6.111 Lecture # 8

Topics for Today: (as time permits)

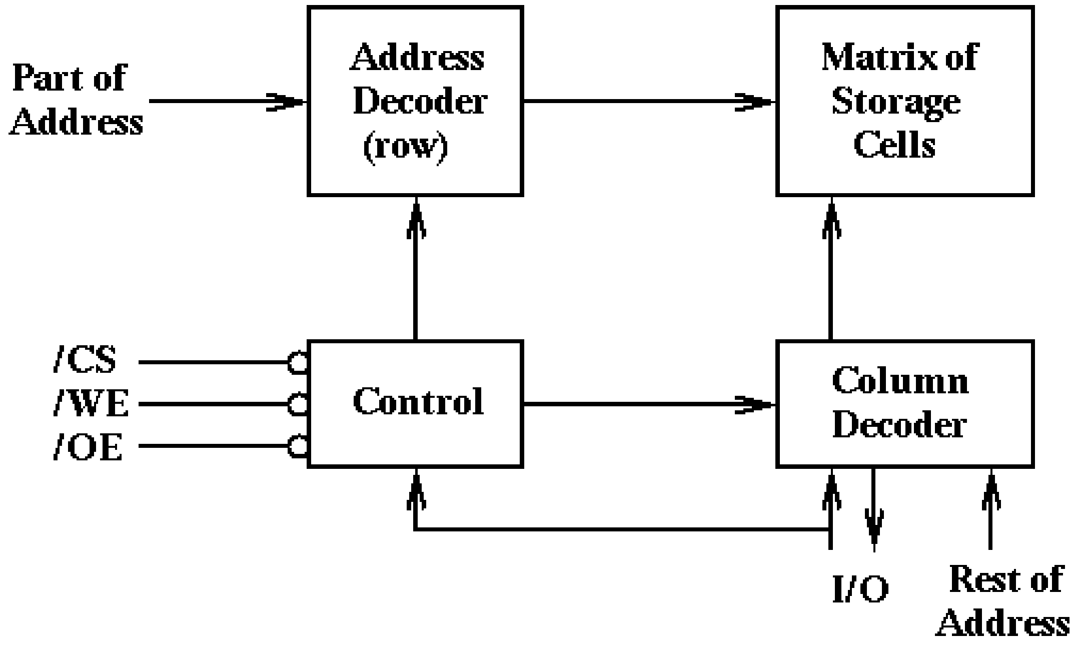
1. **Memories**
2. **Assembling 'packages' for designs**
3. **Discussion of design procedure**
4. **Development of a design example using a finite state machine**

Preview:

No class Monday (Student Holiday)
Wednesday: quiz rev iew and discussion of Phase II
Friday: Quiz 1

Memories are usually organized as 2- dimensional arrays of cells

Address is split into two parts e.g. 4k = 4096 addresses = 2^{14}
might have 7 bits of address for each of row and column



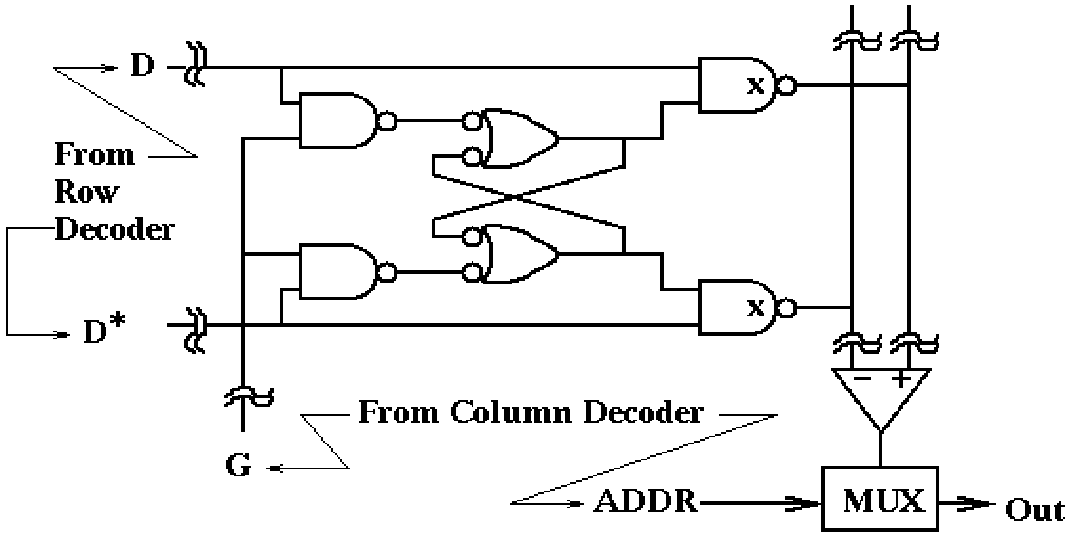
Conceptual Memory Cell: This is what goes at each intersection of the row and column lines (i.e. there are a lot of these!)

Note how this is like a 'D-Latch' The lines D and D* are from the row decoder and control.

