# Chapter 11: System Design Methodology

Prof. Soo-Ik Chae

# Objectives

After completing this chapter, you will be able to:

- Describe the features of finite-state machines (FSMs)
- Understand how to model FSMs
- Describe basic structures of register-transfer designs
- Describe basic structures and features of algorithmic-state machine (ASM) charts
- Understand how to model ASMs
- Describe the features and structures of datapath and control designs
- Describe the realizations of RTL designs
- Understand the relationship between iterative logic and FSMs

# Finite-State Machines (FSMs)

- A finite-state machine (FSM) $M$ is a quintuple

  $M = (I, O, S, \delta, \lambda)$

  where $I$, $O$, and $S$ are finite, nonempty sets of inputs, outputs, and states, respectively.

  $\delta$: $I \times S \rightarrow S$ is the state transition function;

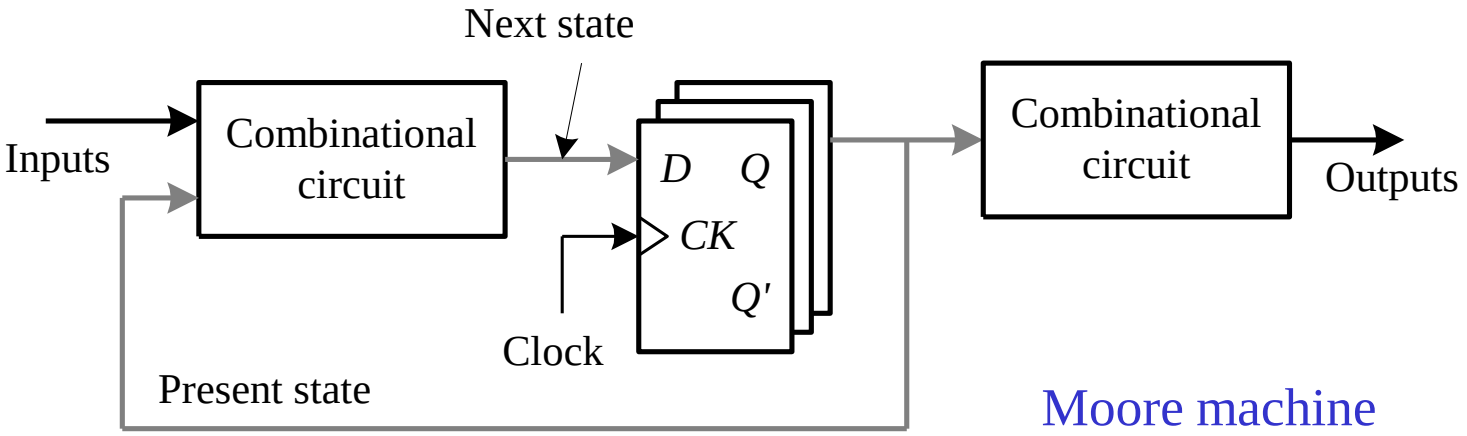
  $\lambda$ is the output function such that

  $\lambda : I \times S \rightarrow O$ for Mealy machine;

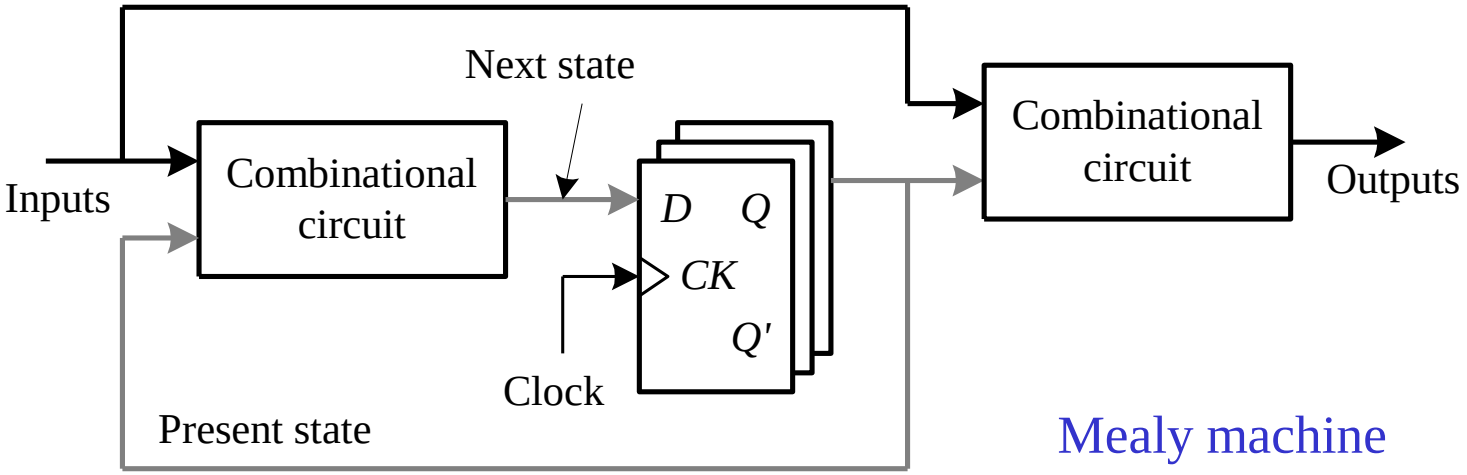
  $\lambda : S \rightarrow O$ for Moore machine.

- The design issues of a finite-state machine used in digital systems are to compute both the state transition and output functions from inputs and (present) states.

# Types of Sequential Circuits




Moore machine




Mealy machine

# State Encoding

- Common FSM encoding options:
  - One-hot code
  - Binary code
  - Gray code
  - Random code

| State | Binary | Gray | One hot |
|---|---|---|---|
| *A* | 00 | 00 | 1000 |
| *B* | 01 | 01 | 0100 |
| *C* | 10 | 11 | 0010 |
| *D* | 11 | 10 | 0001 |

One-hot encoding is usually used in FPGA-based design, because there are a lot of registers in theses devices.

# How to Realize an FSM?

- Explicit fsm: States are declared explicitly.
  - It is so easy to write clearly and correctly.
  - Every synthesis tool supports this type FSM.
  - Almost all FSMs are written in this type.
- Implicit fsm: States are not declared explicitly. The synthesis tools infer the state from the activity within a cyclic (always …) behavior.
  - It is so hard to write clearly and correctly.
  - Not every synthesis tool supports this type FSM.

# (Explicit) FSM Realization

- An explicit fsm is usually called FSM for short.
  - Two major realization issues:
    - state-register declaration and update
    - the computation of both next-state and output functions
- State register can be declared as
  - one register: harder to follow and model.
  - two registers: easier to follow and model.
- Output and next state functions:
  - continuous assignment
  - function
  - always block
  - a combination of the above

# FSM Modeling Styles

- Style 1: one state register
  - part 1: initialize and update the state register

```
always @(posedge clk or negedge reset_n)
    if    (!reset_n) state <= A;
    else  state <= next_state (state, x) ;
```

  - part 2: determine next state using function

```
function next_state (input present_state, x)
    case (present_state ) …
endcase endfunction
```

  - part 3: determine output function

```
always @(state or x) case (state) … or
    assignment statements
```

# FSM Modeling Styles

- Style 2: two state registers
  - part 1: initialize and update the 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

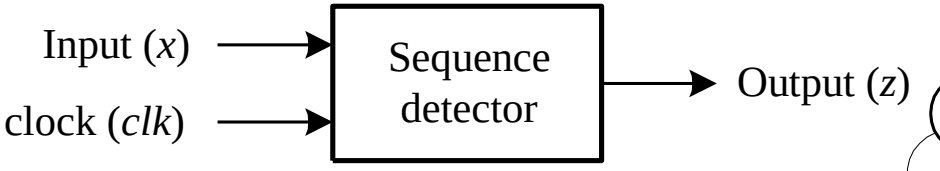
```
always @(present_state or x) case (present_state ) … or
assignment statements
```

  - part 3: determine output function

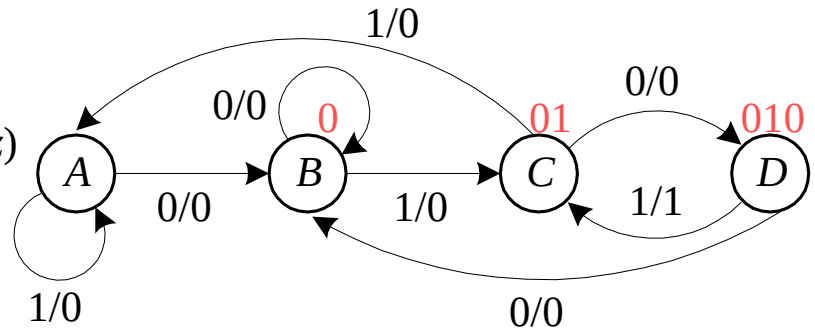
```
always @(present_state or x) case (present_state ) … or
assignment statements
```

# An FSM Example --- A 0101 Sequence Detector

- Design a finite-state machine to detect the pattern 0101 in the input sequence *x*. Assume that overlapping pattern is allowed.

