Register Transfer Level Design

Chapter 8

Steve Oldridge
Dr. Sidney Fels

Topics

- RTL Notation
- RTL in HDL
- Algorithmic State Machines
- Sequential Binary Multiplier
  - Control Logic
  - HDL
- Design with Multiplexors
- Race-Free Design
- Latch-Free Design

Register Transfer Level Notation

- A Circuit is described as:
  - The set of registers in the system
  - Operations performed on the data in those registers
  - Control that supervises the sequence of operations

RTL Notation Examples

- R2 ← R1
- If (T1 = 1) then R2 ← R1
- If (T3 = 1) then (R2 ← R1, R1 ← R2)
- R1 ← R1 + R2
- R3 ← R3 + 1
- R4 ← shr R4
- R5 ← 0

RTL in HDL

- Assign S = A + B
- Always @(A,B)
```
S = A+B;
```
- Always @(negedge clock)
```
Begin
 RA = RA + RB;
 RD = RA;
End
```
- Always @(negedge clock)
```
Begin
 RA <= RA + RB;
 RD <= RA;
End
```

Arithmetic Operators

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	%	modulus
	**	exponentiation

Logic Operators

Logic	~	negation (complement)
(bitwise	&	AND
or	|	OR
reduction)	^	exclusive-OR (XOR)
Logical	!	negation
	&&	AND
	||	OR

Shift Operators

Shift	>>	logical right shift
	<<	logical left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
	{ , }	concatenation

Relational Operators

Relational	>	greater than
	<	less than
	==	equality
	!=	inequality
	===	case equality
	!==	case inequality
	>=	greater than or equal
	<=	less than or equal

Order of Operations

<table>
<tr><td>+ − ! ~ & ~ & | ~| ^ ~^ ^~ (unary)</td><td>Highest precedence</td></tr>
<tr><td>**</td><td rowspan="12">↓</td></tr>
<tr><td>*/%</td></tr>
<tr><td>+− (binary)</td></tr>
<tr><td><< >> <<< >>></td></tr>
<tr><td>< <= > >=</td></tr>
<tr><td>== != === !==</td></tr>
<tr><td>& (binary)</td></tr>
<tr><td>^ ^~ ~^ (binary)</td></tr>
<tr><td>| (binary)</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?: (conditional operator)</td></tr>
<tr><td>{} {{}}</td><td>Lowest precedence</td></tr>
</table>

Looping

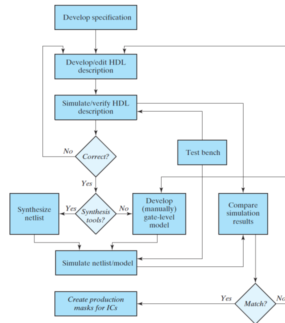
- Repeat (n)
- Forever
- While (condition)
- For (k=0; k<=3; k++)

Looping Example

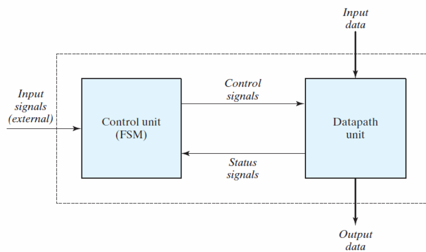
- Initial

```
Begin
  clock = 1'b0;
  repeat (16)    // creates an 8 cycle clock
    #5 clock = ~clock;
  end;
```

