Logic Design with Verilog IV: Design Example (Bus) and Synthesizable Code

Jae W. Lee (jaewlee@snu.ac.kr)

Department of Computer Science and Engineering

Seoul National University

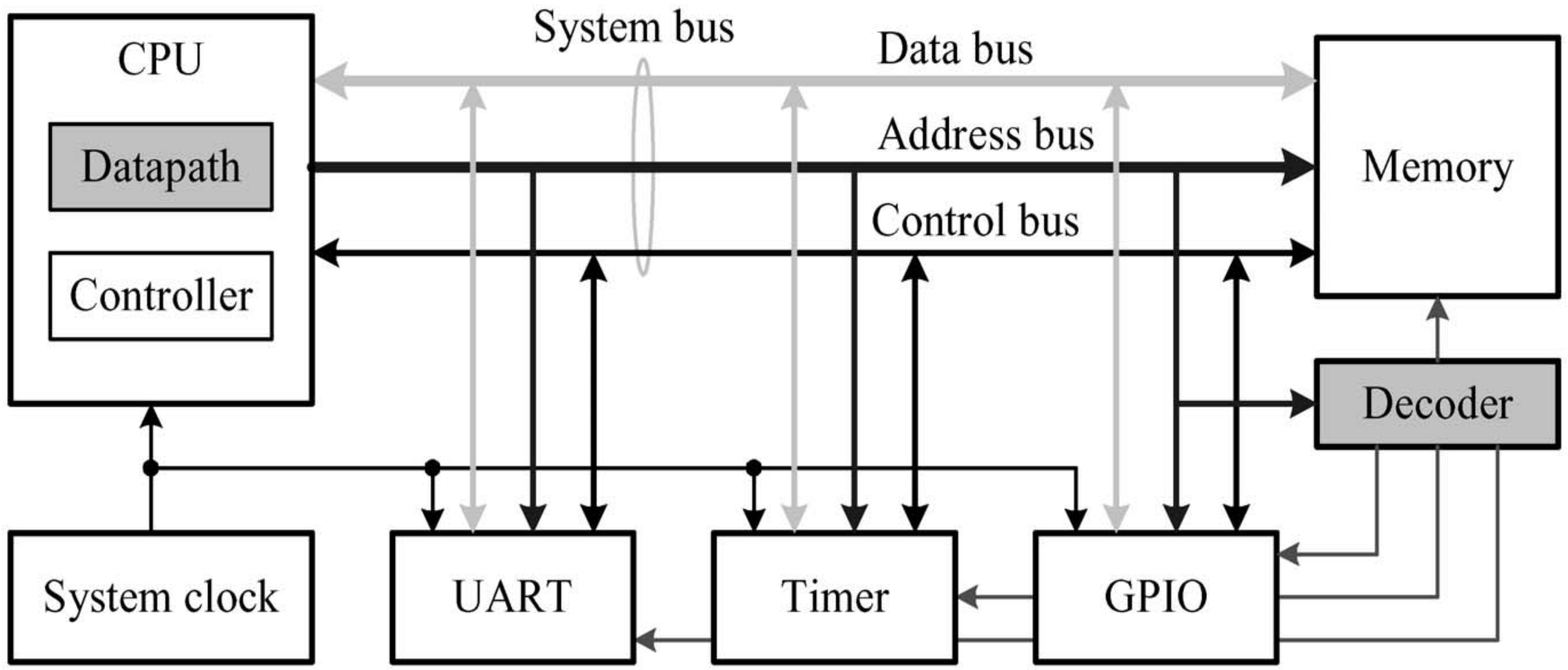
Slide credits: Prof. Ming-Bo Lin (Digital System Designs and Practices Using Verilog HDL and FPGAs)

Outline

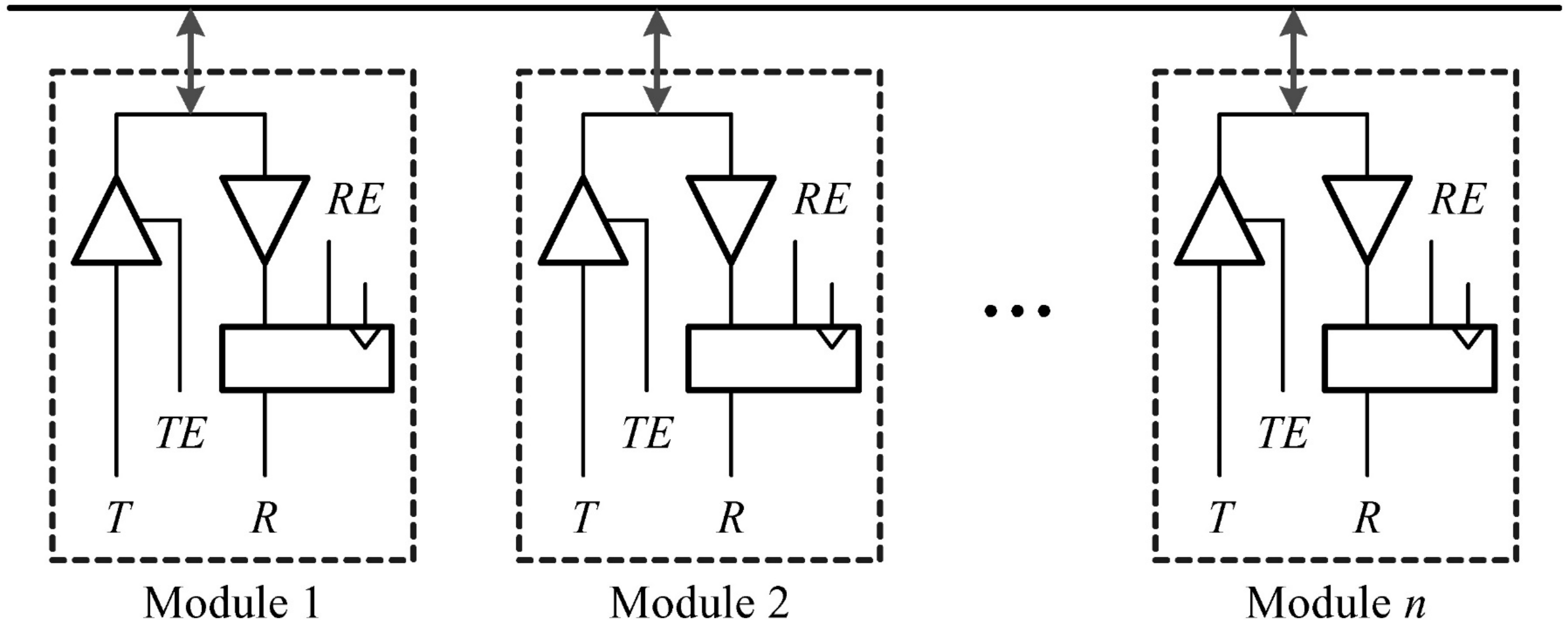
- Design example: bus
  - Tri-state based bus
  - Multiplexer-based bus
  - Bus arbitration
  - Simple bus controller example
- Verilog synthesis flow
  - Design environment and constraints
  - Architecture of logic synthesizers
  - Synthesizable operators and constructs
  - Logic optimization

Bus: A Basic Microprocessor System

- Bus: a set of wires used to transport information between two or m ore devices in a digital system



Bus: A Tristate Bus

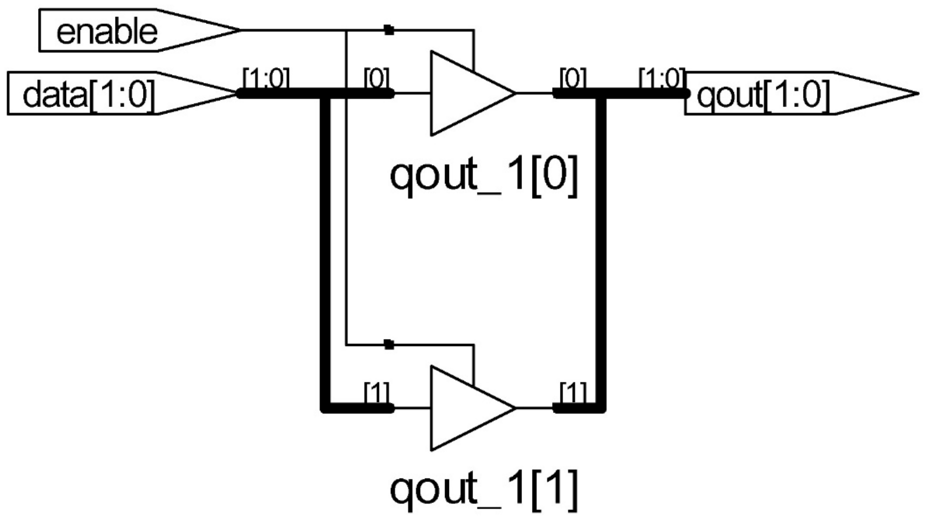


T : transmit TE : transmit enable R: receive RE : receive enable

Bus: A Tristate Bus

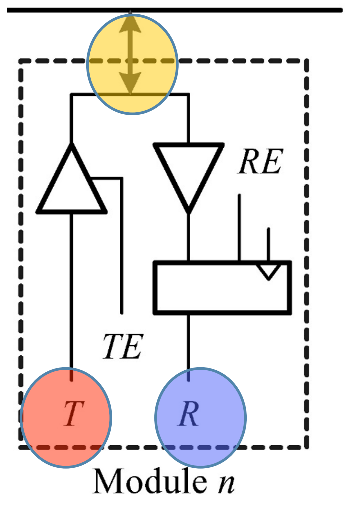
- Verilog code

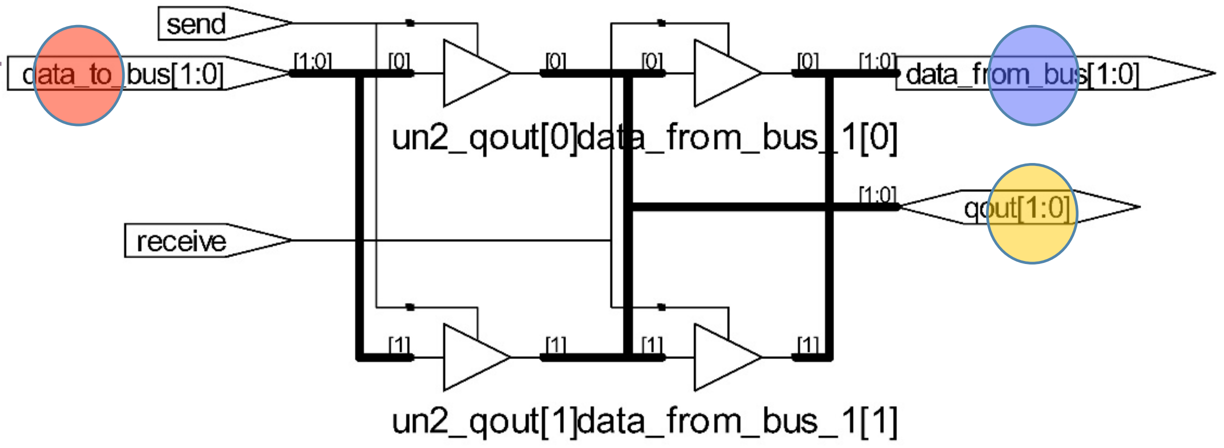
```
// a tristate bus example
module tristate_bus (data, enable, qout);
parameter               N = 2;    // define bus width
input                   enable;
input  [N-1:0]          data;
output [N-1:0]          qout;

// the body of tristate bus
assign qout = enable ? data : {N{1'bz}};
endmodule
```