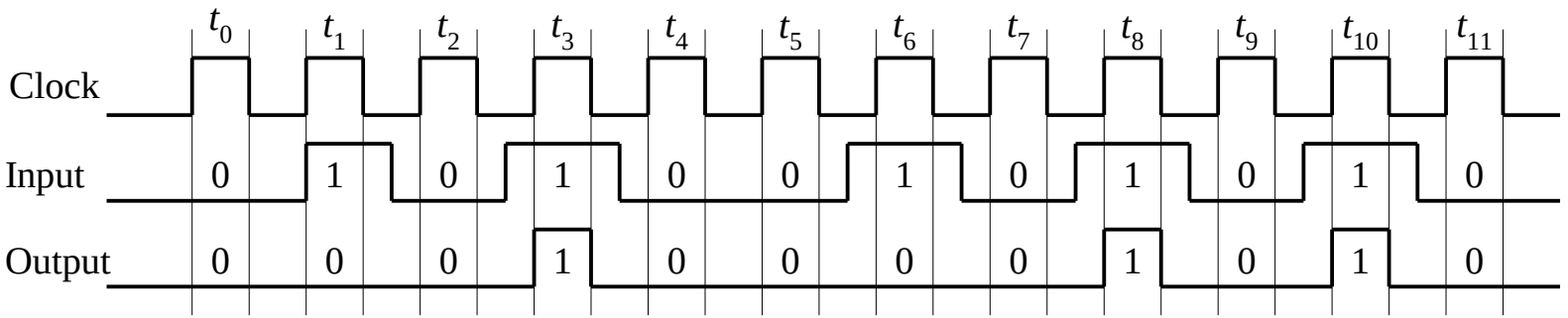



Block diagram




State diagram

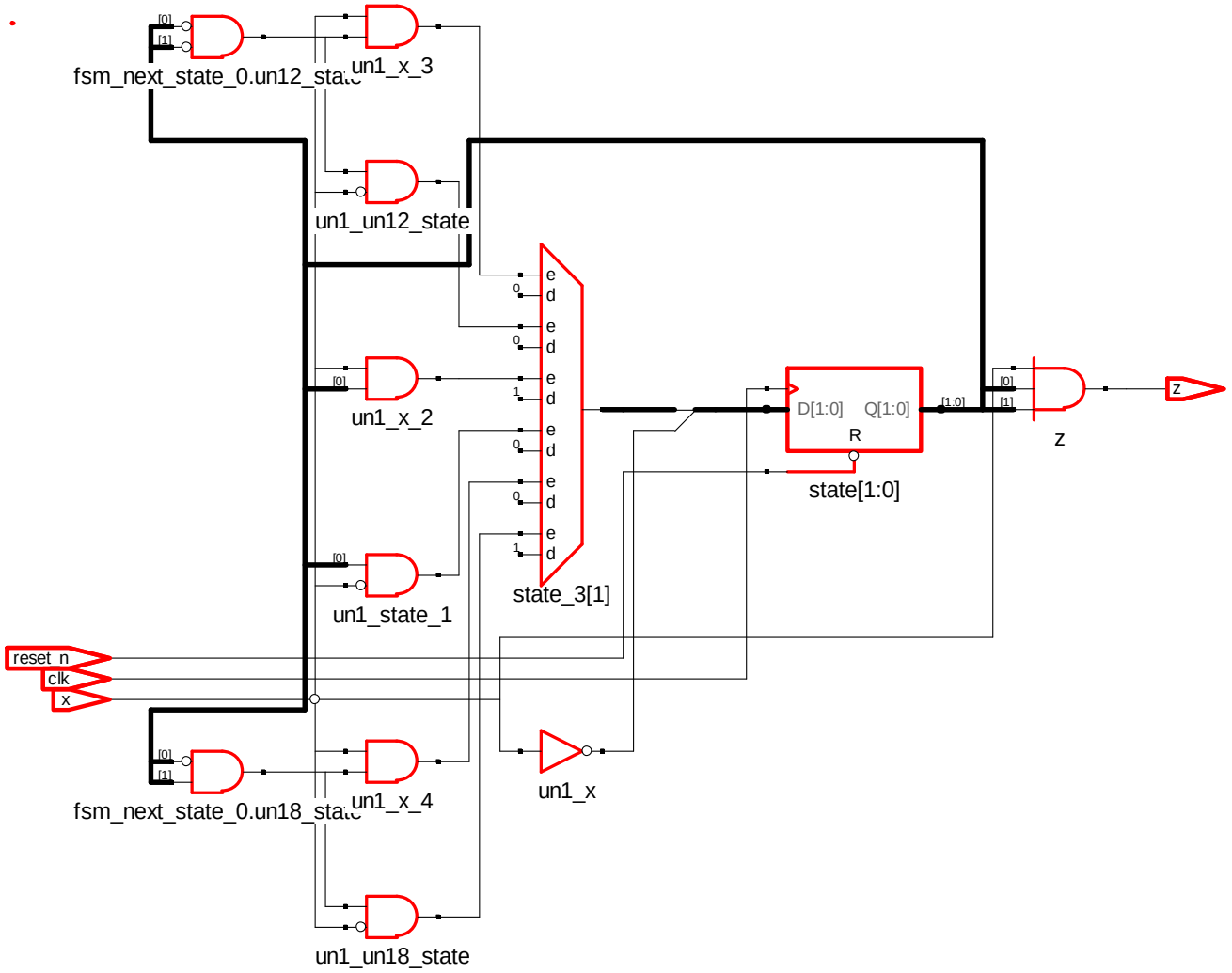
Timing chart



# An FSM Example --- The Sequence Detector: Style 1a

```
// a behavioral description of 0101 sequence detector – style 1a
// Mealy machine example --- Using only state register and one always block.
module sequence_detector_mealy (clk, reset_n, x, z);
input  x, clk, reset_n;
output wire z;
reg  [1:0] state;
parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// the body of sequence detector --- Using only one always block.
always @(posedge clk or negedge reset_n) begin
   if (!reset_n) state <= A; else state <= fsm_next_state(state, x); end
function [1:0] fsm_next_state (input [1:0] present_state, input x);
reg [1:0] next_state;
begin 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
   fsm_next_state = next_state;
end endfunction
assign z = (x == 1 && state == D);
endmodule
```

# An FSM Example --- The Sequence Detector: Style 1a



# An FSM Example --- The Sequence Detector: style 1b

```
// a b ehavioral description of 0101 sequence detector: style 1b
// Mealy machine example --- Using only one state register and two always blocks.
module sequence_detector_mealy (clk, reset_n, x, z);
input  x, clk, reset_n;
output reg z;
// Local declaration
reg  [1:0] state;
parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// the body of sequence detector --- Using two always blocks.
always @(posedge clk or negedge reset_n) begin
  if (!reset_n) state <= A; else state <= fsm_next_state(state, x); end
function [1:0] fsm_next_state (input [1:0] present_state, input x);
  …. Same as that of style 1a
end endfunction
// evaluate output function z
always @(state or x) case (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
```

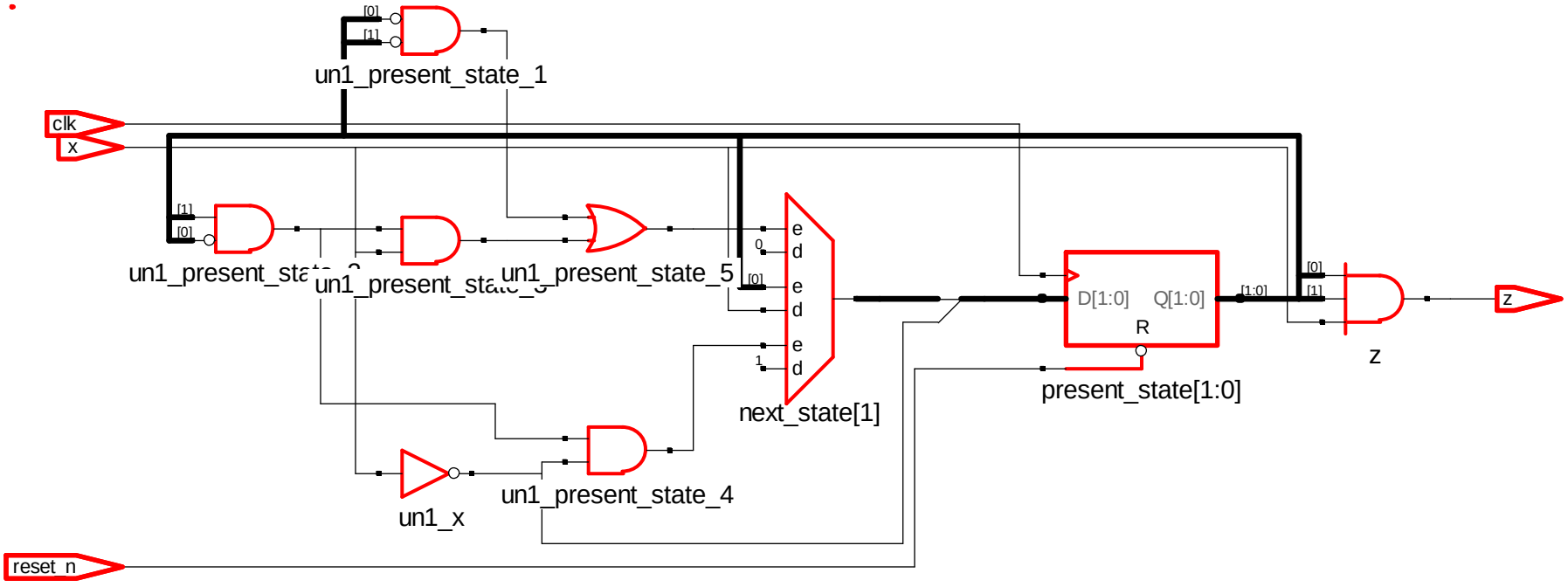
# An FSM Example --- The Sequence Detector: style 2

```
// a behavioral description of 0101 sequence detector: style 2
// Mealy machine example --- using two state registers and three always blocks.
module sequence_detector_mealy (clk, reset_n, x, z);
input  x, clk, reset_n;
output z;
// local declaration
reg [1:0] present_state, next_state; // present state and next state
reg    z;
parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// initialize to state A and update state register
always @(posedge clk or negedge reset_n)
    if     (!reset_n) present_state <= A;
    else  present_state <= next_state;          // update present state
```

# An FSM Example --- The Sequence Detector: style 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
// evaluate output function z
always @(present_state or x)
   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
```

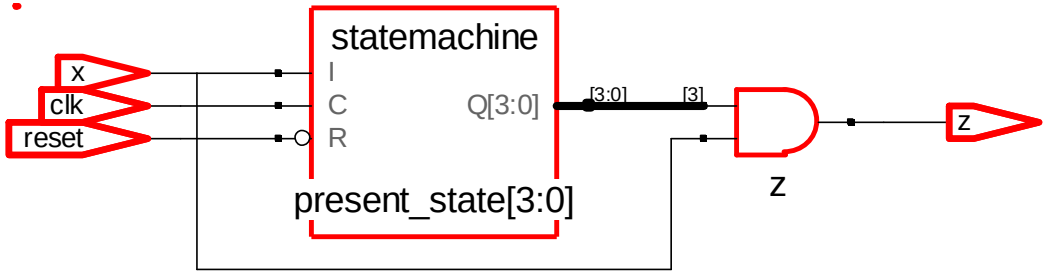
# An FSM Example --- The Sequence Detector: style 2



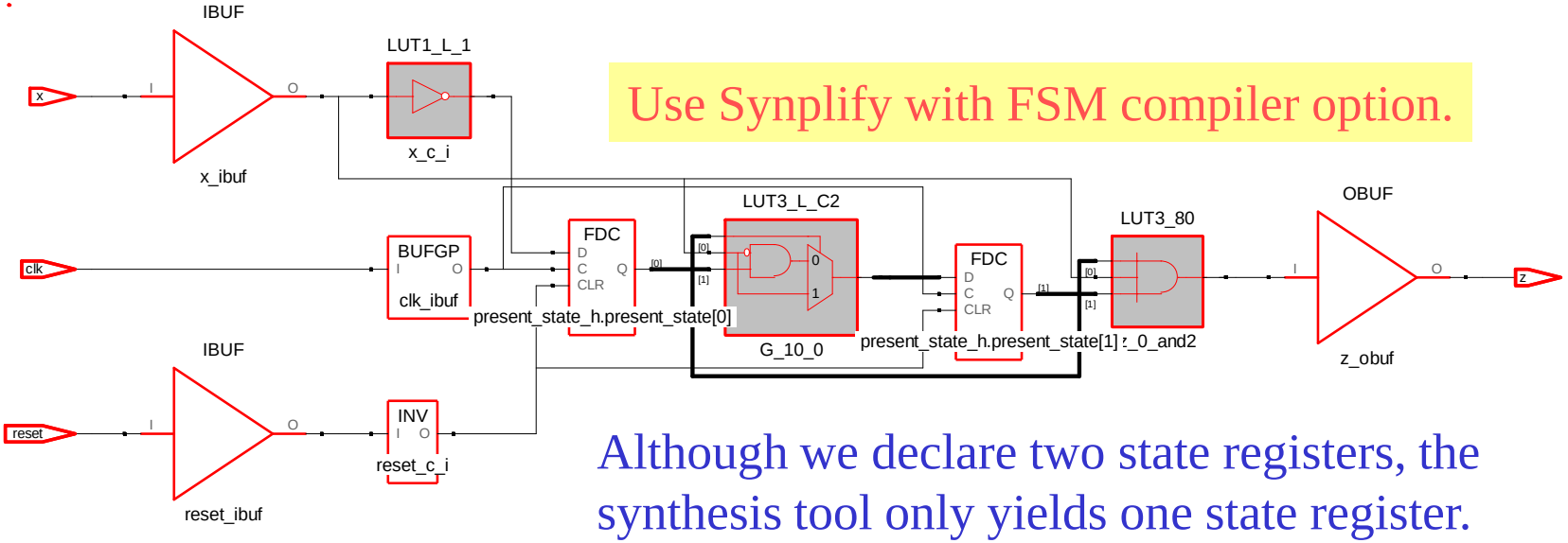
# An FSM Example --- The Sequence Detector

- Both modeling styles, using one state register or two state registers, yield the same circuit as shown below.

RTL result




- Gate-level result




Use Synplify with FSM compiler option.

Although we declare two state registers, the synthesis tool only yields one state register.

# Coding Style

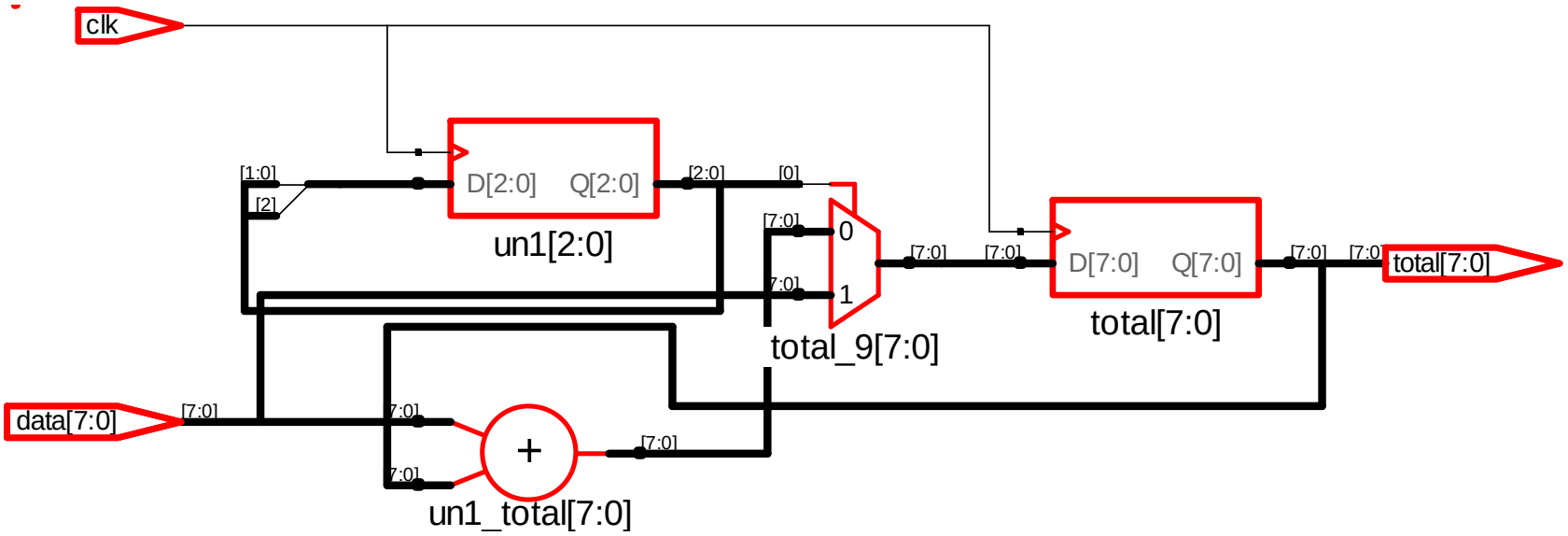
- State name should be described using parameters: parameter or localparam.
- Combinational logic for computing the next state should be separated from the state registers, i.e., using its own always block or a function.
- The combinational logic of the output function should be implemented with a case statement.

# Example: An Implicit FSMs

- Due to the difficulty of writing a correct implicit FSM program, it is not suggested to use this method to realize an FSM circuit.