D = D* = '1' => 'Read': cell contents go onto sense lines

D = D* = '0' => This row is not addressed. Output is high (collectors open and some other row drives sense lines)

D = /D* and G = '1': 'Write': D is latched onto cell when G goes low



Output of this cell is 'open collector' and so "pulls down" the sense lines that go to the column decoding MUX

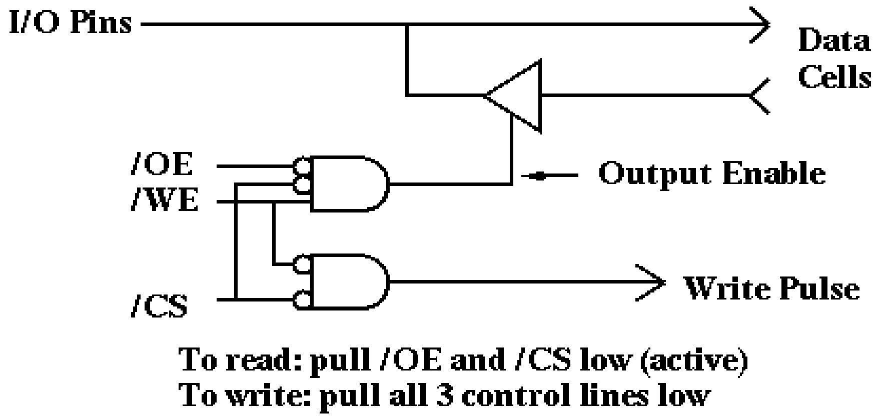
Control Lines:

Often Active Low

OE is 'Output Enable'

WE is 'Write Enable'

CS is 'Chip Select'

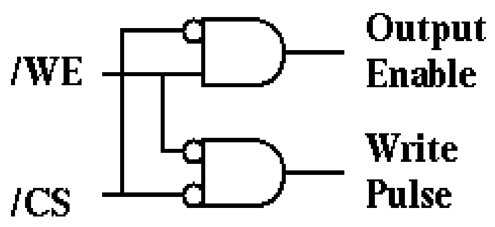

I/O Pins
Data
Cells
/OE
/WE
/CS
Output Enable
Write Pulse
To read: pull /OE and /CS low (active)
To write: pull all 3 control lines low

If: /WE is LOW,
/CS is LOW,
/OE is HIGH,
Data pins are input
Input data is written to chip

If /WE is HIGH
/CS is LOW
/OE is LOW
Data pins are output
Date is read from memory

Some have simpler control structure

Some parts have 2
Control lines:



The /OE line is in many cases redundant (but having the extra line to use can be convenient)

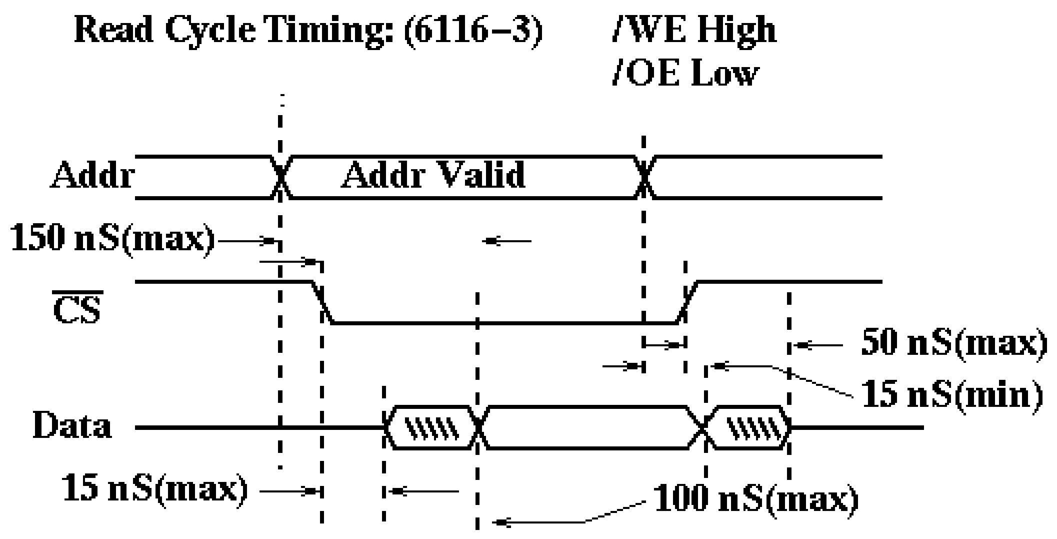
In these parts,

Read => /CS = LOW and /WE = HIGH

Write => /CS = LOW and /WE = LOW

Read Cycle Timing

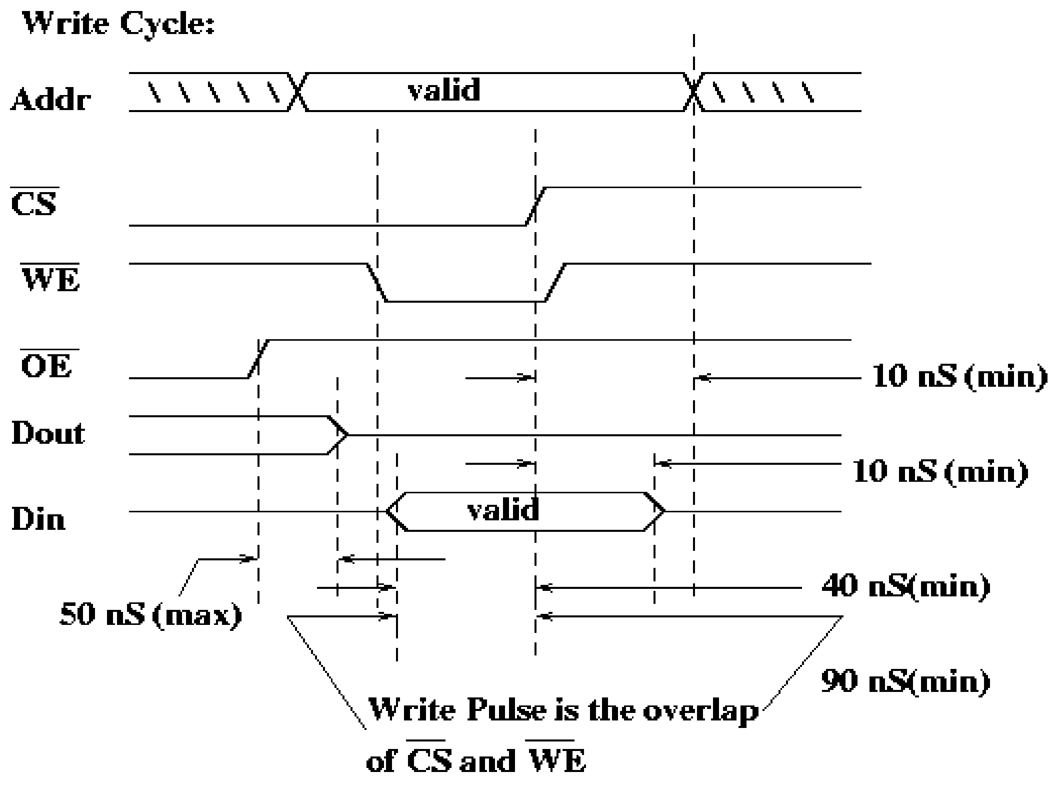
Address takes a little while to propagate into the right places
It takes a bit less time for the part to 'grab' the output pins
(invalid data may be on them initially)
And note it takes a little while after /CS goes high for the part to let go of the output pins
And if Address goes invalid before /CS goes high, there may be invalid data on the output pins



