

Roadmap

Part 1: Modeling, Verification and Synthesis of Combinational Logic

- Design Flow for HDL-Based Design Methodology
- Modeling, Verification, and Synthesis of Combinational Logic
- Structural decomposition and top-down design
- Verilog language constructs
- Test plan, test bench, simulation, and verification

Part 2: Modeling, Verification, and Synthesis of Sequential Logic

- RTL data flow constructs and operators
- Control flow constructs
- Sequential (=) vs. concurrent (<=) assignments

• Part 3: Synchronous Finite State Machines and Datapath Controllers

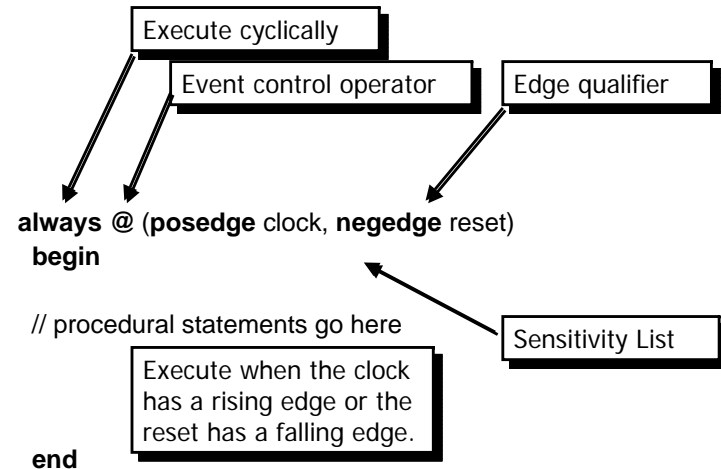
- Mealy, Moore machines
- Behavioral modeling for latch-free synthesis
- Race-Free Synthesis of Datapath Controllers

CSE/EE 40462 Sequential Logic.1

Copyright 2006 Michael D. Ciletti, ND - 2006

Modeling Synchronous Logic with Cyclic Behaviors

- Use edge-sensitive cyclic behaviors to model flip-flops and sequential logic.



CSE/EE 40462 Sequential Logic.2

Copyright 2006 Michael D. Ciletti, ND - 2006

Modeling Synchronous Logic (1 of 2)

Example: D Flip-Flop with asynchronous set / reset

```

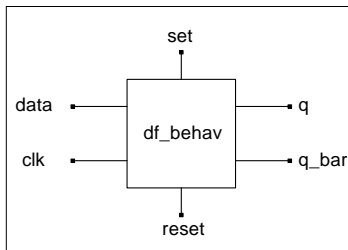
module df_behav (input data, set, clk, reset, output reg q, output q_bar
);
    
```

How does a synthesis tool identify the synchronizing signal?

```

    assign q_bar = ~ q;
    always @ (posedge clk, negedge reset, negedge set)
    begin
        if (reset == 0) q <= 0;
        else if (set == 0) q <= 1;
        else q <= data;
    end
endmodule
    
```

Verilog
2001



Non-blocking assignment operator

CSE/EE 40462 Sequential Logic.3

Copyright 2006 Michael D. Ciletti, ND - 2006

Modeling Synchronous Logic (2 of 2)

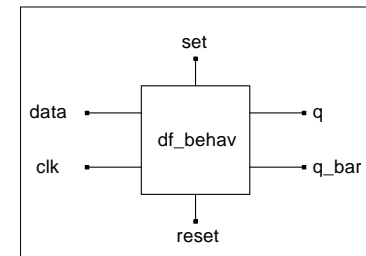
Example: D flip-flop with synchronous set / reset

```

module df_behav (input data, set, clk, reset, output reg q, output q_bar
);
    
```

```

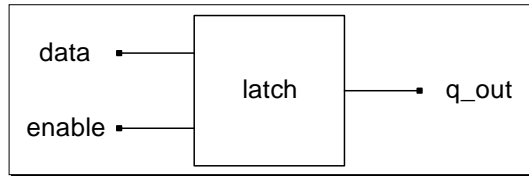
    assign q_bar = ~ q;
    always @ (posedge clk) // Flip-flop with synchronous set/reset
    begin
        if (reset == 0) q <= 0;
        else if (set == 0) q <= 1;
        else q <= data;
    end
endmodule
    
```



