



# **L9: Digital Systems Design Hierarchical Design**





# Digital Systems Design



- **Start with a simple (one block) block diagram.**
  - Provide a global English description.
  - Describe the approach (algorithm) in English.
  - Describe the input and output signals.
- **Synchronize the inputs.**
- **Expand the blocks. Use hierarchy.**
  - Repeat the expansion until all blocks are simple.
    - More than two levels may be desirable.
- **Implement these simple blocks and test them.**
  - Code them in VHDL, compile, and simulate.
    - Consider glitch free requirements if FSM outputs are system outputs.
- **Use structural instantiation in VHDL to wire the blocks.**
- **Actually build some subset if you need to provide proof of principle.**
- **Test the completed design by simulation and then by testing the hardware implementation.**



# Example



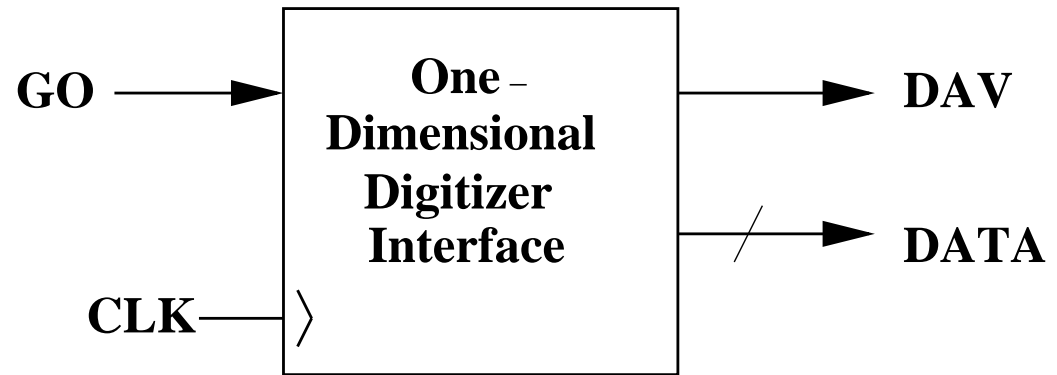
**Digitizer to record position of a cursor.**

**This system detects the position of a probe for one dimension. It is extendable to two dimensions. This is one way that digitizing tablets work.**

**A grid of wires is arrayed in the surface to be scanned.**

**The cursor (a physical device) produces a magnetic field which links with the wires in the surface. These wires are connected to a detector under control of a counter. The detector synchronously detects a phase reversal to signal that this wire is right under the cursor.**

**The normal state is that the digitizer is ready to make a measurement and the computer interface issues a GO signal (which is asynchronous to the digitizer clk). The digitizer then determines the data and provides it at the output and asserts DAV to tell the computer interface that data is available.**



**GO – Tell the digitizer to get the data.**

**DAV – Data is available to be read.**

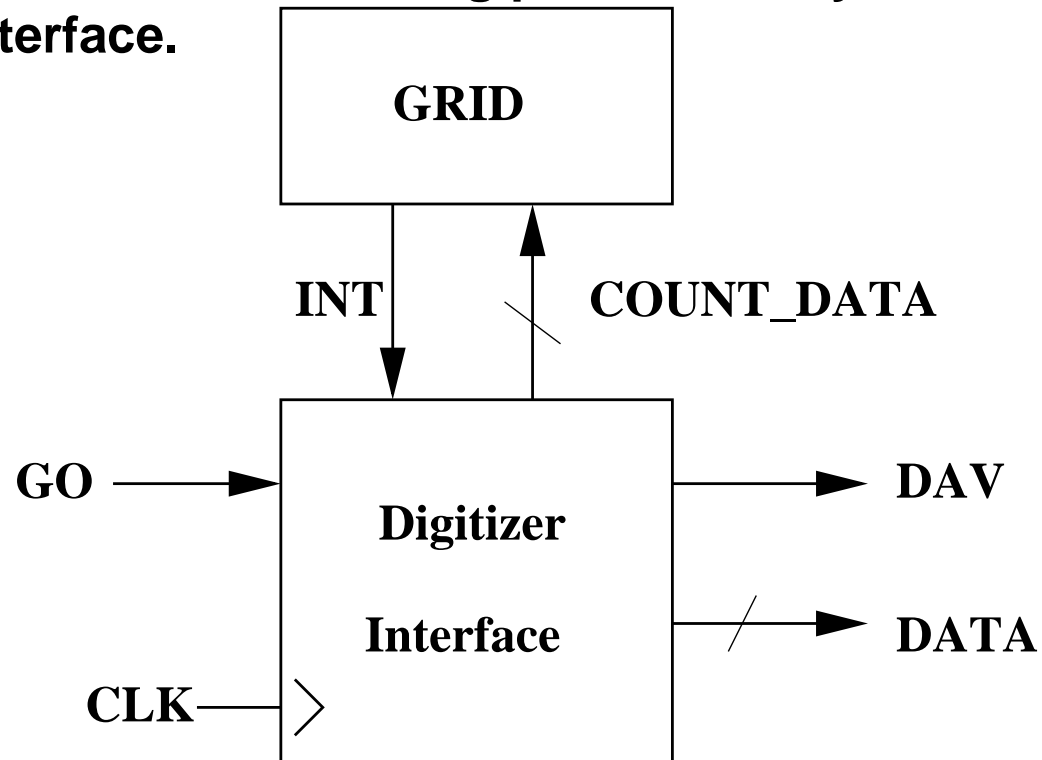
**DATA – Represents the X position of the pen.**



# Go to Two Blocks



First we isolate all the analog parts and only concern ourselves with the digital interface.



**GO** – Tell the digitizer to get the data.

**DAV** – Data is available to be read.

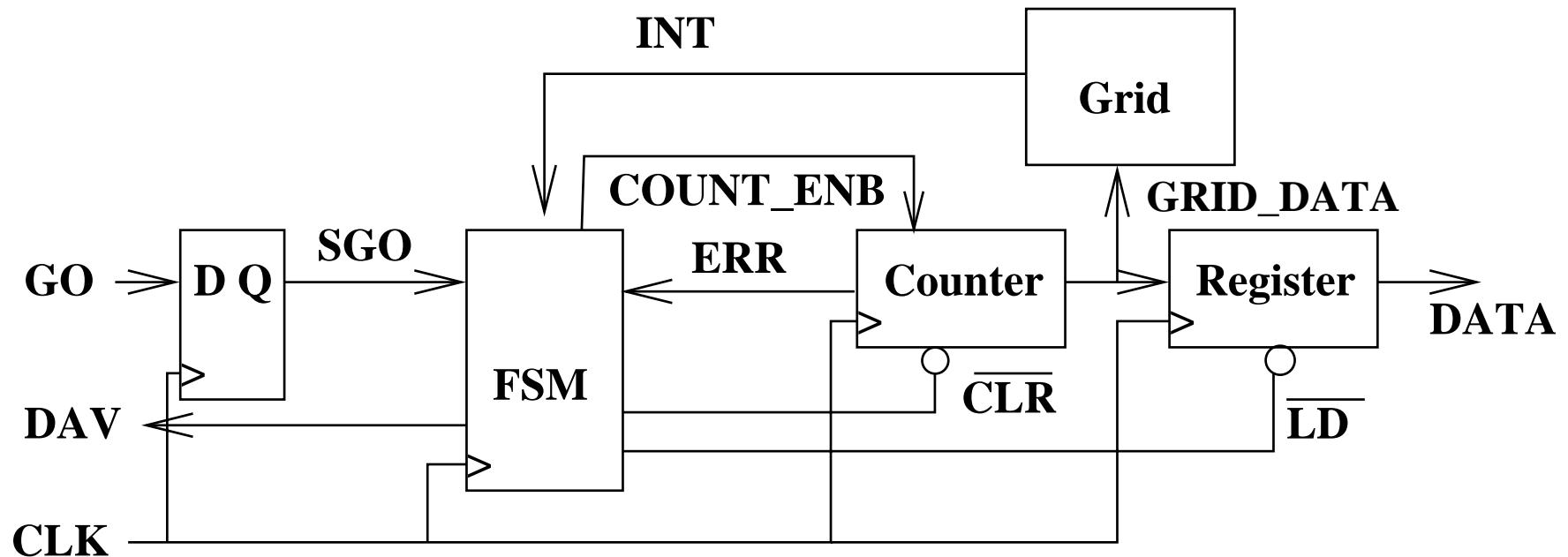
**DATA** – Represents the X position of the pen.

