Dice Game Implementation

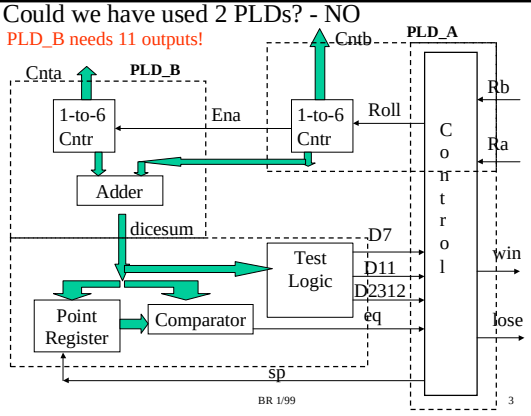
- **Why was dice game implemented in three 22V10 PLDs?**
- **What are the resources needed by the Dice Game?**
  - **Outputs: 6 for dice values (3 bits each dice), win, loss**
  - **Inputs: Ra, Rb, Reset, Clk**
  - **6 FFs for dice, minimum 3 FFs for FSM, 4 FFs for Point register (13 total)**
- **22V10 has**
  - **12 input-only pins, 10 input/output pins**
  - **10 FFs (1 FF per input/output pin)**
- **Need at least TWO 22V10s just for FFs**

Partitioning of Design

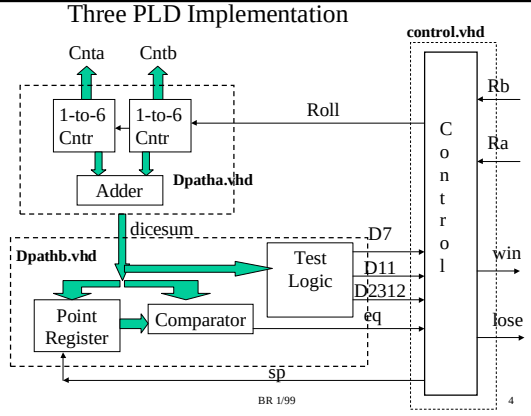
- **Splitting a design over multiple PLDs is called "partitioning"**
- **Number of inputs will not be a concern**
  - **Will be limited by # of FFs, # of outputs**
- **Could we have used just two PLDs? (see next page)**
  - **PLD_A: 3 FFs for Cntr, 3 FFs for FSM, 7 outputs (Cntb[2:0], Win, Lose, Ena, Sp). Could even use one-hot encoding for FSM (3 more FFs, three more outputs).**
  - **PLD_B: 3 FFs for Cntr, 4 FFs for Point Register, 7 outputs (Cnta[2:0], Eq, D7, D11, D2312). But 4 FFs for Cntr consumes 4 outputs so 11 outputs total!!!!**
  - **This division of the logic will not work.**
- **Some other partition of the logic could possibly be found. Three PLDs gives good observation of internal signal values.**

Could we have used 2 PLDs? - NO

PLD_B needs 11 outputs!



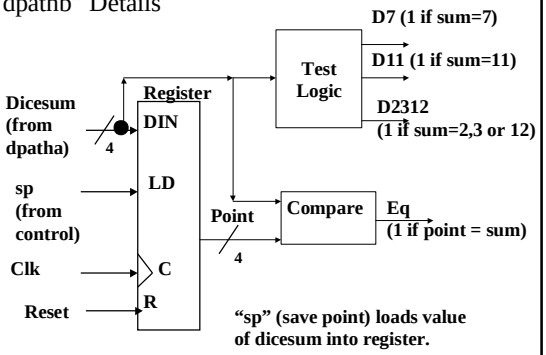
Three PLD Implementation



VHDL For Three PLD Solution

- **Up to this time, have been using VHDL to specify boolean equations.**
- **VHDL has high-level statements that allow more natural specification of a problem**
- **Example: What is the boolean equation for signal "D7"? (=1 when dicesum = 7).**
- **VHDL Boolean equation:**
  D7 <= (not dsum(3)) and dsum(2) and dsum(1) and dsum (0);
- **High level VHDL Statement:**
  D7 <= '1' when (dsum = "0111") else '0';