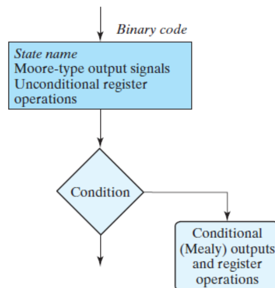
Logic Synthesis Process



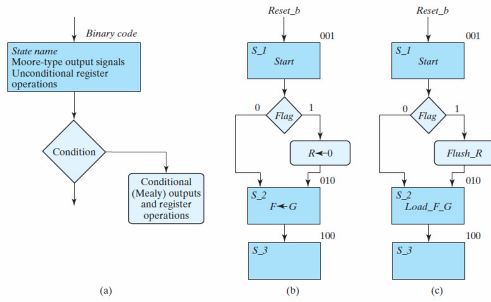
Algorithmic State Machines



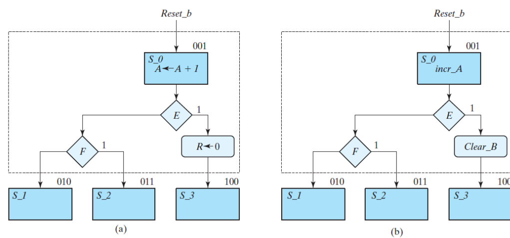
ASM Charts



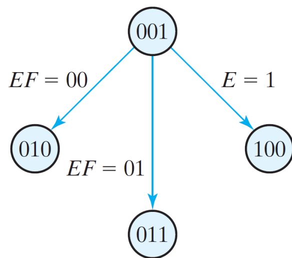
ASM Chart Example



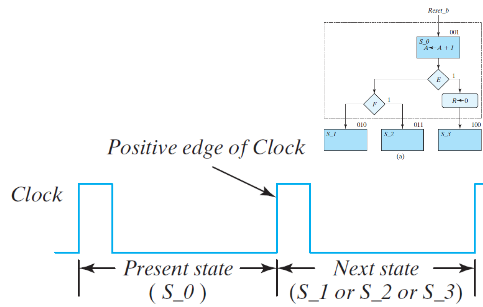
ASM Block



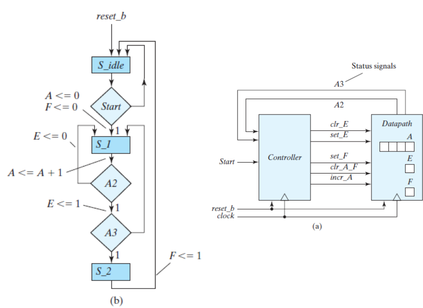
Simplifications of ASM



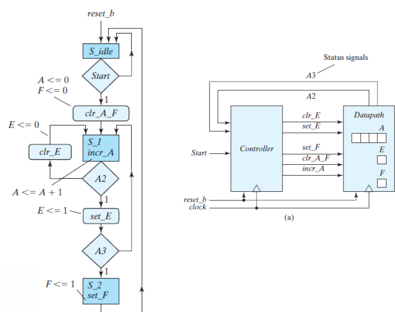
Timing Considerations



Algorithmic State Machine and Datapath Chart (ASMD)



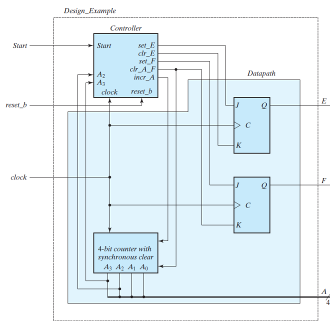
ASMD Example



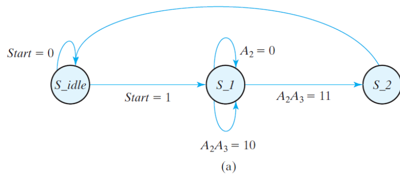
Timing Sequence

Counter				Flip-Flops			
A_3	A_2	A_1	A_0	E	F	Conditions	State
0	0	0	0	1	0	$A_2 = 0, A_3 = 0$	*S_1*
0	0	0	1	0	0		
0	0	1	0	0	0		
0	0	1	1	0	0		
0	1	0	0	0	0	$A_2 = 1, A_3 = 0$	
0	1	0	1	1	0		
0	1	1	0	1	0		
0	1	1	1	1	0		
1	0	0	0	1	0	$A_2 = 0, A_3 = 1$	
1	0	0	1	0	0		
1	0	1	0	0	0		
1	0	1	1	0	0		
1	1	0	0	0	0	$A_2 = 1, A_3 = 1$	
1	1	0	1	1	0		*S_2*
1	1	0	1	1	1		*S_idle*

Controller and Hardware Datapath



RTL via State Diagrams



(a)