Bus: A Bidirectional Bus

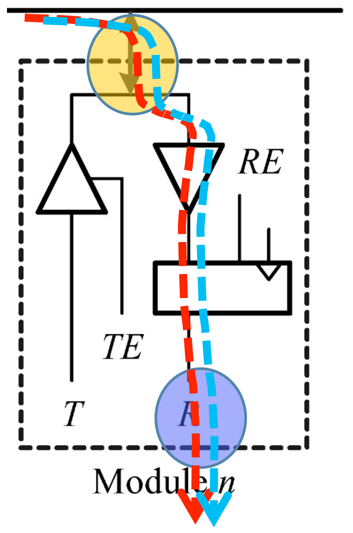
```
// a bidirectional bus example
module bidirectional_bus (data_to_bus, send, receive, data_from_bus, qout);
parameter               N = 2;          // define bus width
input                   send, receive;
input   [N-1:0]         data_to_bus;
output  [N-1:0]         data_from_bus;
inout   [N-1:0]         qout;           // bidirectional bus
// the body of tristate bus
assign data_from_bus = receive ? qout : {N{1'bz}};
assign qout = send ? data_to_bus : {N{1'bz}};
endmodule
```

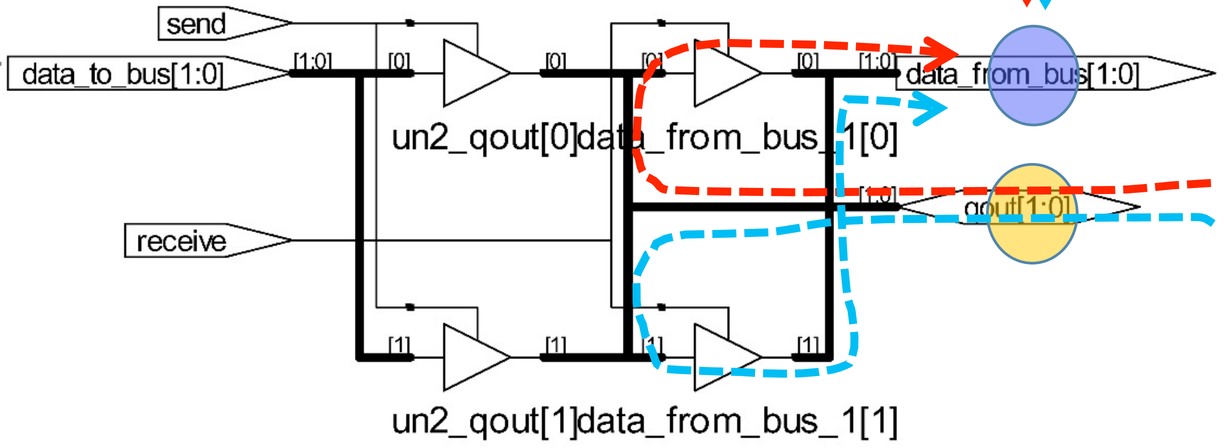




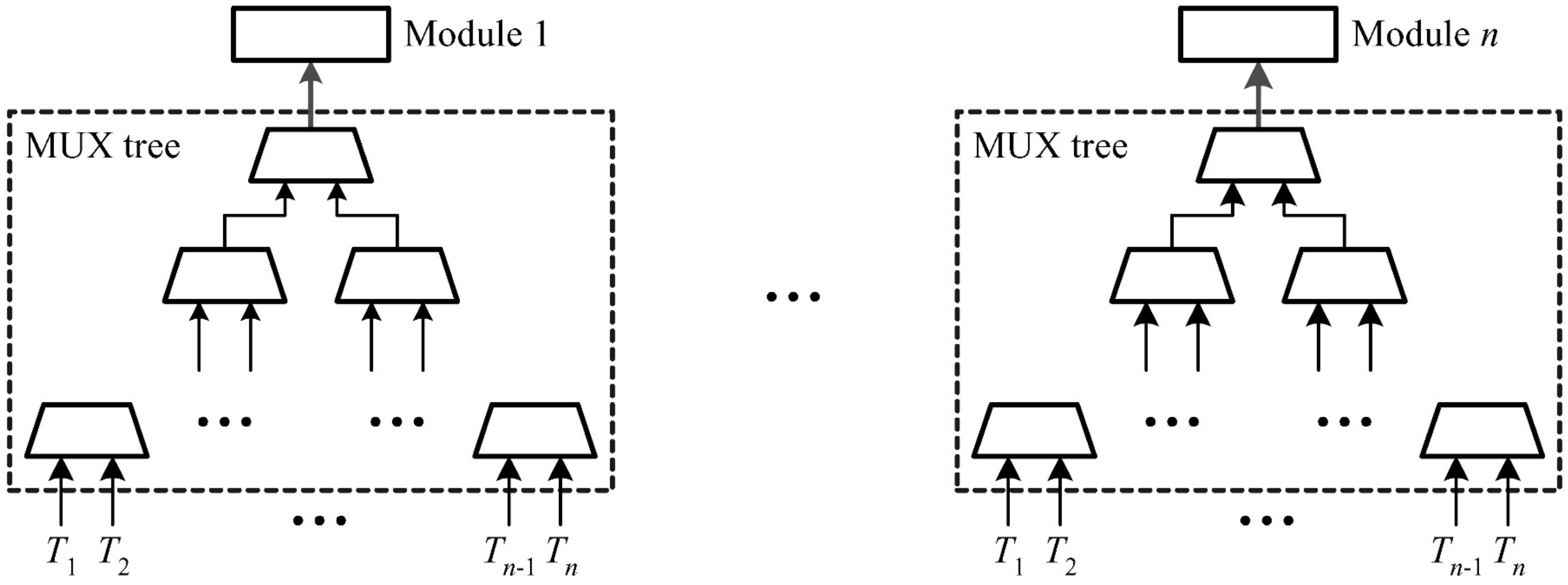
Bus: A Bidirectional Bus

```
// a bidirectional bus example
module bidirectional_bus (data_to_bus, send, receive, data_from_bus, qout);
parameter               N = 2;          // define bus width
input                   send, receive;
input   [N-1:0]         data_to_bus;
output  [N-1:0]         data_from_bus;
inout   [N-1:0]         qout;           // bidirectional bus
// the body of tristate bus
assign data_from_bus = receive ? qout : {N{1'bz}};
assign qout = send ? data_to_bus : {N{1'bz}};
endmodule
```





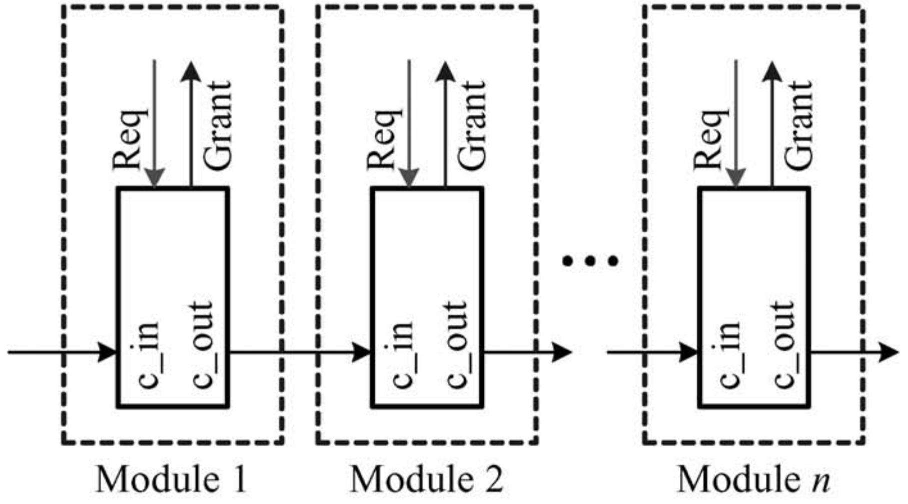
Bus: A Multiplexer-Based Bus



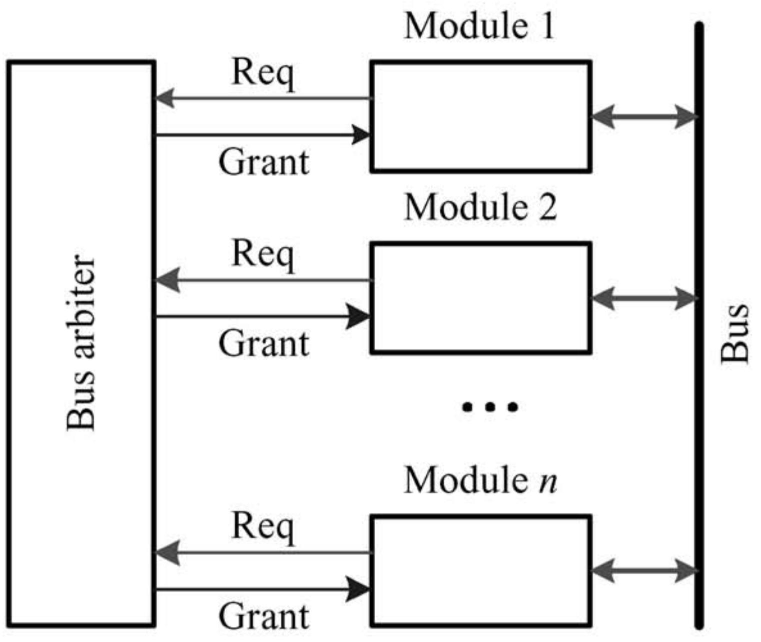
The propagation delay is much less than the tristate bus when the modules attached to it are large enough
It can avoid the large amount of capacitive load by the tristate bus

Bus: Arbitration

- Bus arbitration
  - The operation that chooses one transmitter from multiple ones attempting to transmit data on the bus
- Types of bus arbitration schemes
  - Daisy-chain: fixed priority
  - Radial: fixed, round robin, etc.