Write cycle timing is a little more complex

==> It is most important that Address and Data must BOTH be valid during the write pulse <==

==> It is also important that <u>Address</u> must be fixed and valid during the <u>Whole</u> of the write pulse <==



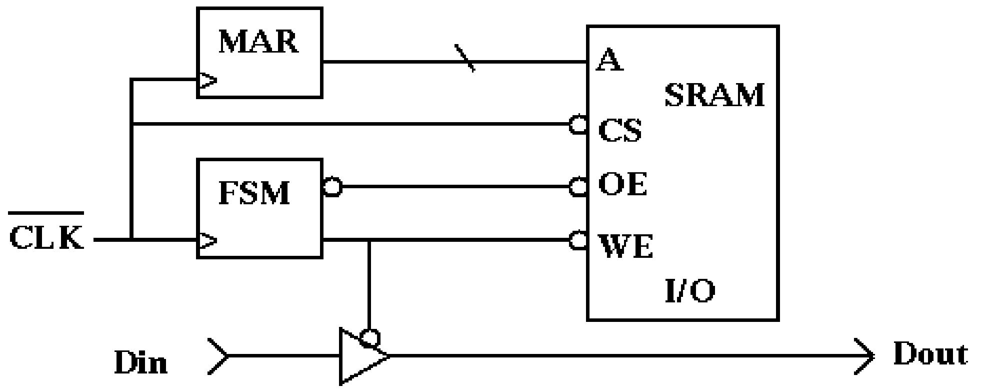
==> Data must be valid for a period at the end of the write pulse <==

Tristated or unstable address lines can wind up writing garbage to a large number of memory locations!

Here is a general purpose suggestion for handling memory in a FSM controlled system.

You can do it more simply in Lab 2

Driving /CS with /CLK ensures 'clean' write pulses and reduces the possibility of bus contention. Both WRITE and READ operations are enabled only on the second half of the clock cycle (before the positive going edge)

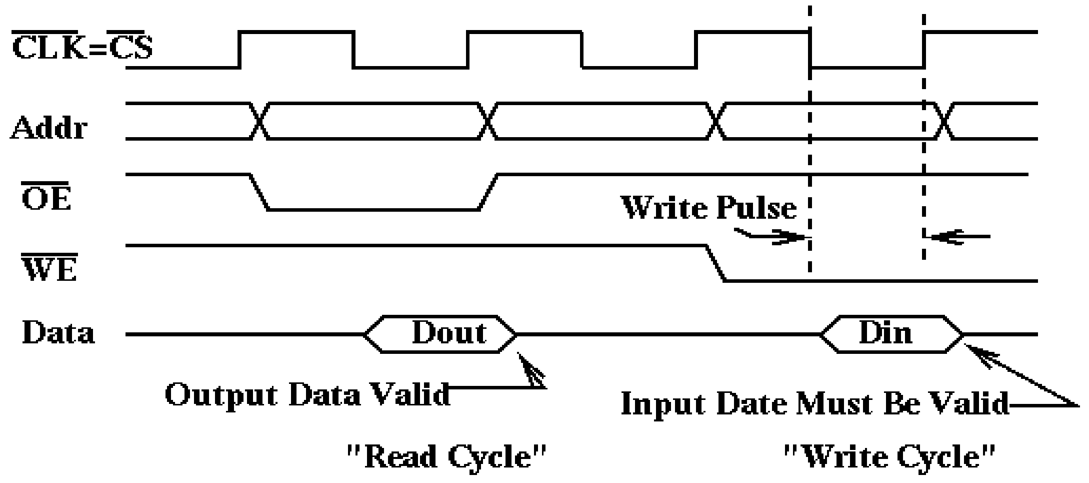


Here is a suggested setup for controlling memory
FSM is the control for this thing
Note: if the data bus is driven ONLY by D and SRAM,
you can just ground $\overline{\text{OE}}$ and not bother with it in the FSM!

This timing diagram illustrates how the scheme on the previous slide might work.

It assumes Addr changes after the positive going clock edge and so is stable when the clock is low.

Also, if the control lines are driven by a FSM, they will change after the positive going clock edge too.



Packages contain bits (or larger pieces of code) that you may re-use.
They are introduced by statements such as:

```
    use work.gridpkg.all;
```

To set up a package you first write and test the pieces

(perhaps using smaller PLD's than you plan for the actual project to save computation time) and then:

-Assemble all of the generic and port declarations from the entities into a file called (for example) gridpkg.vhd

-Then put all of the files into a single file: do something like:

```
 cat gridpkg.vhd synchronizer.vhd reg.vhd ctr.vhd fsmt.vhd > all.vhd
```

-Set the device to the target device, C374I.