CSE/EE 40462 Sequential Logic.4

Copyright 2006 Michael D. Ciletti, ND - 2006

Transparent Latch (Cyclic Behavior)



```

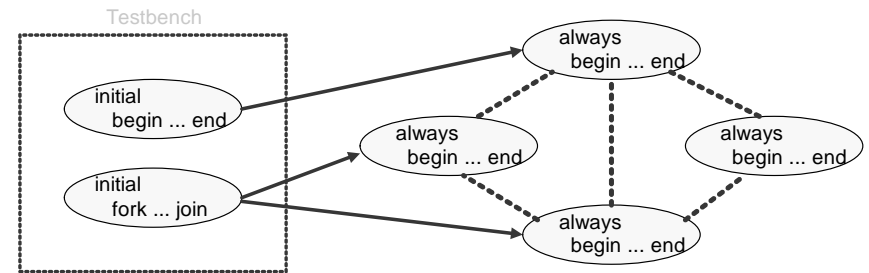
module tr_latch (q_out, enable, data);
output    q_out;
input    enable, data;
reg      q_out;
always @ (enable, data)    // always @ (enable or data)

begin
    if (enable) q_out <= data;
end
endmodule
    
```

Preferred model for synthesis of a latch

Concurrent Behaviors

Cyclic and single-pass behaviors execute concurrently with each other and with continuous assignments and primitives.



• A module may contain multiple, concurrently interacting behaviors!!!

Do not attempt to nest behaviors.

Verilog Models of Combinational Logic

Descriptive Options

Verilog Options

- Logic Gates → • Primitives
- Truth Tables → • User-Defined Primitives (UDPs)
- Boolean Equations → • Continuous Assignments

Structural

Behavioral

RTL / Dataflow

• Cyclic Behaviors

Algorithm

$R1 \leftarrow R1 + R2$ Add content of $R2$ to $R1$
 $R3 \leftarrow R3 + 1$ Increment $R3$ by 1 (count up)
 $R4 \leftarrow \text{shr } R4$ Shift right $R4$
 $R5 \leftarrow 0$ Clear $R5$ to 0

Dataflow / RTL Behavioral Modeling

- Dataflow / RTL (register transfer level) models of combinational logic describe concurrent operations on datapath signals, usually in a synchronous machine.
- Register operations with language operators
- Assignment of value to variables
- Control flow constructs

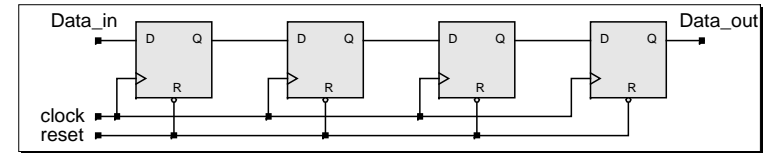
Verilog Operators

Precedence	Symbol	Operator
Highest	+ - ! ~	Unary
	**	Exponentiation
	* / %	Multiply, divide, modulus
	+ -	Add, subtract
	<< >> <<< >>>	Shift
	< <= > >=	Relational
	== != === !==	Equality
	& ~&	Binary, Reduction
	^ ~^ ~^	
	~	Logical
Lowest	&&	
		Conditional

CSE/EE 40462 Sequential Logic.9

Copyright 2006 Michael D. Ciletti, ND - 2006

RTL Example: shift Register



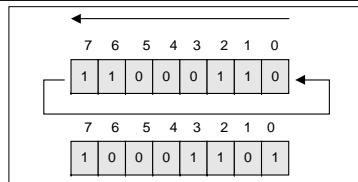
```

module Shift_reg4 (output Data_out, input Data_in, clock, reset);
reg [3: 0] Data_reg;
assign Data_out = Data_reg[0];
always @ (negedge reset or posedge clock)
begin
    if (reset == 1'b0) Data_reg <= 4'b0;
    else Data_reg <= {Data_in, Data_reg[3:1]};
end
endmodule
    
```