"dpathb" Details



VHDL for “dpathb”

Entity (input/outputs) declaration

```
entity dpathb is
  port (  clk,reset:  in std_logic;
        dicesum:    in std_logic_vector(3 downto 0);
        sp:         in std_logic;
        point:      out std_logic_vector(3 downto 0);
        eq:         out std_logic;
        d7_i:       out std_logic;
        d11_i:      out std_logic;
        d2312_i:    out std_logic
          );
end dpathb;
```


VHDL for “dpathb” (cont)

DFFs for point register

Note that ‘D’ is equal to dicesum when sp is asserted else keeps its same state - this is a register!!!

```
architecture a of dpathb is
 signal q,d: std_logic_vector (3 downto 0);
begin
   point <= q;      -- point register output

-- Flip flops for point register
  stateff: process (clk,reset)
   begin
   if (reset = '1') then
     q   <= "0000";
   elsif (clk'event and clk='1') then
     q <= d;
   end if;
  end process stateff;

  -- equations for D inputs of point register
d <= dicesum when (sp = '1') else q;
```


Boolean equations vs. High Level

I could have specified the boolean equations for each ‘d’ FF input of the point register as:

```
d(0) <= (sp and dicesum(0) )   or  ((not sp) and (q(0)) ;
d(1) <= (sp and dicesum(1) )   or  ((not sp) and (q(1)) ;
d(2) <= (sp and dicesum(2) )   or  ((not sp) and (q(2)) ;
d(3) <= (sp and dicesum(3) )   or  ((not sp) and (q(3)) ;
```

However, it is much easier (and clearer!) to simply write:

```
d <=  dicesum when (sp = ‘1’) else  q;
```

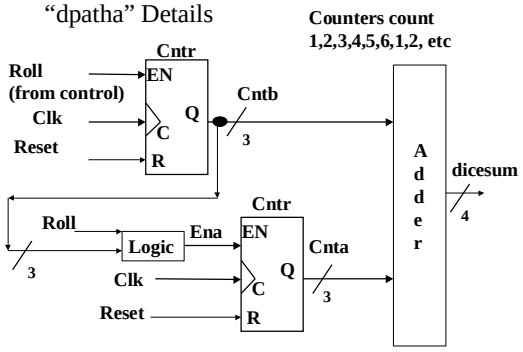
This statement DOES REPRESENT the above boolean equations; it is simply expressed differently.

VHDL for “dpathb” (cont)

```
-- other equations
  eq <= '1' when (dicesum = q) else '0';

  d7_i <= '1' when (dicesum = "0111") else '0';

  d11_i <= '1' when (dicesum = "1011") else '0';|

  d2312_i <= '1' when ( (dicesum = "0010") or
                        (dicesum = "0011") or
                        (dicesum = "1100") )
             else '0';

end a;
```

Again, we could have written boolean equations, but this is clearer.