**INT** – Indicates that the cursor was detected.

**COUNT\_DATA** – Specifies grid wire to be energized.



# Enough Blocks



**GO** – Tell the digitizer to get the data.

**DAV** – Data is available to be read.

**DATA** – Represents the X position of the pen.

**INT** – Indicates that the cursor was detected.

**GRID\_DATA** – Specifies the grid wire to be energized.

**SGO** – A synchronized version of GO.

**COUNT\_ENB** – This enables the counter to count.

**ERR** – This indicates counter overflow without cursor detection.

**$\overline{\text{CLR}}$**  – This clears the counter.

**$\overline{\text{LD}}$**  – This loads the register with the counter data.



# FSM Specification

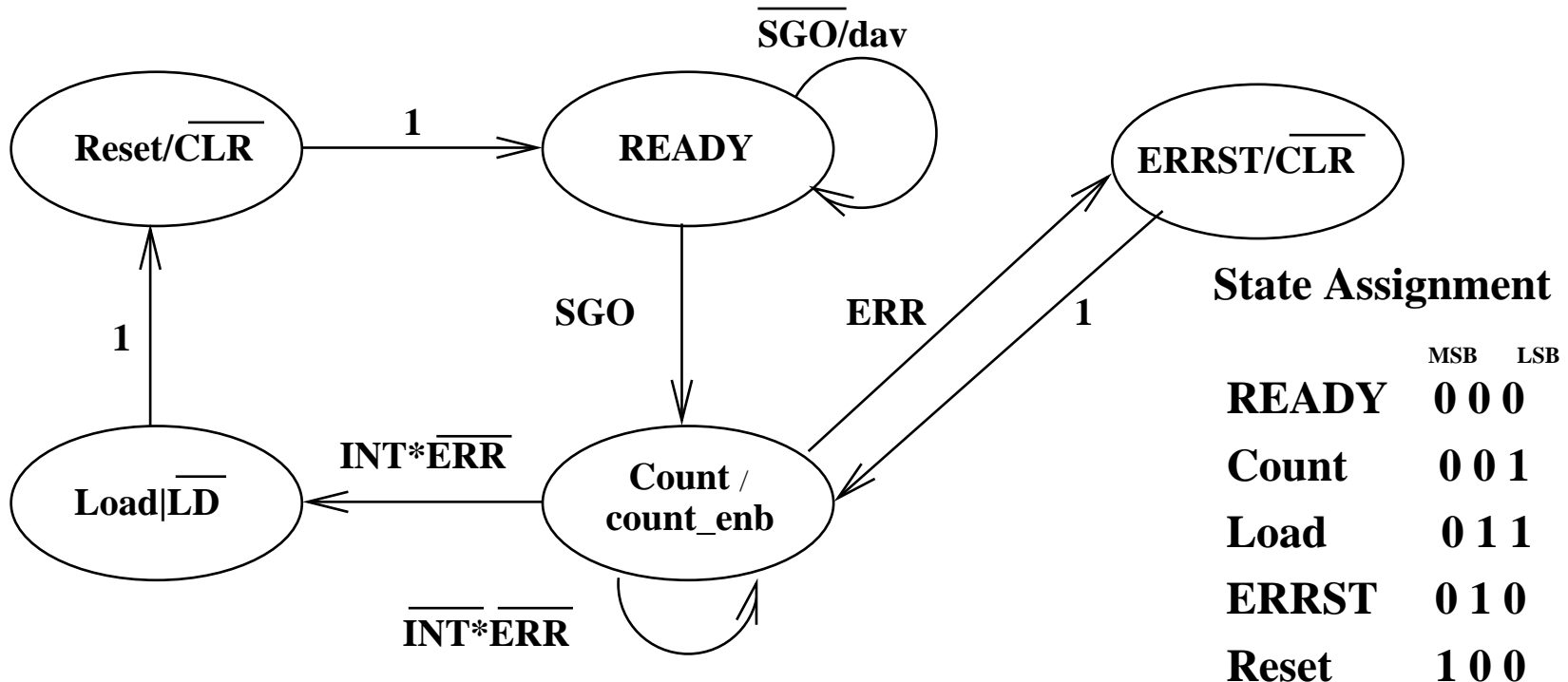


## Inputs:

- SGO** Synchronized version of GO
- INT** Cursor detected
- ERR** Grid overflow (position not detected)

## Outputs:

- DAV** Data is available.
- LD** Load count\_data (grid\_data) into the output register.
- CLR** Clear the counter.
- COUNT** Enable the counter to count.



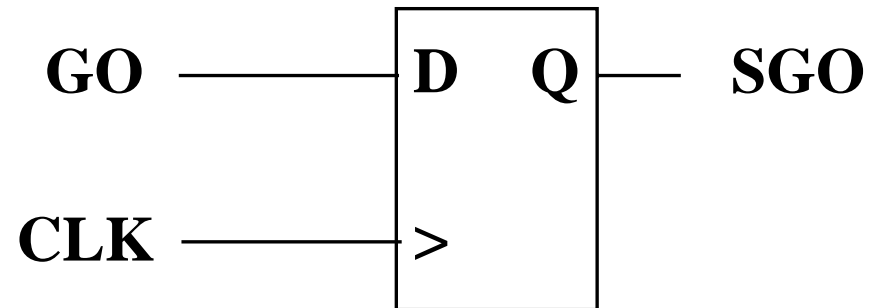


# Implement the Blocks in VHDL



Don't forget to synchronize! Even though it is simple.

```
library ieee;  
use ieee.std_logic_1164.all;  
entity synchronizer is  
  port (go, clk : in std_logic;  
        sgo      : out std_logic);  
end synchronizer;
```



```
architecture behavioral of synchronizer is  
begin -- behavioral  
  sync:process(clk)  
  begin  
    if rising_edge(clk) then  
      sgo <= go;  
    end if;  
  end process sync;  
  -- Why not use an internal signal and two flip flops?  
end architecture behavioral;
```