CSE/EE 40462 Sequential Logic.10

Copyright 2006 Michael D. Ciletti, ND - 2006

RTL Example: Simple Barrel Shifter



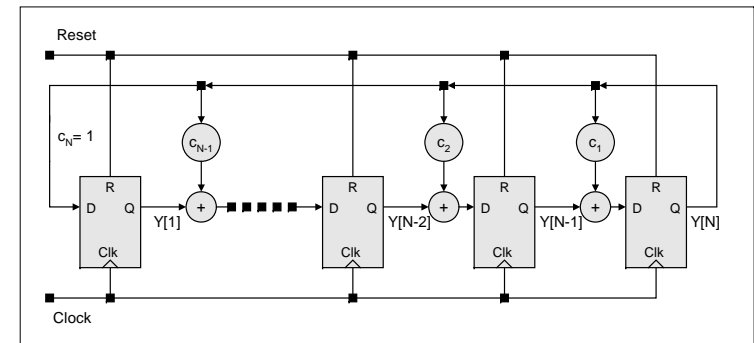
```

module barrel_shifter (output reg [7: 0] Data_out,
    input [7: 0] Data_in, input load, clock, reset
);
always @ (posedge reset or posedge clock)
begin
    if (reset == 1'b1) Data_out <= 8'b0;
    else if (load == 1'b1) Data_out <= Data_in;
    else Data_out <= {Data_out [6: 0], Data_out [7]};
end
endmodule
    
```

CSE/EE 40462 Sequential Logic.11

Copyright 2006 Michael D. Ciletti, ND - 2006

RTL Example: LFSR (1 of 3)



CSE/EE 40462 Sequential Logic.12

Copyright 2006 Michael D. Ciletti, ND - 2006

RTL Example: LFSR (2 of 3)

```

module Auto_LFSR_RTL # (parameter Length = 8,
    initial_state = 8'b1001_0001, // 91h
    [1: Length] Tap_Coefficient = 8'b_1111_0011)
input Clock, Reset, output reg [1: Length] Y;
  
```

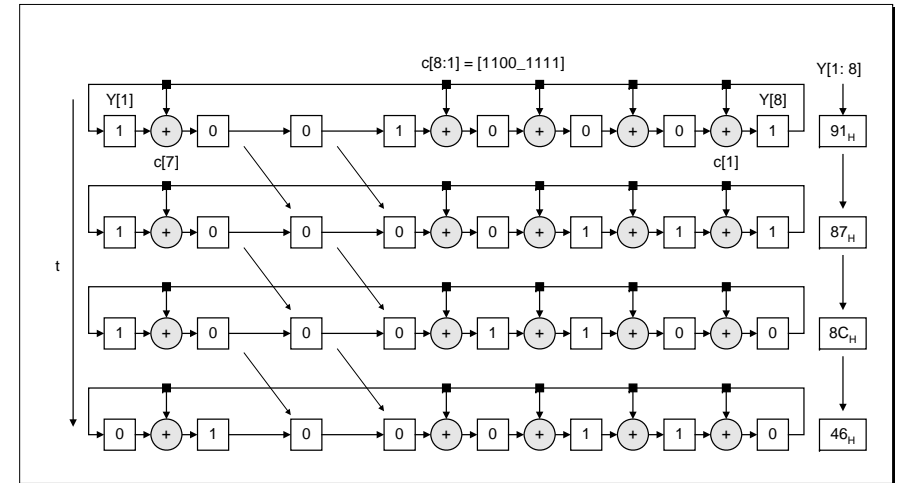
```

always @ (posedge Clock) if (!Reset) Y <= initial_state;
  else begin
    Y[1] <= Y[8];
    Y[2] <= Tap_Coefficient[7] ? Y[1] ^ Y[8] : Y[1];
    Y[3] <= Tap_Coefficient[6] ? Y[2] ^ Y[8] : Y[2];
    Y[4] <= Tap_Coefficient[5] ? Y[3] ^ Y[8] : Y[3];
    Y[5] <= Tap_Coefficient[4] ? Y[4] ^ Y[8] : Y[4];
    Y[6] <= Tap_Coefficient[3] ? Y[5] ^ Y[8] : Y[5];
    Y[7] <= Tap_Coefficient[2] ? Y[6] ^ Y[8] : Y[6];
    Y[8] <= Tap_Coefficient[1] ? Y[7] ^ Y[8] : Y[7];
  end
endmodule
  