-Compile this file (without it being the Top design).

Now you will have something you can use as a library package if you use the parts as specified in the entity declarations.

Hierarchical Design

Start with a one-block block diagram.

Expand to major blocks.

Repeat expansion until blocks are simple.

Implement these simple blocks and test.
(Code them in VHDL and simulate.)

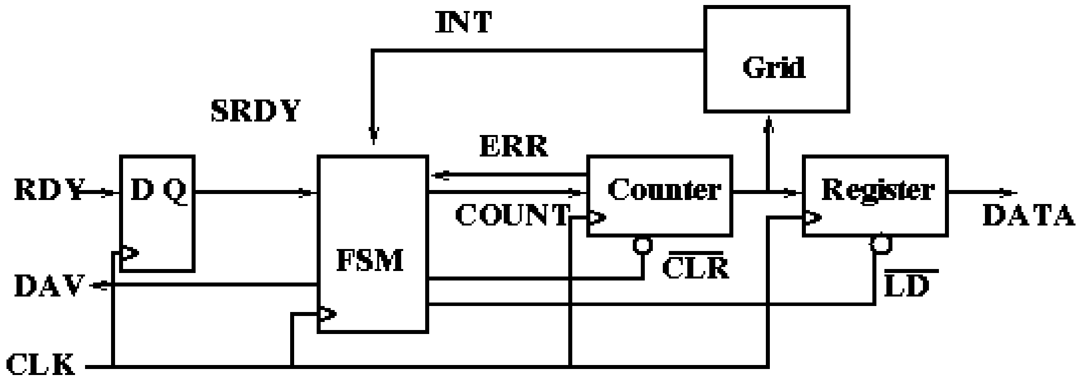
Wire the blocks together.
(Use structural instantiation in VHDL.)

Test the design.

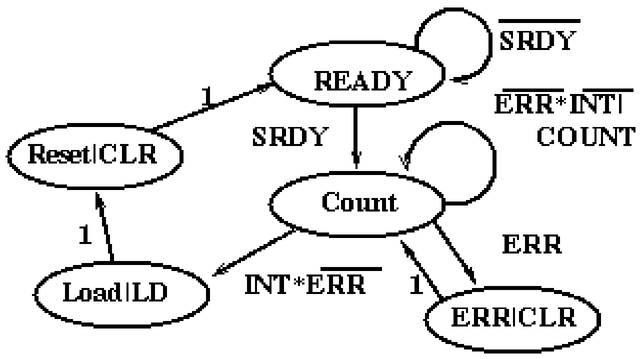
Stay Tuned: we will illustrate these steps.

Example: Digitizer Interface, FSM Control

- Position detection using an array of wires
  - Generate magnetic field with a coil (not shown here)
  - Count while sweeping over the array (contents of Counter)
- Detect position of a cursor:
  - By phase reversal Or other artifact of signal detectionv(INT signal)
  - Put count into a register (/LD is low)
- Implement a 'Handshake'
  - Set handshake line (dav) when signal is ready
  - Wait for ready signal (rdy) before counting (SRDY is synchronized RDY)



Here is the conventionally drawn FSM diagram of the system we are going to implement:



States:

Ready: waiting for the synchronized RDY signal from the user (_ of handshake)

Count: counter is incrementing itself along with the position sensor of the grid

ERR: Counter has overflowed, which means sensor was not found

Load: Counter was interrupted by finding the sensor: contents (count) is the position Count is loaded onto output counter

Reset: transient state - counter is cleared and transition is made to Ready

I/O Signals for FSM

Input:

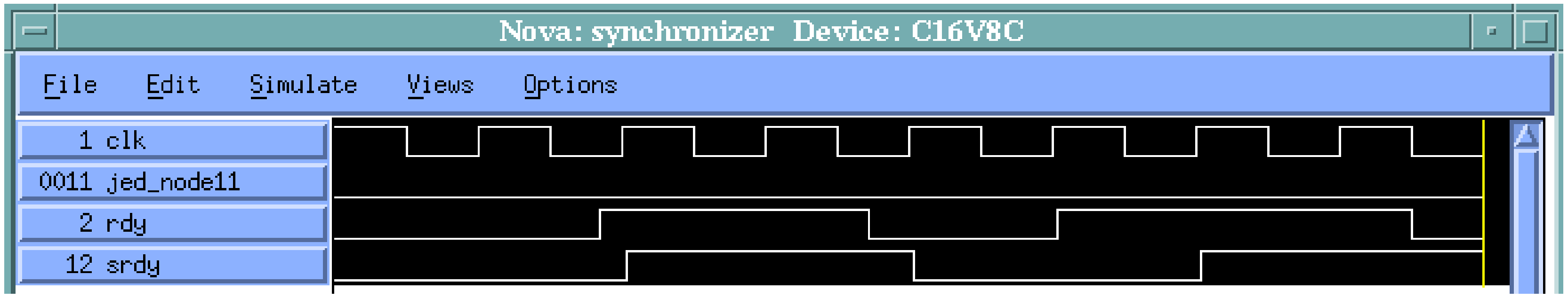
SRDY	**Synchronized GO (receiver ready)**
INT	**Grid position is detected (assumed synchronized)**
ERR	**Grid overflow (position not detected)**

Output:

DAV	**Data is ready**
LD	**Load count into the output register**
CLR	**Clear the counter**
COUNT	**Enable the counter to count**

One smaller of the blocks is the syhchronizer:

```
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  port (rdy, clk : in  std_logic;
        srdy     : out std_logic);
end synchronizer;

architecture behavioral of synchronizer is
begin  -- behavioral
  sync:process(clk)
  begin
    if rising_edge(clk) then
      srdy <= rdy;
    end if;
  end process sync;
end architecture behavioral;
```