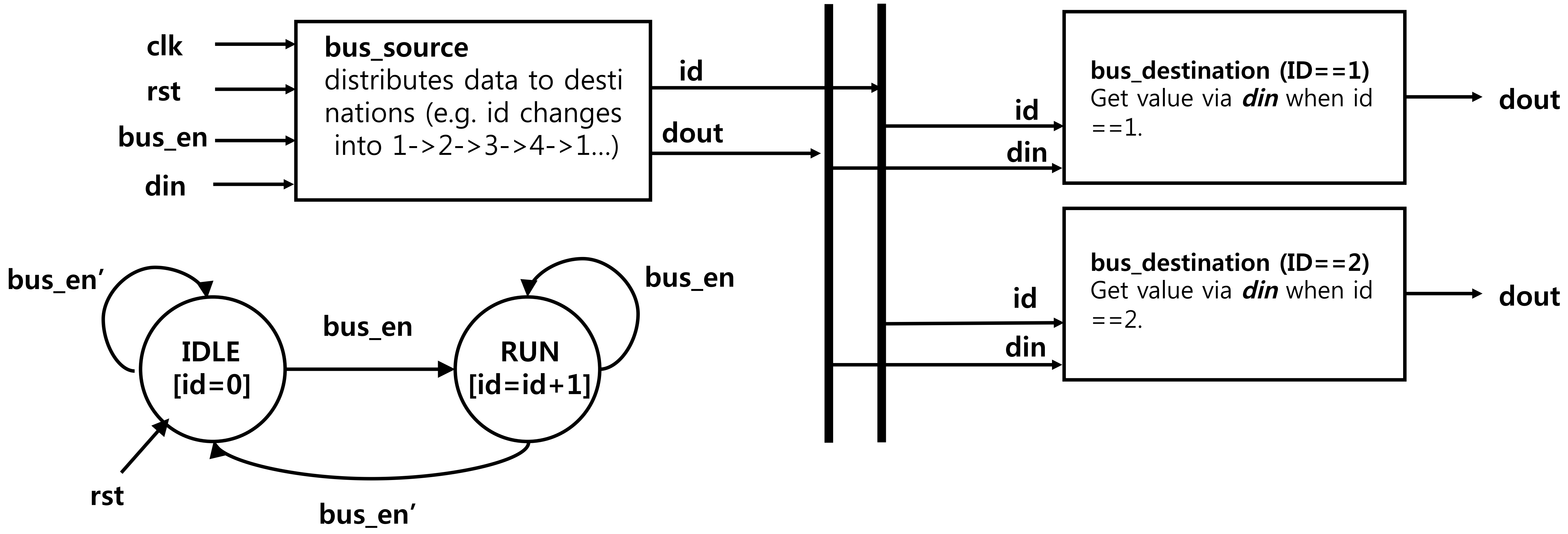
(a) Daisy-chain arbitration



(b) Radial arbitration

Bus: Simple bus controller example

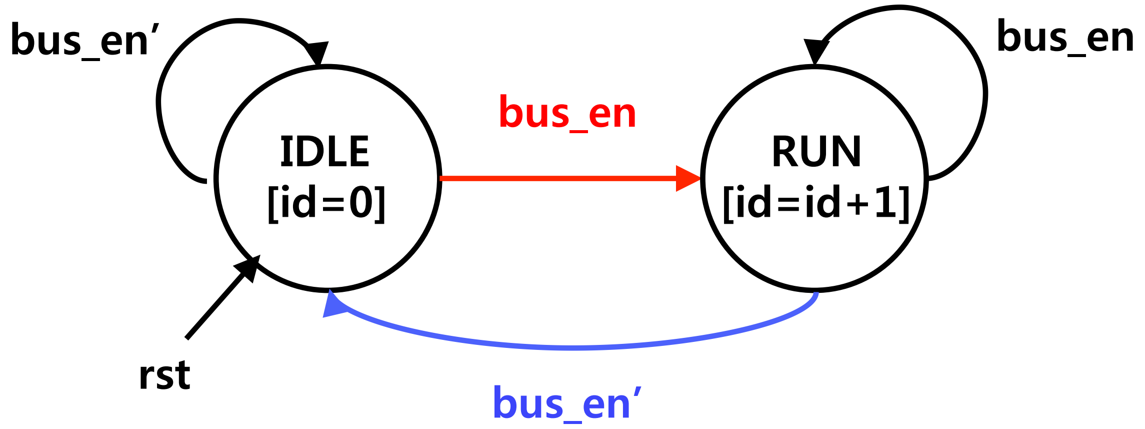
- Overview



Bus: Simple bus controller example

- Source

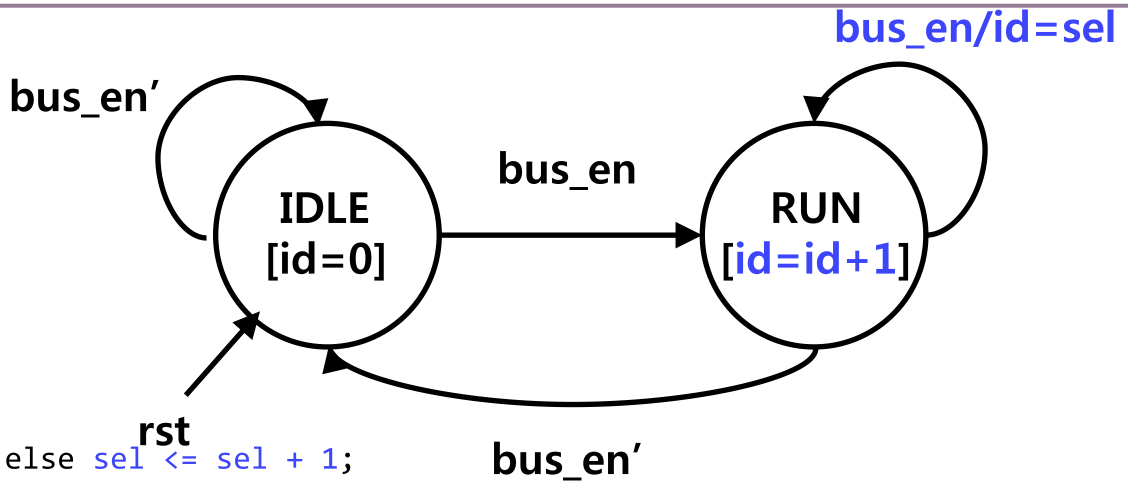
```
module bus_source (clk, rst, bus_en, din, dout, id);
parameter NUM_DESTINATION=2;
input                clk, rst, bus_en;
input  [31:0]        din;
output [31:0]        dout;
output reg [2:0]     id;
reg [2:0]            sel;
reg                  sel_rst;
reg [1:0]            curr_state, next_state;
parameter IDLE=2`d0, RUN=2`d1;
assign dout = din;

  //part 1: initialize to state INIT and update current state register
  always @(posedge clk or posedge rst)
    if(rst) curr_state <= IDLE; else curr_state <= next_state;

  //part 2: determine next state
  always @(*)
    case(curr_state)
      IDLE: if(bus_en) next_state <= RUN; else next_state <= curr_state;
      RUN: if(bus_en) next_state <= curr_state; else next_state <= IDLE;
    endcase
```



Bus: Simple bus controller example

- Source (cont'd)

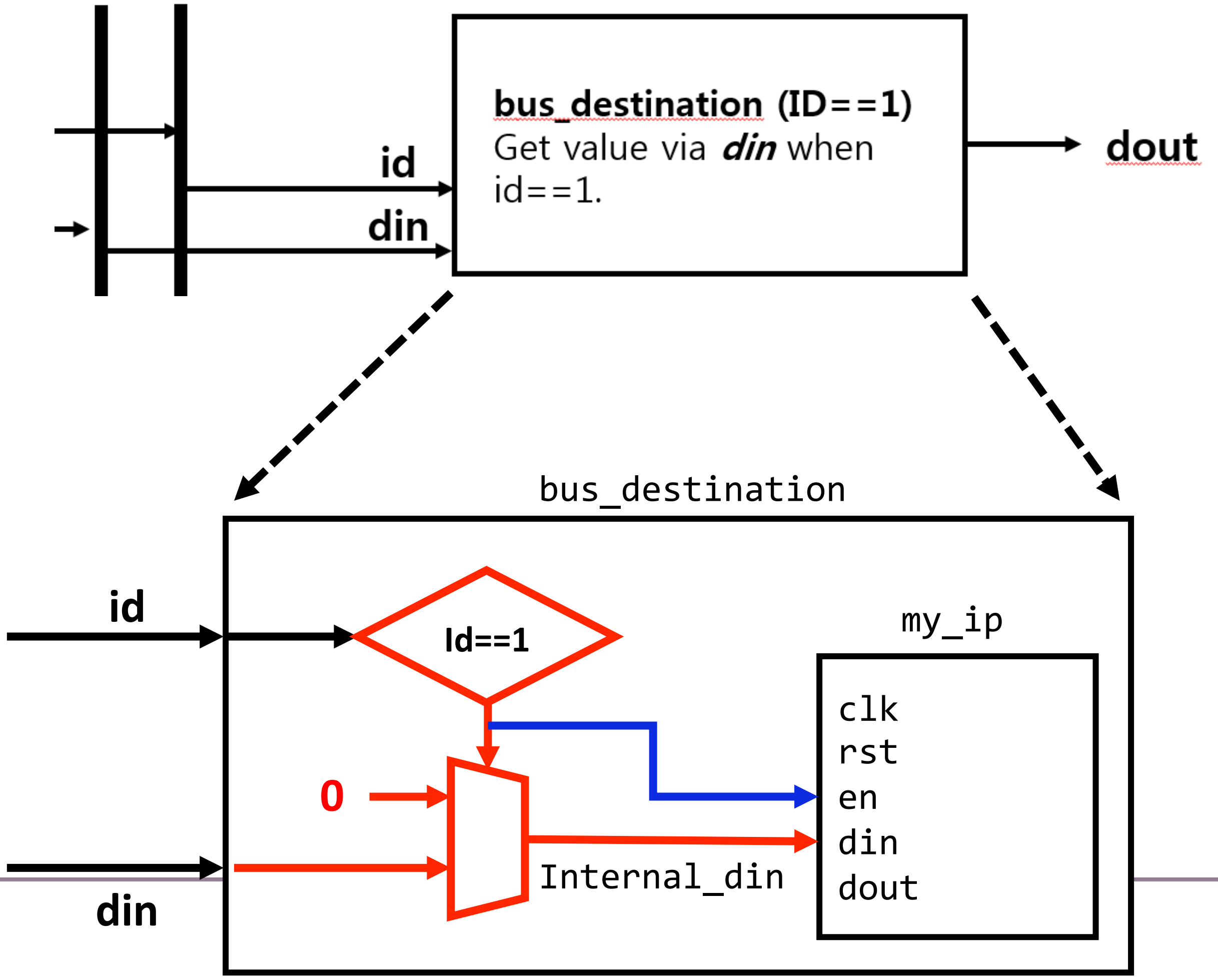
```
//part 3: determine output and internal register
//Output id is determined
always @(*)
  case(curr_state)
    IDLE: id <= 0;
    RUN: if(bus_en) id <= sel; else id <= 0;
    default: id <= id;
  endcase

//If sel_rst == 0, increment id counter (sel)
always @(posedge clk or posedge sel_rst)
  if(sel_rst) sel <= 0; else if(sel==NUM_DESTINATION) sel <= 1; else sel <= sel + 1;

//Setting internal register to control the counter
always @(*)
  case(curr_state)
    IDLE: if(bus_en) sel_rst <= 0; else sel_rst <= 1;
    RUN: if(bus_en) sel_rst <= 0; else sel_rst <= 1;
    default: sel_rst <= sel_rst;
  endcase
```