S_idle ⟶ *S_1, clr_A_F*: $A \leftarrow 0, F \leftarrow 0$

S_1 ⟶ *S_1, incr_A*: $A \leftarrow A + 1$

if $(A_2 = 1)$ then *set_E*: $E \leftarrow 1$

if $(A_2 = 0)$ then *clr_E*: $E \leftarrow 0$

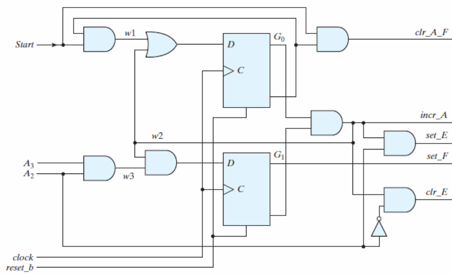
S_2 ⟶ *S_idle, set_F*: $F \leftarrow 1$

(b)

State Table

Present-State Symbol	Present State G_1	Present State G_0	Inputs *Start*	Inputs A_2	Inputs A_3	Next State G_1	Next State G_0	Outputs *set_E*	Outputs *clr_E*	Outputs *set_F*	Outputs *clr_A_F*	Outputs *incr_A*
S_idle	0	0	0	X	X	0	0	0	0	0	0	0
S_idle	0	0	1	X	X	0	1	0	0	0	1	0
S_1	0	1	X	0	X	0	1	0	1	0	0	1
S_1	0	1	X	1	0	0	1	1	0	0	0	1
S_1	0	1	X	1	1	1	1	1	0	0	0	1
S_2	1	1	X	X	X	0	0	0	0	1	0	0

Controller Circuitry



HDL for our Example

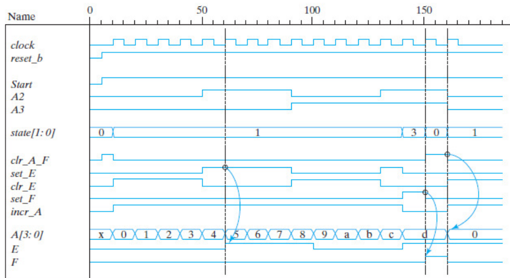
```
module Design_Example_RTL
// carefully consider unused state code implications for output and next
    state

 ( output reg     [3:0]    A,
   output reg              E, F,
   input                   Start, clock, reset_b
 );
 reg              [1: 0]   state, next_state;
 reg                       clr_E, set_E, clr_F, set_F, clr_A_F, incr_A;
 parameter                 S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;
 wire                      A2 = A[2], A3 = A[3];
```

```
// control unit
  always @ (posedge clock, negedge reset_b)
    if (reset_b ==0) state <= S_idle;
    else state <= next_state;

  always @ (state, Start, A2, A3) begin
    next_state = state;
    clr_E = 0;
    set_E = 0;
    clr_A_F = 0;
    set_F = 0;
    incr_A = 0;
    case (state)
      S_idle:          if (Start) begin next_state = S_1; clr_A_F = 1; end
      S_1: begin incr_A = 1; if (A2 == 0) clr_E = 1; else
                begin set_E = 1; if (A3) next_state = S_2; end
              end
      S_2: begin set_F = 1; next_state = S_idle; end
     //default:        next_state = S_idle;
    endcase
   end
```

```
// datapath unit
  always @ (posedge clock) begin
    if (clr_E)       E <= 0;
    if (set_E)       E <= 1;
    if (clr_A_F)   begin A <= 0; F <= 0; end
    if (set_F)       F <= 1;
    if (incr_A)     A <= A + 1;
end
endmodule
```