# Synchronizer Testing (Simulation)



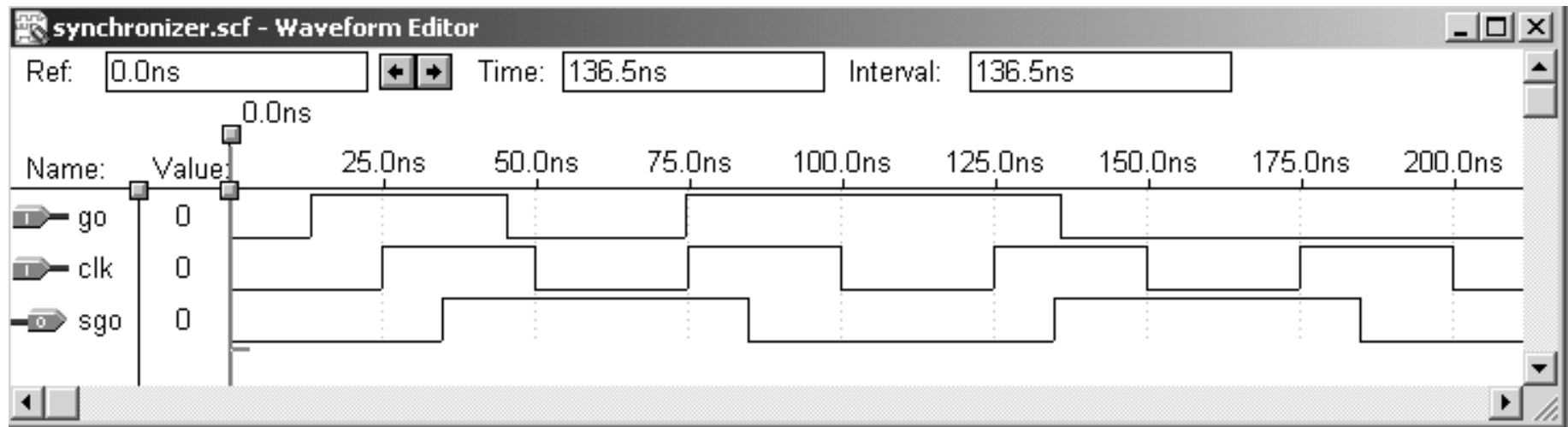
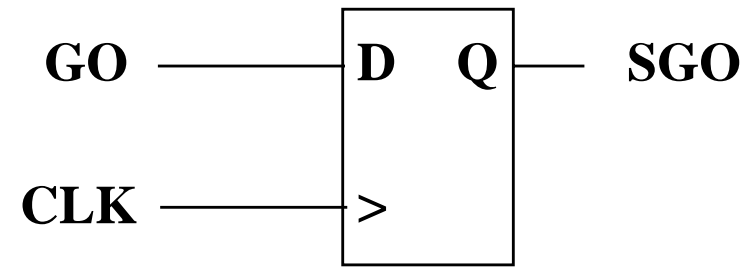
The synchronizer is simple, but important.

Should we also synchronize INT?

Do so if there is a doubt.

But, INT only happens after count\_data has changed.

So, it is already synchronized.

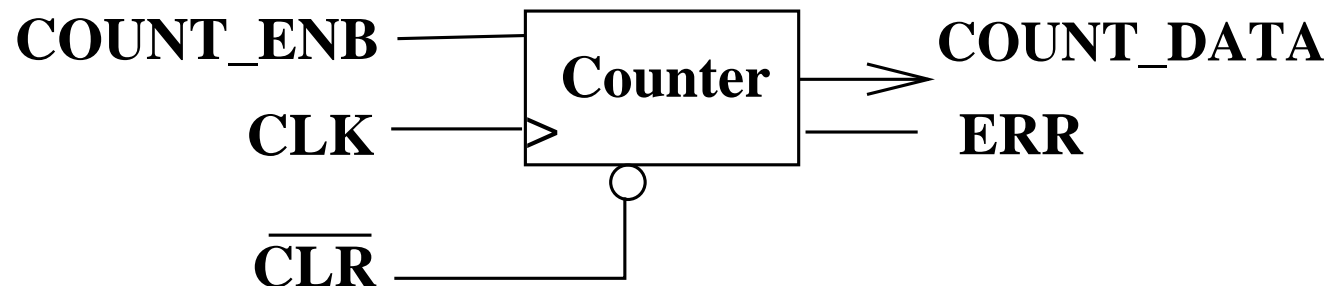




# ctr.vhd



```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
entity ctr is  
  generic (size: integer := 3);  
  port (count_enb, n_clr, clk : in std_logic;  
        err : out std_logic;  
        count_data : out std_logic_vector(size- 1 downto 0));  
end ctr;
```



**COUNT\_DATA** – This is the counter output.

**COUNT\_ENB** – This enables the counter to count.

**ERR** – This indicates counter overflow without cursor detection.

**$\overline{\text{CLR}}$**  – This clears the counter.

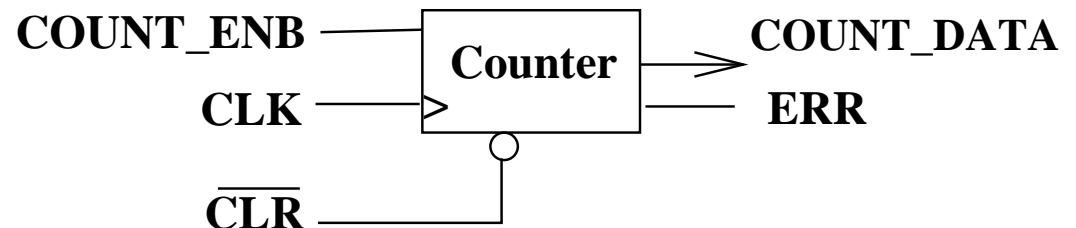


# ctr.vhd Architecture



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');
    count_data <= 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_enb = '1' then
                cnt_int <= cnt_int + 1;
            end if;
        end if;
    end process state_transition;
end behavioral;
```



# Test of ctr.vhd (Simulation)



**ERR is the carry out.**

**Note the glitch on ERR**

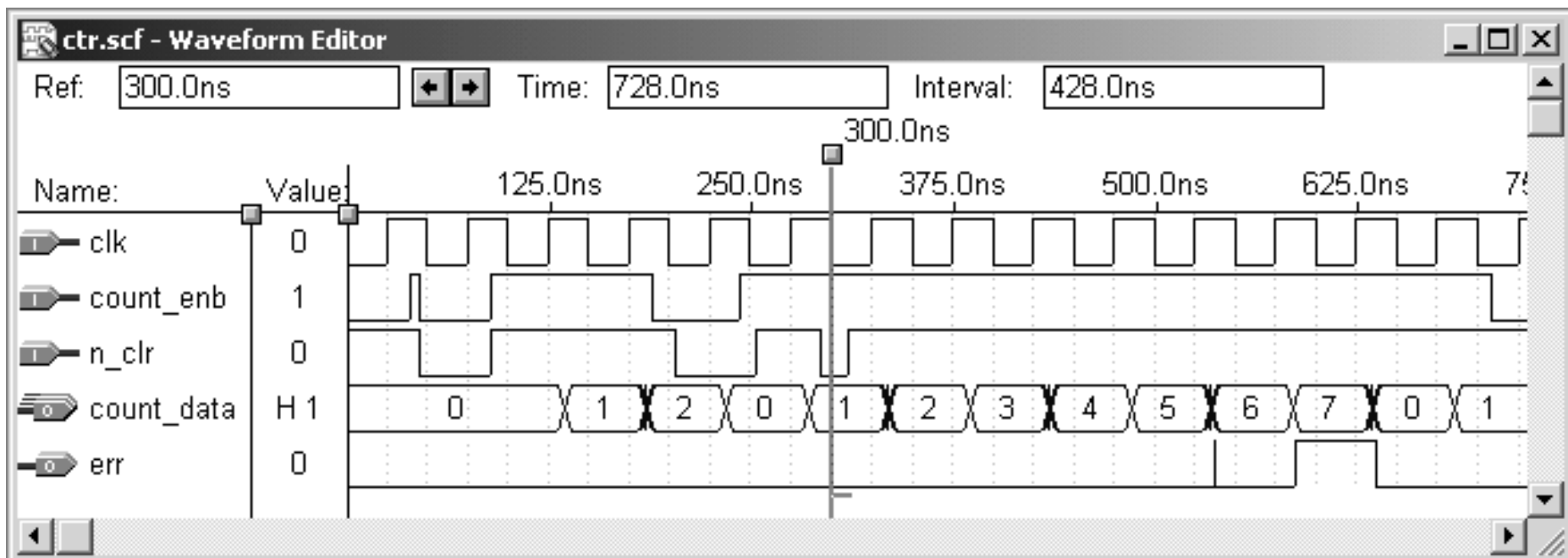
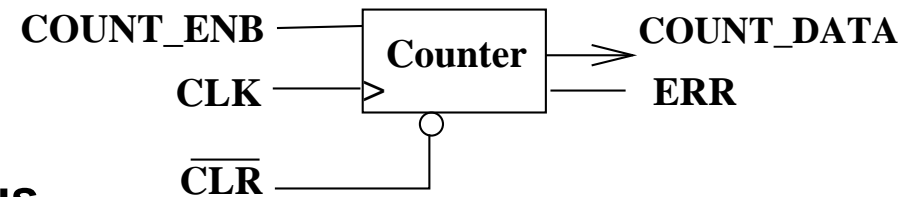
**Note there is no carry in.**

**count\_enb and n\_clr are synchronous**

**and have an effect only when bracketing a rising clk edge.**

**The testing (simulation) is only for a size of three.**

**Do you think it will work for size eight?**





# reg.vhd

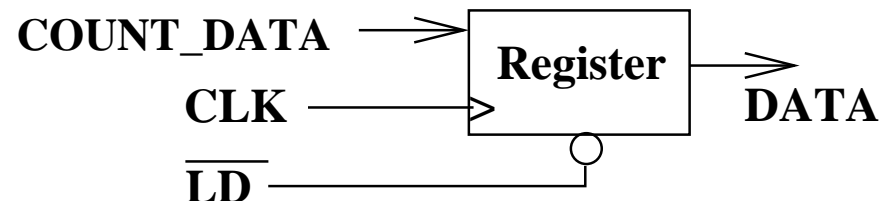


- This is a loadable register whose width is a generic.
- ~~Se~~ has a default- one number to change.
- Instantiation as a component can define size.

