6-6

HDL for Registers and Counters

J.J. Shann

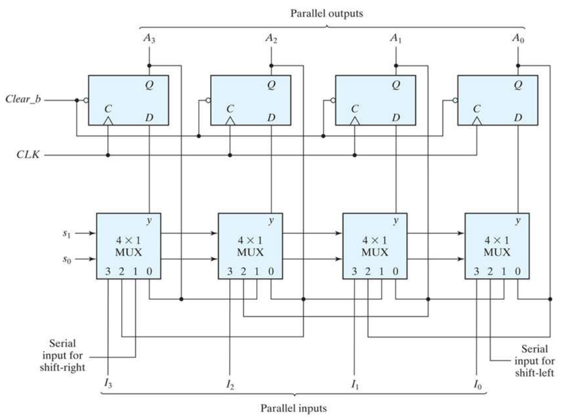
HDL Models of Registers and Counters

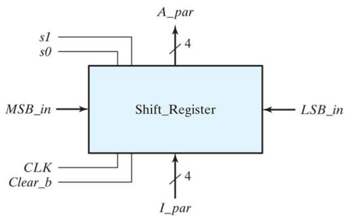
- HDL models of registers and counters:
  - Behavioral level: describes only the operations of the register, as prescribed by a function table, w/o a preconceived structure.
  - Structural level: shows the ckt in terms of a collection of components such as gates, flip-flops, and multiplexers.
    - The various components are instantiated to form a *hierarchical* description of the design.

* When a machine is complex, a *hierarchical* description creates a physical partition of the machine into simpler and more easily described units.

A. Shift Register

- Universal shift register:
  - p. 353, Fig 6.7, p.354, Table 6.3





Function table

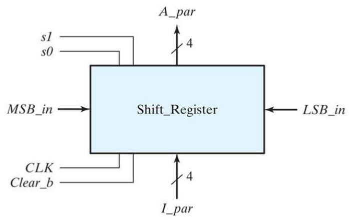
Mode Control		
s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

J.J. Shann HDL-126

■126

HDL Example 6.1: Universal Shift Register (Behavioral Model)

- HDL Example 6.1: 4-bit Universal Shift Register (Behavioral Model)-- p. 353, Fig 6.7(a), p.354, Table 6.3



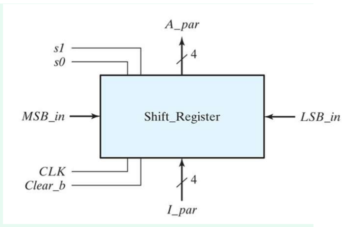
Mode Control		
s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

```
module Shift_Register_4_beh (             // V2001, 2005
   output  reg [3: 0]  A_par,             // Register output
   input       [3: 0]  I_par,             // Parallel input
   input               s1, s0,            // Select inputs
                       MSB_in, LSB_in,    // Serial inputs
                       CLK, Clear_b       // Clock and Clear_b
);
```

-127

■127

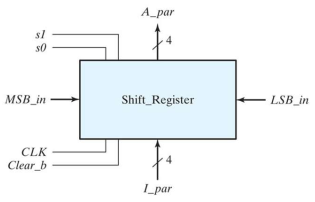
```
module Shift_Register_4_beh (
    output  reg [3: 0]  A_par,
    input       [3: 0]  I_par,
    input               s1, s0,
                        MSB_in, LSB_in,
                        CLK, Clear_b
);
    always @ (posedge CLK, negedge Clear_b)
      if (~Clear_b) A_par <= 4'b0000;
      else
        case ({s1, s0})
          2'b00: A_par <= A_par;    // No change
          2'b01: A_par <= {MSB_in, A_par[3: 1]};  // Shift right
          2'b10: A_par <= {A_par[2: 0], LSB_in};     // Shift left
          2'b11: A_par <= I_par;     // Parallel load of input
        endcase
endmodule
```

