

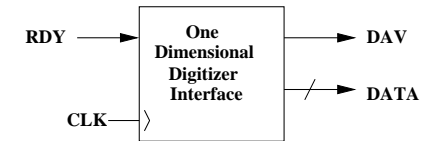
9.1 Hierarchical Design

Monday, February 26, 2001

- Start with a one-block diagram.
 - Expand to major blocks.
 - Repeat expansion until blocks are simple.
- Implement these simple blocks and test.
 - Code them in VHDL and simulate.
- Use structural instantiation in VHDL to wire the blocks together.
- Test the design.

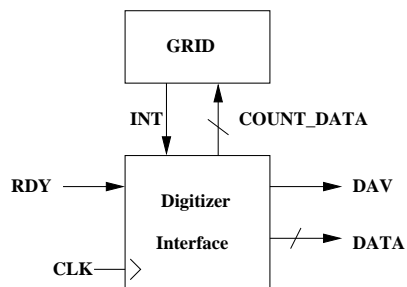
9.2 One-block Block Diagram

- Global English Description
 - Describe the function.
 - Describe the input and output signals.
- Example - Digitizer to Record Position
 - Simplify it so there is only one dimension.



RDY – The receiver is ready.
DAV – Data is available to be read.
DATA – Represents the X position of the pen.

9.3 Go to Two Blocks

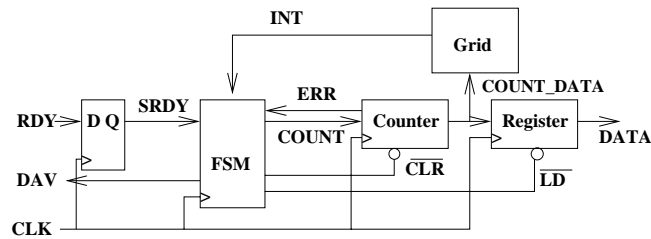


RDY – The receiver is ready.
DAV – Data is available to be read.
DATA – Represents the X position of the pen.
INT – Indicates that the cursor was detected.
COUNT_DATA – Specifies grid wire to be energized.

9.4 English Description of the Digitizer

- Position detection using an array of wires
 - Generate magnetic field with a coil.
 - Count while sweeping over the array.
- Detect position of a cursor:
 - By phase reversal
 - Or other artifact of signal detection
 - Put count into a register.
- Implement a 'Handshake'.
 - Set handshake line (dav) when signal is ready.
 - Wait for ready signal (rdy) before counting.

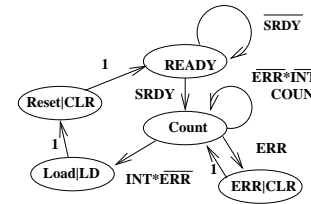
9.5 Enough Blocks



RDY – The receiver is ready.
 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 the grid wire to be energized.
 SRDY – A synchronized version of RDY.
 COUNT – This enables the counter to count.
 ERR – This indicates counter overflow without cursor detection.
 CLR – This clears the counter.
 LD – This loads the register with the counter data.

9.6 FSM Specification

- Input:
 - SRDY Synchronized version of RDY
 - INT Cursor detected
 - ERR Grid overflow (position not detected)
- Output:
 - DAV Data available
 - /LD Load COUNT_DATA into the output register.
 - /CLR Clear the counter.
 - COUNT Enable the counter to count.



State Assignment

	MSB	LSB
READY	0	0
Count	0	1
Load	0	1
ERR	0	1
Reset	1	0

9.7 Implement the Blocks with VHDL

- Don't forget to synchronize! Here is synchronizer.vhd.

```
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;
-- Why not use an internal signal and two flip-flops?
end architecture behavioral;
```

9.8 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.

9.9 reg.vhd

```
--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;
        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;
```

9.10 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 reg is
  generic (size: integer := 4);
  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) and n_ld = '0' then
      data <= count_data;
    end if;
  end process;
end architecture behavioral;
-- this produces
-- Warning: 'n_ld'should be referenced in the sensitivity list.
-- Warning: 'count_data'should be referenced in the sensitivity list.
-- Abort: Can't handle expression 's'event' in final equations.
```

9.11 Test Program for reg: uses ctr (Next!)