```
// an implicit FSM example
module sum_3data_implicit(clk, data, total);
parameter N = 8;
input   clk;
input   [N-1:0] data;
output [N-1:0] total;
reg      [N-1:0] total;
// we do not declare state registers.
always begin
    @(posedge clk) total <= data;
    @(posedge clk) total <= total + data;
    @(posedge clk) total <= total + data;
end
endmodule
```

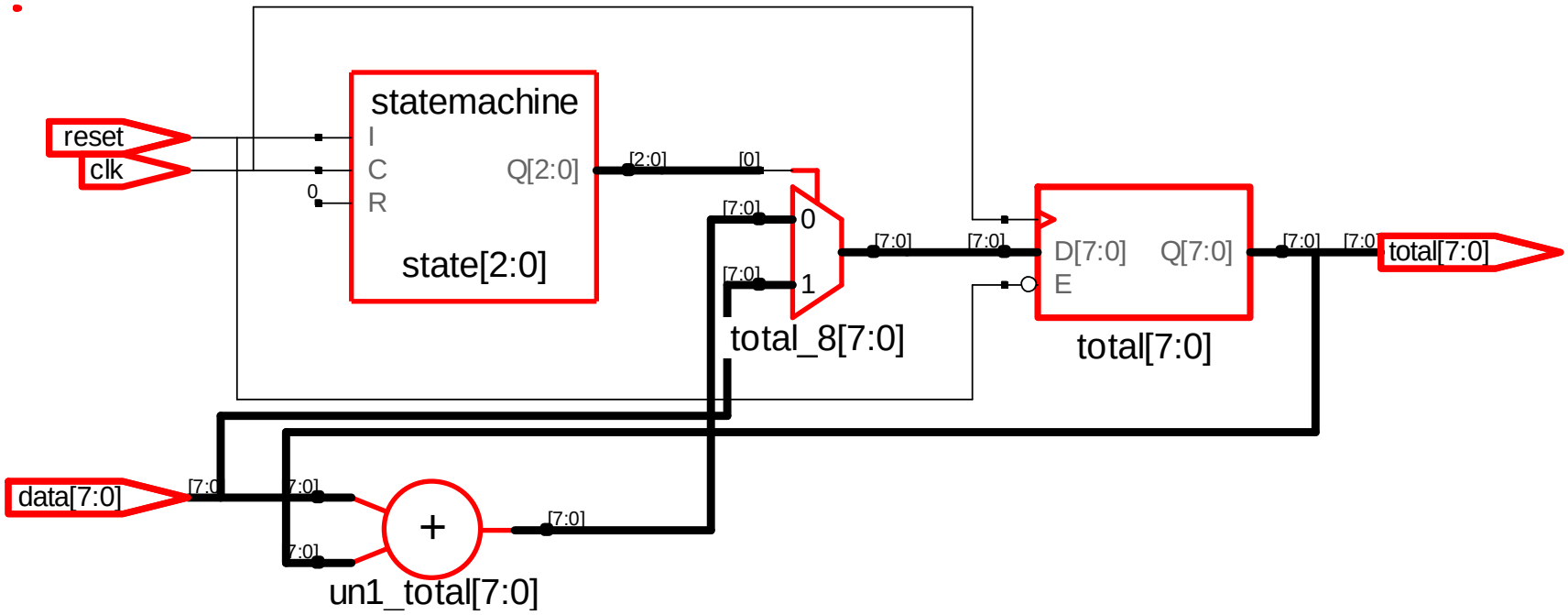
# Example: An Implicit FSMs



# Example: An Explicit FSMs

```
// an explicit FSM example
module sum_3data_explicit(clk, reset, data, total);
parameter N = 8;
input   clk, reset;
input   [N-1:0] data;
output [N-1:0] total;
reg      [N-1:0] total;
reg     [1:0]   state;     // declare state registers explicitly
parameter A = 2'b00, B =2'b01, C = 2'b10;
always @(posedge clk)
  if (reset) state <= A; else
  case (state)
     A: begin total <= data; state <= B; end
     B: begin total <= total + data; state <= C; end
     C: begin total <= total + data; state <= A; end
  endcase
endmodule
```

# Example: An Explicit FSMs




Both implicit and explicit fsm have the same synthesized results.

# Register-Transfer-Level Design

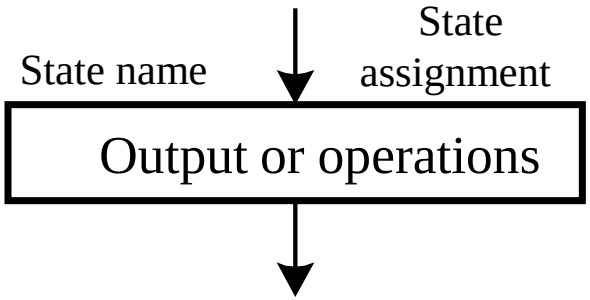
- Features of register-transfer-level (RTL) designs
  1. are sequential machines.
  2. are structural.
  3. concentrate on functionality, not details of logic design.
- Two types of register-transfer-level design are
  - ASM (algorithmic-state machine) chart
  - Datapath and controller (DP + CU)

# ASM Charts (or SM charts)

- ASM (algorithmic state machine) charts
  - specify RTL operations on the per-cycle basis.
  - show clearly the flow of control from state to state.
  - are very suitable for data path-controller architectures.
- An ASM chart is composed of
  - State block
  - Decision block
  - Conditional output block

# ASM State Blocks

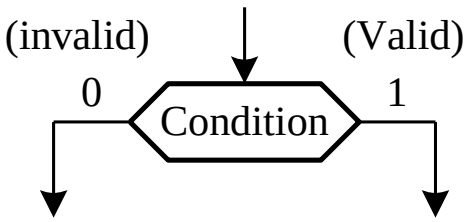
- A state block
  - specifies a machine state and a set of unconditional RTL operations associated with the state.
  - may execute as many actions as you want and all actions in a state block occur in parallel.
  - occupies a clock period along with its related operations.



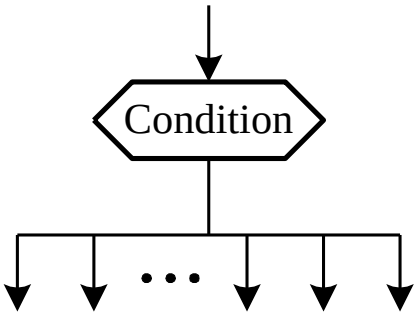
# Decision Blocks

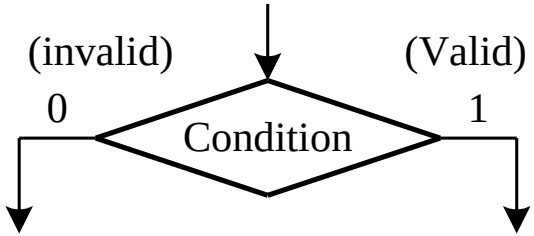
- A decision block
  - describes the condition under which ASM will execute specific actions.
  - selects the next state based on the value of primary input or present state.
  - can be drawn in either two-way selection or multi-way selection.

Two-way selection



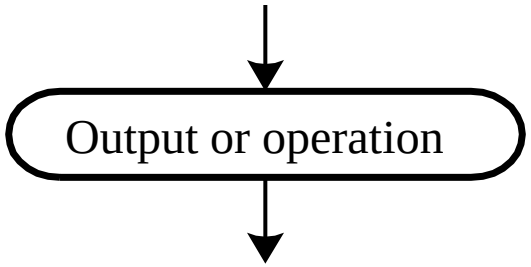

Multi-way selection



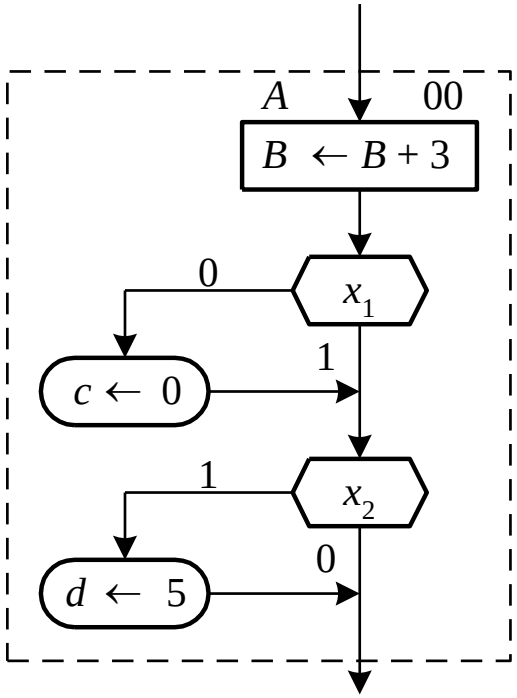
# Conditional Output Blocks

- A conditional output block
  - describes the RTL operations executed under conditions specified by one or more decision blocks.
  - receives signals from the output of a decision block or the other conditional output block.
  - can only evaluate present state or primary input value on present cycle.

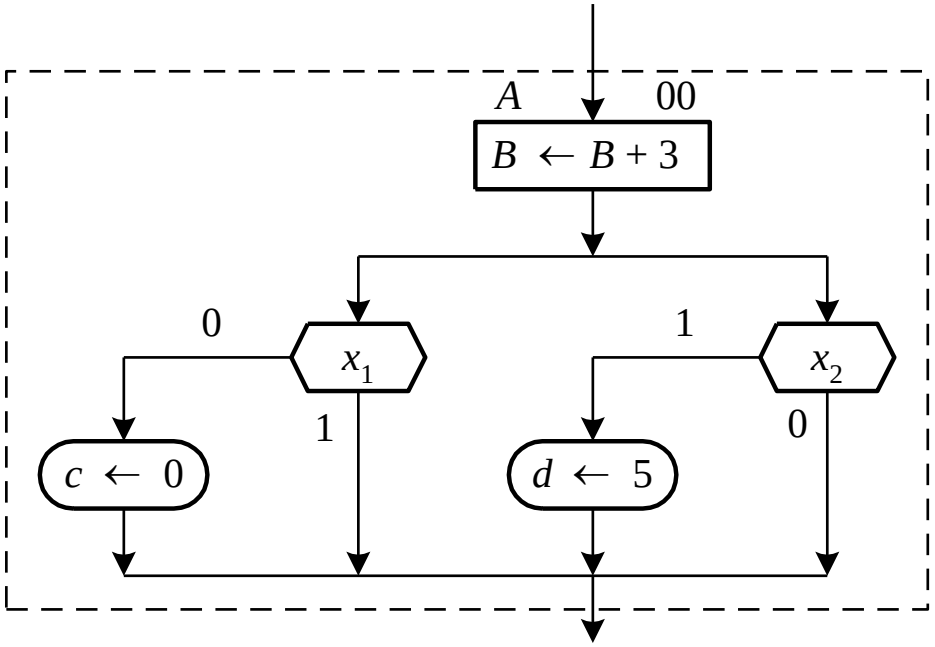


# ASM Blocks

- An ASM block has exactly one entrance path and one or more exit paths.




Serial testing



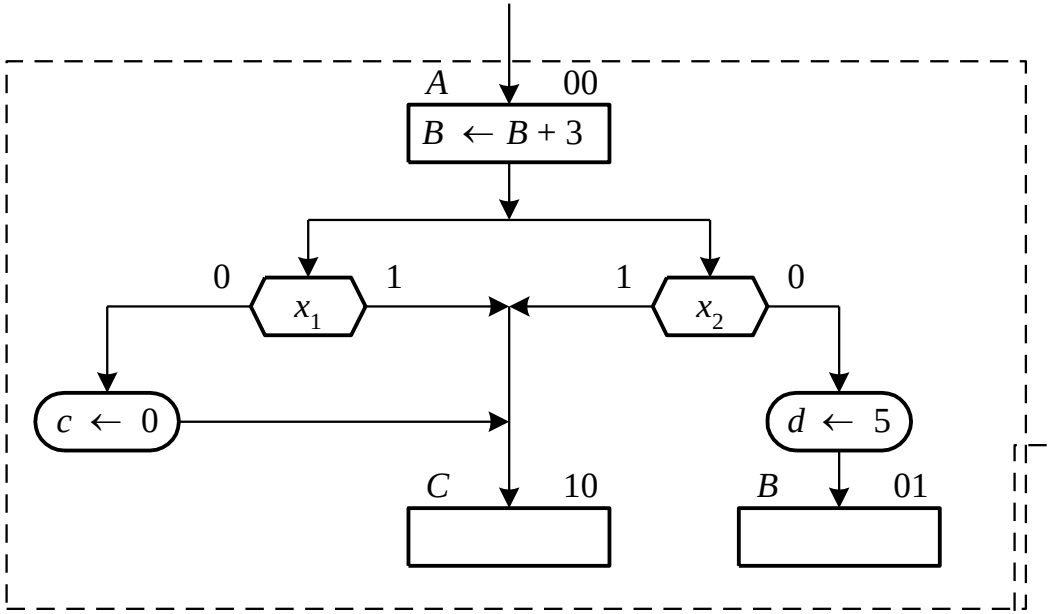

Parallel testing

# ASM Charts --- ASM Blocks

- ASM blocks
  - An ASM block contains one state block and a serial-parallel network of decision blocks and conditional output blocks.
  - Each ASM block describes the operations executed in one state.
- The basic rules for constructing an ASM chart
  1. Each state and its associated set of conditions must define a unique next state.
  2. Every path of the network of conditions must terminate at a next state.
  3. There cannot exist any loop in the network of conditions.

# Invalid ASM Blocks

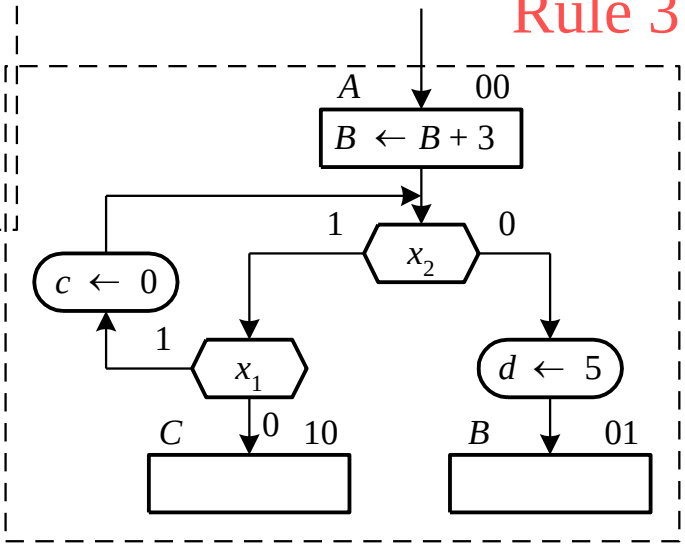
- Undefined next state: why? Try to explain it!




Rule 1? (x1,x2)= (0,0)

Undefined exit path:
why? Try to explain it!

Rule 3?



# ASM Modeling Styles

- Style 1a: one state register and one always block. (Not a good style!!!)
  - part 1: initialize, determine, update the state register, and determine RTL operations.