```

CSE/EE 40462 Sequential Logic.13

Copyright 2006 Michael D. Ciletti, ND - 2006

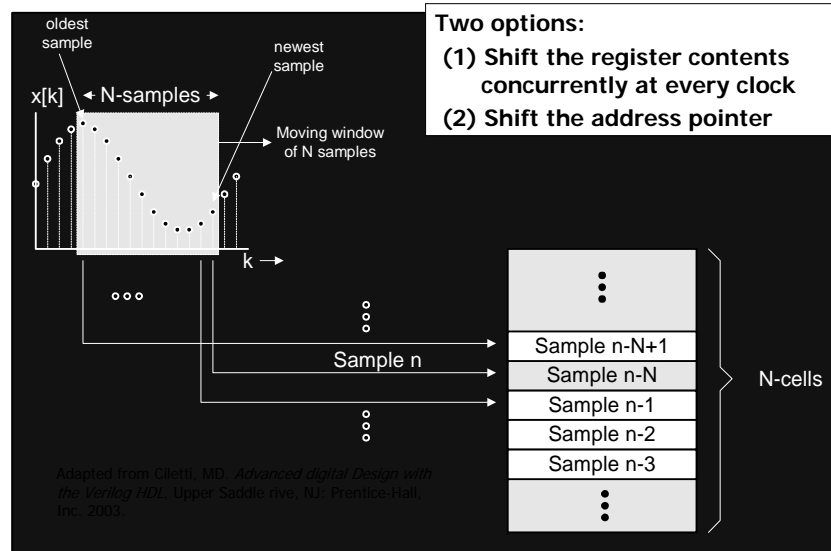
RTL Example: LFSR (3 of 3)



CSE/EE 40462 Sequential Logic.14

Copyright 2006 Michael D. Ciletti, ND - 2006

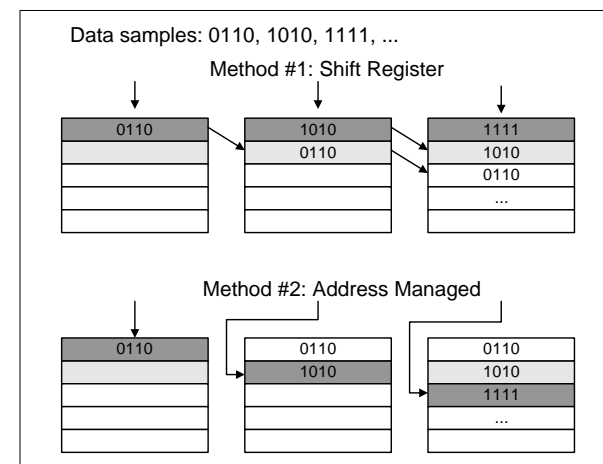
RTL Example: Circular Buffer



CSE/EE 40462 Sequential Logic.15

Copyright 2006 Michael D. Ciletti, ND - 2006

Data Movement



CSE/EE 40462 Sequential Logic.16

Copyright 2006 Michael D. Ciletti, ND - 2006

Circular Buffer #1 (Shift Register)

```

module Circular_Buffer_1 (cell_3, cell_2, cell_1, cell_0, Data_in, clock, reset);
parameter buff_size = 4;
parameter word_size = 8;
output [word_size -1: 0] cell_3, cell_2, cell_1, cell_0;
input [word_size -1: 0] Data_in;
input clock, reset;
reg [word_size -1: 0] Buff_Array [buff_size -1: 0];
assign cell_3 = Buff_Array[3], cell_2 = Buff_Array[2];
assign cell_1 = Buff_Array[1], cell_0 = Buff_Array[0];
integer k;

always @ (posedge clock) begin
  if (reset == 1) for (k = 0; k <= buff_size -1; k = k+1)
    Buff_Array[k] <= 0;
  else for (k = 1; k <= buff_size -1; k = k+1) begin
    Buff_Array[k] <= Buff_Array[k-1];
  end
  Buff_Array[0] <= Data_in;
end
endmodule