“dpatha” Details



Ena = 1 when Cntb = 6 and Roll = 1

VHDL for dpatha

```
ntity dpatha is
  port (  clk,reset:  in std_logic;
          roll : in std_logic;
          dicesum: out std_logic_vector(3 downto 0);
          douta: out std_logic_vector(2 downto 0);
          doutb: out std_logic_vector(2 downto 0)
       );
```

```
architecture a of dpatha is
  signal cnta, cntb:  std_logic_vector(2 downto 0);
  signal en_a: std_logic;  -- enable for counter A
  signal sum: std_logic_vector(3 downto 0);
  signal c1,c2,c3: std_logic;  -- carry signals

 begin

  douta <= cnta;
  doutb <= cntb;
  dicesum <= sum;

  en_a <= '1' when ((cntb = "110") and (roll = '1'))
              else '0';

-- State Flip Flops
  stateff: process (clk,reset)
   begin
    if (reset = '1') then
     cnta <= "001";    -- initialize both counters to '1'
     cntb <= "001";
    elsif (clk'event and clk='1') then
```

```
if (roll = '1') then
    case cntb is
              when "001" =>    cntb <= "010";
              when "010" =>    cntb <= "011";
              when "011" =>    cntb <= "100";
              when "100" =>    cntb <= "101";
              when "101" =>    cntb <= "110";
              when "110" =>    cntb <= "001";
              when others  =>  cntb <= "001";
   end case;
   end if;

   if (en_a = '1') then
    case cnta is
              when "001" =>    cnta <= "010";
              when "010" =>    cnta <= "011";
              when "011" =>    cnta <= "100";
              when "100" =>    cnta <= "101";
              when "101" =>    cnta <= "110";
              when "110" =>    cnta <= "001";
              when others  =>  cnta <= "001";
   end case;
  end if;
 end if;
 end process stateff;
```



```
- sum equations to add cnta + cntb
  -- sum = a xor b xor ci
  -- cout = (a and b) or Ci(a or b)

  -- bit 0, no carry in
  sum(0) <= cnta(0) xor cntb(0);
  c1  <= cnta(0) and cntb(0);

  -- bit 1, c1 is carry in
  sum(1) <= cnta(1) xor cntb(1) xor c1;
  c2  <= (cnta(1) and cntb(1)) or (c1 and (cnta(1) or cntb(1)));

  -- bit 2, c2 is carry in
  sum(2) <= cnta(2) xor cntb(2) xor c2;
  c3  <= (cnta(2) and cntb(2)) or (c2 and (cnta(2) or cntb(2)));

  -- bit 3 is carry3 since no counter bits
  sum(3) <= c3;
end a;
```

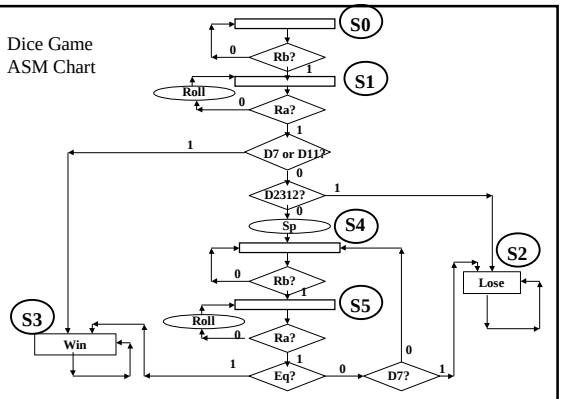
"dpatha" adder equations.

We could have simply written:

sum <= cnta + cntb;

This is a valid operation in VHDL!

Dice Game
ASM Chart



VHDL for control

```
entity control is
  port (  clk,reset:  in std_logic;
          d7_i:  in std_logic;
          d11_i:  in std_logic;
          d2312_i:  in std_logic;
          ra:     in std_logic;
          rb:     in std_logic;
          eq:    in std_logic;
          sp:   out std_logic;
          roll: out std_logic;
          win : out std_logic;
          lose: out std_logic;
          q0 : out std_logic;
          q1 : out std_logic;
          q4 : out std_logic;
          q5 : out std_logic
      );

end control;
```

Used one-hot encoding. Outputs q0, q1, lose, win,q4, q5 correspond to states S0, S1, S2, S3, S4, S5.

The one hot equations were written by inspection. The VHDL file contains boolean equations for DFF inputs and ra,rb, eq,sp outputs.


```
architecture a of control is
 -- FFs for Finite State Machine
   signal q, d : std_logic_vector(5 downto 0);
begin

-- State Flip Flops
 stateff: process (clk,reset)
  begin
  if (reset = '1') then
    q   <= "000001";
  elsif (clk'event and clk='1') then
    q <= d;
  end if;
 end process stateff;

-- FF equations
  d(0) <= q(0) and (not rb);

 d(1) <= (q(0) and rb) or  (q(1) and (not ra));

 d(2) <= q(2) or  (q(1) and ra and (d7_i or d11_i))
           or  (q(5) and ra and eq);

 d(3) <= q(3) or (q(1) and ra and (not d7_i) and
                  (not d11_i) and D2312_i)
           or (q(5) and ra and (not eq) and (d7_i)) ;

 d(4) <= (q(1) and ra and (not d7_i) and
            (not d11_i) and (not D2312_i))
           or (q(4) and (not rb))
           or (q(5) and ra and (not eq) and (not d7_i));

 d(5) <= (q(4) and rb) or  (q(5) and not ra);

 win <= q(2);
 lose <= q(3);

 q0 <= q(0);
 q1 <= q(1);
 q4 <= q(4);
 q5 <= q(5);
 sp <= q(1) and ra and (not d7_i) and
           (not d11_i) and (not d2312_i);

 roll <= (q(1) and (not ra)) or (q(5) and (not ra));
end a;
```


Is there an easier way to do VHDL for a FSM?