```
always @(posedge clk or negedge start_n)
    if (!start_n) state <= A; else  state <=  …;
```

- Style 1b: one state register and two always blocks
  - part 1: initialize, determine, and update the state register.

```
always @(posedge clk or negedge start_n)
    if (!start_n) state <= A; else  state <=  …;
```

  - part2: determine RTL operations

```
always @(posedge clk) case (state ) … or
assignment statements
```

# ASM Modeling Styles

- Style 2: two state registers and three always blocks (the best style !!!!)
  - part 1: initialize and update the state register

    always @(posedge clk or negedge start_n)

    if (!start_n) present_state <= A; else present_state <= next_state;
  - part 2: determine next state

    always @(present_state or x) case (present_state ) … or

    assignment statements
  - part 3: determine RTL operations

    always @(posedge clk) case (present_state ) … or

    assignment statements

# An ASM Example --- Booth Serial Multiplier

- Problem Specification: Booth multiplication algorithm

  Assume that: $X = x_{n-1}x_{n-2}\cdots x_1x_0$

  $Y = y_{n-1}y_{n-2}\cdots y_1y_0$

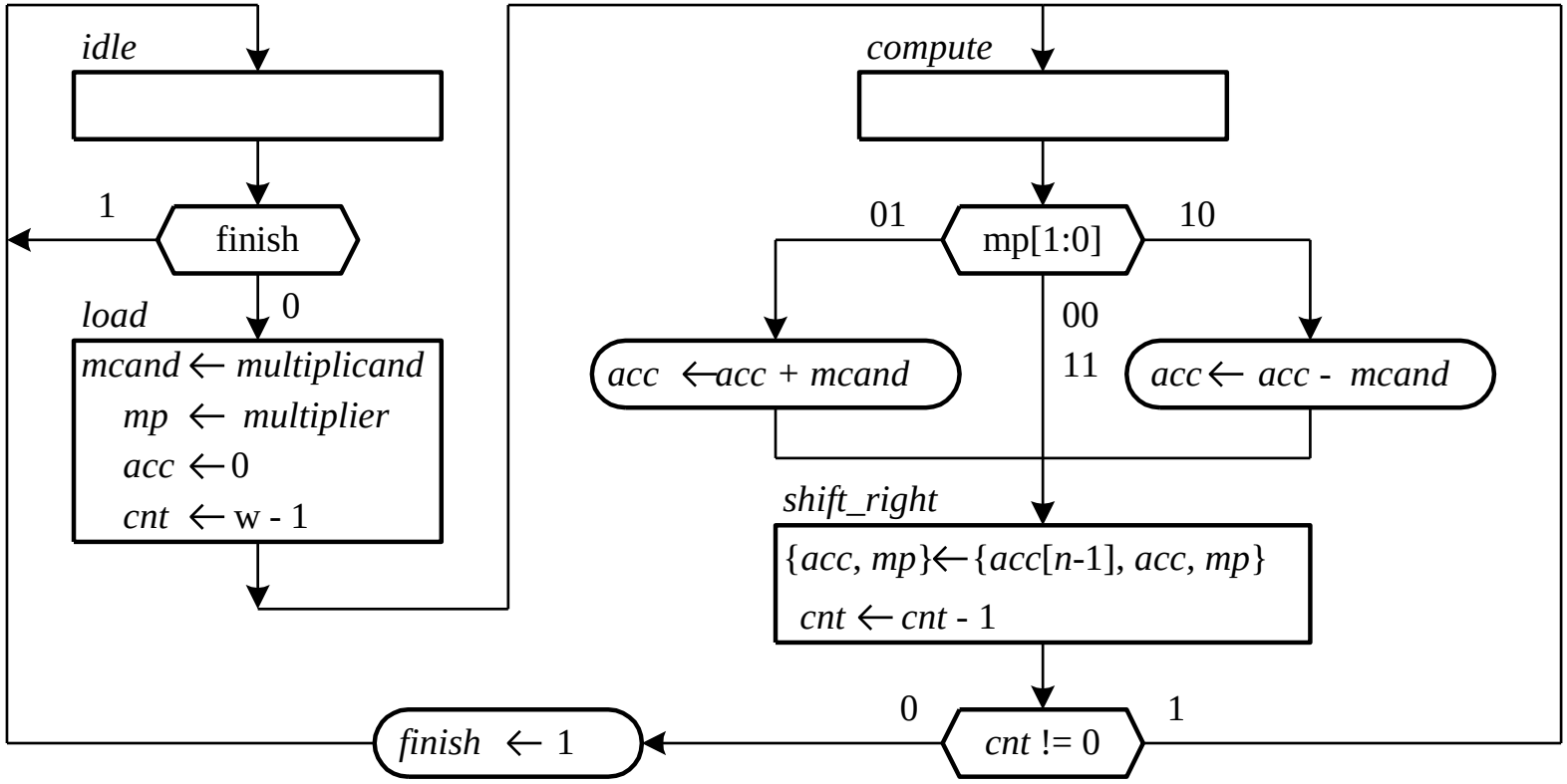
- At *i*th step:
  - Add 0 to partial product $P$ if $x_i x_{i-1} = 00$
  - Add $Y$ to partial product $P$ if $x_i x_{i-1} = 01$
  - Subtract $Y$ from partial product $P$ if $x_i x_{i-1} = 10$
  - Add 0 to partial product $P$ if $x_i x_{i-1} = 11$

# An ASM Example --- Booth Serial Multiplier

| | | *acc* | *mp* | *mp*(0) | *cnt* | |
|---|---|---|---|---|---|---|
| *mp* = 77H | | 0 0 0 0 0 0 0 0 | 0 1 1 1 0 1 1 1 | 0 | 7 | *acc* ⟵ *acc* - *mcand* |
| *mcand* = 13H | - | 0 0 0 1 0 0 1 1 | | | | |
| | | 1 1 1 0 1 1 0 1 | | | | |
| | | 1 1 1 1 0 1 1 0 | 1 0 1 1 1 0 1 1 | 1 | 7 | right shift *acc:mp* |
| | | 1 1 1 1 1 0 1 1 | 0 1 0 1 1 1 0 1 | 1 | 6 | right shift *acc:mp* |
| | | 1 1 1 1 1 1 0 1 | 1 0 1 0 1 1 1 0 | 1 | 5 | right shift *acc:mp* |
| | + | 0 0 0 1 0 0 1 1 | | | | *acc* ⟵ *acc* +*mcand* |
| | | 0 0 0 1 0 0 0 0 | | | | |
| | | 0 0 0 0 1 0 0 0 | 0 1 0 1 0 1 1 1 | 0 | 4 | right shift *acc:mp* |
| | - | 0 0 0 1 0 0 1 1 | | | | |
| | | 1 1 1 1 0 1 0 1 | | | | |
| | | 1 1 1 1 1 0 1 0 | 1 0 1 0 1 0 1 1 | 1 | 3 | right shift *acc:mp* |
| | | 1 1 1 1 1 1 0 1 | 0 1 0 1 0 1 0 1 | 1 | 2 | right shift *acc:mp* |
| | | 1 1 1 1 1 1 1 0 | 1 0 1 0 1 0 1 0 | 1 | 1 | right shift *acc:mp* |
| | + | 0 0 0 1 0 0 1 1 | | | | *acc* ⟵ *acc* +*mcand* |
| | | 0 0 0 1 0 0 0 1 | | | | |
| 08D5H | ⟶ | 0 0 0 0 1 0 0 0 | 1 1 0 1 0 1 0 1 | 0 | 0 | right shift *acc:mp* |

# An ASM Example --- Booth Serial Multiplier

- ASM chart approach



# An ASM Example --- Booth Serial Multiplier: Style 2 (1/3)

```
// Booth multiplier: style 2 --- using three always blocks
module booth (clk, start_n, multiplier, multiplicand, product, finish);
parameter W = 8;  // word size
parameter N = 3;   // N = log2(W)
input  clk, start_n;
input  [W-1:0]   multiplier, multiplicand;
output [2*W-1:0] product;
wire   [2*W-1:0] product;
reg    [W-1:0]   mcand, acc;
reg    [N-1:0]    cnt;
reg    [W:0]      mp;       // one extra bit
reg    [1:0]      present, next;
output reg        finish;    // finish flag
parameter idle = 2'b00, load = 2'b01, compute = 2'b10, shift_right = 2'b11;
// the body of the w-bit booth multiplier
// initialize to state idle
always @(posedge clk or negedge start_n)
    if (!start_n) present <= idle; else  present <= next;
```

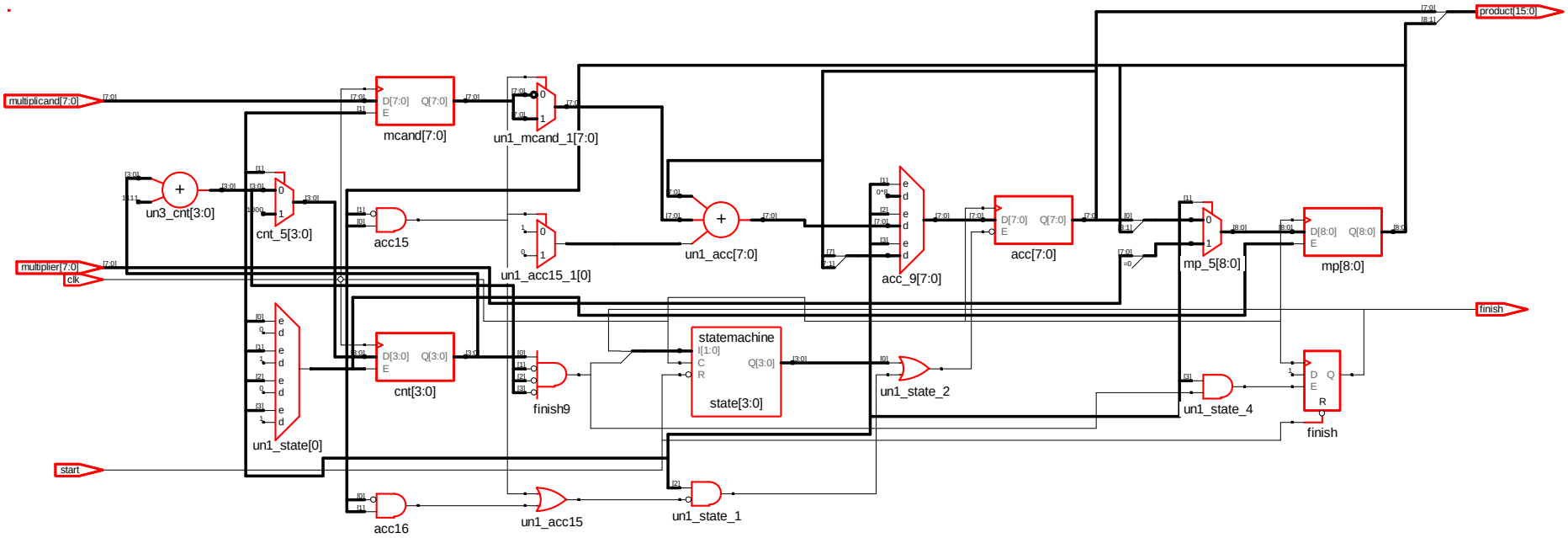
# An ASM Example --- Booth Serial Multiplier: Style 2 (2/3)