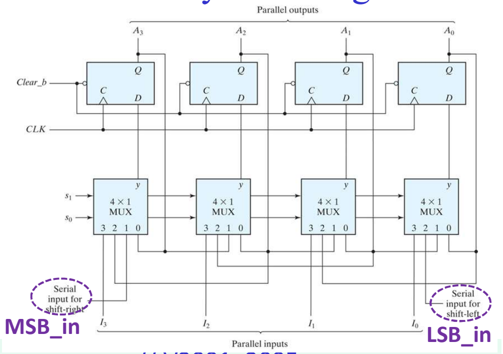


Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

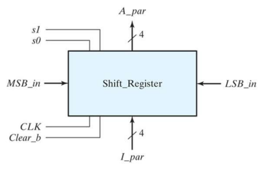
HDL Example 6.2: Universal Shift Register (Structural Model)

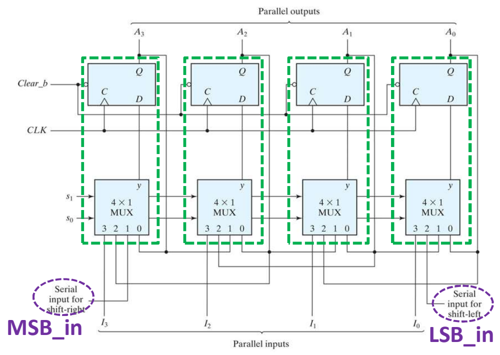
- HDL Example 6.2: 4-bit University Shift Register (Structural Model)
  - p. 353, Fig 6.7(a)(b)



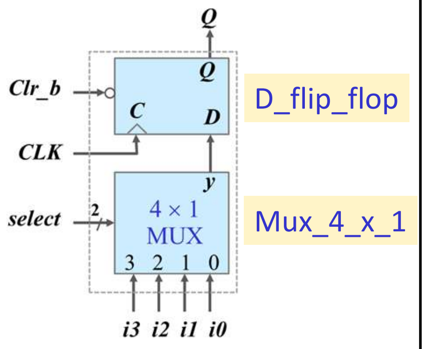


```
module Shift_Register_4_str (                          // V2001, 2005
    output   [3: 0]   A_par,                           // Parallel output
    input    [3: 0]   I_par,                           // Parallel input
    input             s1, s0,                          // Mode select
    input             MSB_in, LSB_in, CLK, Clear_b     // Serial inputs, clock, clear
);
```

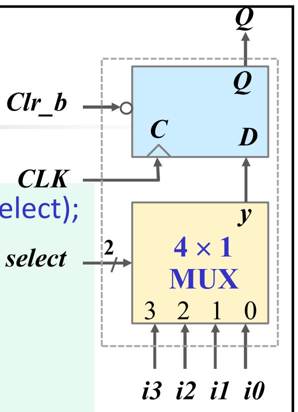




Shift_Register_4_str



stage

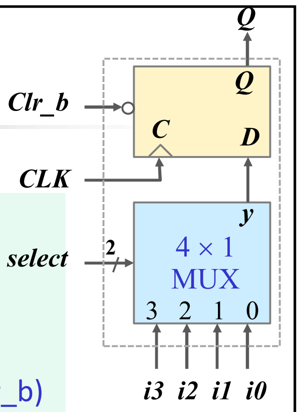


— 4x1 multiplexer (behavioral model):

```
module Mux_4_x_1 (mux_out, i0, i1, i2, i3, select);
   output          mux_out;
   input           i0, i1, i2, i3;
   input    [1: 0] select;
   reg             mux_out;

   always @ (select, i0, i1, i2, i3)
      case (select)
         2'b00:    mux_out = i0;
         2'b01:    mux_out = i1;
         2'b10:    mux_out = i2;
         2'b11:    mux_out = i3;
      endcase
endmodule
```

– D flip-flop (behavioral model):

```
module D_flip_flop (Q, D, CLK, Clr_b);
   output   Q;
   input    D, CLK, Clr_b;
   reg      Q;

   always @ (posedge CLK or negedge Clr_b)
      if (!Clr_b) Q <= 1'b0; else Q <= D;
endmodule
```