```

CSE/EE 40462 Sequential Logic.17

Copyright 2006 Michael D. Ciletti, ND - 2006

Circular Buffer #2 (Address managed)

```

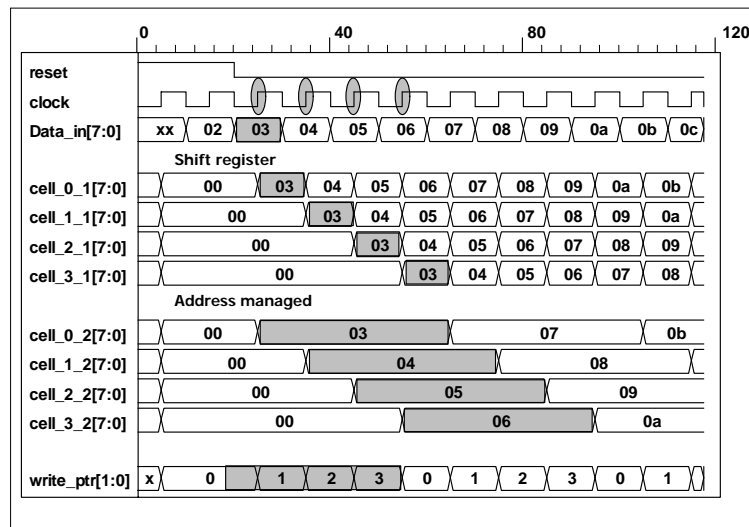
module Circular_Buffer_2 (cell_3, cell_2, cell_1, cell_0, Data_in, clock, reset);
parameter buff_size = 4, word_size = 8;
output [word_size -1: 0] cell_3, cell_2, cell_1, cell_0;
input [word_size -1: 0] Data_in;
input clock, reset;
reg [word_size -1: 0] Buff_Array [buff_size -1: 0];
assign cell_3 = Buff_Array[3], cell_2 = Buff_Array[2];
assign cell_1 = Buff_Array[1], cell_0 = Buff_Array[0];
integer k;
parameter write_ptr_width = 2; // Width of write pointer
parameter max_write_ptr = 3;
reg [write_ptr_width -1 : 0] write_ptr; // Pointer for writing
always @ (posedge clock)
  if (reset == 1) begin write_ptr <= 0;
    for (k = 0; k <= buff_size -1; k = k+1) Buff_Array[k] <= 0; end
  else begin Buff_Array[write_ptr] <= Data_in;
    if (write_ptr < max_write_ptr) write_ptr <= write_ptr + 1; else write_ptr <= 0;
  end
endmodule

```

CSE/EE 40462 Sequential Logic.18

Copyright 2006 Michael D. Ciletti, ND - 2006

Circular Buffer: Simulation Results



CSE/EE 40462 Sequential Logic.19

Copyright 2006 Michael D. Ciletti, ND - 2006

Mixed Models (1 of 2)

- *Structural and behavioral models may be mixed within the same description.*

```

module Add_mix_16 (sum, c_out, a, b, c_in);
output sum, c_out;
input a, b, c_in;
wire [15:0] sum, a, b;
wire c_in, c_in4, c_in8, c_in12, c_out;

  Add_rca_4 M1 (sum[3:0], c_in4, a[3:0], b[3:0], c_in);
  Add_rca_4 M2 (sum[7:4], c_in8, a[7:4], b[7:4], c_in4);
  Adder_4_RTL M3 (sum[11:8], c_in12, a[11:8], b[11:8], c_in8);
  Adder_4_RTL M4 (sum[15:12], c_out, a[15:12], b[15:12], c_in12);
endmodule