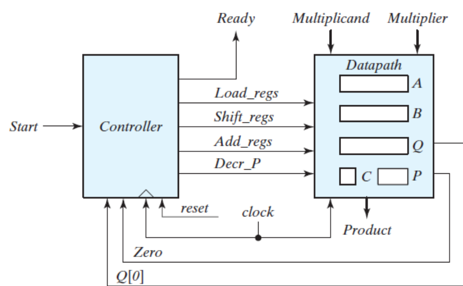
```
// test bench
module t_Design_Example_RTL;
reg Start, clock, reset_b;
  wire [3: 0] A;
  wire E, F;

  // Instantiate design example
  Design_Example_RTL M0 (A, E, F, Start, clock, reset_b);

  // Describe stimulus waveforms
  initial #500 $finish;              // Stopwatch
  initial
    begin
      reset_b = 0;
      Start = 0;
      clock = 0;
      #5 reset_b = 1; Start = 1;
      repeat (32)
        begin
          #5 clock = ~ clock;        // Clock generator
        end
    end
  initial
  $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

Testbench output

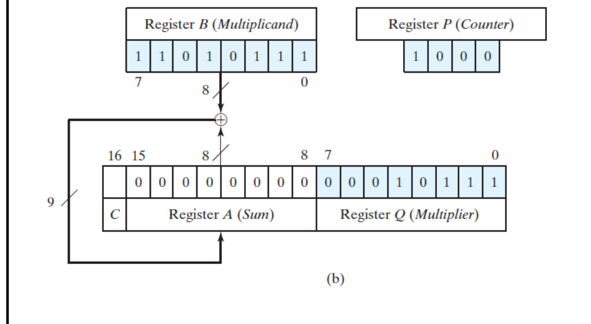


Sequential Binary Multiplier

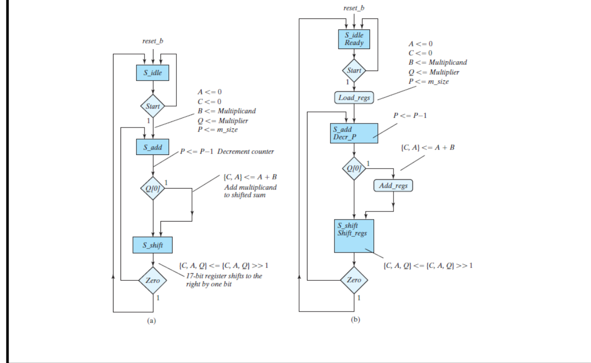
Sequential Binary Multiplier



SBM Datapath



ASMD Chart for Binary Multiplier

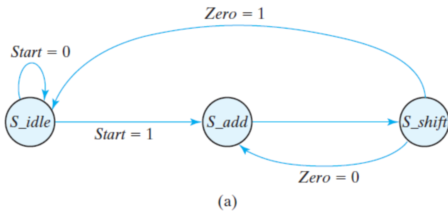


Operation

Multiplicand $B = 10111_2 = 17_H = 23_{10}$ **Multiplier $Q = 10011_2 = 13_H = 19_{10}$**

	C	A	Q	P
Multiplier in Q	0	00000	10011	101
$Q_0 = 1$; add B		10111		
First partial product	0	10111		100
Shift right CAQ	0	01011	11001	
$Q_0 = 1$; add B		10111		
Second partial product	1	00010		011
Shift right CAQ	0	10001	01100	
$Q_0 = 0$; shift right CAQ	0	01000	10110	010
$Q_0 = 0$; shift right CAQ	0	00100	01011	001
$Q_0 = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right CAQ	0	01101	10101	000
Final product in $AQ = 0110110101_2 = 1b5_H$				

Control State Diagram

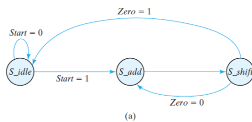


(a)

Control Register Operations

State Transition		Register Operations
From	To	
S_idle		Initial state
S_idle	*S_add*	$A <= 0$, $C <= 0$, $P <= dp_width$
S_add	*S_shift*	$P <= P - 1$ if (*Q[0]*) then ($A <= A + B$, $C <= C_{out}$)
S_shift		shift right {*CAQ*}, $C <= 0$

Control State Machine

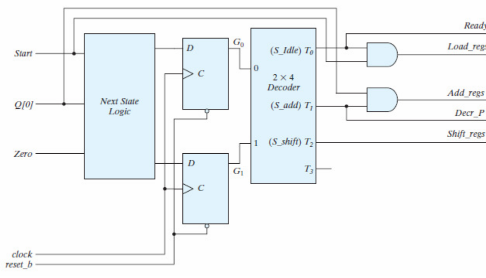


State	Binary	Gray Code	One-Hot
S_idle	00	00	001
S_add	01	01	010
S_shift	10	11	100

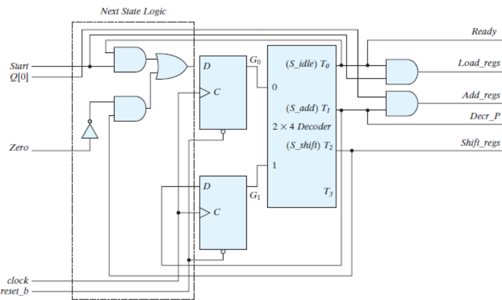
State Machine + Control Signals

Present-State Symbol	Present State G_1	Present State G_0	Inputs *Start*	Inputs *Q[0]*	Inputs *Zero*	Next State G_1	Next State G_0	*Ready*	*Load_regs*	*Decr_P*	*Add_regs*	*Shift_regs*
S_idle	0	0	0	X	X	0	0	1	0	0	0	0
S_idle	0	0	1	X	X	0	1	1	1	0	0	0
S_add	0	1	X	0	X	1	0	0	0	1	0	0
S_add	0	1	X	1	X	1	0	0	0	1	1	0
S_shift	1	0	X	X	0	0	1	0	0	0	0	1
S_shift	1	0	X	X	1	0	0	0	0	0	0	1

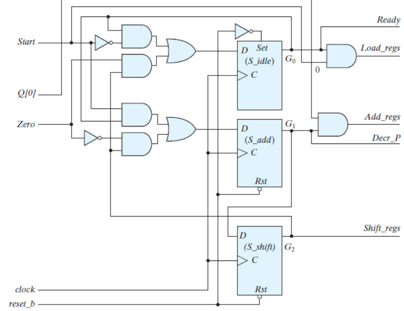
Sequence Register and Decoder



Sequential Binary Multiplier Sequence Register and Decoder



One Hot Representation

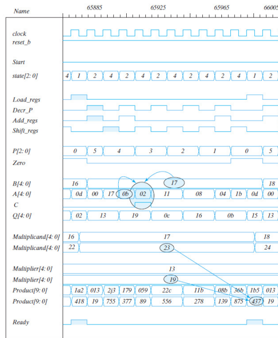