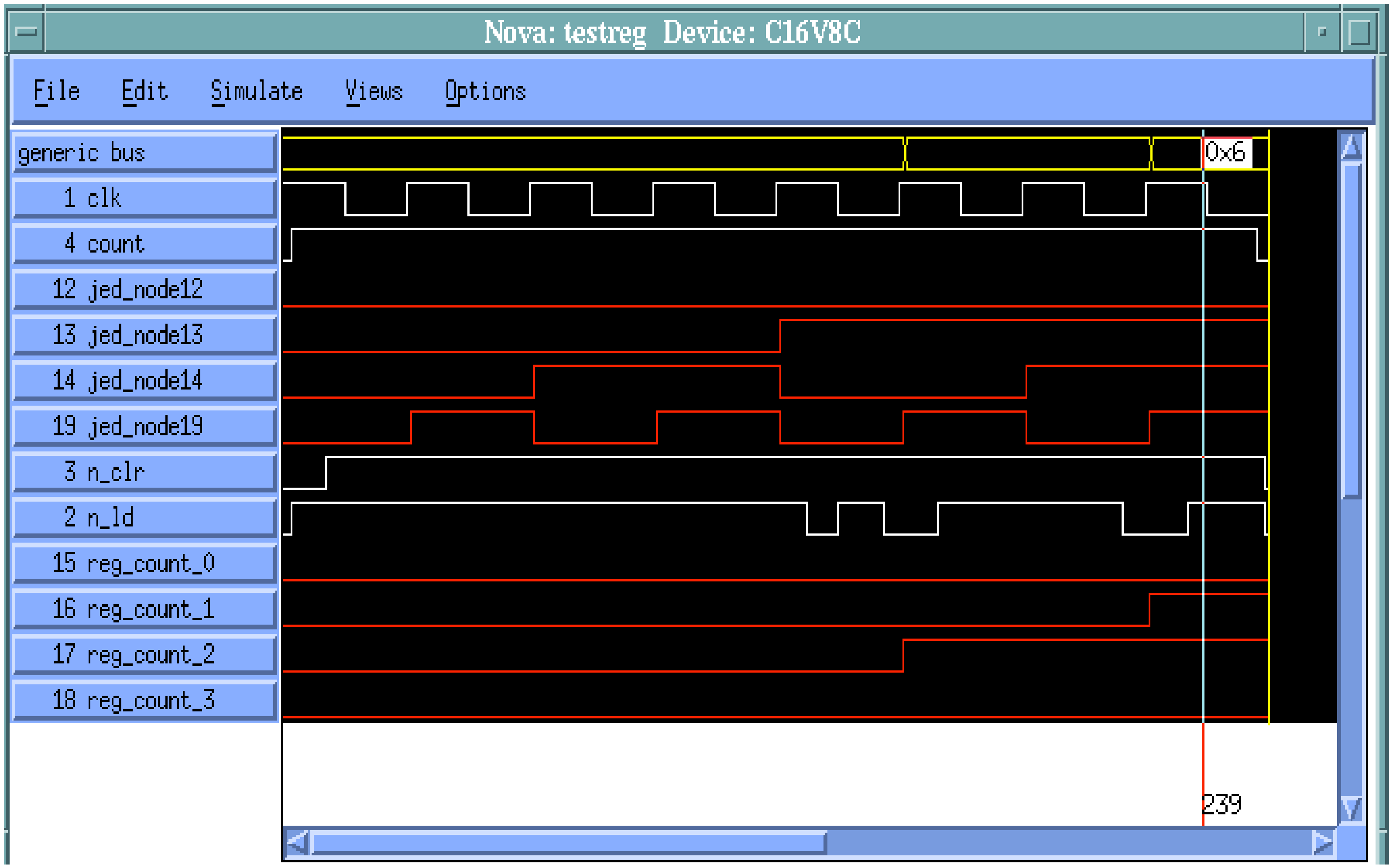


Second Part: This is a loadable register whose width is a generic.
size has a default - one number to change
instantiation as a component can define size

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is
  generic (size: integer := 4);
  port (n_ld, clk : in  std_logic;
        grid : in std_logic_vector(size - 1 downto 0);
        data : out std_logic_vector(size - 1 downto 0));
end reg;
architecture behavioral of reg is
begin  -- behavioral
  regff:process(clk)
  begin
    if rising_edge(clk) then
      if n_ld = '0' then
        data <= grid;
      end if;
    end if;
  end process;
end architecture behavioral;
```

Now we are going to test the resister, using a counter which we have already designed and will discuss next.

```
library ieee;
use ieee.std_logic_1164.all;
use work.gridpkg.all;
entity testreg is
    generic (gridsize : integer := 4); -- adjustable
    port (count, n_clr, n_ld, clk : in std_logic;   -- simple inputs
          reg_count : out std_logic_vector(gridsize-1 downto 0));
                                                     -- to see if it works
end testreg;