```

CSE/EE 40462 Sequential Logic.20

Copyright 2006 Michael D. Ciletti, ND - 2006

Concurrent vs. Sequential Assignment

- The order of execution of procedural statements in a cyclic behavior may depend on the order in which the statements are listed
- A blocked procedural assignment (=) cannot execute until the previous statement executes
- Expression substitution is recognized by synthesis tools

Example: Sequential Assignment (1 of 2)

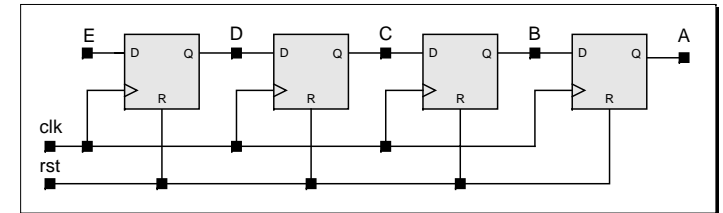
```

module shiftreg_PA (E, A, clk, rst);
output A;
input E;
input clk, rst;
reg A, B, C, D;
    
```

```

always @ (posedge clk or posedge rst) begin
  if (reset) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
    A = B;
    B = C;
    C = D;
    D = E;
  end
end
endmodule
    
```

Synthesis Result:



Example: Sequential Assignment (2 of 2)

```

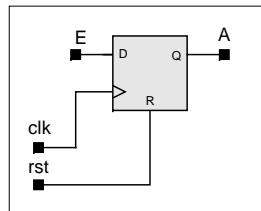
module shiftreg_PA (E, A, clk, rst);
output A;
input E;
input clk, rst;
reg A, B, C, D;
    
```

Reverse the order of the assignments:

```

always @ (posedge clk or posedge rst) begin
  if (reset) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
    D = E;
    C = D;
    B = C;
    A = B;
  end
end
endmodule
    
```

Synthesis Result:



Concurrent Assignments (Non-Blocking)

- Nonblocking assignment (<=) statements execute *concurrently* (in parallel) rather than sequentially
- The order in which nonblocking assignments are listed has no effect.
- Mechanism: the RHS of the assignments are sampled, then assignments are updated
- Assignments are based on values held by RHS before the statements execute
- Result: No dependency between statements

Example: Concurrent Assignment

```
module shiftreg_nb (A, E, clk, rst);
  output      A;
  input       E;
  input       clk, rst;
  reg         A, B, C, D;
```

```
always @ (posedge clk or posedge rst) begin
  if (rst) begin A <= 0; B <= 0; C <= 0; D <= 0; end
  else begin
    A <= B;           //      D <= E;
    B <= C;           //      C <= D;
    C <= D;           //      B <= D;
    D <= E;           //      A <= B;
  end
end
endmodule
```

The result of synthesis is independent of the order in which the nonblocking statements are listed.

Loops in Verilog

```
• repeat loop:
...
word_address = 0;
repeat (memory_size)
  begin
    memory [ word_address] = 0;
    word_address = word_address + 1;
  end
• for loop
reg [15: 0] demo_register;
integer K;
...
for (K = 4; K; K = K - 1)
  begin
    demo_register [K + 10] = 0;
    demo_register [K + 2] = 1;
  end
...
• while loop
begin: count_of_1s
  reg [7: 0] temp_reg;
  count = 0;
  temp_reg = reg_a;
  while (temp_reg)
    begin
      if (temp_reg[0]) count = count + 1;
      temp_reg = temp_reg >> 1;
    end
end
• forever loop
forever #5 clock = ~ clock;
```

Memories

Memories are declared as a **reg** type with an appended word range.