```
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start,
    clock, reset_b);
// Default configuration: 5-bit datapath
 parameter                               dp_width = 5;        // Set to width of
    datapath
 output   [2*dp_width - 1: 0]   Product;
 output                           Ready;
 input          [dp_width - 1: 0]   Multiplicand, Multiplier;
 input                                    Start, clock, reset_b;

 parameter                        BC_size =        3;                   // Size
    of bit counter
 parameter                        S_idle =   3'b001,          // one-hot code
                                  S_add =   3'b010,
                                  S_shift =  3'b100;

 reg      [2: 0]                  state, next_state;
 reg      [dp_width - 1: 0]       A, B, Q;            // Sized for datapath
 reg                              C;
 reg      [BC_size - 1: 0]        P;
 reg                              Load_regs, Decr_P, Add_regs, Shift_regs;
 assign                   Product = {A,Q};
 wire                             Zero = (P == 0);     // counter is zero
 wire                             Ready = (state == S_idle);    // controller status
```

```
// control unit

 always @ (posedge clock, negedge reset_b)
   if (~reset_b) state <= S_idle; else state <= next_state;

   always @ (state, Start, Q[0], Zero) begin
   next_state = S_idle;
   Load_regs = 0;
   Decr_P = 0;
   Add_regs = 0;
   Shift_regs = 0;
   case (state)
     S_idle:        begin if (Start) next_state = S_add; Load_regs = 1; end
     S_add:         begin next_state = S_shift; Decr_P = 1; if (Q[0]) Add_regs =
    1; end
     S_shift:       begin Shift_regs = 1; if (Zero) next_state = S_idle;
          else next_state = S_add; end
     default:       next_state = S_idle;
   endcase
 end
```

```
// datapath unit

  always @ (posedge clock) begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
endmodule
```

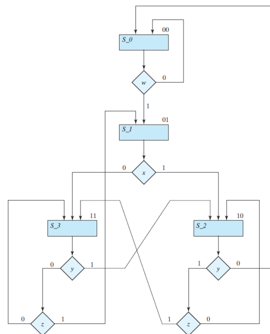
Testbench



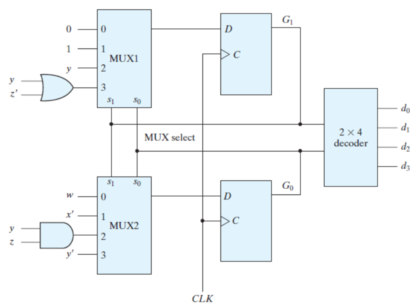
Behavioral HDL Description of a Parallel Multiplier