```

library ieee;
use ieee.std_logic_1164.all;
entity reg is
  generic (size: integer := 3);
  port (n_ld, clk : in std_logic;
        count_data : 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 <= count_data;
      end if;
    end if;
  end process;
end architecture behavioral;

```



- DATA – Represents the X position of the pen.
- COUNT\_DATA – Specifies the grid wire to be energized.
- $\overline{LD}$  – This loads the register with the counter data.



# regng.vhd – This Doesn't Work.



```
-- One can't combine other conditions with rising_edge(clk).
library ieee;
use ieee.std_logic_1164.all;
entity regng is
  generic (size: integer := 3);
  port (n_id, clk : in std_logic;
        count_data : in std_logic_vector(size- 1 downto 0);
        data : out std_logic_vector(size- 1 downto 0));
end regng;
architecture behavioral of regng is
begin -- behavioral
  regff:process(clk)
  begin
    if rising_edge(clk) and n_id = '0' then
      data <= count_data;
    end if;
  end process;
end architecture behavioral;
-- this produces
-- Error: line 13: File c:\documents: Unsupported feature
-- error: signal parameter in a subprogram is not supported
```

This is NOT allowed.





# regtst.vhd



```

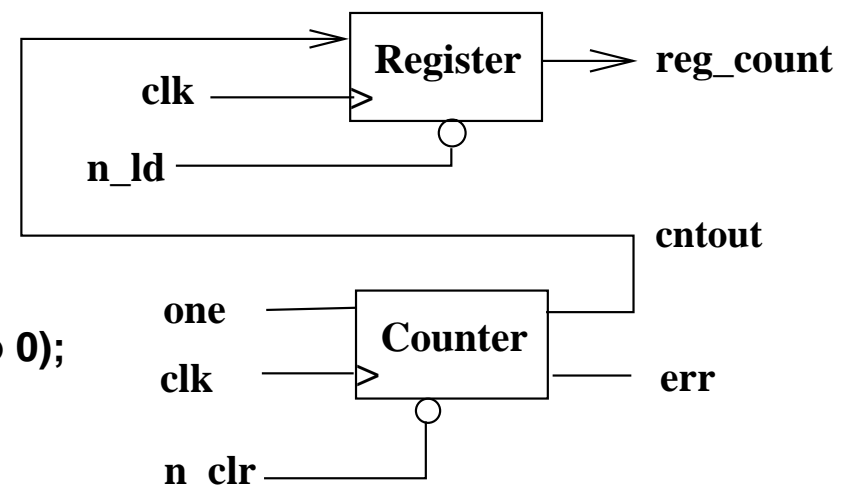
-- Test of register using counter as input
library ieee;
use ieee.std_logic_1164.all;
entity regtst is -- to see if the register works
  generic (gridsize : integer := 3); -- adjustable
  port (n_clr, n_ld, clk : in std_logic;
        reg_count : out std_logic_vector(gridsize 1downto 0));
end regtst;
-- purpose: assemble counter and register
architecture test of regtst is
  signal cntout : std_logic_vector(gridsize 1downto 0);
  signal err, one : std_logic; -- counter overflow
  component ctr
    generic (size: integer := 3);
    port (count_enb, n_clr, clk : in std_logic;
          err : out std_logic;
          count_data :
            out std_logic_vector(size- 1 downto 0));
  end component;
  component reg
    generic (size: integer := 3);
    port (n_ld, clk : in std_logic;
          count_data : in std_logic_vector(size- 1 downto 0);
          data : out std_logic_vector(size- 1 downto 0));
  end component;

```

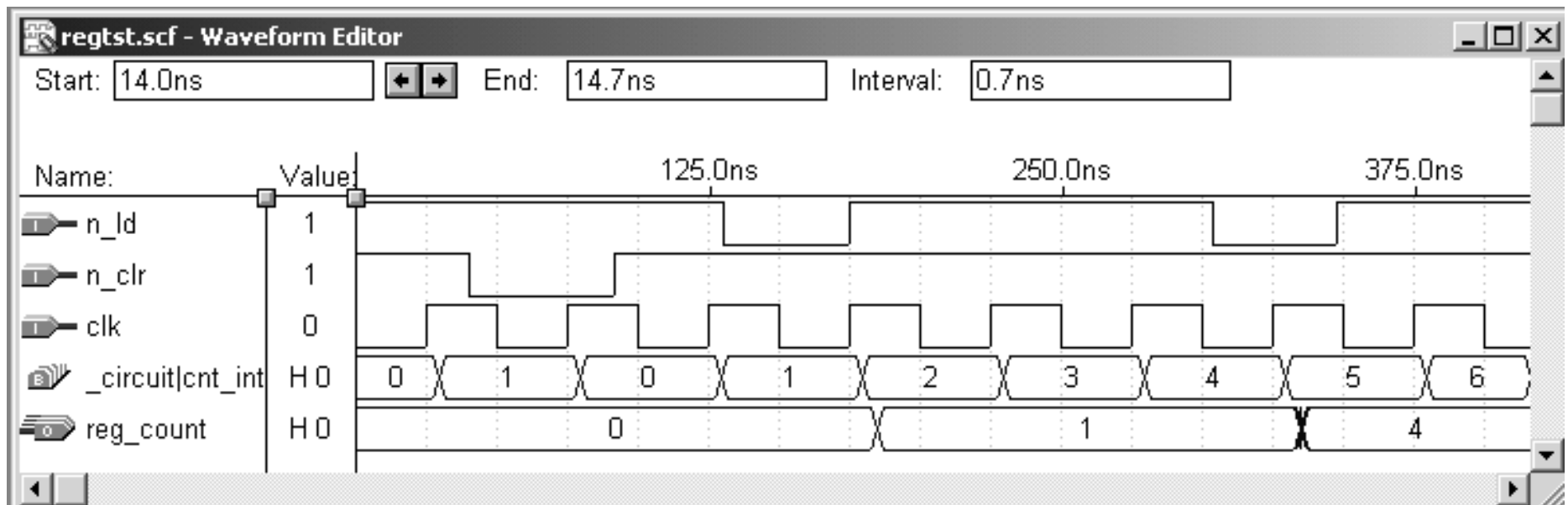
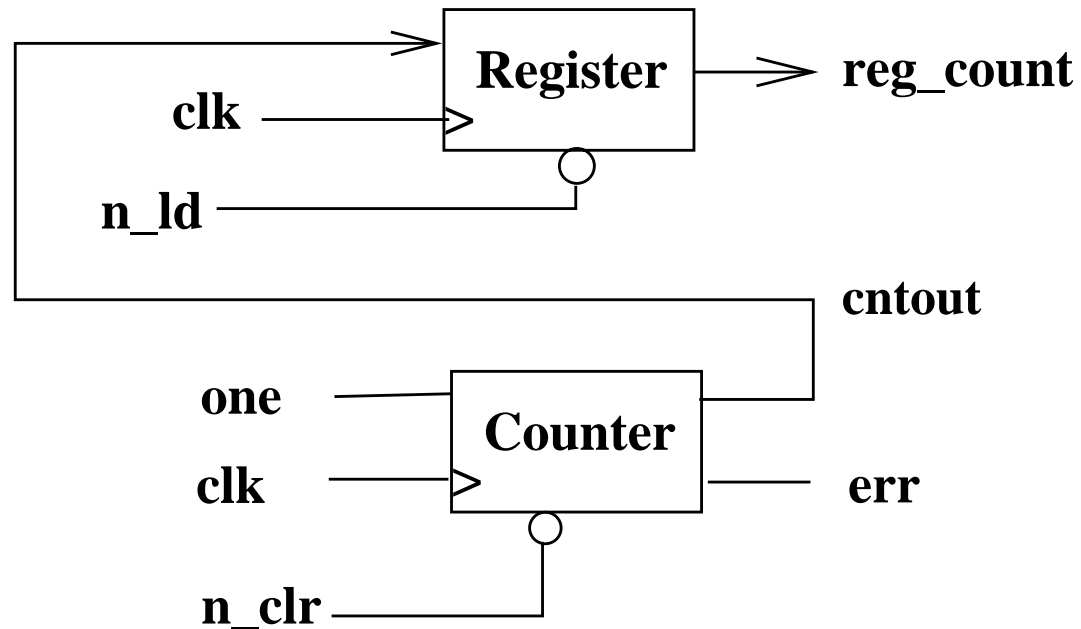
```