Example:

```
...
parameter word_size = 32, mem_size = 1024;
reg [31: 0] memory [0: 1023];
reg [9: 0] word_address;
...
word_address = 0;
repeat (memory_size)
  begin
    memory [word_address] = 0;
    word_address = word_address + 1;
  end
end
...
```

Verilog Models: Combinational and Sequential Logic

Descriptive Options

Verilog Options

- Logic Gates → • Primitives
- Truth Tables → • User-Defined Primitives (UDPs)
- Boolean Equations → • Continuous Assignments

- Cyclic Behaviors

RTL / Dataflow

Algorithm

Structural

Behavioral

Algorithmic Behavioral Modeling (1 of 3)

```

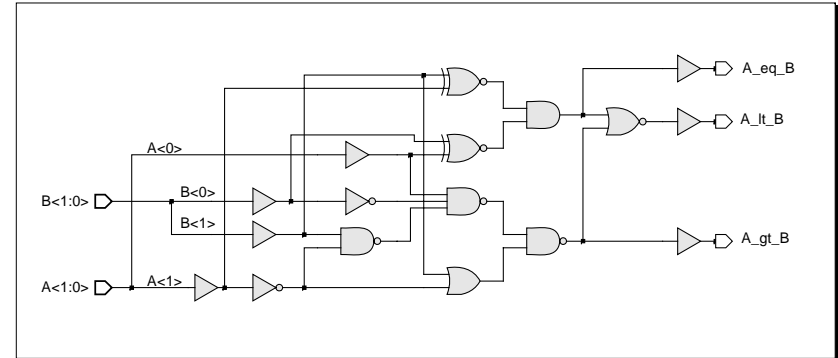
module compare_2_algo (A_lt_B, A_gt_B, A_eq_B, A, B);
  output    A_lt_B, A_gt_B, A_eq_B;
  input [1: 0]  A, B;

  reg      A_lt_B, A_gt_B, A_eq_B;

  always @ (A, B)    // Level-sensitive behavior
  begin
    A_lt_B = 0;
    A_gt_B = 0;
    A_eq_B = 0;
    if (A == B)      A_eq_B = 1;
    else if (A > B)  A_gt_B = 1;
    else             A_lt_B = 1;
  end
endmodule

```

Algorithmic Behavioral Modeling (3 of 3)



Algorithmic Behavioral Modeling (2 of 3)

```

module Auto_LFSR_RTL # (parameter Length = 8,
  initial_state = 8'b1001_0001, // 91h
  [1: Length] Tap_Coefficient = 8'b_1111_0011),
  input Clock, Reset, output reg [1: Length] Y;
  integer Cell_ptr;

  always @ (posedge Clock)
  begin
    if (Reset == 0) Y <= initial_state; // Arbitrary initial state, 91h
    else begin for (Cell_ptr = 2; Cell_ptr <= Length; Cell_ptr = Cell_ptr + 1)
      if (Tap_Coefficient [Length - Cell_ptr + 1] == 1)
        Y[Cell_ptr] <= Y[Cell_ptr - 1] ^ Y [Length];
      else
        Y[Cell_ptr] <= Y[Cell_ptr - 1];
        Y[1] <= Y[Length];
    end
  end
endmodule

```

Multi-Cycle Operations (1 of 2)

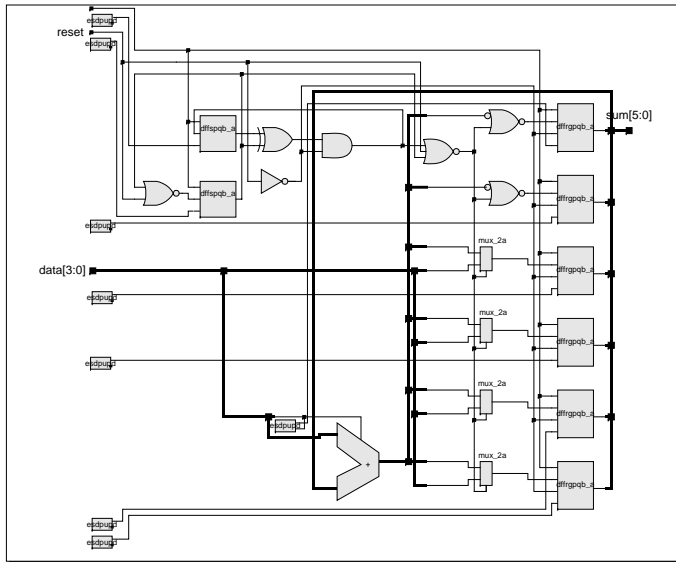
```

module add_4cycle (output reg [5: 0] sum, input [3: 0] data,
  input clk, reset);

  always @ (posedge clk) begin: add_loop
    if (reset) disable add_loop; else sum <= data;
    @ (posedge clk) if (reset) disable add_loop; else sum <= sum + data;
    @ (posedge clk) if (reset) disable add_loop; else sum <= sum + data;
    @ (posedge clk) if (reset) disable add_loop; else sum <= sum + data;
  end
endmodule

