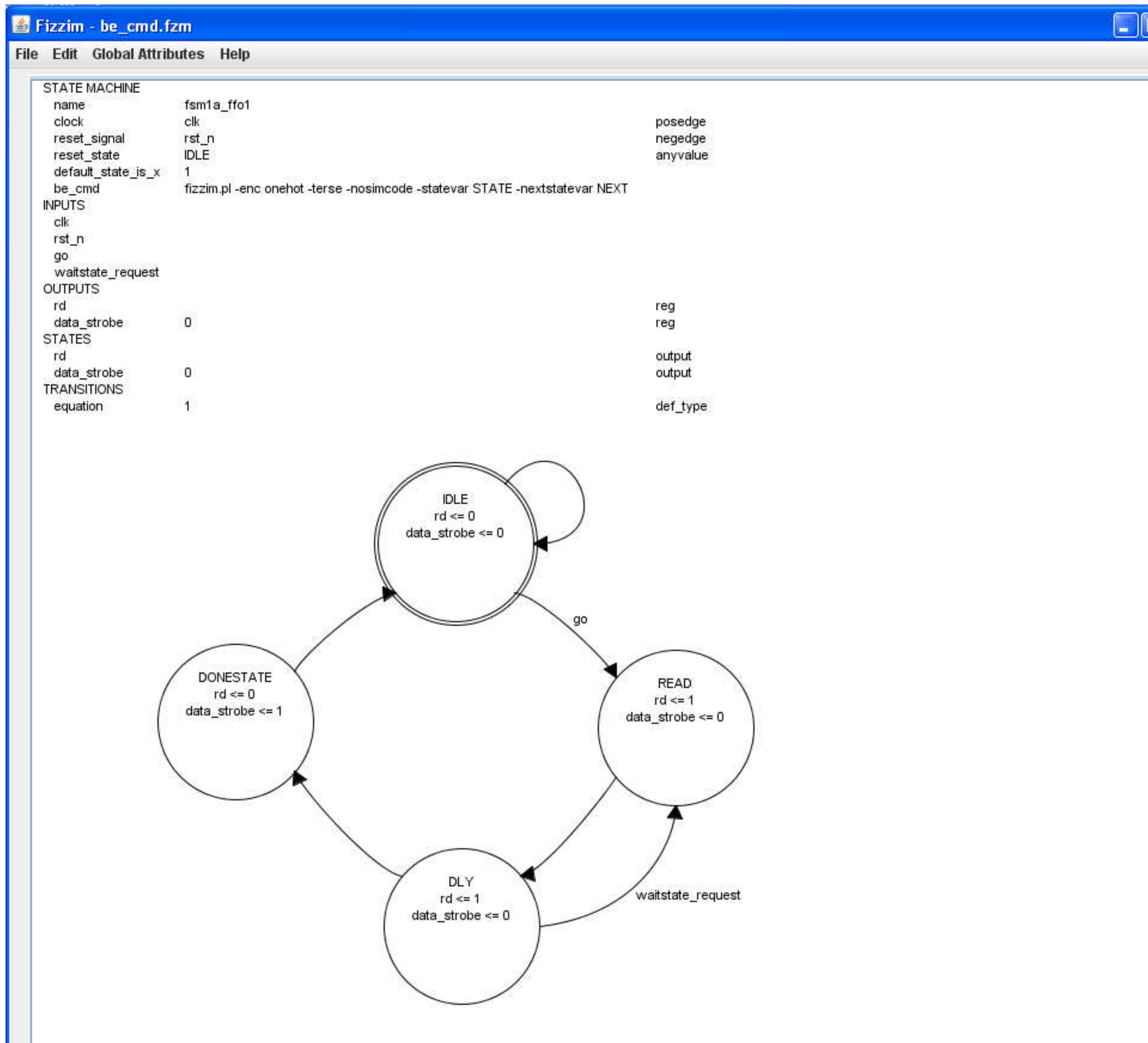
A common example is having to move the while fsm because the attribute table grew too big. Suppose I added a be_cmd and ended up with this:




I can move the whole fsm by selecting all the states either by selecting each state individually (click, ctl-click, ctl-click, etc), or by drawing a box around the whole fsm:




And then just drag it to the new location.




Notice that my arcs didn't change.

# 29 –terse (-sunburst) option

Cliff Cummings of Sunburst Design is one of the industry's top Verilog experts. He participates in standards activities, teaches Verilog and SystemVerilog classes, and presents frequently on all things Verilog.