begin -- test
  one <= '1';
  count_circuit: ctr
    port map (count_enb => one,
              n_clr => n_clr,
              clk => clk,
              err=> err,
              count_data => cntout);
  reg_circuit: reg
    port map (n_ld => n_ld,
              clk => clk,
              count_data => cntout,
              data => reg_count);
end test;

```



# Test of reg.vhd (Simulation)





# fsm.vhd

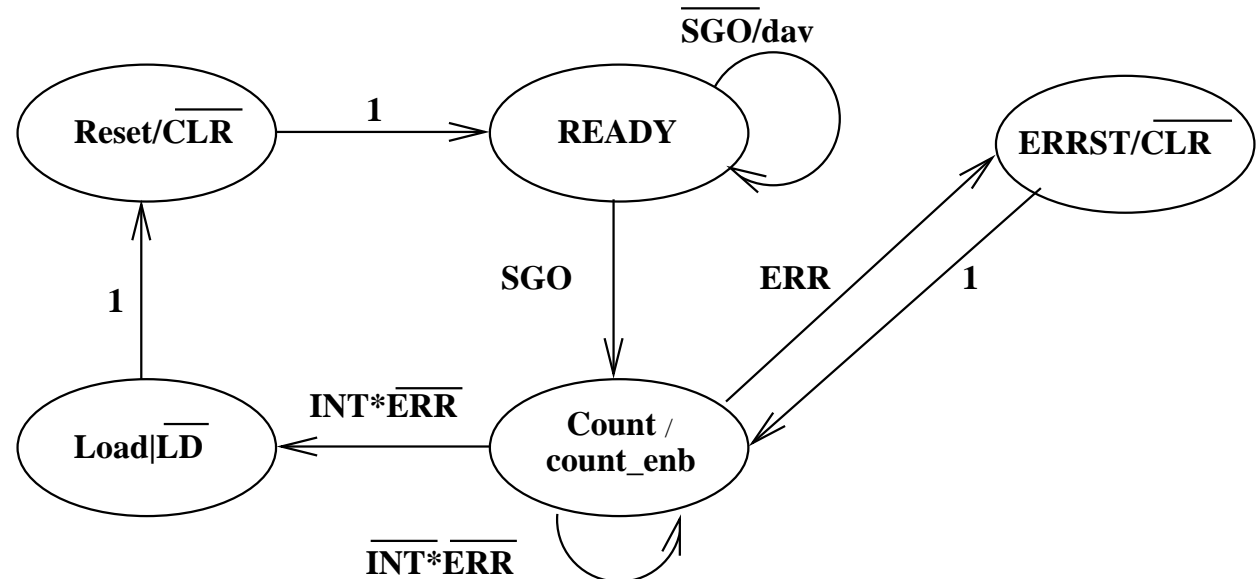


There are multiple ways of defining states.

Does one use constants or enumerated types?

Sometimes, the efficiency of making state assignments is not needed.

Here we assign them with the attribute `enum_encodings`.



```

library ieee;
use ieee.std_logic_1164.all;
entity fsm is
  port (sgo, int, err, clk : in std_logic;
        dav, count_enb, n_clr, n_ld : out std_logic);
end fsm;

```