Bus: Simple bus controller example

- Destination

```
// a bidirectional bus example
module bus_destination (clk, rst, id, din, dout);
parameter ID=1;
input               id;
input   [31:0]      din;
output  [31:0]      dout;

wire [31:0] internal_din;
wire en;

assign internal_din = (id==ID) ? din : 0;
assign en = (id==ID) ? 1 : 0;

MY_IP my_ip (
          .clk(clk),
          .rst(rst),
          .en(en),
          .din(internal_din),
          .dout(dout)
);

endmodule
```

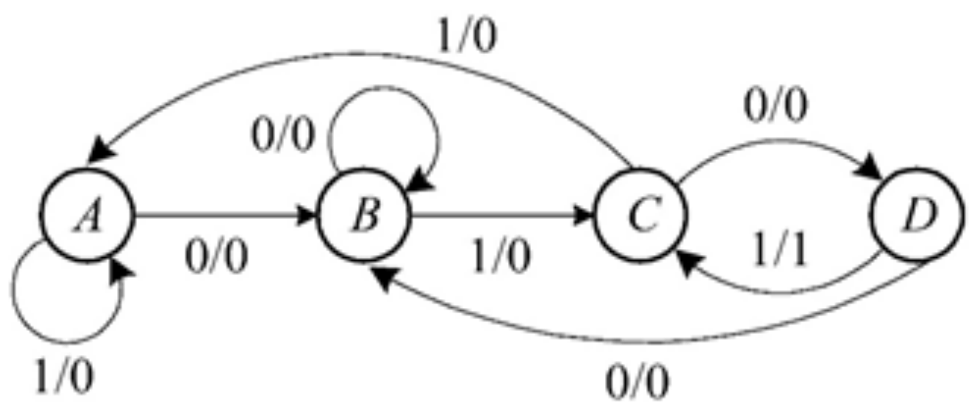


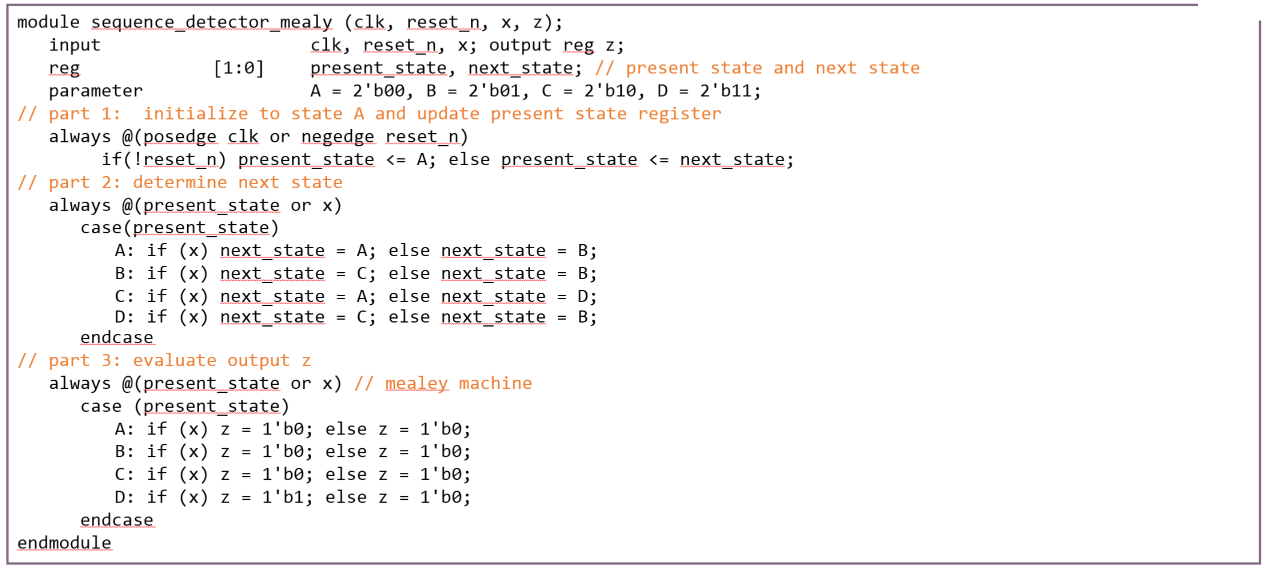
Outline

- Design example: bus
  - Tri-state based bus
  - Multiplexer-based bus
  - Bus arbitration
  - Simple bus controller example

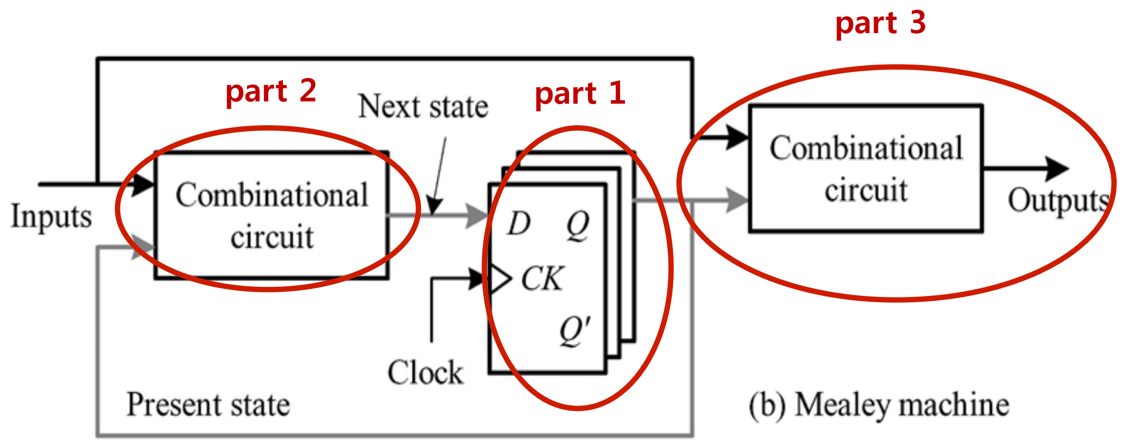
- Verilog synthesis flow
  - Design environment and constraints
  - Architecture of logic synthesizers
  - Synthesizable operators and constructs
  - Logic optimization

Logic Synthesizer





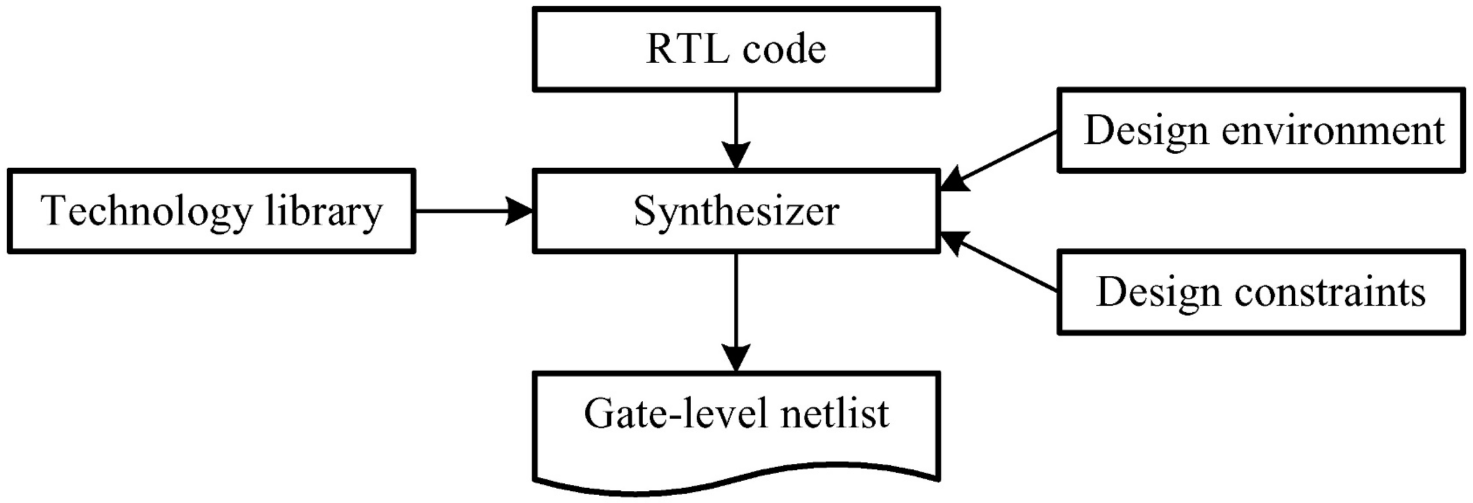
```
module sequence_detector_mealy (clk, reset_n, x, z);
   input                        clk, reset_n, x; output reg z;
   reg              [1:0]       present_state, next_state; // present state and next state
   parameter                    A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// part 1:  initialize to state A and update present state register
   always @(posedge clk or negedge reset_n)
        if(!reset_n) present_state <= A; else present_state <= next_state;
// part 2: determine next state
   always @(present_state or x)
      case(present_state)
         A: if (x) next_state = A; else next_state = B;
         B: if (x) next_state = C; else next_state = B;
         C: if (x) next_state = A; else next_state = D;
         D: if (x) next_state = C; else next_state = B;
      endcase
// part 3: evaluate output z
   always @(present_state or x) // mealey machine
      case (present_state)
         A: if (x) z = 1'b0; else z = 1'b0;
         B: if (x) z = 1'b0; else z = 1'b0;
         C: if (x) z = 1'b0; else z = 1'b0;
         D: if (x) z = 1'b1; else z = 1'b0;
      endcase
endmodule
```