```
module Algorithmic_Binary_Multiplier #(parameter dp_width = 5) (
  output [2*dp_width - 1: 0] Product, input [dp_width - 1: 0] Multiplicand, Multiplier);
  reg       [dp_width - 1: 0]            A, B, Q;                              // Sized
    for datapath
  reg                                    C;
  integer k;
  assign                     Product = {C, A, Q};

  always @ (Multiplier, Multiplicand) begin
    Q = Multiplier;
    B = Multiplicand;
    C = 0;
    A = 0;
    for  (k = 0; k <= dp_width -1; k = k+1) begin
      if (Q[0]) {C, A} = A + B;
      {C, A, Q} = {C, A, Q} >> 1;
    end
  end
endmodule
```

Design with Multiplexors



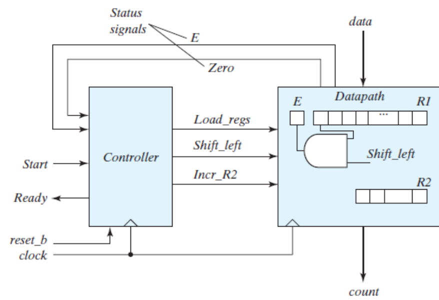
Design with Multiplexors

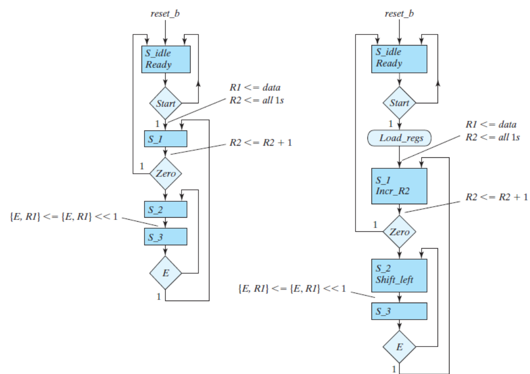


State Table

Present State		Next State		Input Condition	Inputs	
G_1	G_0	G_1	G_0	s	MUX1	MUX2
0	0	0	0	w'		
0	0	0	1	w	0	w
0	1	1	0	x		
0	1	1	1	x'	1	x'
1	0	0	0	y'		
1	0	1	0	yz'		
1	0	1	1	yz	$yz' + yz = y$	yz
1	1	0	1	$y'z$		
1	1	1	0	y		
1	1	1	1	$y'z'$	$y + y'z' = y + z'$	$y'z + y'z' = y'$

Mux Design Example

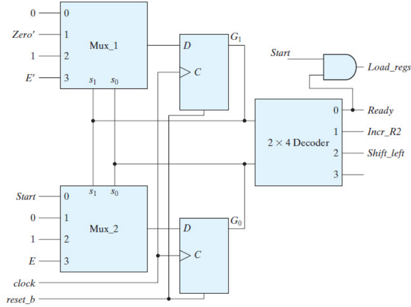




State Table

Present State		Next State		Input Conditions	Multiplexer Inputs	
G_1	G_0	G_1	G_0		MUX1	MUX2
0	0	0	0	$Start'$		
0	0	0	1	$Start$	0	$Start$
0	1	0	0	$Zero$		
0	1	1	0	$Zero'$	$Zero'$	0
1	0	1	1	None	1	1
1	1	1	0	E'		
1	1	0	1	E	E'	E

Mux-based Controller




```
module Count_Ones_STR_STR (count, Ready, data, Start, clock, reset_b);
// Mux – decoder implementation of control logic
// controller is structural
// datapath is structural

  parameter R1_size = 8, R2_size = 4;
  output  [R2_size -1: 0]     count;
  output                      Ready;
  input   [R1_size -1: 0]             data;
  input                               Start, clock, reset_b;
  wire                                Load_regs, Shift_left, Incr_R2, Zero, E;