```

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

```



# FSM Output Architecture



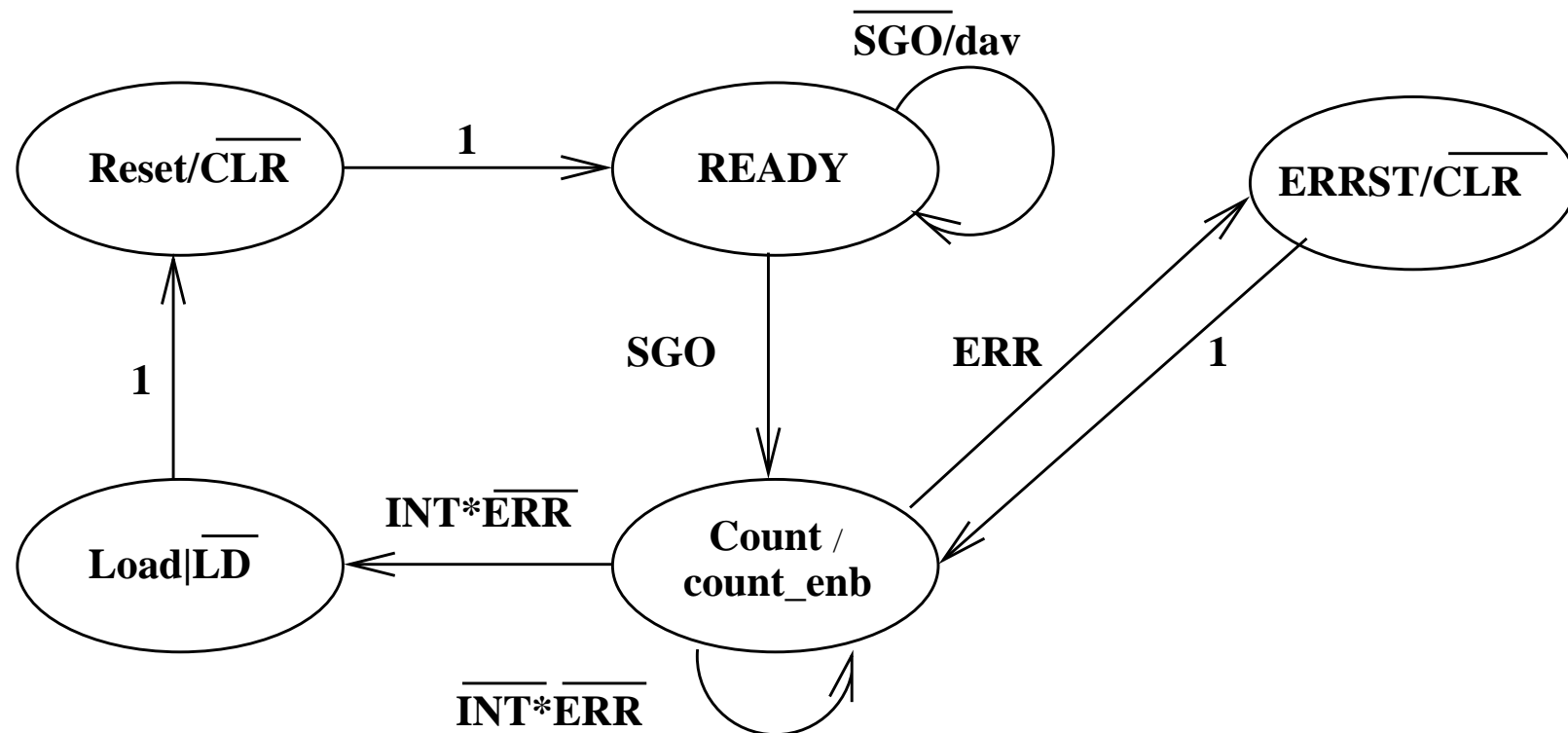
begin -- behavioral

n\_clr <= '0' when (state = Reset) or (state = ERRST) else '1';

n\_ld <= '0' when state = Load else '1';

count\_enb <= '1' when state = Count else '0';

dav <= '1' when (state = READY) and (sgo = '0') else '0';





# Rest of fsm.vhd Architecture



state\_transitions:process(clk)

begin

if rising\_edge(clk) then

case state is

when READY =>

if sgo = '0' then

state <= READY;

else state <= Count;

end if;

when Count =>

if err = '1' then state <= ERRST;

elsif int = '0' then state <= Count;

else state <= Load;

end if;

when Load =>

state <= Reset;

when Reset =>

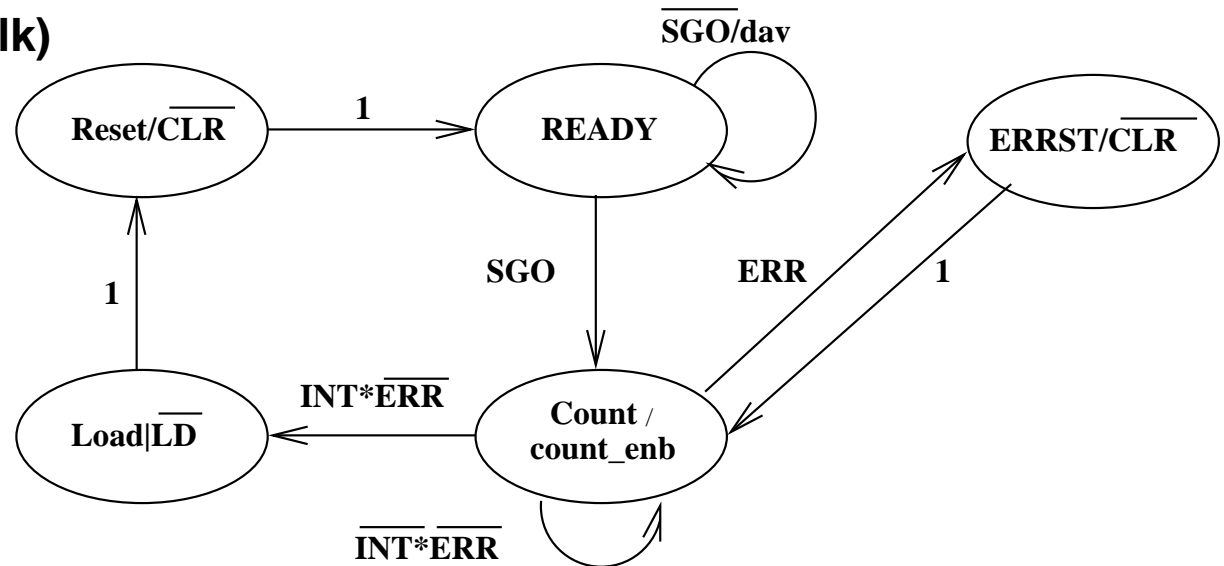
state <= READY;

when ERRST =>

state <= Count; -- don't need "when others" as all cases guaranteed

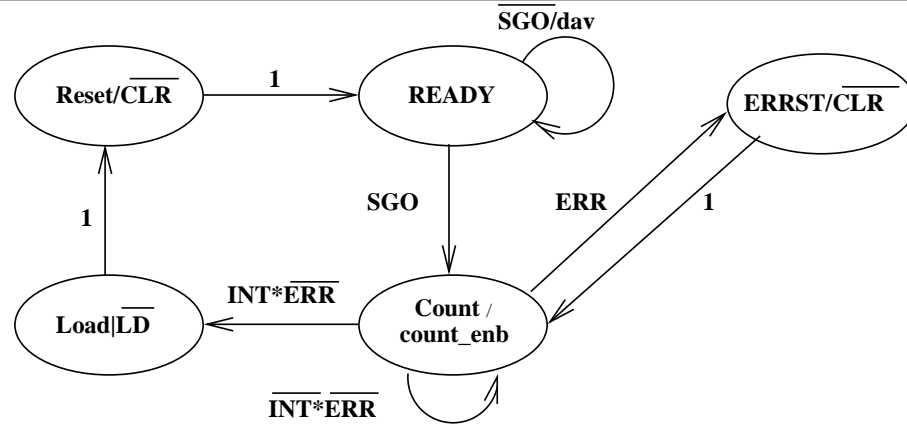
end case;

end if;



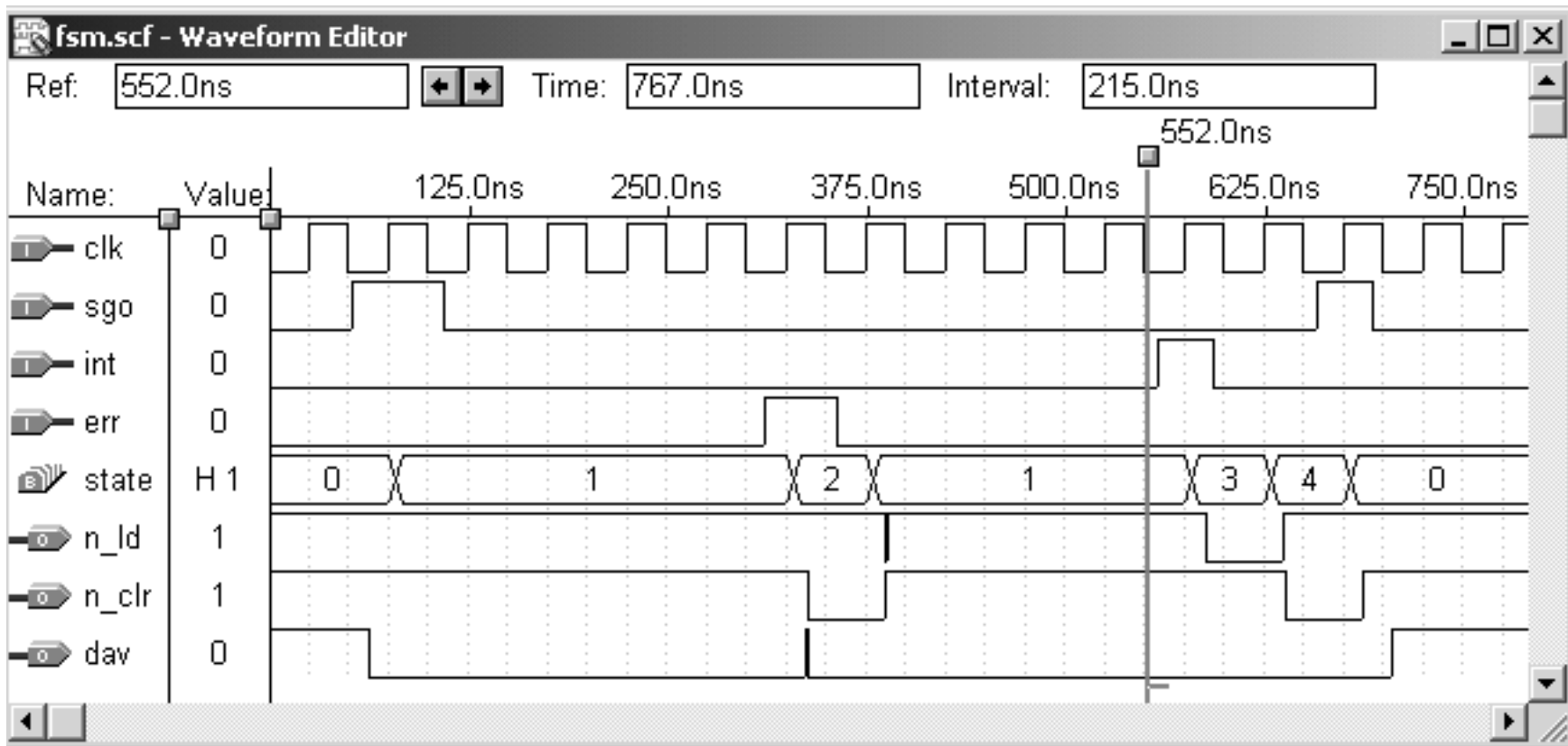


# Testing of fsm.vhd (Simulation)



Exercise all state transitions.

**READY**    **000**  
**Count**    **001**  
**Load**      **011**  
**ERRST**    **010**  
**Reset**     **100**





# Putting it All Together



- **First, test all the components.**
- **Put the xxx.vhd files in the same directory.**
  - If they aren't already there.
- **Create a top level file that instantiates the components and wires them together.**