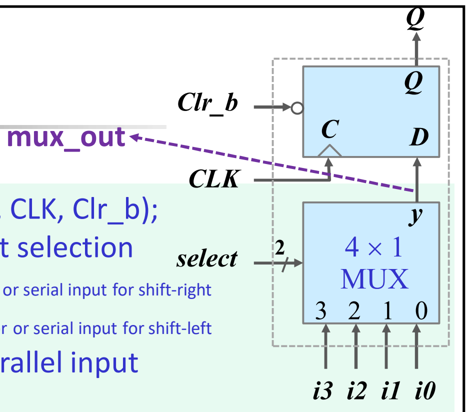


J.J. Shann HDL-132

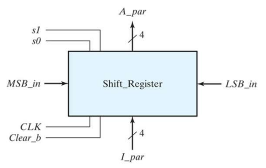
■132

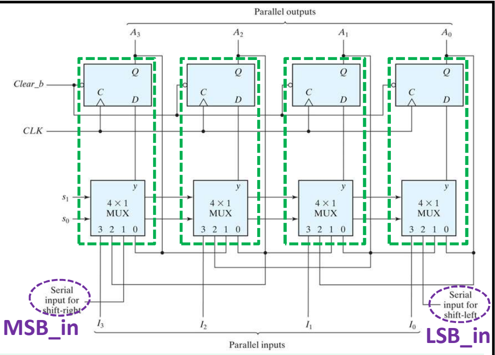
– 1-stage of shift register:

```
module stage (i0, i1, i2, i3, Q, select, CLK, Clr_b);
   input              i0,  //circulation bit selection
                      i1,  //data from left neighbor or serial input for shift-right
                      i2,  //data from right neighbor or serial input for shift-left
                      i3;  //data from parallel input
   output             Q;
   input     [1: 0]   select;       //stage mode control bus
   input              CLK, Clr_b;   //Clock, Clear for flip-flops
   wire               mux_out;

// instantiate mux and flip-flop
  Mux_4_x_1 M0 (mux_out, i0, i1, i2, i3, select);
  D_flip_flop  M1 (Q, mux_out, CLK, Clr_b);
endmodule
```



■133





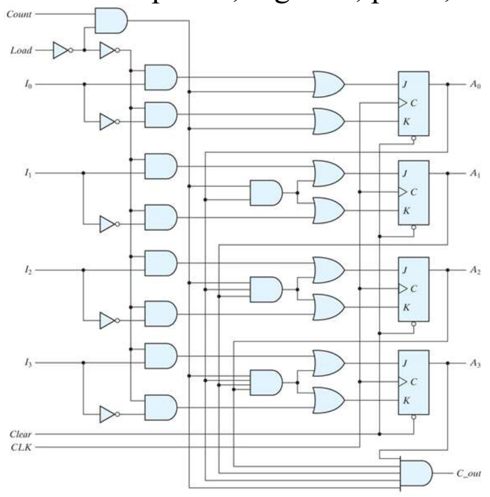
```
module Shift_Register_4_str (
    output    [3: 0]   A_par,
    input     [3: 0]   I_par,
    input              s1, s0,
    input              MSB_in, LSB_in, CLK, Clear_b      // Serial inputs, clock, clear
);

// bus for mode control
    wire  [1:0]     select = {s1, s0};

// Instantiate the four stages
    stage ST0 (A_par[0], A_par[1], LSB_in, I_par[0], A_par[0], select, CLK, Clear_b);
    stage ST1 (A_par[1], A_par[2], A_par[0], I_par[1],A_par[1], select, CLK, Clear_b);
    stage ST2 (A_par[2], A_par[3], A_par[1], I_par[2], A_par[2], select, CLK, Clear_b);
    stage ST3 (A_par[3], MSB_in, A_par[2], I_par[3], A_par[3], select, CLK, Clear_b);
endmodule
```