(b) Mealey machine

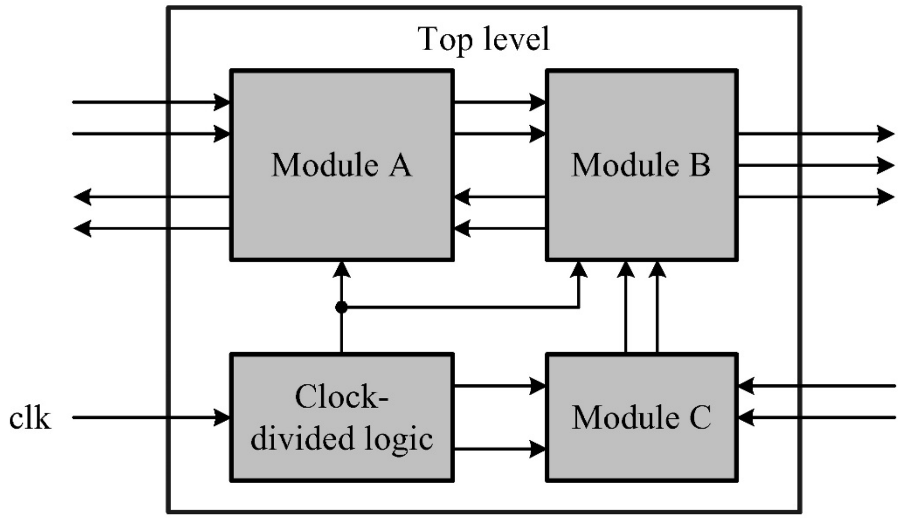
Logic Synthesis Environment

- Design environment
- Design constraints
- RTL code
- Technology library



Design Environment

- The process parameters
  - Technology library
  - Operating conditions
    - Process of technology
    - Operating voltage and temperature
- I/O port attributes
  - Drive strength of input port
  - Capacitive loading of output port
  - Design rule constraints
- Statistical wire-load model
  - Pre-layout static timing analysis



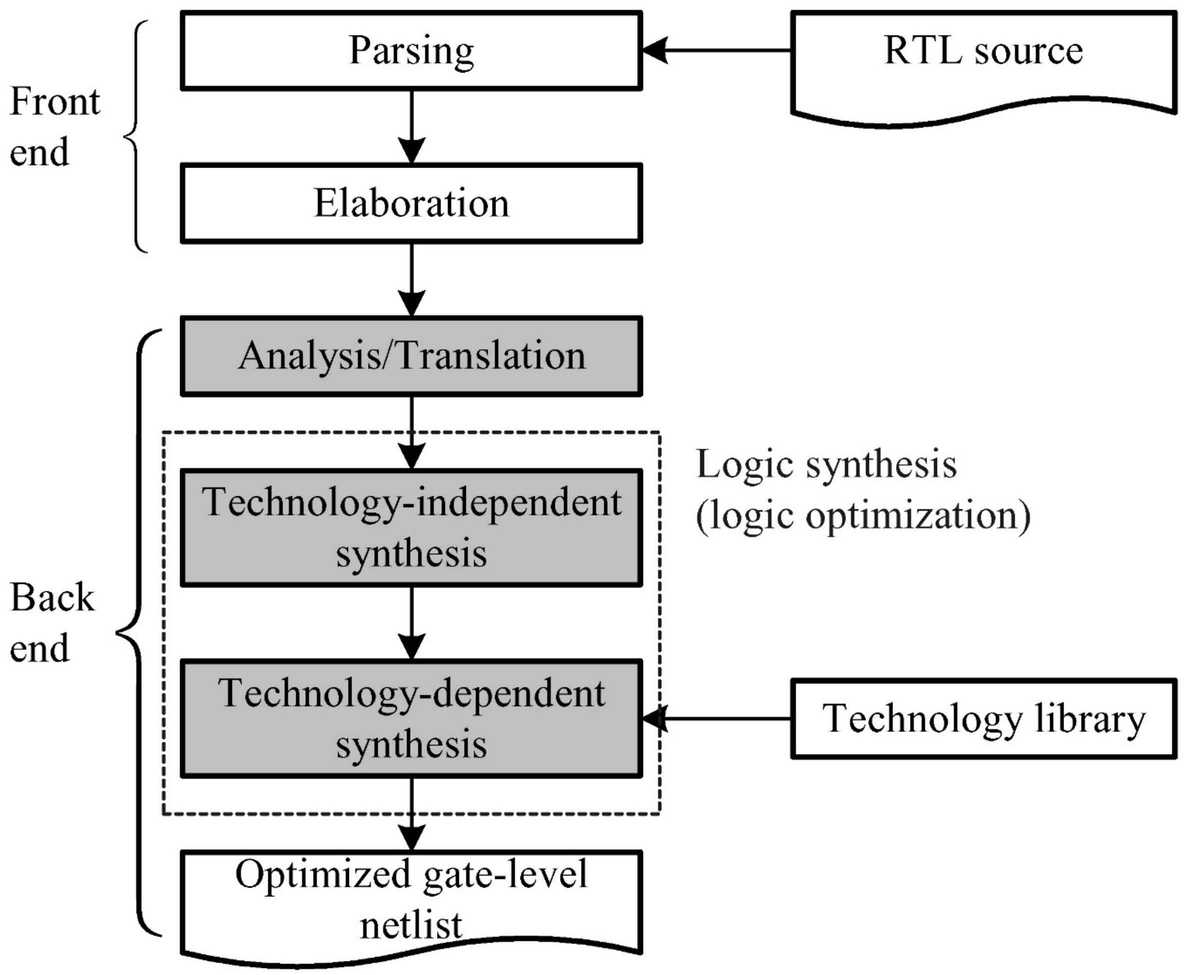
Design Constraints

- Include clock, input delay and output delay
  - e.g., 100MHz, every input/output has 0.5ns delay

- Clock signal specification
  - Period : clock period
  - Duty cycle : proportion of time during which a component is operated
  - Transition time : rise time and fall time of the clock
  - Skew : clock network delay

- Delay specifications
  - Maximum
  - Minimum

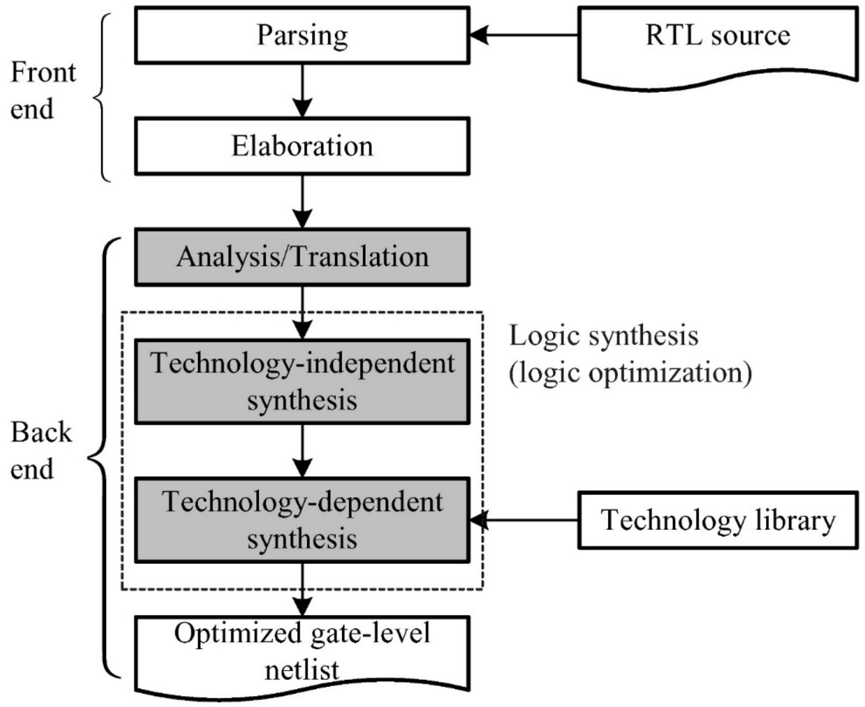
Architecture of Logic Synthesizers


Front end
Parsing
RTL source
Elaboration
Back end
Analysis/Translation
Logic synthesis (logic optimization)
Technology-independent synthesis
Technology-dependent synthesis
Technology library
Optimized gate-level netlist

Architecture of Logic Synthesizers

- Front end
  - Parsing phase
    - Check the syntax of the source code
    - Create internal components
  - Elaboration phase
    - Connect the internal components
    - Unroll loops & expand generate-loops
    - Set up parameters passing for tasks and functions