- There is an easier way to write the VHDL for the finite state machine code
- Will use a “case” statement for specifying the FSM action
  - Will generate the same boolean equations
  - Will be more readable
- Will also use symbolic names for states (S0, S1, etc)
  - Can change state encoding very easily.

```
architecture a of control_alt is

  -- FFs for Finite State Machine

  signal q, d : std_logic_vector(5 downto 0);
  constant S0: std_logic_vector(5 downto 0) := "000001";
  constant S1: std_logic_vector(5 downto 0) := "000010";
  constant S2: std_logic_vector(5 downto 0) := "000100";
  constant S3: std_logic_vector(5 downto 0) := "001000";
  constant S4: std_logic_vector(5 downto 0) := "010000";
  constant S5: std_logic_vector(5 downto 0) := "100000";

begin


-- State Flip Flops
  stateff: process (clk,reset)
   begin
   if (reset = '1') then
     q   <= S0;
   elsif (clk'event and clk='1') then
     q <= d;
   end if;
  end process stateff;
```

Will define symbolic names for the states.

To use new state encoding, only have to change definition of symbolic names!

This uses one-hot encoding.


```
-- q is present state, d is next state.
 clogic: process (q, ra, rb, d7_i, d11_i, d2312_i)
 begin

  -- defaults
  win <= '0'; lose <= '0';
  sp <= '0';  roll <= '0';
  d <= q;  -- default is to stay in same state.

  case q is
    when S0 => if (rb = '1') then  d <= S1; end if;
    when S1 =>
              if (ra = '1') then
                if (d7_i = '1' or d11_i = '1') then
                  d <= S3;
                elsif (d2312_i = '1') then
                  d <= S2;
                else
                  sp <= '1';
                  d <= S4;
                end if;
              else
                roll <= '1';
              end if;
    when S2 =>  lose <= '1';
    when S3 =>  win <= '1';
    when S4 =>  if (rb = '1') then d <= S5; end if;
    when S5 =>
              if (ra = '1') then
                if (eq = '1') then
                  d <= S3;
                elsif (d7_i = '1') then
                  d <= S2;
                else
                  d <= S4;
                end if;
              else
                roll <= '1';
              end if;
    when others =>  d <= S0;
  end case;
 end process clogic;
```



```
architecture a of control_alt is

  -- FFs for Finite State Machine

  signal q, d : std_logic_vector(2 downto 0);
  constant S0: std_logic_vector(2 downto 0) := "000";
  constant S1: std_logic_vector(2 downto 0) := "001";
  constant S2: std_logic_vector(2 downto 0) := "010";
  constant S3: std_logic_vector(2 downto 0) := "011";
  constant S4: std_logic_vector(2 downto 0) := "100";
  constant S5: std_logic_vector(2 downto 0) := "101";

begin


-- State Flip Flops
  stateff: process (clk,reset)
   begin
   if (reset = '1') then
     q   <= S0;
   elsif (clk'event and clk='1') then
     q <= d;
   end if;
  end process stateff;
```

Symbolic names changed to define a binary counting order encoding for States!

No other changes necessary to code!!!

Summary

- High level VHDL can let you describe digital systems easier and faster. These descriptions are more understandable to an external reader.
- Still MUST KNOW implications of a high level VHDL statement -- ie. What gates get generated?
  - Sum <= Cnta + Cntb; Easy to write, but what kind of adder gets synthesized? There are many different ways to build an adder, and each one has a different tradeoff in terms of speed and gate count!
- Take EE 4743/EE 6743 to find out more about Digital System design!