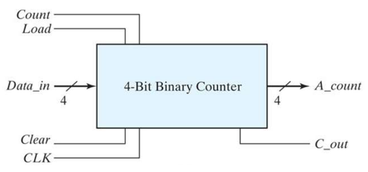
```
module stage (i0, i1, i2, i3, Q, select, CLK, Clr_b);
  input           i0, i1, i2, i3;
  output          Q;
  input   [1: 0]  select;
  input           CLK, Clr_b;
  wire            mux_out;

// instantiate mux and flip-flop
 Mux_4_x_1 M0 (mux_out, i0, i1, i2, i3, select);
 D_flip_flop  M1 (Q, mux_out, CLK, Clr_b);
endmodule
```

J.J. Shann HDL-134

B. Synchronous Counter

- Synchronous counter:
  - p. 365, Fig 6.14; p.366, Table 6.6



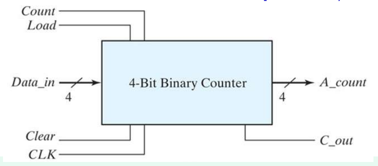


Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

J.J. Shann HDL-135

HDL Example 6.3: Synchronous Counter

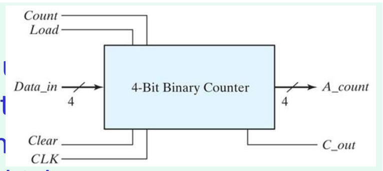
- HDL Example 6.3: 4-bit Binary Counter with Parallel Load-- p.366, Table 6.6, p. 365, Fig 6.14



Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

```
module Binary_Counter_4_Par_Load (
   output  reg  [3:0]  A_count,     // Data output
   output              C_out,       // Output carry
   input  [3:0]        Data_in,     // Data input
   input               Count,       // Active high to count
                       Load,        // Active high to load
                       CLK,         // Positive edge sensitive
                       Clear_b      // Active low
);
```

```
module Binary_Counter_4_Par_Load (
   output  reg  [3:0]  A_count,     // Data ou
   output              C_out,       // Output
   input  [3:0]        Data_in,     // Data in
   input               Count,       // Active high to count
                       Load,
                       CLK,
                       Clear_b
);
```

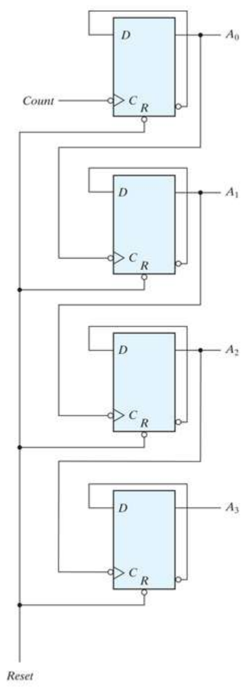


Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

```
   assign C_out = Count & (~Load) & (A_count == 4'b1111);
   always @ (posedge CLK, negedge Clear_b)
      if (~Clear_b)        A_count <= 4'b0000;
      else if (Load)       A_count <= Data_in;
      else if (Count)      A_count <= A_count + 1'b1;
      else                 A_count <= A_count;    // redundant statement
endmodule
```

C. Ripple Counter

- Ripple counter:
  - p.356, Fig 6.8(b)



Summary of HDL

- HDLs are extremely important tools for modern digital designers.
- HDLs are used for both *simulation* and *synthesis*.
- **Writing HDL code ⇒ describing *real hardware*!**
  1. Sketch a *block diagram* of your system.
  2. Identify which portions are *combinational logic* and which portions are *sequential circuits or FSMs*.
  3. Write HDL code for each portion, using the correct idioms to imply the kind of hardware needed.

Guidelines of Blocking and Nonblocking Assignments (p. HDL-94)

- Use *continuous assignments* **assign** to model simple ***combinational*** logic.
- Use **always @ (*)** and *blocking assignments* (=) to model more complicated ***combinational*** logic where the **always** statement is helpful.
- Use **always @ (posedge clk)** and *nonblocking assignments* (<=) to model ***synchronous sequential*** logic.
- Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

▪144