→ Complete description of the input circuit

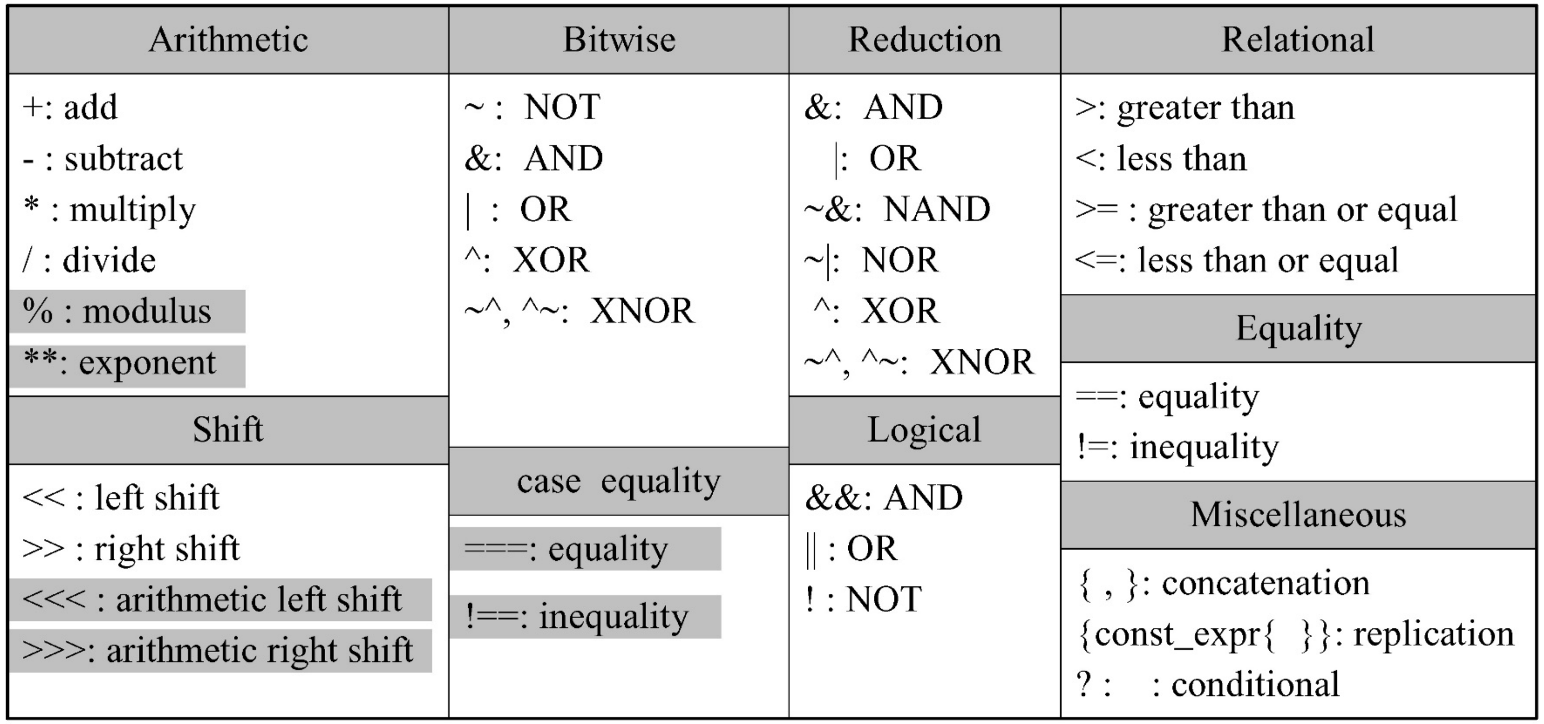

Front end
Parsing
RTL source
Elaboration
Back end
Analysis/Translation
Technology-independent synthesis
Logic synthesis (logic optimization)
Technology-dependent synthesis
Technology library
Optimized gate-level netlist

Writing Synthesizable Code

- Synthesizable operators
- Synthesizable constructs
  - assignment statement
  - if .. else statement
  - case statement
  - loop structures
  - always statement

Synthesizable Operators

- Analysis/translation step extracts the logical functions from a language structure



Arithmetic	Bitwise	Reduction	Relational
+: add	~ : NOT	&: AND	>: greater than
- : subtract	&: AND	|: OR	<: less than
* : multiply	| : OR	~&: NAND	>= : greater than or equal
/ : divide	^: XOR	~|: NOR	<=: less than or equal
% : modulus	~^, ^~: XNOR	^: XOR	
**: exponent		~^, ^~: XNOR	

Shift	case equality	Logical	Equality
<< : left shift	===: equality	&&: AND	==: equality
>> : right shift	!==: inequality	|| : OR	!=: inequality
<<< : arithmetic left shift		! : NOT	
>>>: arithmetic right shift			

Miscellaneous
{ , }: concatenation
{const_expr{ }}: replication
? : : conditional

Operators with gray blocks are generally the exceptions for synthesis
Operands can be wire net type, reg variable type and parameter, as well as functions

Synthesizable Constructs: if-else

- Usually synthesize into a 2-to-1 multiplexer
  - Nested if-else → priority-encoded, cascaded combination of multiplexers

- For combinational logic
  - Completely specified?

```
always @(enable or data)
    if (enable) y = data;  //infer a latch
```

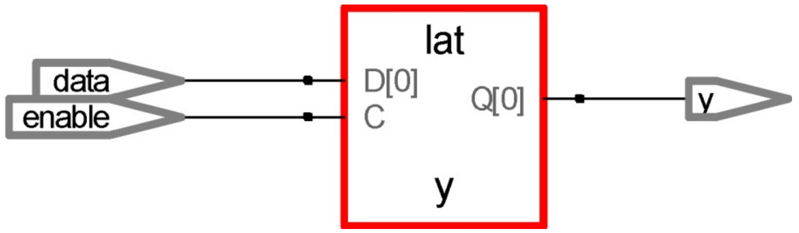
```
always @(enable or data)
    if (enable) y = data;
    else y = y; //infer a latch
```

```
always @(enable or data)
    if (enable) y = data;
    else y = x; //infer a mux
```

Synthesizable Constructs: if-else

- Latch inference --- Incomplete if-else statements

```
// creating a latch
module latch_infer_if(enable, data, y);
input          enable, data;
output reg     y;

always @(enable or data)
    if (enable) y = data;   // infer a latch for y
```



Synthesizable Constructs: if-else

- For sequential logic
  - Completely specified?

```
always @(posedge clk)
    if (enable) y <= data;
    else y <= y;     // a redundant expression
```

  - Synthesis tool will remove the redundant expression

Synthesizable Constructs: if-else

- Coding style
  - Avoid using any latches in a design
  - Assign outputs for all input conditions to avoid inferred latches
  - Two ways to avoid latch inference:

```
always @(enable or data)
    y = 1'b0;    // initialize y to its initial value.
    if (enable) y = data;
```

```
always @(enable or data)
    if (enable) y = data;
    else y = x;  // complete if-else statement
```

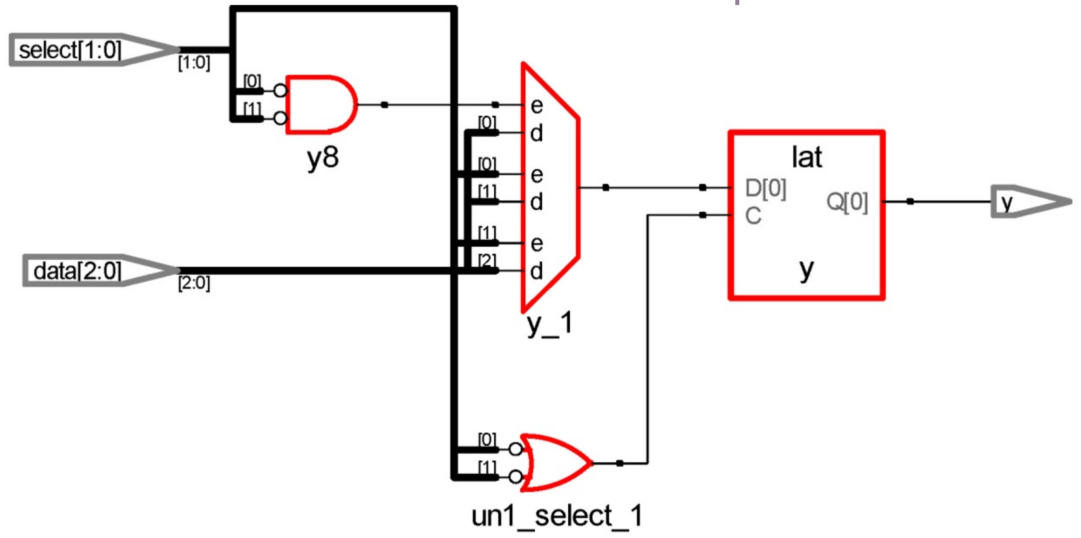
Synthesizable Constructs: case Statement

- A case statement
  - Infers a multiplexer
  - Completely specified?

Synthesizable Constructs: case Statement

- Latch inference --- Incomplete case statements

```
// Creating a latch
module latch_infer_case(select, data, y);
…
output reg   y;
always @(select or data)
    case (select)
       2'b00: y = data[select];
       2'b01: y = data[select];
       2'b10: y = data[select];
      // infer a latch
endcase
```



Synthesizable Constructs: case Statement

- Latch inference --- Complete case statements

```
module mux(select, data, y);
…
output reg   y;
always @(select or data)
    case (select)
       2'b00: y = data[select];
       2'b01: y = data[select];
       2'b10: y = data[select];
      default: y = 2'b11; // mux
    endcase
```

Synthesizable Constructs: Pos/Neg signals

- Mixed use of posedge/level signals

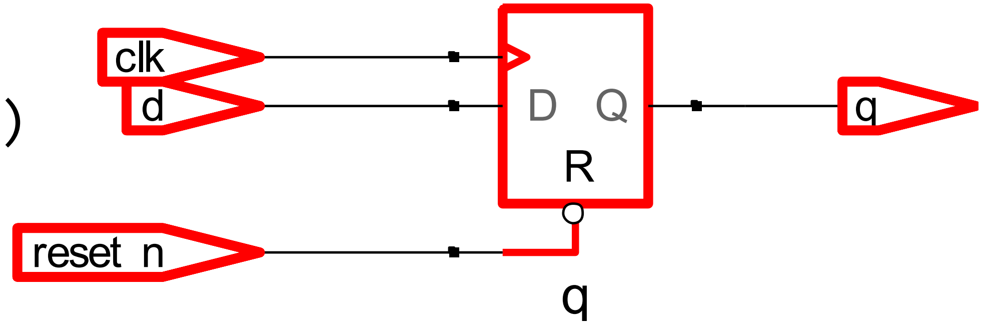
```
// the mixed usage of posedge and level signals
// the result cannot be synthesized
module DFF_bad (clk, reset, d, q);
…
// the body of DFF
always @(posedge clk or reset)
begin
     if (reset)     q <= 1'b0;
     else           q <= d;
end
```