 Controller_STR    M0 (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero,
    clock, reset_b);
 Datapath_STR    M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2,
    clock);

endmodule
```

```
module Controller_STR (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero, clock, reset_b);
  output      Ready;
  output       Load_regs, Shift_left, Incr_R2;
  input                       Start;
  input                       E, Zero;
  input                        clock, reset_b;
  supply0    GND;
  supply1    PWR;
  parameterS0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;  // Binary code
  wire                        Load_regs, Shift_left, Incr_R2;
  wire                        G0, G0_b, D_in0, D_in1, G1, G1_b;
  wire                        Zero_b = ~Zero;
  wire                        E_b = ~E;
  wire [1:0]  select = {G1, G0};
  wire [0:3]  Decoder_out;

  assign                              Ready = ~Decoder_out[0];
  assign                              Incr_R2 = ~Decoder_out[1];
  assign                              Shift_left = ~Decoder_out[2];
  and                                          (Load_regs, Ready, Start);
  mux_4x1_beh                         Mux_1       (D_in1, GND, Zero_b, PWR, E_b, select);
  mux_4x1_beh                         Mux_0       (D_in0, Start, GND, PWR, E, select);
  D_flip_flop_AR_b      M1            (G1, G1_b, D_in1, clock, reset_b);
  D_flip_flop_AR_b      M0            (G0, G0_b, D_in0, clock, reset_b);
  decoder_2x4_df        M2            (Decoder_out, G1, G0, GND);

endmodule
```

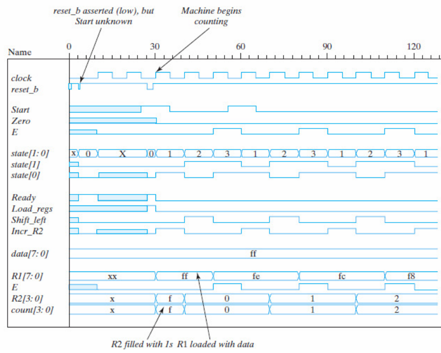
```
module Datapath_STR (count, E, Zero, data, Load_regs, Shift_left,
    Incr_R2, clock);
parameter               R1_size = 8, R2_size = 4;
output  [R2_size -1: 0] count;
output                  E, Zero;
input [R1_size -1: 0]   data;
input                   Load_regs, Shift_left, Incr_R2, clock;
wire [R1_size -1: 0]    R1;
supply0                 Gnd;
 supply1                       Pwr;
 assign Zero = (R1 == 0);

 Shift_Reg              M1     (R1, data, Gnd, Shift_left, Load_regs,
    clock, Pwr);
 Counter                M2     (count, Load_regs, Incr_R2, clock,
    Pwr);
 D_flip_flop_AR         M3     (E, w1, clock, Pwr);
 and                           (w1, R1[R1_size -1], Shift_left);
endmodule
```

Testing

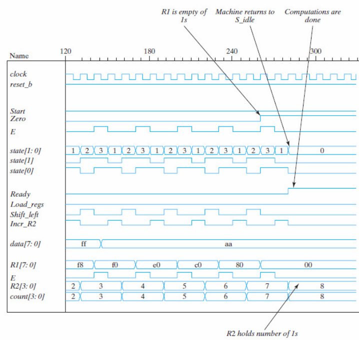
```
module t_Count_Ones;
  parameter R1_size = 8, R2_size = 4;
  wire [R2_size -1: 0]    R2;

  wire [R2_size -1: 0]         count;
  wire                    Ready;
  reg  [R1_size -1: 0]         data;
  reg                     Start, clock, reset_b;
  wire [1: 0] state;      // Use only for debug
  assign state = {M0.M0.G1, M0.M0.G0};

  Count_Ones_STR_STR M0 (count, Ready, data,
   Start, clock, reset_b);
```

```
initial #650 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
#1 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
     data = 8'Hff;
    # 25 Start = 1;
    # 35  Start = 0;
    #55 Start = 1;
    #65 Start = 0;
    #145 data = 8'haa;
    #395 Start = 1;
    #405 Start = 0;
    #27 reset_b = 0;
    #29 reset_b = 1;
    #355 reset_b = 0;
    #365 reset_b = 1;
  join
endmodule
```

Testbench results



Testbench Results



Race Conditions

- Verification is important
  - Simulation and Physical Circuit must match
- Three common problems
  - Feedback path exists between data and control
  - Simulated procedural blocks transition immediately isn't true of real circuits
  - Order of execution of multiple blocked assignments to a variable is indeterminate

Latch Free Synthesis

- Feedback free assignments will synthesize to logic
- Input and output should change automatically
  - Inputs must ALL be specified in sensitivity list
  - Otherwise synthesis will use latches
  - Always@* (verilog)