  - While one can also include some VHDL that will produce circuitry, one is advised to have architectures that are instantiation alone.
    - Except for wires.
  - Do not mix other VHDL with instantiations.
- **You may declare components in the architecture section.**
- **Or, you can precede your entity/architecture pair with a package declaration.**
- **Either way it is a good idea to have only one entity/architecture pair per file (though not required).**



# Package Declaration



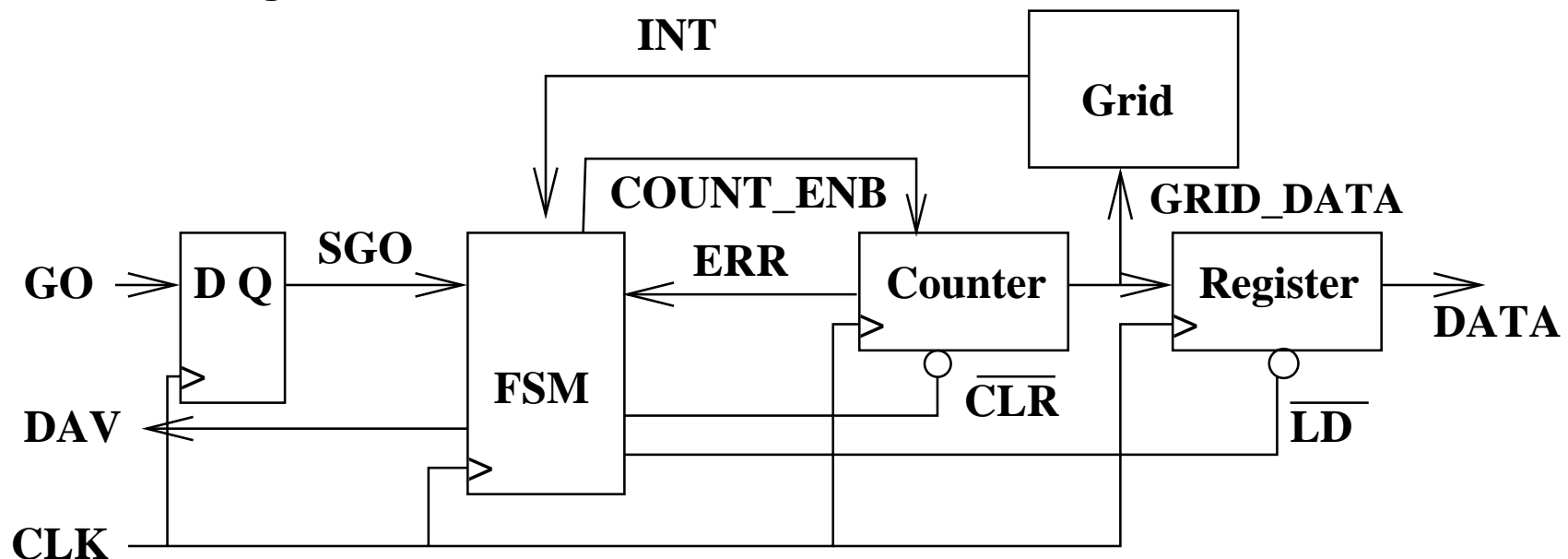
```
-- This is the first part of the file – grid.vhd
library ieee;
-- Take generic and port declarations from the entity.
use ieee.std_logic_1164.all;
package gridpkg is
  component synchronizer
    port (go, clk : in std_logic;
          sgo : out std_logic); end component;
  component fsm
    port (sgo, int, err, clk : in std_logic;
          dav, count_enb, n_clr, n_ld : out std_logic); end component;
  component ctr
    generic (size: integer := 3);
    port (count_enb, n_clr, clk : in std_logic;
          err : out std_logic;
          count_data : out std_logic_vector(size- 1 downto 0)); end component;
  component reg
    generic (size: integer := 3);
    port (n_ld, clk : in std_logic;
          count_data : in std_logic_vector(size- 1 downto 0);
          data : out std_logic_vector(size- 1 downto 0)); end component;
-- constant gridsize: integer := 4;
end gridpkg;
```



# grid.vhd- Entity