Error: Do not mix posedge/negedge use with plain (level) signal references

Synthesizable Constructs: Pos/Neg signals

- Mixed use of posedge/negedge signals

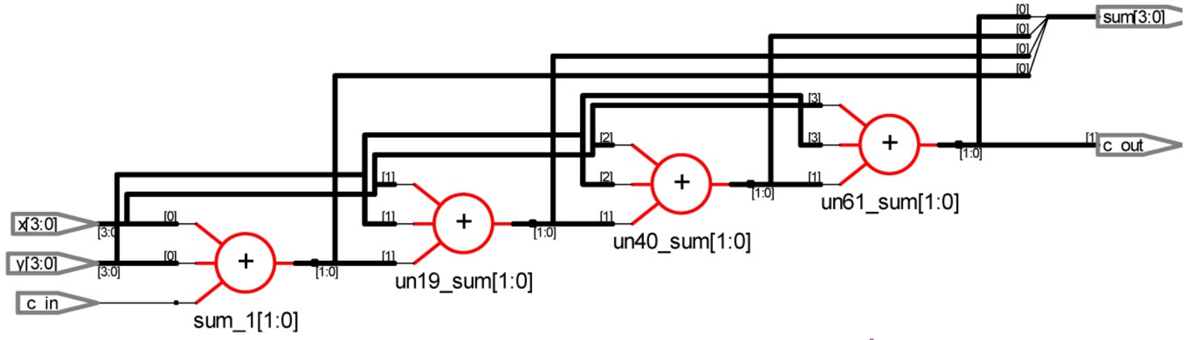
```
// the mixed usage of posedge/negedge signal
module DFF_good (clk, reset_n, d, q);
…
// the body of DFF
always @(posedge clk or negedge reset_n)
begin
     if (!reset_n)  q <= 1'b0;
     else           q <= d;
end
```



Synthesizable Constructs: Loops

- Loop statements *for*, *while*, and *forever* are synthesizable except that *while* and *forever* must contain timing control @(posedge clk) or @(negedge clk)
- *repeat* is generally not synthesizable

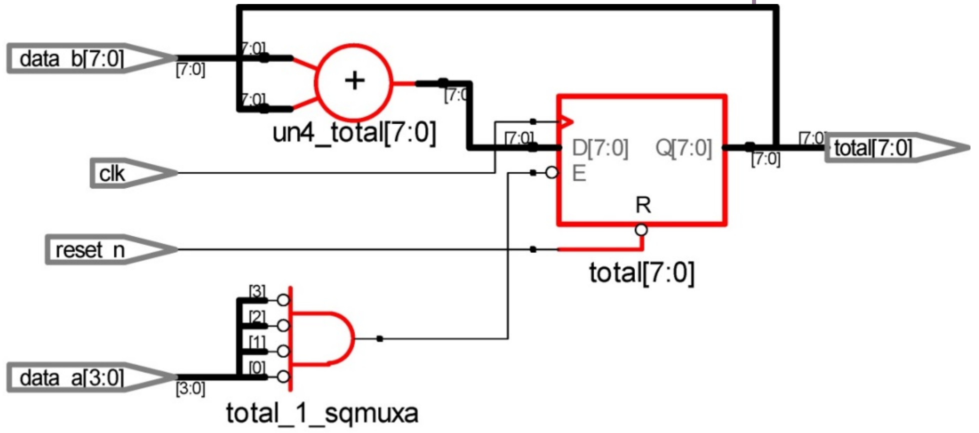
```
// an N-bit adder using for loop.
module nbit_adder_for( x, y, c_in, sum, c_out);
parameter       N = 4;     // default size
…
integer         i;
…
always @(x or y or c_in) begin
   co = c_in;
   for (i = 0; i < N; i = i + 1)
        {co, sum[i]} = x[i] + y[i] + co; // At the elaboration phase, the always
block is expanded into the following statements
   c_out = co;
end
```



Synthesizable Constructs: Loops

- Loop structures – Incorrectly synthesizable example

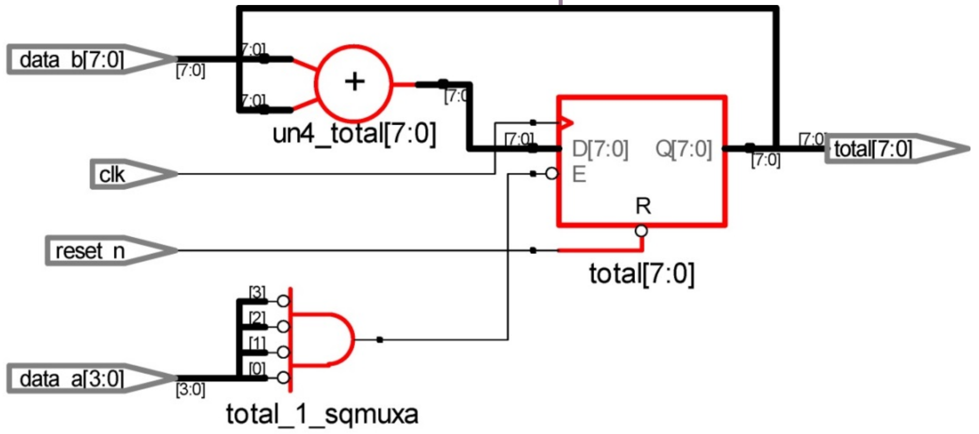
```
// a multiple cycle example --- This is an incorrect version.
…
parameter    N = 8;
parameter    M = 4;
input        clk, reset_n;
…
integer i;
// what does the following statement do?
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) total <= 0;
    else  for (i = 0; i < M; i = i + 1)
        if (data_a[i] == 1) total <= total + data_b;
end
```



Q: Why the synthesized result is like this?
Due to the blocking statement!

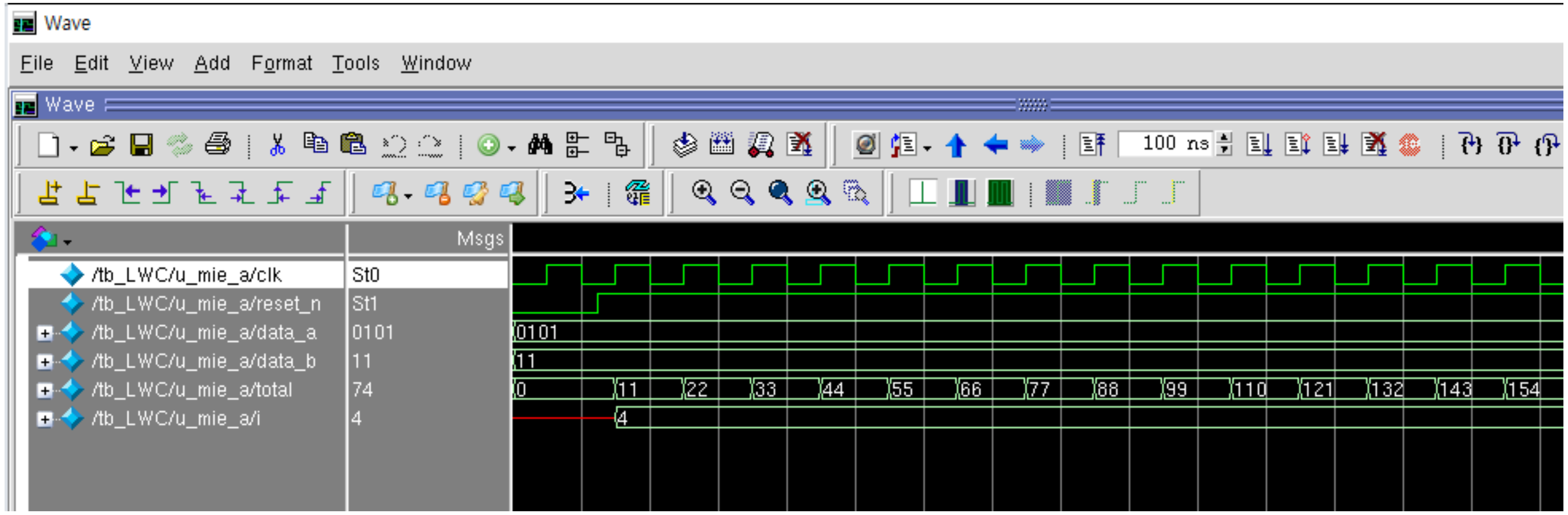
Synthesizable Constructs: Loops

```
// a multiple cycle example --- This is an incorrect version.
…
parameter    N = 8;
parameter    M = 4;
input        clk, reset_n;
…
integer i;
// what does the following statement do?
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) total <= 0;
    else  begin
      if (data_a[0] == 1) total <= total + data_b;
      if (data_a[1] == 1) total <= total + data_b;
      if (data_a[2] == 1) total <= total + data_b;
      if (data_a[3] == 1) total <= total + data_b;
    end
end
```



Synthesizable Constructs: Loops

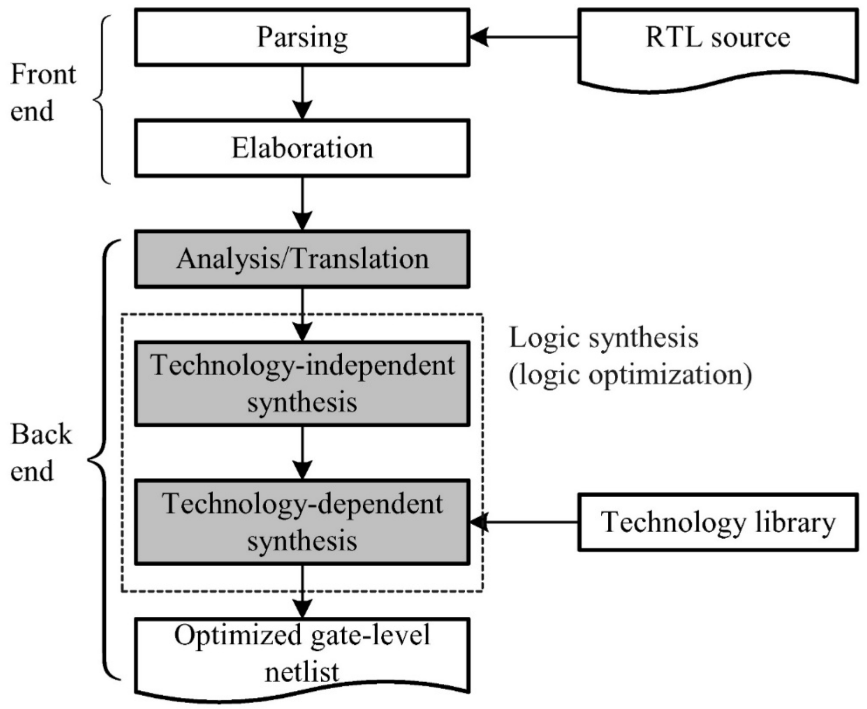
- Loop structures – Incorrectly synthesizable example



```
/tb_LWC/u_mie_a/clk      St0
/tb_LWC/u_mie_a/reset_n  St1
/tb_LWC/u_mie_a/data_a   0101   0101
/tb_LWC/u_mie_a/data_b   11     11
/tb_LWC/u_mie_a/total    74     0 11 22 33 44 55 66 77 88 99 110 121 132 143 154
/tb_LWC/u_mie_a/i        4      4
```