```
// determine next state
always @(*) case (present)
      idle:       if (!finish) next = load; else next = idle;
      load:       next = compute;
      compute:    next = shiftright;
      shiftright: if (cnt == 0) next = idle; else next = compute;
   endcase
// execute RTL operations
always @(posedge clk) begin
    if (!start_n) finish <= 0;
    case (present)
        idle: ;
        load: begin
            mp <= {multiplier,1'b0};
            mcand  <= multiplicand;
            acc  <= 0; cnt <= W - 1;
        end
```

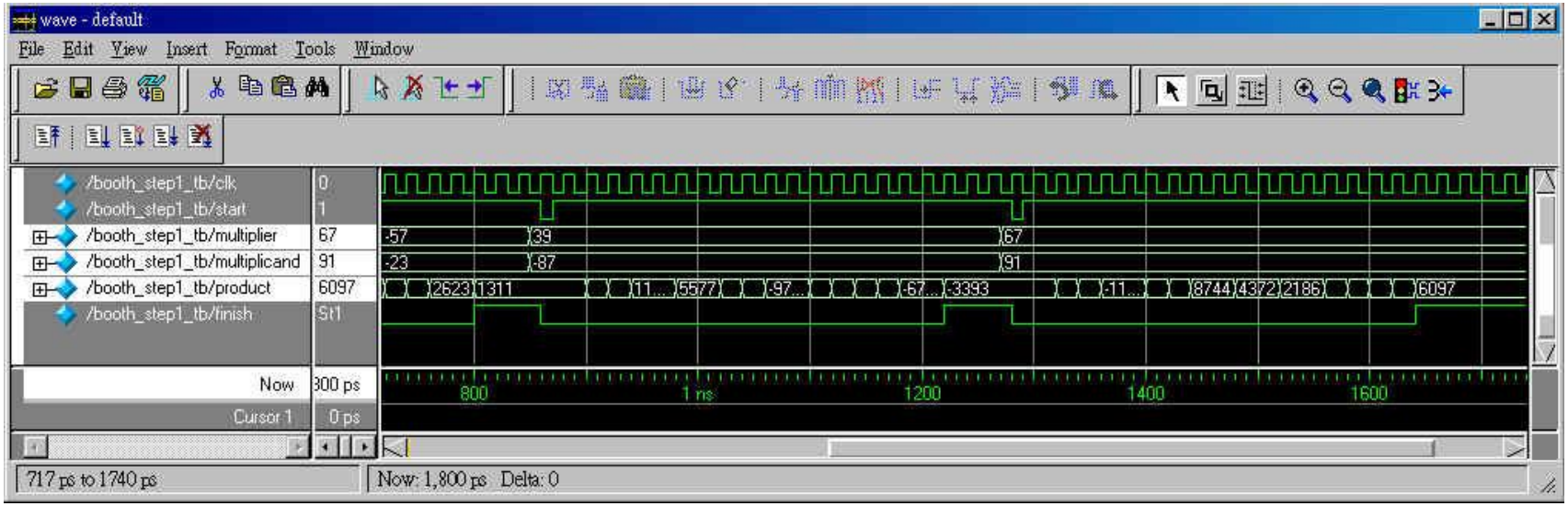
# An ASM Example --- Booth Serial Multiplier: Style 2 (3/3)

```
    compute: begin
        case (mp[1:0])
            2'b01: acc <= acc + mcand;
            2'b10: acc <= acc - mcand;
            default: ; // do nothing
        endcase end
    shift_right: begin
        {acc, mp} <= {acc[W-1], acc, mp[W:1]}; // arithmetic shift right
        cnt <= cnt - 1;
        if (cnt == 0) finish <= 1;
    end
    endcase
end
assign product = {acc, mp[W:1]};
endmodule
```

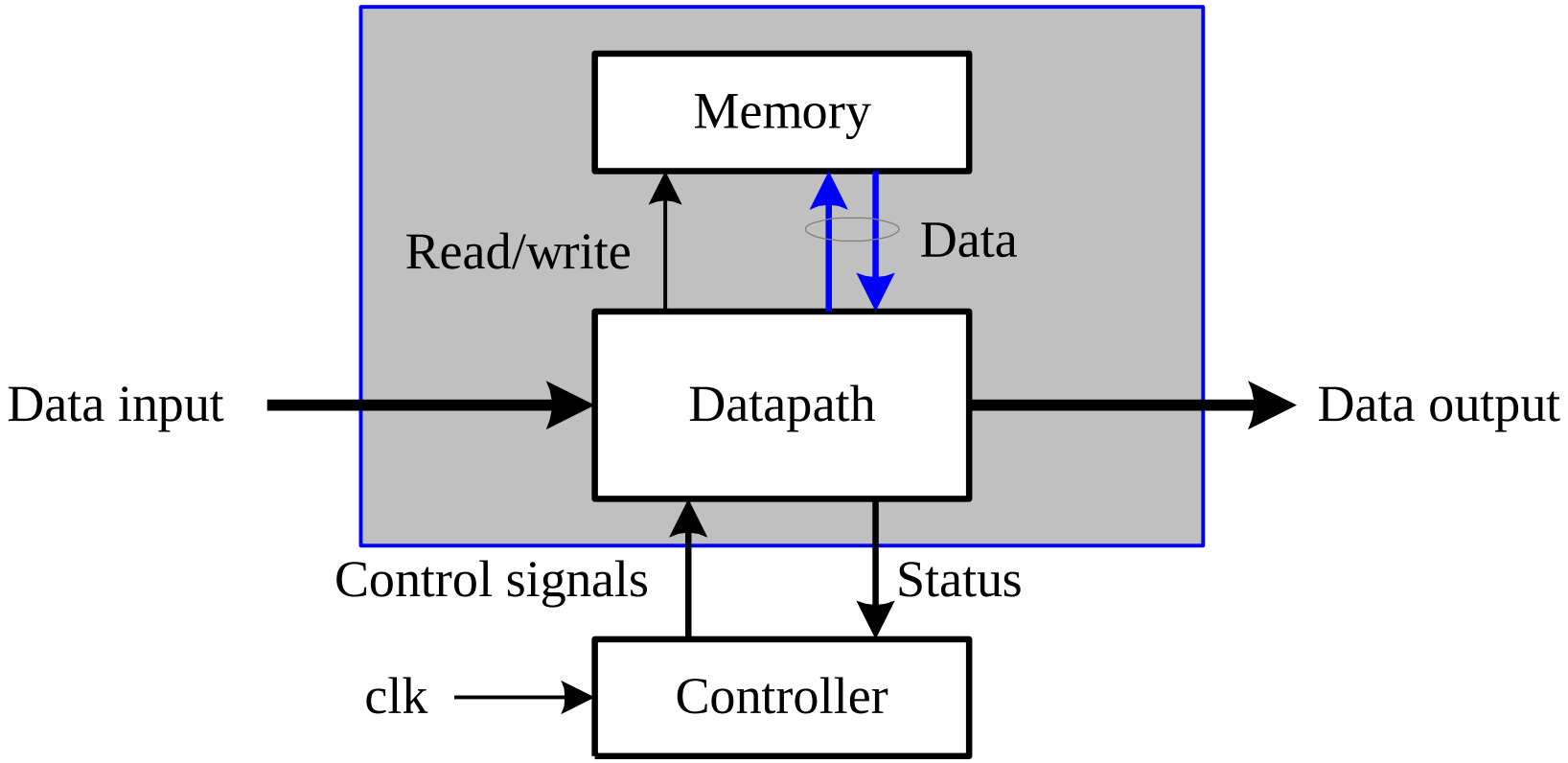
# An ASM Example ---Booth Serial Multiplier



# An ASM Example ---Booth Serial Multiplier



# dp+cu Architecture Design



# dp+cu Architecture Design

- A digital system can be considered as a system composed of three major parts:
  - data path performs all operations that are required in the system.
  - memory temporarily stores the data used and generated by data path unit.
  - control unit controls and schedules all operations performed by the data path unit.

# A dp+cu Approach --- Booth Serial Multiplier

- Datapath and controller units can be easily derived from ASM chart (ASM modeling style 2).
  - Datapath part corresponds to the registers and function units in RTL operation block (part 3).
  - Controller corresponds to the first and second always blocks (part 1 and 2), which determines the next state function, and the control signals in the RTL operation block (part 3).

# A Three-Step Paradigm of DP+CU design

1. Model the design (usually described by an ASM chart) using any modeling style described above as a single module.
2. Extract the datapath from the module and construct it as another independent module.
3. Extract control-unit module and construct top module.
   - Convert the first module to the control module
   - Add a top module that instantiates both datapath and control-unit

# An ASM Example --- Step 1: modeling the problem

```
module counter1 (clk, start_n, data_a, data_b, total, finish);
parameter W = 8;  // word size
parameter N = 3;   // N = log2(W)
input  clk, start_n;
input  [W-1:0]   data_a, data_b;
output reg [W-1:0] total;
reg     [N-1:0]    cnt;
reg     [1:0]       present, next;
output reg        finish;    // finish flag
localparam idle = 1'b0, add = 1'b1;
// the body of the w-bit booth multiplier
// part 1: initialize to state idle
always @(posedge clk or negedge start_n)
    if (!start_n) present <= idle; else  present <= next;
// part 2: determine next state
always @(*) case (present)
      idle:         if (!finish) next = add; else next = idle;
      add:         if (cnt == 0) next = idle; else next = add;
   endcase
```

# An ASM Example --- Step 1

```
// part 3: execute RTL operations
always @(posedge clk or negedge start_n) begin
    if (!start_n) begin finish <= 0; total <= 0; end
    case (present)
        idle: cnt <= N - 1;
        add: begin
            if (data[cnt] == 1) total <=  total + data_b;
            cnt <= cnt  - 1;
            if (cnt == 0) finish <= 1; end
     endcase   end
endmodule
```

# An ASM Example --- Step 2: extract datapath

```
module counter2 (clk, start_n, data_a, data_b, total, finish);
parameter W = 8;  // word size
parameter N = 3;   // N = log2(W)
input  clk, start_n;
input  [W-1:0]   data_a, data_b;
output reg [W-1:0] total;
reg     [N-1:0]    cnt;
reg     [1:0]       present, next;
output reg        finish;    // finish flag
localparam idle = 1'b0, add = 1'b1;
datapath #(8)  dp (.clk(clk), .acc_reset_n(start_n), .acc_load(load),
                        .acc_data_b(data_b), .acc_total(total));
// part 1: initialize to state idle: the same as in step 1
always @(posedge clk or negedge start_n)
    if (!start_n) present <= idle: else  present <= next;
// part 2: determine next state:the same as in step 1
always @(*) case (present)
      idle:         if (!finish) next = add; else next = idle;
      add:          if (cnt == 0) next = idle; else next = add;
   endcase
```

# An ASM Example --- Step 2

```
// part 3: execute RTL operations: extracted the datapath out form the part 3

always @(posedge clk or negedge start_n) begin
    if (!start_n) begin finish <= 0; total <= 0; end
    case (present)
        idle: cnt <= N - 1;
        add: begin
            if (data[cnt] == 1) total <=  total + data_b;
            cnt <= cnt  - 1;
            if (cnt == 0) finish <= 1; end
     endcase   end
assign load = (present == add) && (data[cnt] == 1);
endmodule
```

# An ASM Example --- Step 2