-- purpose: assemble counter and register
architecture test of testreg is
  signal gridcnt : std_logic_vector(gridsize-1 downto 0);  -- internal
count
  signal err : std_logic;       -- counter overflow

begin  -- test
    count_circuit: ctr
        port map (count => count, n_clr => n_clr, clk => clk,
                  err=> err, grid => gridcnt);
    reg_circuit: reg
        port map (n_ld => n_ld, clk => clk, grid => gridcnt,
                  data => reg_count);
end test;
```


Nova: testreg Device: C16V8C
File
Edit
Simulate
Views
Options
generic bus
1 clk
4 count
12 jed_node12
13 jed_node13
14 jed_node14
19 jed_node19
3 n_clr
2 n_ld
15 reg_count_0
16 reg_count_1
17 reg_count_2
18 reg_count_3
0x6
239

Comments on Register Testing (Simulation)

Creation of buses often helps.
Note we have specified a bus (a group of lines)
  Beware ==> one cannot use buses to specify inputs. <==
  Buses merely provide a way of displaying signal values.

Load behavior of the register is as expected

You can see the counter doing its thing

Register loads on a clock edge when the /LD line is low

Next is the counter (which we have already used), but this should already be familiar:

As promised earlier, here is a clearable counter with a carry out
Note this one has generic (adjustable) size

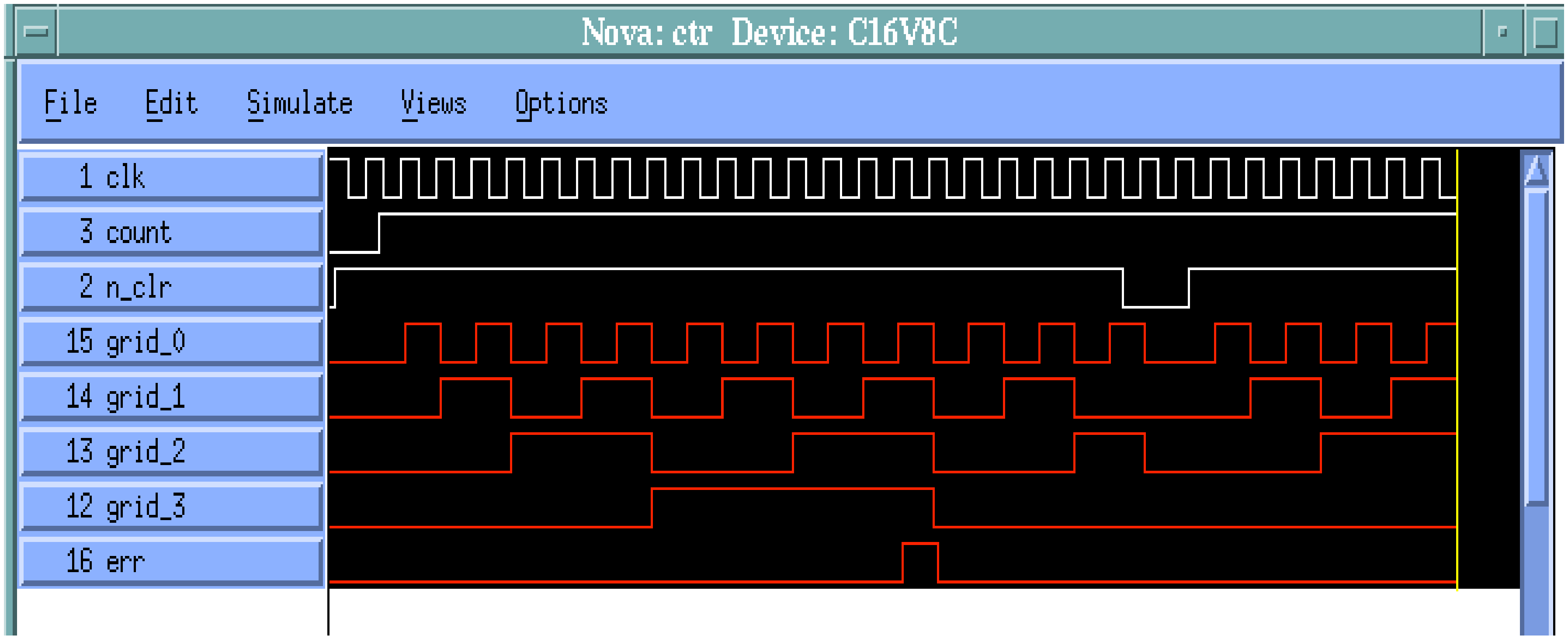
```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity ctr is
  generic (size: integer := 4);
  port (count, n_clr, clk : in  std_logic;
        err : out std_logic;
        grid : out std_logic_vector(size - 1 downto 0));
end ctr;
architecture behavioral of ctr is
  signal cnt_int : std_logic_vector(size - 1 downto 0);
  signal all_ones : std_logic_vector(size - 1 downto 0);
begin  -- behavioral
  all_ones <= (others => '1');
  grid <= cnt_int;
  err <= '1' when cnt_int = all_ones else '0';
  state_transition:process(clk)
  begin
    if rising_edge(clk) then
      if n_clr = '0' then
        cnt_int <= (others => '0');
      elsif count = '1' then
        cnt_int <= cnt_int + 1;
      end if;
    end if;
  end process state_transition;
end behavioral;
```

Here is the simulation of the counter:

It counts (see the progression of grid_0 to grid_3)

RCO (err) is asserted at the right time (1111)

The thing clears synchronously when n_clr is brought low)



Now the control part of the system is the FSM: fsmt.vhd

There are multiple ways of defining states:

Does one use constants or enumerated types?

In some cases, one doesn't need the "efficiency" of making the state assignment. The system will do it if we don't. But here we do it.

Here is the entity statement and the beginning of the architecture:

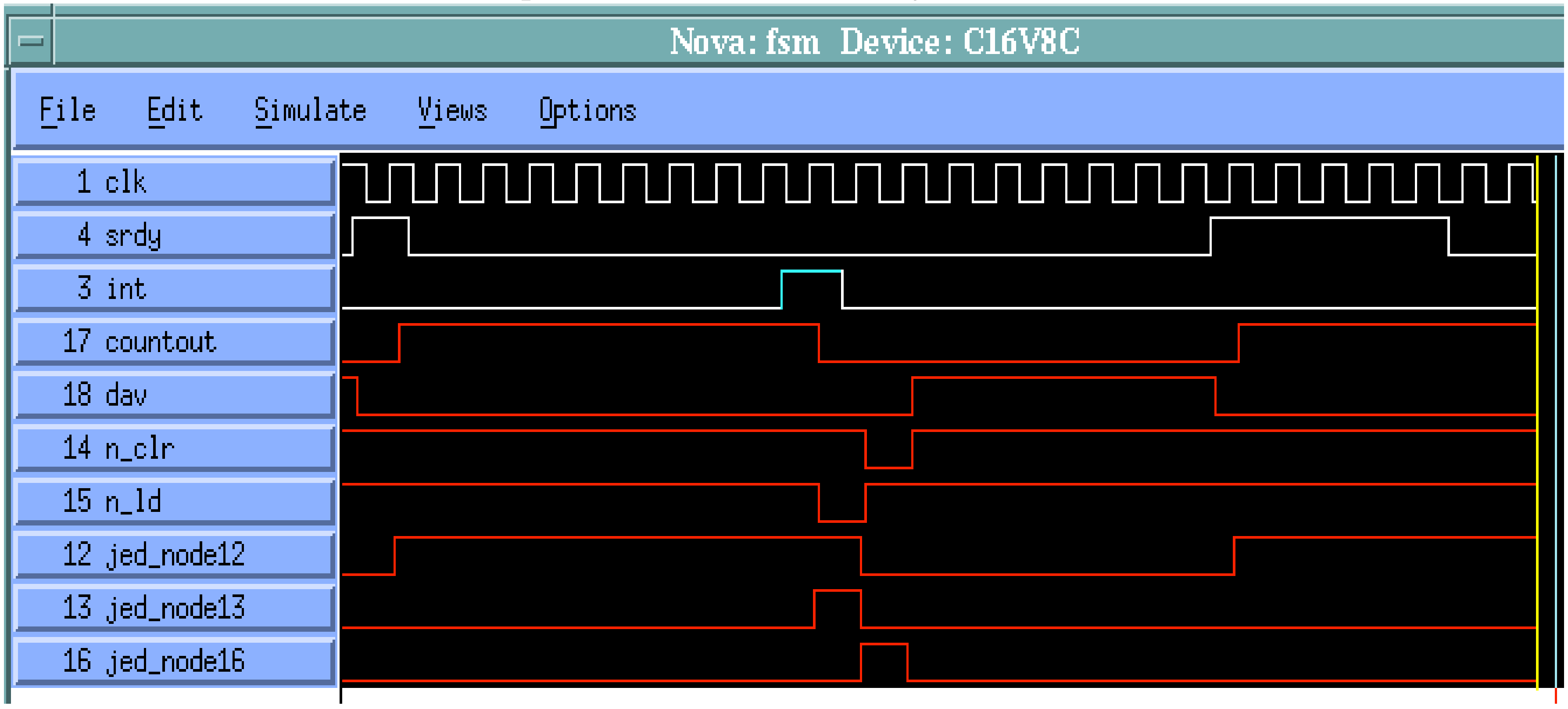
```
library ieee;
use ieee.std_logic_1164.all;