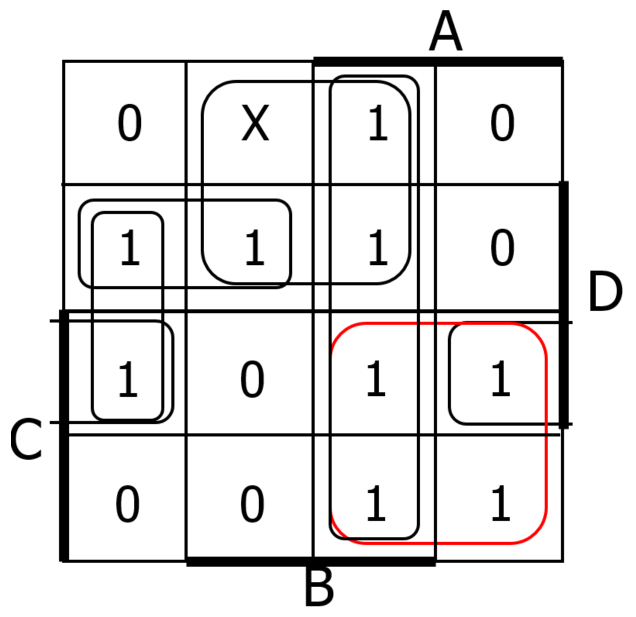
Architecture of Logic Synthesizers

- Back end
  - Analysis/translation
    - Include managing the design hierarchy
    - Extract the FSM
    - Explore resource sharing
  - Logic synthesis (**logic optimization**)
    - Create a new gate network computing the functions specified by a set of Boolean functions, one per primary output
    - Balance a number of concerns : functional metric and non-functional metric (area, power, delay)
  - Netlist generation


Front end
Parsing
RTL source
Elaboration
Analysis/Translation
Logic synthesis (logic optimization)
Technology-independent synthesis
Back end
Technology-dependent synthesis
Technology library
Optimized gate-level netlist

Logic Optimization

- Combinational circuit
  - e.g., Minimum cover selection on K-map



6 prime implicants:

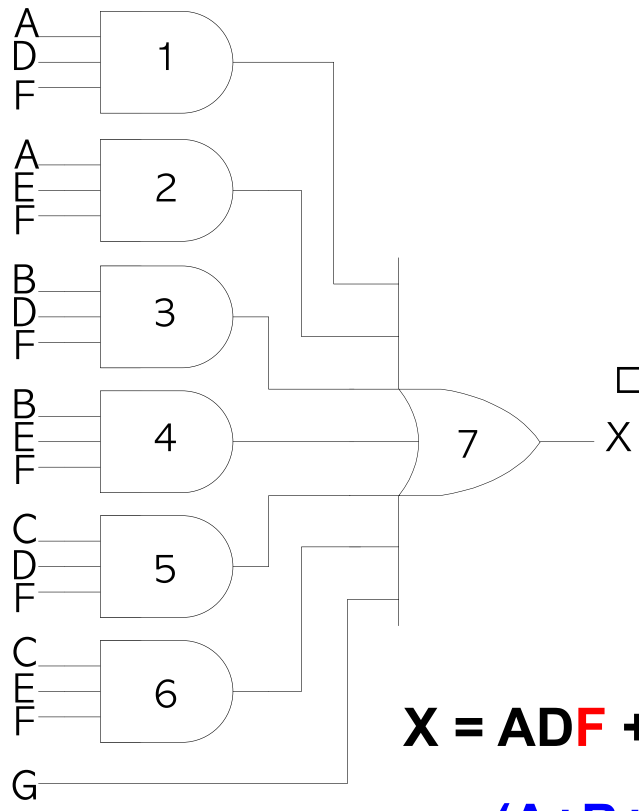
A'B'D, BC', AC, A'C'D, AB, B'CD

essential

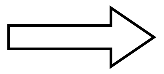
minimum cover: AC + BC' + A'B'D

Logic Optimization

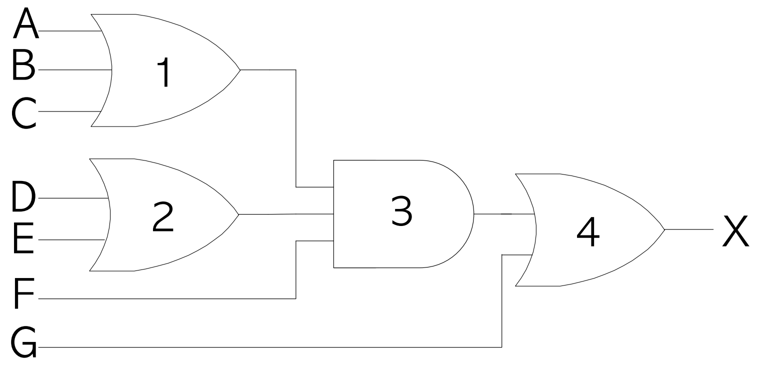
- Combinational circuit
  - e.g., multi-level logic



Two-level logic



More than two logic gates on any path



Multilevel logic

X = ADF + AEF + BDF + BEF + CDF + CEF + G

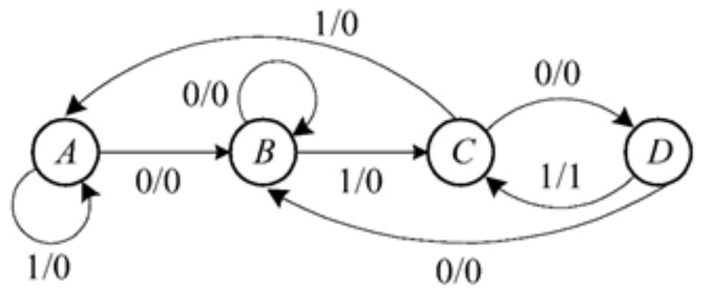
= (A+B+C) (D + E) F + G

❑ **How can we increase the logic level? (conversion into multi-level?)**

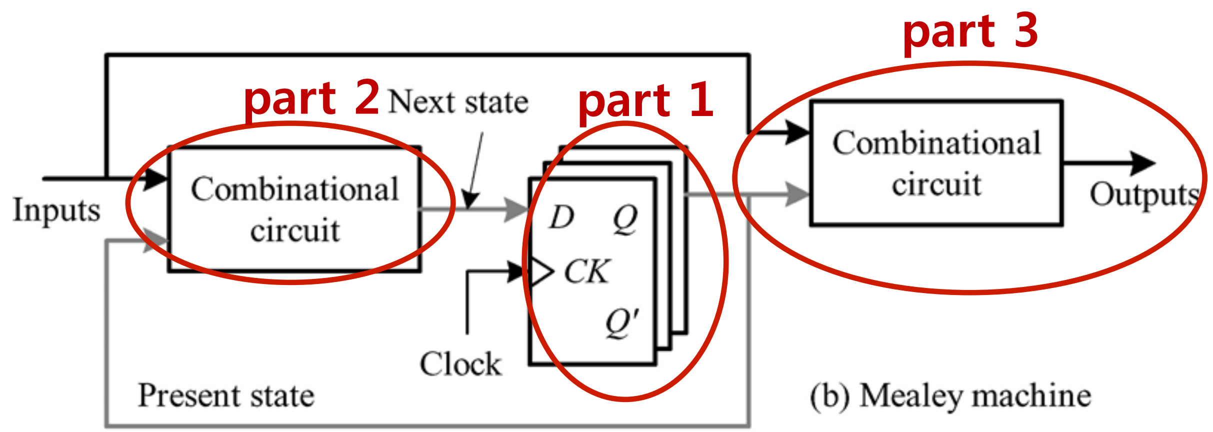
Factoring

Logic Optimization

- FSM optimization in sequential circuit
  - **State minimization in FSM + input/output/state encoding** + combinational circuit optimization



```
module sequence_detector_mealy (clk, reset_n, x, z);
   input                    clk, reset_n, x; output reg z;
   reg            [1:0]     present_state, next_state; // present state and next state
   parameter                A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// part 1:  initialize to state A and update present state register
   always @(posedge clk or negedge reset_n)
        if(!reset_n) present_state <= A; else present_state <= next_state;
// part 2: determine next state
   always @(present_state or x)
      case(present_state)
         A: if (x) next_state = A; else next_state = B;
         B: if (x) next_state = C; else next_state = B;
         C: if (x) next_state = A; else next_state = D;
         D: if (x) next_state = C; else next_state = B;
      endcase
// part 3: evaluate output z
   always @(present_state or x) // mealey machine
      case (present_state)
         A: if (x) z = 1'b0; else z = 1'b0;
         B: if (x) z = 1'b0; else z = 1'b0;
         C: if (x) z = 1'b0; else z = 1'b0;
         D: if (x) z = 1'b1; else z = 1'b0;
      endcase
endmodule
```



states → # F/Fs

Logic Optimization

- FSM optimization in sequential circuit
  - State Minimization
    - Fewer states require fewer state bits
    - Fewer bits require fewer logic equations
  - Encodings: State, Inputs, Outputs
    - State encoding with fewer bits has fewer equations to implement
      - However, each may be more complex
    - State encoding with more bits (e.g., one-hot) has simpler equations
      - Complexity directly related to complexity of state diagram
    - Input/output encoding may or may not be under designer control