```
// define the datapath module
module datapath (clk, acc_reset_n, acc_load, acc_data_b, acc_total);
paramenter N = 8;
input clk, acc_reset_n, acc_load;
input [N-1:0] acc_data_b;
output reg [N-1:0] acc_total;
always @(posedge clk or negedge acc_reset_n)
   if ( !acc_reset_n) acc_total <= 0;
   else if (acc_load) acc_total <= acc_total + acc_data_b;
endmodule
```

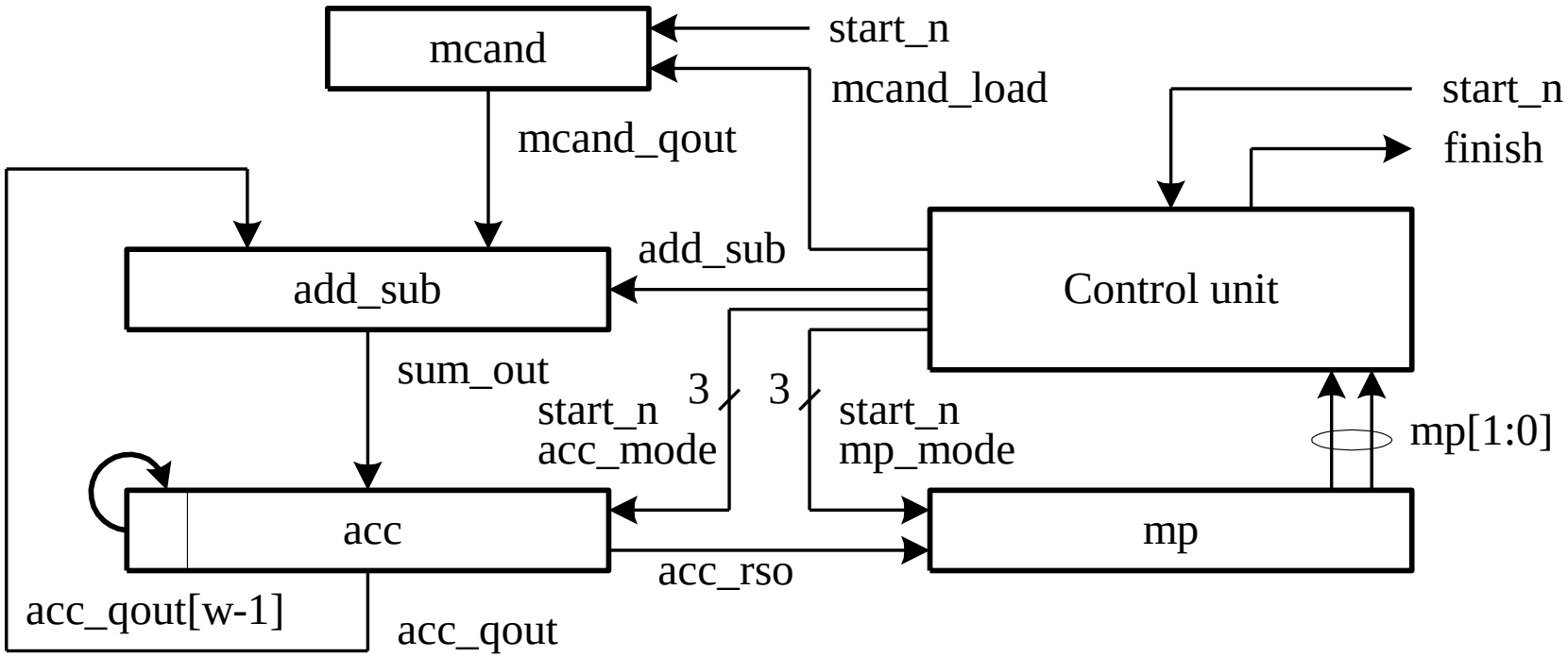
# An ASM Example --- Step 3: convert the 1st module

```
// converting the first module to the control module
module controller (cu_clk, cu_reset_n, cu_data_a, cu_load, cu_finish);
parameter W = 8;  // word size
parameter N = 3;   // N = log2(W)
input  cu_clk, cu_reset__n;
input  [W-1:0]   cu_data_a;
output reg [W-1:0] total;
reg     [N-1:0]    cnt;
reg     [1:0]       present, next;
output reg        finish;    // finish flag
localpram idle = 1'b0, add = 1'b1;
// the body of the w-bit booth multiplier
// part 1: initialize to state idle
always @(posedge clk or negedge start_n)
    if (!start_n) present <= idle; else  present <= next;
….
endmodule
```

# An ASM Example --- Step 3: construct top module

```
// converting the first module to the control module
module top (clk, start_n, data_a, data_b, total, finish);
parameter W = 8;  // word size
parameter N = 3;   // N = log2(W)
input  clk, start__n;
input  [W-1:0] data_a, data_b;
output wire [W-1:0] total;
wire   load;
output wire  finish;    // finish flag
localpram idle = 1'b0, add = 1'b1;
// datapath
datapath #(8)    dp (.clk(clk), .acc_reset_n(start_n), .acc_load(load),
                        .acc_data_b(data_b), .acc_total(total));
// controller
controller cu (.cu_clk(clk), .cu_reset_n(start_n), .cu_data_a(data_a),
                    .cu_load(load), .cu_finish(finish));
endmodule
```

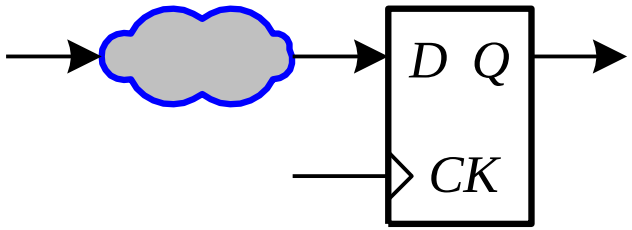
# A dp+cu Approach --- Booth Serial Multiplier




datapath

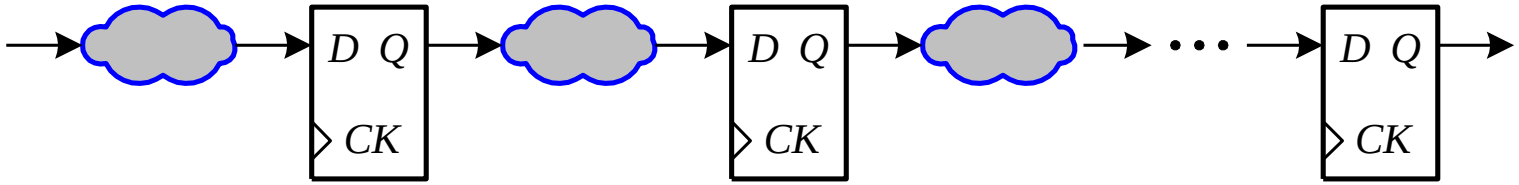
# Realization Options of RTL Design

- The rationale behind these options is a tradeoff among performance (throughput, operating frequency), space (area, hardware cost), and power consumption.
- Single cycle uses combinational logic only to realize the required functions.
  - It may require a quite long propagation time to finish a computation of required functions.



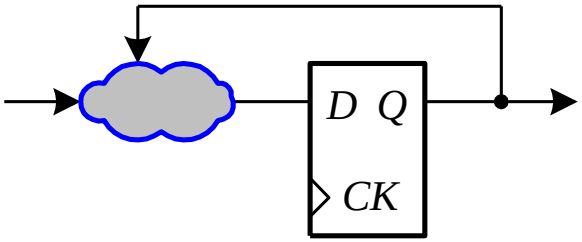
# Realization Options of RTL Design

- Multiple cycle executes the required functions in consecutive clock cycles.
  - Linear structure performs straightforward the required functions without sharing resources.

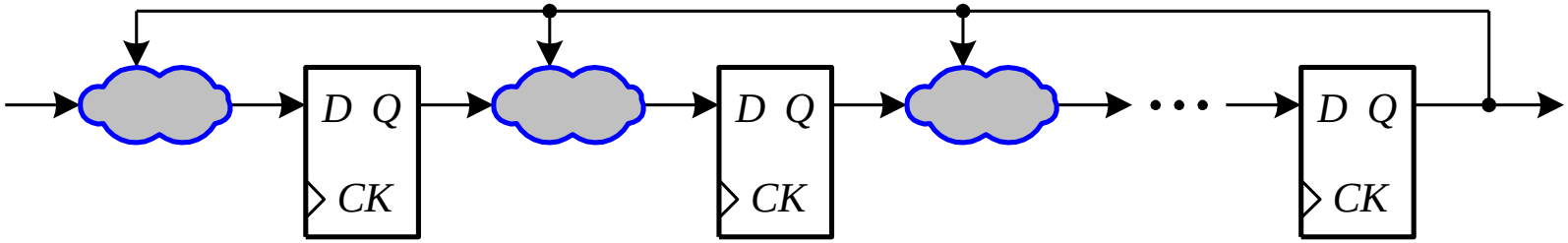


# Realization Options of RTL Design

- Nonlinear (feedback or feed-forward) structure performs the required functions with sharing resources by using feedback or feed-forward connection.



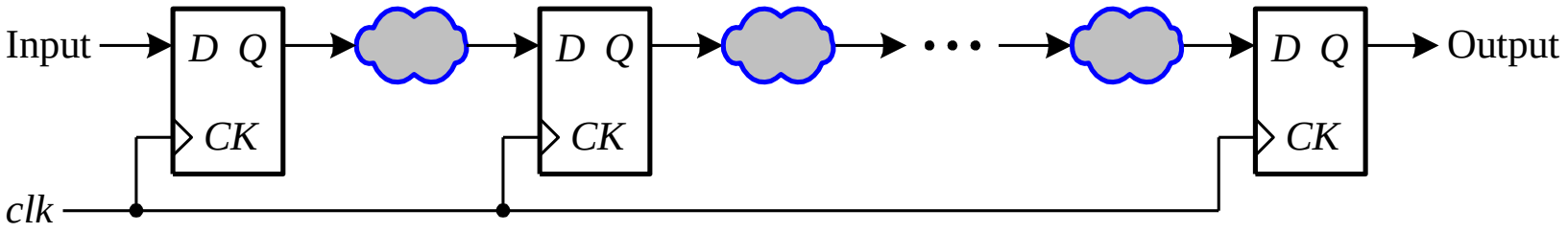

Single-stage nonlinear structure




Multiple-stage nonlinear structure

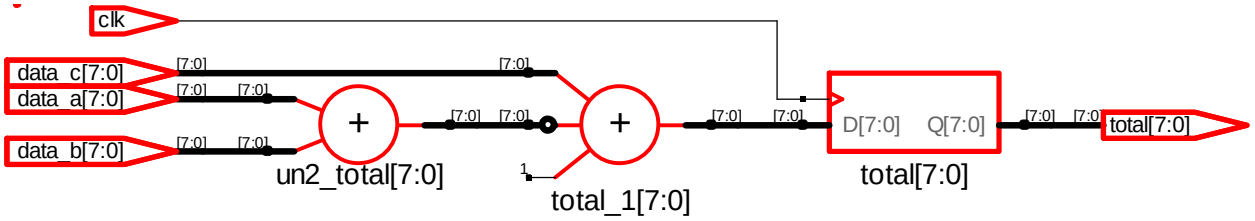
# Realization Options of RTL Design

- Pipeline is also a multiple cycle structure in which a new data can be fed into the structure at each clock cycle. Hence, it may output a result per clock cycle after the pipeline is fully filled.



# A Single-Cycle Example

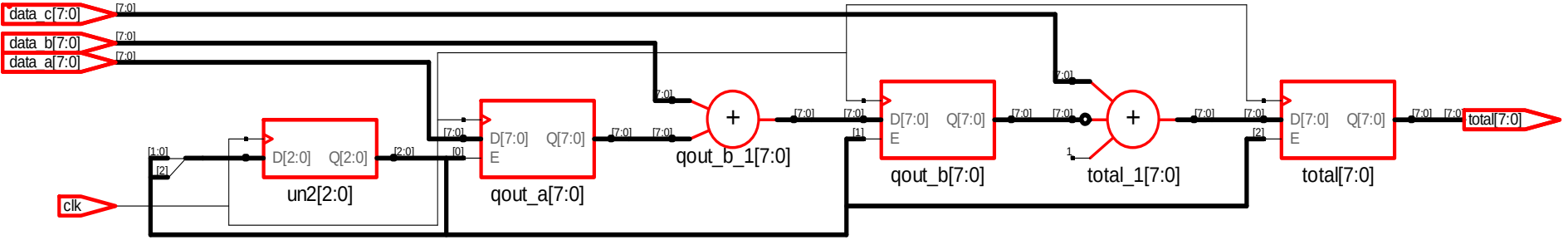
```
// a single-cycle example
module single_cycle_example(clk, data_a, data_b, data_c, total);
parameter N = 8;
input   clk;
input   [N-1:0] data_a, data_b, data_c;
output  reg [N-1:0] total;
// compute total = data_c - (data_a + data_b)
always @(posedge clk) begin
    total  <= data_c - (data_a + data_b);
end
endmodule
```



# A Multiple-Cycle Example

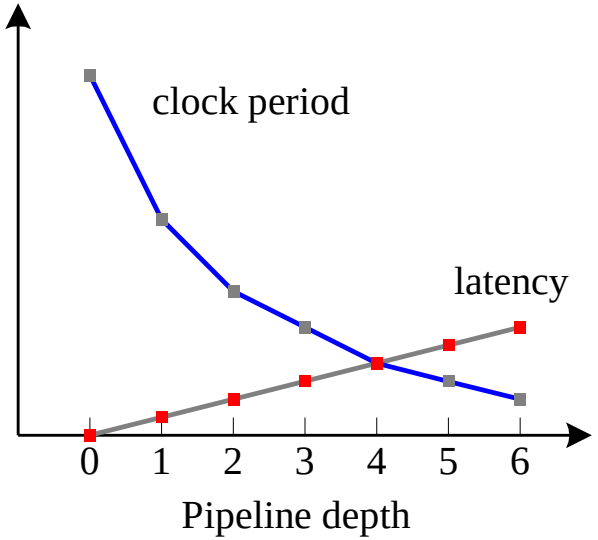
```
// a multiple cycle example --- an implicit FSM
module multiple_cycle_example(clk, data_a, data_b, data_c, total);
parameter N = 8;
input   clk;
input   [N-1:0] data_a, data_b, data_c;
output  reg [N-1:0] total;
reg     [N-1:0] qout_a, qout_b;
// compute total = data_c - (data_a + data_b)
// a, b, and c are sampled once in three clock edges
always @(posedge clk) begin
      qout_a <= data_a;
     @(posedge clk) qout_b <= qout_a + data_b;
     @(posedge clk)     total <= data_c - qout_b;
end
endmodule
```

# A Multiple-Cycle Example



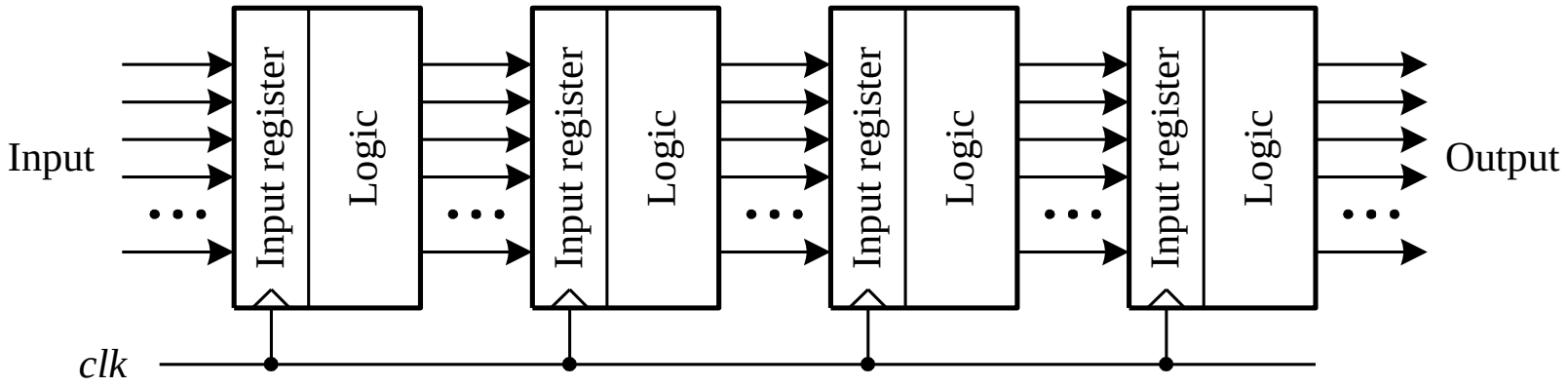
# Pipeline Principles

- Pipelining: a pipeline is composed of several stages with each stage consisting of a combinational logic circuit and a register, called pipeline register.
- Pipeline latency: the number of cycles between the presentation of an input value and the appearance of its associated output.



# Pipelined Systems

- The pipelined clock $f_{pipe}$ is set by the slowest stage and is larger than the frequency used in the original cascaded system.




- For the *i*th-stage, the smallest allowable clock period $T_i$ is determined by the following condition:

$$T_i > t_{FF} + t_{setup} + t_{d,i} - t_{skew,i+1}$$

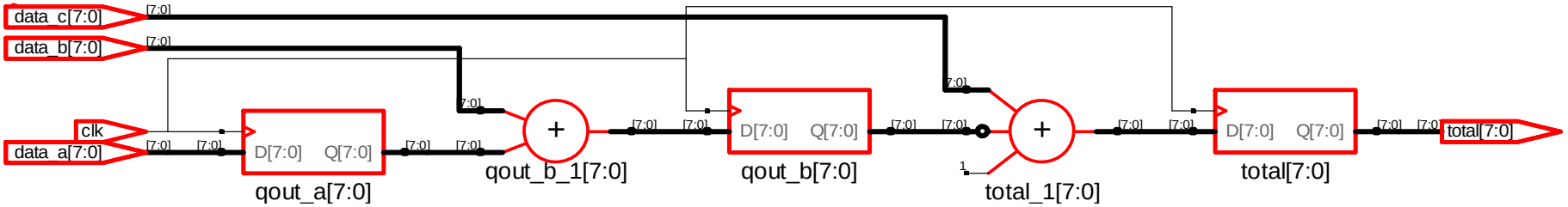
- The pipelined clock period for an *m*-stage pipeline is chosen to be:

$$T_{pipe} = \max\{T_i \mid i = 1, 2, ..., m\}$$

# A Simple Pipeline Example --- Not a Good One: why?