Cliff is a firm believer in "less is more". He prefers a coding style that eliminates any and all unnecessary syntax (like begin/end blocks, etc). fizzim.pl has an option (-terse or –sunburst) that will produce this sort of output.

Here's an example:

(Example: cliff_terse_example.fzm)

Standard output looks like this:

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



  // 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
    pre_rd = 0; // default
    case (state)
      IDLE: begin
        if (go) begin
          nextstate = READ;
          pre_rd = 1;
        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
    rd <= 0; // default
    case (nextstate)
      DLY : begin
        rd <= 1;
      end
      READ: begin
        rd <= 1;
      end
    endcase
  end
end

// 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

endmodule
```

The sunburst version looks like this:

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

  // 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
    pre_rd = 0; // default
    case (state)
      IDLE: begin
        if     (go) begin
                              nextstate = READ;
                              pre_rd = 1;
        end
        else                  nextstate = IDLE;
      end
      DLY  : if (ws)          nextstate = READ;
             else             nextstate = DONE;
      DONE:                   nextstate = IDLE;
      READ:                   nextstate = DLY;
    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)     rd <= 0;
    else begin
      rd <= 0; // default
      case (nextstate)
        DLY : rd <= 1;
        READ: rd <= 1;
      endcase
    end
  end

  // 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

endmodule
```

## 30 SystemVerilog output

Beginning in revision 3.0, fizzim.pl can produce output in SystemVerilog format.

SystemVerilog is invoked by specifying "-language SystemVerilog" on the command line (or in the be_cmd attribute string – see the section on be_cmd).

The code is structured to follow coding guidelines from Cliff Cummings (as taught in his SystemVerilog class). The primary changes are in the following areas:

1. Use of logic data type instead of wire and reg
2. Use of enumerated types instead of parameters for state names. In most waveform viewers, the eliminates the need for special code to be able to see the state names. Because of this, the "-simcode" option defaults to off when the language is SystemVerilog (the default is on normally).
3. Use of always_comb, always_ff instead of always_at *
4. Use unique case instead of "//synopsys full_case parallel_case" in onehot encoding (unless the attribute "onehot_pragma" is set).

The heros output for cliff_classic looks like this in SystemVerilog:

```
module cliff_classic (
  output logic ds,
  output logic rd,
  input logic clk,
  input logic go,
  input logic rst_n,
  input logic ws
);


  // state bits
  enum logic [2:0] {
    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
    XXX = 'x
  } state, nextstate;


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

endmodule
```

The heros output for cliff_classic looks like this in SystemVerilog:

```
module cliff_classic (
  output logic ds,
  output logic rd,
  input logic clk,
  input logic go,
  input logic rst_n,
  input logic ws
);


  // state bits
  enum {
    IDLE_BIT,
    DLY_BIT,
    DONE_BIT,
    READ_BIT
  } index;

  enum logic [3:0] {
    IDLE = 4'b1<<IDLE_BIT,
    DLY  = 4'b1<<DLY_BIT,
```

```
    DONE = 4'b1<<DONE_BIT,
    READ = 4'b1<<READ_BIT,
    XXX = 'x
  } state, nextstate;


  // comb always block
  always_comb begin
    nextstate = XXX; // default to x because default_state_is_x is set
    unique case (1'b1)
      state[IDLE_BIT]: begin
        if (go) begin
          nextstate = READ;
        end
        else begin
          nextstate = IDLE;
        end
      end
      state[DLY_BIT]: begin
        if (ws) begin
          nextstate = READ;
        end
        else begin
          nextstate = DONE;
        end
      end
      state[DONE_BIT]: begin
        begin
          nextstate = IDLE;
        end
      end
      state[READ_BIT]: begin
        begin
          nextstate = DLY;
        end
      end
    endcase
  end

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

  // datapath sequential always block
  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      ds <= 0;
      rd <= 0;
    end
    else begin
      ds <= 0; // default
      rd <= 0; // default
      unique case (1'b1)
        nextstate[DLY_BIT]: begin
          rd <= 1;
```

```
        end
        nextstate[DONE_BIT]: begin
          ds <= 1;
        end
        nextstate[READ_BIT]: begin
          rd <= 1;
        end
      endcase
    end
  end

endmodule
```

## 31 Future directions / wishlist

- 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.

# 32 Acknowledgements

The authors would like to acknowledge the following individuals for their assistance in mapping our the feature set and reviewing the output:

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

Cliff Cummings – Sunburst Design

# 33 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)