```

Multi-Cycle Operations (2 of 2)



CSE/EE 40462 Sequential Logic.33

Copyright 2006 Michael D. Ciletti, ND - 2006

Tasks (1 of 2)

```
module adder_task (output reg [3: 0] sum, output reg c_out, input
  clk, reset, c_in, input [3: 0] data_a, data_b);
```

```
  always @ (posedge clk, posedge reset)
    if (reset) {c_out, sum} <= 0;
    else add_values (c_out, sum, data_a, data_b, c_in);
```

```
  task add_values (output reg c_out, output reg [3: 0] sum,
    input [3: 0] data_a, data_b, input c_in);
    begin
      {c_out, sum} <= data_a + (data_b + c_in);
    end
  endtask
endmodule
```

CSE/EE 40462 Sequential Logic.34

Copyright 2006 Michael D. Ciletti, ND - 2006

Functions (1 of 2)

```
module arithmetic_unit (output [4: 0] result_1, output [3: 0] result_2,
  input [3: 0] operand_1, operand_2);
```

```
  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = largest_operand (operand_1, operand_2);
```

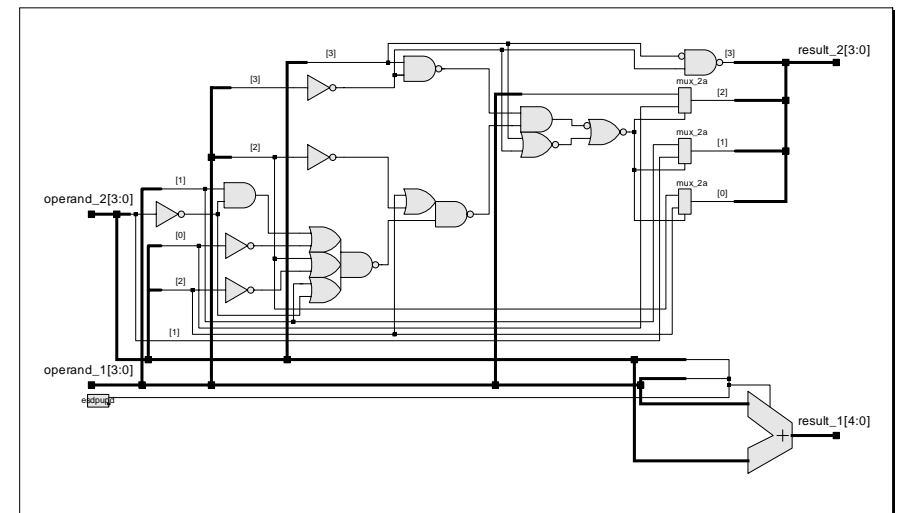
```
  function [4: 0] sum_of_operands (input [3: 0] operand_1,
    operand_2);
    sum_of_operands = operand_1 + operand_2;
  endfunction
```

```
  function [3: 0] largest_operand (input [3: 0] operand_1, operand_2);
    largest_operand = (operand_1 >= operand_2) ? operand_1 :
    operand_2;
  endfunction
endmodule
```

CSE/EE 40462 Sequential Logic.35

Copyright 2006 Michael D. Ciletti, ND - 2006

Functions (2 of 2)



CSE/EE 40462 Sequential Logic.36

Copyright 2006 Michael D. Ciletti, ND - 2006