```
// a simple pipeline example --- Not a good one
module simple_pipeline(clk, data_a, data_b, data_c, total);
parameter N = 8;
input   clk;
input   [N-1:0] data_a, data_b, data_c;
output  reg [N-1:0] total;
reg     [N-1:0] qout_a, qout_b;
// compute total = data_c - (data_a + data_b)
always @(posedge clk) begin
    qout_a <= data_a;
    qout_b <= qout_a + data_b;
    total     <= data_c - qout_b;              // t(n) = c(n-1) – (b(n-2) + a(n-3))
end
endmodule
```

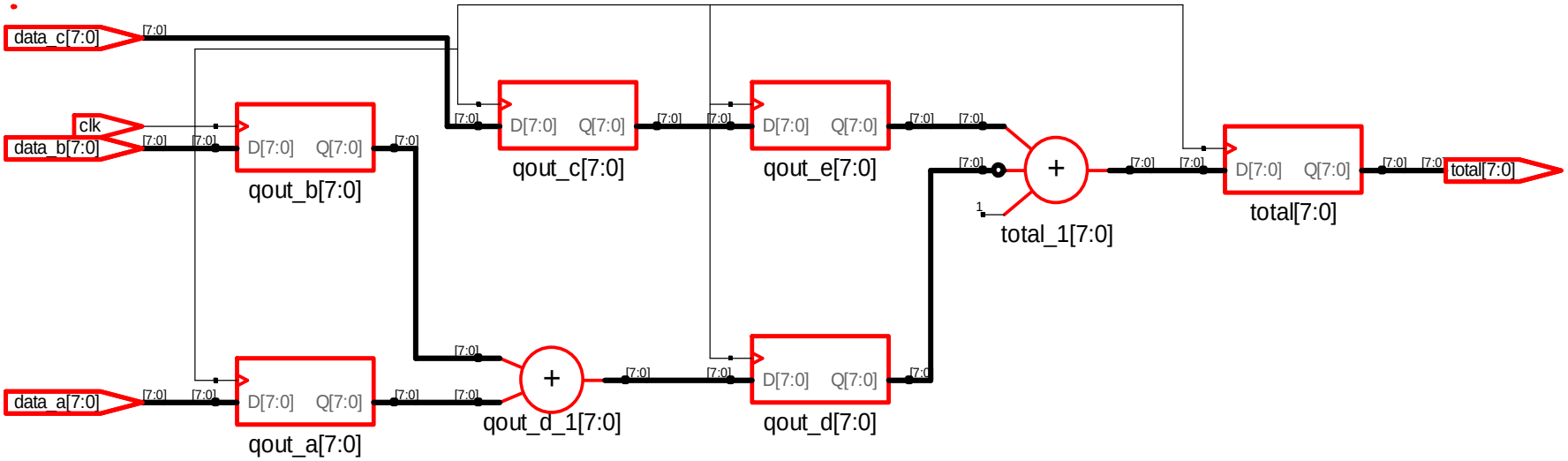
# A Simple Pipeline Example --- Not a Good One: why?



# A Simple Pipeline Example

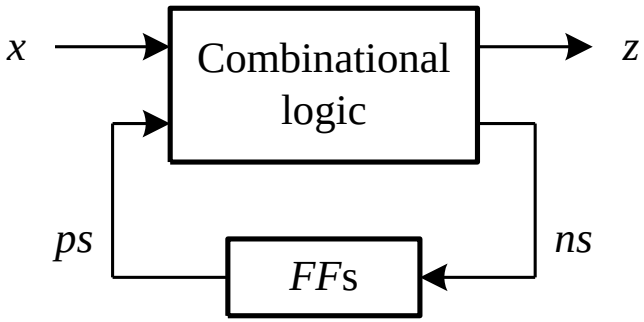
```
// a simple pipeline example
module pipeline_example(clk, data_a, data_b, data_c, total);
parameter N = 8;
input   clk;
input   [N-1:0] data_a, data_b, data_c;
output  reg [N-1:0] total;
reg     [N-1:0] qout_a, qout_b, qout_c, qout_d, qout_e;
// compute total = data_c - (data_a + data_b)
always @(posedge clk) begin
   qout_a <= data_a; qout_b <= data_b; qout_c <= data_c;
   qout_d <= qout_a + qout_b;
   qout_e <= qout_c;
      total  <= qout_e - qout_d;              // t(n) = c(n-3) – (b(n-3) + a(n-3))