entity fsm is
  port (srdy, int, errin, clk : in  std_logic;
        dav, countout, n_clr, n_ld  : out std_logic);
end fsm;

architecture behavioral of fsm is
  type StateType is (READY, Count, Load, ERR, Reset);
  attribute enum_encoding of StateType: type is
    "000 001 011 010 100";
  signal state : StateType;
```

```
begin  -- behavioral
  n_clr <= '0' when (state = Reset) or (state = ERR) else '1';
  n_ld  <= '0' when state = Load else '1';
  countout <= '1' when state = Count else '0';
  dav   <= '1' when (state = READY) and (srdy = '0') else '0';
  state_transitions:process(clk)
  begin
    if rising_edge(clk) then
      case state is
        when READY =>
          if srdy = '0' then state <= READY;
          else state <= Count;
          end if;
        when Count =>
          if errin = '1' then state <= ERR;
          elsif int = '0' then state <= Count;
          else state <= Load;
          end if;
        when Load =>
          state <= Reset;
        when Reset =>
          state <= READY;
        when ERR =>
          state <= Count;
        -- don't need "when others" as all cases guaranteed
      end case;
    end if;
  end process state_transitions;
end architecture behavioral;
```

FSM Testing (Simulation)

Exercise all state transitions

An advantage of using constants rather than enumerated types is that the state names are visible. One has to poke around to see which jedec nodes encode the state!



State | 0 | 1 | 3 | 4 | 0 | 1

(0 => READY, 1 => Count, 3=> Load, 4 => RESET)

Now we build all the component parts into a package: this is the header of that package

```
use ieee.std_logic_1164.all;
package gridpkg is
  component synchronizer
  port (rdy, clk : in  std_logic;
        srdy     : out std_logic);
  end component;
  component fsm
  port (srdy, int, errin, clk : in  std_logic;
        dav, countout, n_clr, n_ld  : out std_logic);
  end component;
  component ctr
  generic (size: integer := 4);
  port (count, n_clr, clk : in  std_logic;
        err : out std_logic;
        grid : out std_logic_vector(size - 1 downto 0));
  end component;
  component reg
  generic (size: integer := 4);
  port (n_ld, clk : in  std_logic;
        grid : in std_logic_vector(size - 1 downto 0);
        data : out std_logic_vector(size - 1 downto 0));
  end component;
end gridpkg;
----------------------------
```

And then assemble the whole package file to be compiled by:

```
cat gridpkg.vhd synchronizer.vhd reg.vhd ctr.vhd fsmt.vhd > all.vhd
```

To generate gridtop.vhd, the 'top level' of the system:

'Wire' the components together using structural instantiation. (This is isomorphic with physically wiring the pieces together).

The entity only has signals specified in the one-block block diagram.

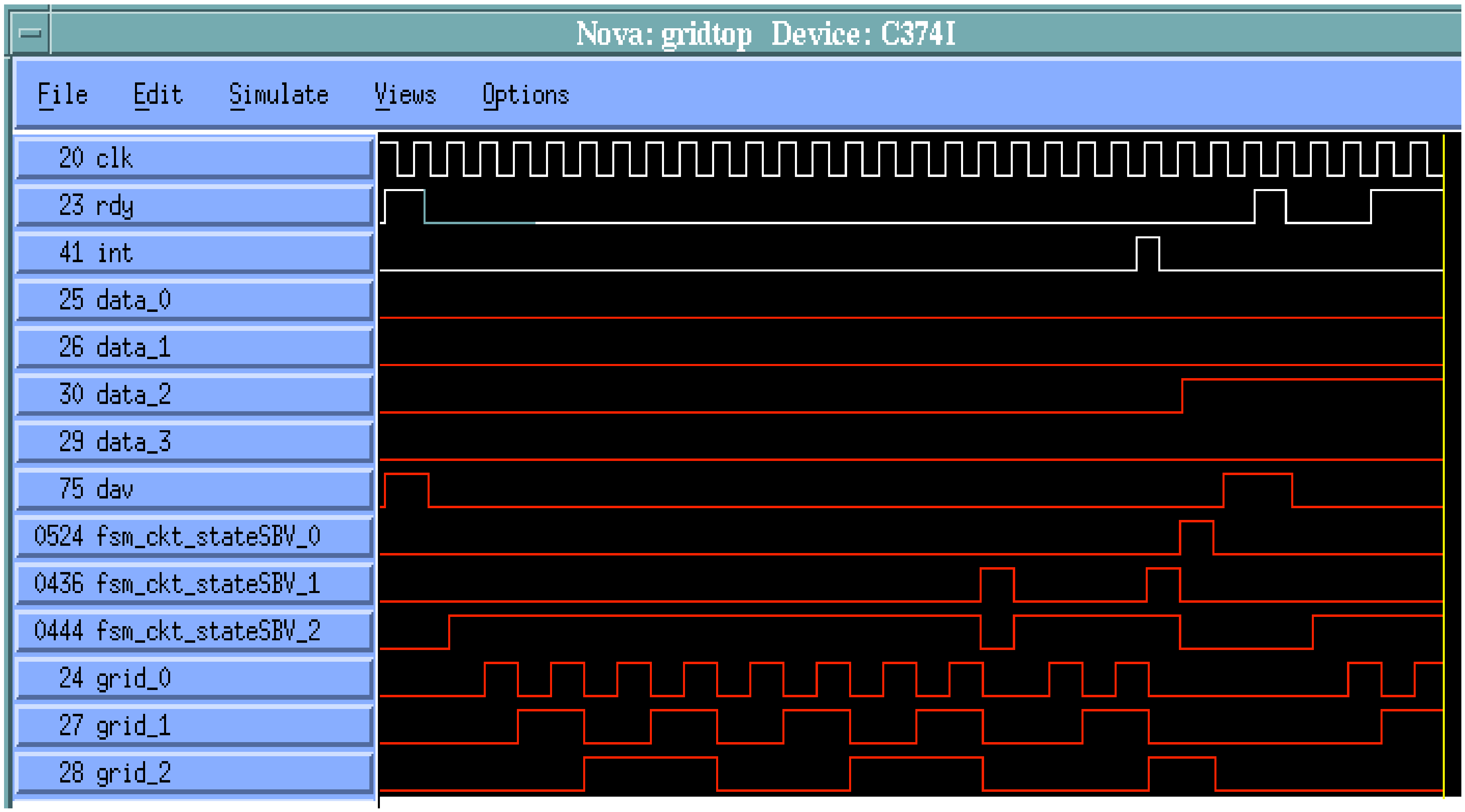
I used a generic, gridsize, for ease of overall testing.

```
library ieee;
use ieee.std_logic_1164.all;
use work.gridpkg.all;  --built according to instructions earlier
entity grid is
  generic (gridsize: integer := 4);
  port (rdy, int, clk : in  std_logic;
        dav : out std_logic;
        data : out std_logic_vector(gridsize - 1 downto 0);
        grid : out std_logic_vector(gridsize - 1 downto 0));
end grid;
```

The architecture part of the assembled system is quite simple, reflecting the structure of the top-level block diagram

```
architecture top of grid is
  signal srdy, err : std_logic;
  signal count, n_clr, n_ld : std_logic;
  signal gridint : std_logic_vector(gridsize - 1 downto 0);
begin
  sync_ckt: synchronizer
    port map (clk => clk, rdy => rdy, srdy => srdy);
  fsm_ckt: fsm
    port map (srdy => srdy, int=> int, errin => err,
              clk => clk, dav => dav, countout => count,
              n_clr => n_clr, n_ld => n_ld);
  ctr_ckt: ctr
    generic map(size => gridsize)
    port map (count => count, n_clr => n_clr, clk => clk,
              err => err, grid => gridint);
  grid <= gridint;
  reg_ckt: reg
    generic map(size => gridsize)
    port map (n_ld => n_ld, clk => clk, grid => gridint,
              data => data);
end top;
```

Finally, we should be able to test the top level functionality (we do need to insert an 'int' signal)


Nova: gridtop Device: C374I
File
Edit
Simulate
Views
Options
20 clk
23 rdy
41 int
25 data_0
26 data_1
30 data_2
29 data_3
75 dav
0524 fsm_ckt_stateSBV_0
0436 fsm_ckt_stateSBV_1
0444 fsm_ckt_stateSBV_2
24 grid_0
27 grid_1
28 grid_2