```
library ieee;
use ieee.std_logic_1164.all;
library work; -- This library statement is optional.
use work.gridpkg.all;
entity grid is
  generic (gridsize: integer := 4); -- if not in the package above
  port (go, int, clk : in std_logic;
        dav : out std_logic;
        data : out std_logic_vector(gridsize- 1 downto 0);
        grid_data : out std_logic_vector(gridsize- 1 downto 0));
end grid;
```





# grid.vhd- Architecture



architecture top of grid is

-- Note the use of generic map to specify the width of the ctr and reg.

```
signal sgo, err : std_logic;
```

```
signal count_enb, n_clr, n_ld : std_logic;
```

```
signal count_data : std_logic_vector(gridsize- 1 downto 0);
```

```
begin
```

```
sync_ckt: synchronizer
```

```
port map (clk => clk, go => go, sgo => sgo);
```

```
fsm_ckt: fsm
```

```
port map (sgo => sgo, int=> int, err => err,  
         clk => clk, dav => dav, count_enb => count_enb,  
         n_clr => n_clr, n_ld => n_ld);
```

```
ctr_ckt: ctr
```

```
generic map(size => gridsize)
```

```
port map (count_enb => count_enb, n_clr => n_clr, clk => clk,  
         err => err, count_data => count_data);
```

```
grid_data <= count_data;
```

```
reg_ckt: reg
```

```
generic map(size => gridsize)
```

```
port map (n_ld => n_ld, clk => clk, count_data => count_data,  
         data => data);
```

```
end top.
```



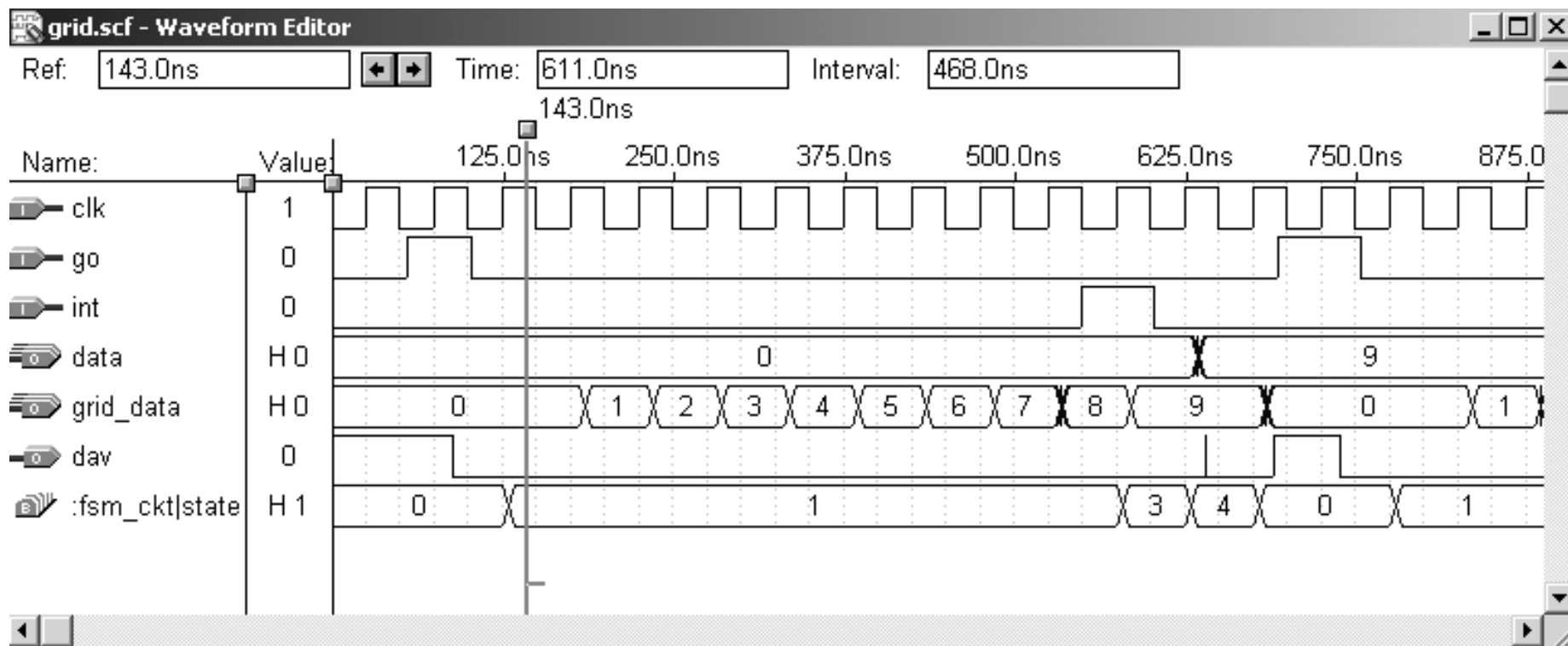
# Overall Testing (Simulation)



**Exercise all functions. Wow, is this tedious!**

**Well, see that some things work. Then hope for the best.**

**You'll do more simulation as the time delay to hardware implementation lengthens and/or the cost of hardware implementation rises and/or the time to test or debug the hardware lengthens.**





# Declare Components (No Package)



architecture top of gridcomp is

-- Note the use of generic map to specify the width of the ctr and reg.

```
signal sgo, err : std_logic;
```

```
signal count_enb, n_clr, n_ld : std_logic;
```

```
signal count_data : std_logic_vector(gridsize- 1 downto 0);
```

```
component synchronizer
```

```
port (go, clk : in std_logic;
```

```
      sgo : out std_logic); end component;
```

```
component fsm
```

```
port (sgo, int, err, clk : in std_logic;
```

```
      dav, count_enb, n_clr, n_ld : out std_logic); end component;
```

```
component ctr
```

```
generic (size: integer := 3);
```

```
port (count_enb, n_clr, clk : in std_logic;
```

```
      err : out std_logic;
```

```
      count_data : out std_logic_vector(size- 1 downto 0)); end component;
```

```
component reg
```

```
generic (size: integer := 3);
```

```
port (n_ld, clk : in std_logic;
```

```
      count_data : in std_logic_vector(size- 1 downto 0);
```

```
      data : out std_logic_vector(size- 1 downto 0)); end component;
```



# Design Rules



- **Use Hierarchical Design**
- **Test everything, all modules and the whole system.**
- **Synchronize all external inputs.**
  - **An asynchronous event must change only one flip- flop and that flip flop must not be used as an input to an output's combinational logic.**
- **Use the same clock edge for all edge triggered flip flops.**
  - **Clock period  $>$ Max (FF delay, input changes) + CL delay + Setup time**
  - **Beware of clock skew.**
- **CLK, /PR, /CLR, or G must NOT have glitches.**
  - **Most combinational logic is glitchy. Counter carry is glitchy.**
  - **When you need glitch free outputs**
    - **Gate clock pulses (carefully)**
    - **Use flip flops as outputs.**



# More Design Rules



- **Keep wires short.**
- **Wire all inputs (even unused ones) of all non Atera parts.**
- **Avoid tristate bus contention. Account for turn off delays.**
- **Avoid high Z address to SRAM when CE is true.**
- **Avoid address changes when write pulse is true.**
- **Use don't cares to simplify combinational logic.**
- **Use names and logic symbols appropriate for algebra and assertion levels. Make your design easy to understand.**
- **When you need glitch-free outputs**
  - **Gate clock pulses carefully to produce glitch free outputs.**
  - **Use flip flops as outputs to produce glitch free outputs.**