```
--Test of register using counter as input
library ieee;
use ieee.std_logic_1164.all;
use work.gridpkg.all;
entity testreg is -- to see if the register works
  generic (gridsize : integer := 4); -- adjustable
  port (count, n_clr, n_ld, clk : in std_logic;
        reg_count : out std_logic_vector(gridsize-1 downto 0));
end testreg;
-- purpose: assemble counter and register
architecture test of testreg is
  -- internal count
  signal gridcnt : std_logic_vector(gridsize-1 downto 0);
  -- gridcnt is the 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, count_data => gridcnt);
  reg_circuit: reg
    port map (n_ld => n_ld, clk => clk, count_data => gridcnt,
              data => reg_count);
end test;
```

9.12 Register Testing (Simulation)

- Creation of busses often helps.
- Beware - one cannot use busses to specify inputs.
- Busses merely provide a way of displaying signal values.



9.13 ctr.vhd

- A clearable counter with a carry out

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

- Architecture is on the next slide.

9.14 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 = '1' then
        cnt_int <= cnt_int + 1;
      end if;
    end if;
  end process state_transition;
end behavioral;
```

9.15 Counter Testing (Simulation)

- ERR is the carry out signal.
- Try out n_clr and count control inputs.



9.16 The Control FSM: fsm.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.

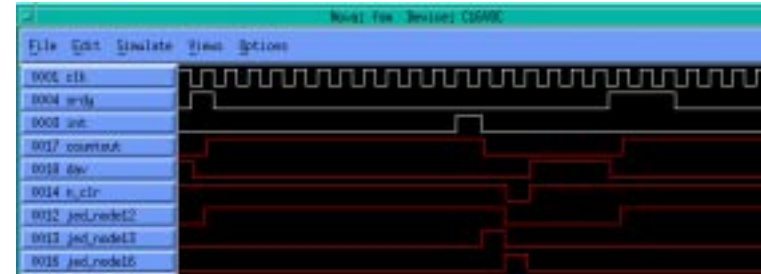
```
library ieee;
use ieee.std_logic_1164.all;
entity fsm is
  port (srdy, int, errin, clk : in std_logic;
        dav, countout, n_clr, n_ld : out std_logic);
end fsm;
architecture behavioral of fsm is
  type StateType is (READY, Count, Load, ERR, Reset);
  attribute enum_encoding of StateType: type 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)
  -- More architecture in the next slide.
```

9.17 More Architecture of fsmt.vhd

```
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;
```

9.18 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!



9.19 Package Declaraton - gridpkg.vhd

```
library ieee;
--Take generic and port declarations from the entity.
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;
          count_data : out std_logic_vector(size - 1 downto 0)); end component;
  component reg
    generic (size: integer := 4);
    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;
end gridpkg;
```

9.20 Putting It All Together

- First, test all the components.
 - Often do this with a simpler PLD (as in a PAL) as the computation time is shorter.
- After testing the components individually, put them all in a single file.
 - cat gridpkg.vhd synchronizer.vhd reg.vhd ctr.vhd fsm.vhd > all.vhd
- Set the device to the target device, C374i.
- Compile this file (without it being the top design).

9.21 gridtop.vhd

- Wire the components together using structural instantiation.
 - 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;
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;
```

9.22 gridtop.vhd Architecture

```
-- Note the use of generic map to specify the
-- width of the ctr and reg.
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, count_data => gridint);
  grid <= gridint;
  reg_ckt: reg
    generic map(size => gridsize)
    port map (n_ld => n_ld, clk => clk, count_data => gridint,
             data => data);
end top;
```

9.23 Overall Testing (Simulation)

- Exercise all functions.

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



9.24 Design Rules

Hierarchical design

Test everything, modules and the whole system.

Use names and Logic Symbols appropriate for algebraic representation and assertion levels.

CLK, /PR, /CLR, and G must NOT have glitches.

Most combinational logic is glitchy.

Carry from a counter is glitchy.

Gate clock pulse CAREFULLY.

Use the same clock edge for all edge triggered flip-flops.

Beware of clock skew!

Clock period >=

Max(FF delay, Input changes) + CL delay + Setup time.

Avoid tri-state 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.

Bypass (decoupling) capacitors are already on your kit.

Use monostables sparingly (if at all).

Keep wires short.

Wire all inputs (even unused ones).

Synchronize all external inputs.

An asynchronous event must change ONLY one flop-flop.