end
endmodule
```

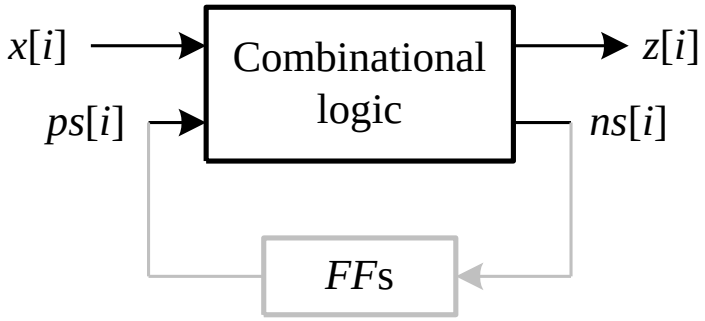
# A Simple Pipeline Example



# FSM vs. Iterative Logic

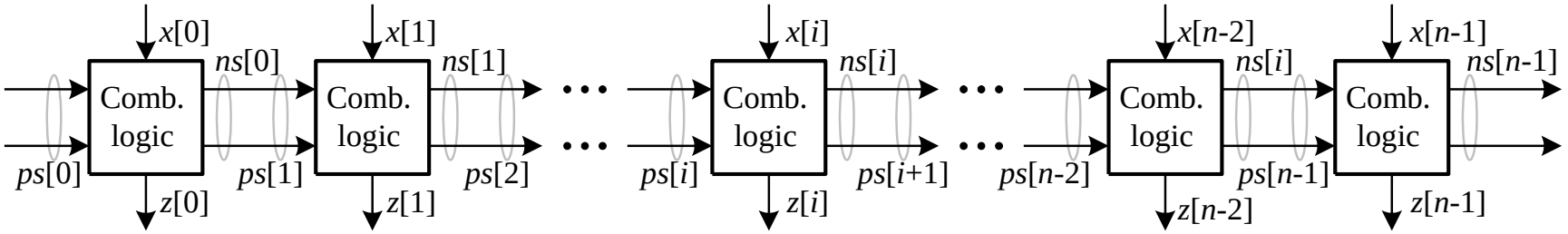
- An iterative logic can be considered as an FSM expansion in time. At each time step, the combinational logic part of the FSM is duplicated and the memory element part is ignored.



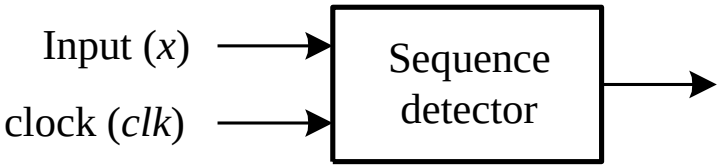

A nonlinear structure may be transferred into a linear one.

- An example of 1-D iterative logic

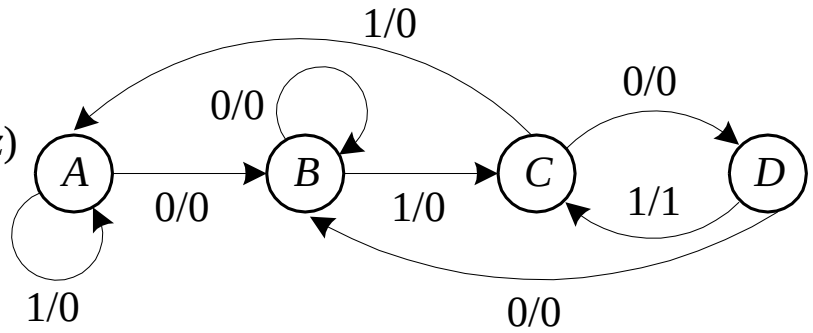


# An FSM Example --- A 0101 Sequence Detector

- Design a finite-state machine to detect the pattern 0101 in the input sequence *x*. Assume that overlapping pattern is allowed.

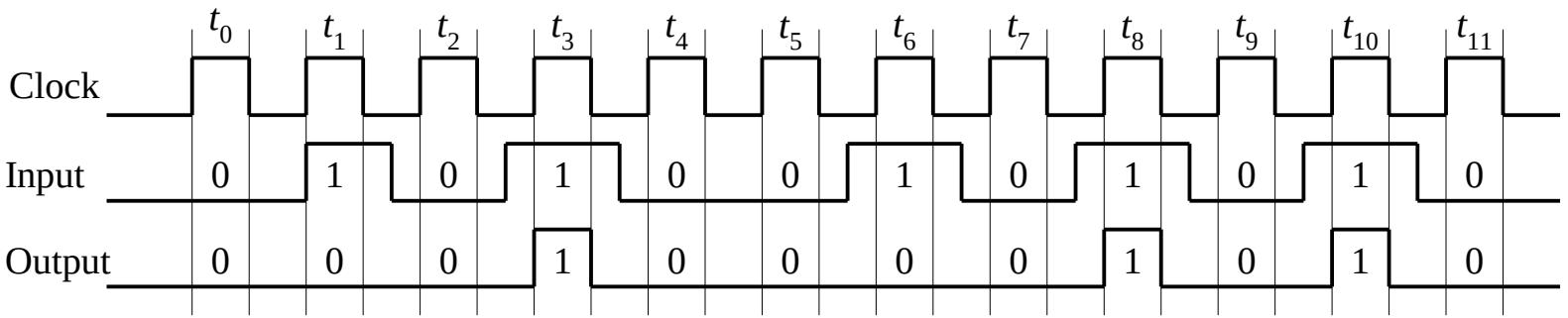



Block diagram

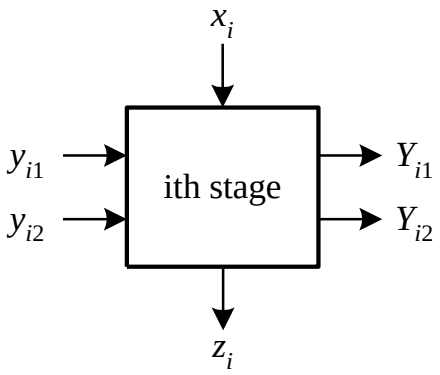



State diagram

Timing chart

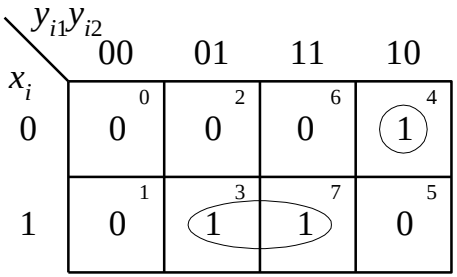


# An Iterative Logic --- A 0101 Sequence Detector

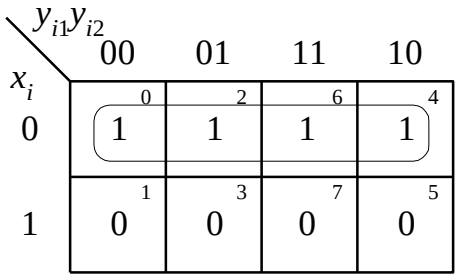



| PS | $x_i$ = 0 | $x_i$ = 1 |
|---|---|---|
| A | B,0 | A,0 |
| B | B,0 | C,0 |
| C | D,0 | A,0 |
| D | B,0 | C,1 |

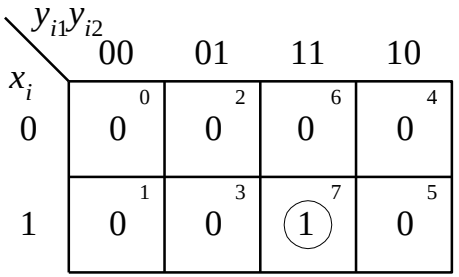
| $y_{i1}$ $y_{i2}$ | $x_i$ = 0: $Y_{i1}$ $Y_{i2}$ | $x_i$ = 1: $Y_{i1}$ $Y_{i2}$ | $x_i$ = 0: $z_i$ | $x_i$ = 1: $z_i$ |
|---|---|---|---|---|
| 00 | 01 | 00 | 0 | 0 |
| 01 | 01 | 10 | 0 | 0 |
| 10 | 11 | 00 | 0 | 0 |
| 11 | 01 | 10 | 0 | 1 |




$Y_{i1} = x_i y_{i2} + x'_i y_{i1} y'_{i2}$

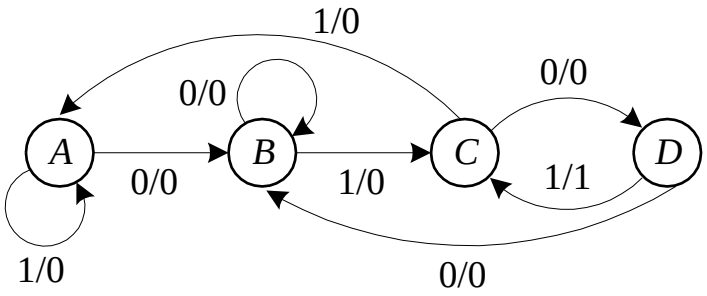



$Y_{i2} = x'_i$

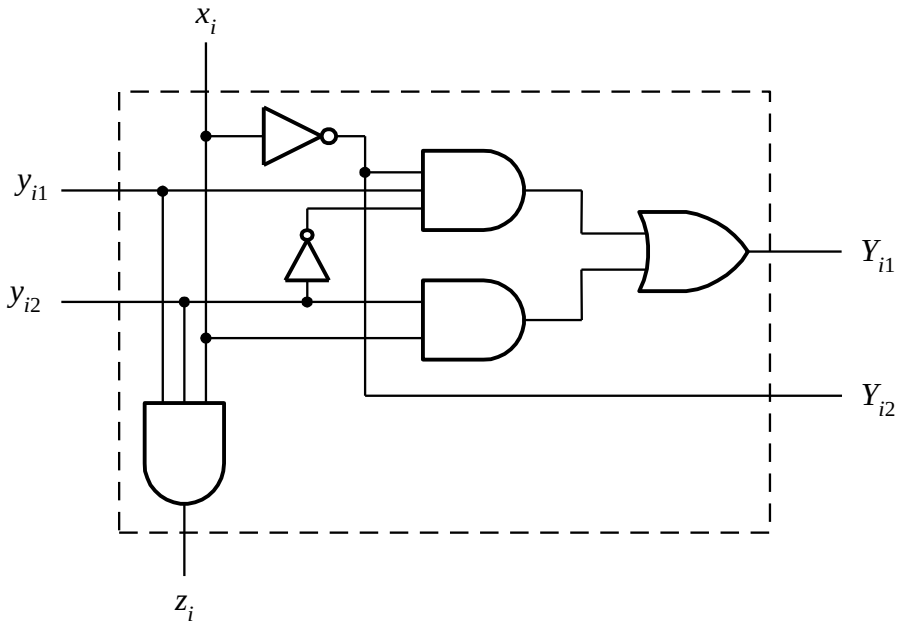


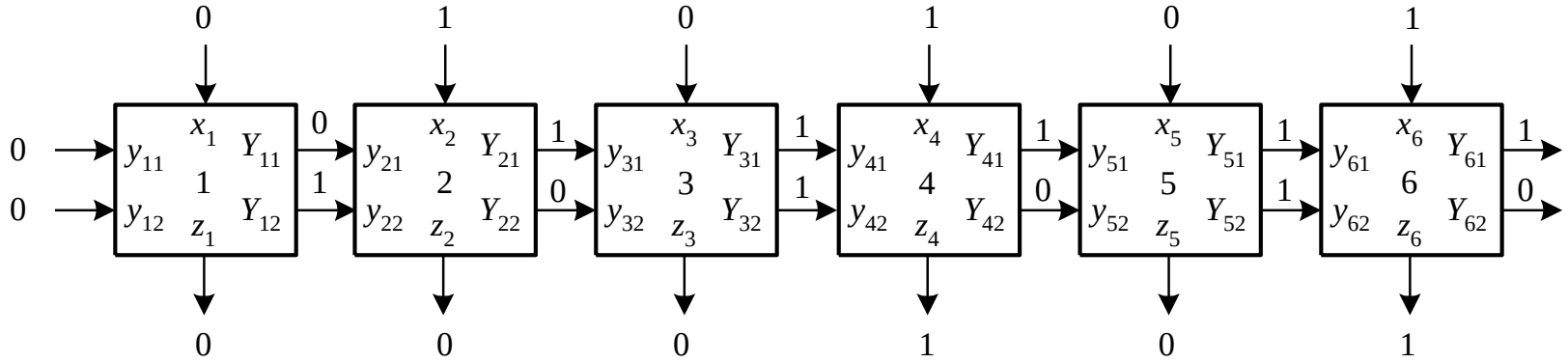

$z_i = x_i y_{i1} y_{i2}$

# An Iterative Logic --- A 6-bit 0101Sequence Detector




State diagram



# An Iterative Logic --- A Sequence Detector

```
// Behavioral description of 0101 sequence detector --- An iterative logic version.
module sequence_detector_iterative (x, z);
parameter n = 6; // define the size of 0101 sequence detector
input  [n-1:0] x;
output wire [n-1:0] z;
// Local declaration.
wire  [n-1:0] next_state_1,  next_state_0 ;
parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// Using generate block to produce an n-bit 0101 sequence detector.
genvar i;
generate for(i = 0; i < n; i = i + 1) begin: detector_0101
  if (i == 0)begin: LSB  // describe LSB cell
    basic_cell bc (x[i], A, z[i], {next_state_1[i], next_state_0[i]}); end
  else begin: rest_bits  // describe the rest bits
    basic_cell bc (x[i], {next_state_1[i-1], next_state_0[i-1]}, z[i], {next_state_1[i],
                   next_state_0[i]}); end
end endgenerate
endmodule
```

# An Iterative Logic --- A Sequence Detector

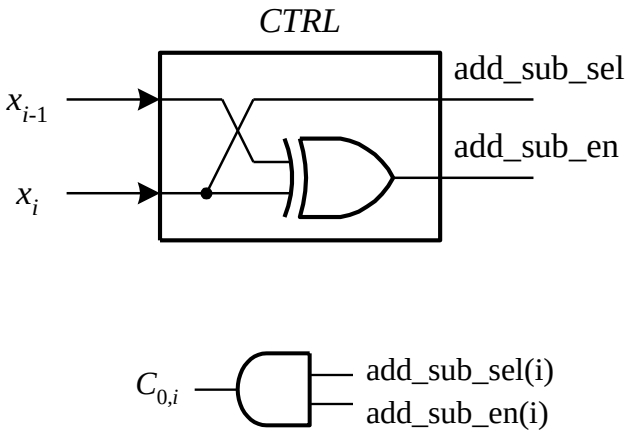
```
// Determine next state using function next_state.
module basic_cell (x, present_state, output_z, next_state);
input x;
input [1:0] present_state;
output reg [1:0] next_state;
output reg output_z;
parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// Determine the next state
always @(present_state or x) case (present_state)
      A: if (x) next_state = A; else next_state = B;
      ……
  endcase
// Determine the output value
always @(present_state or x) case (present_state)
    A: if (x) output_z = 1'b0; else output_z = 1'b0;
      ……
  endcase
endmodule
```

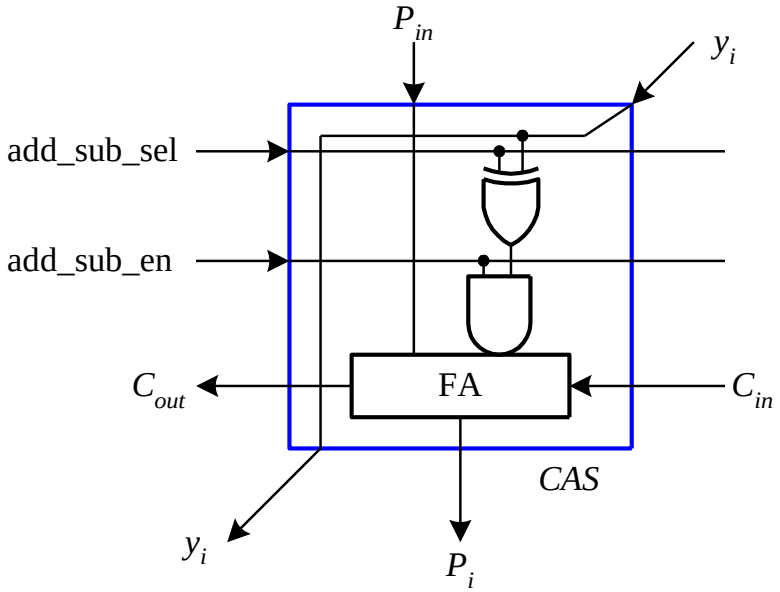
Exactly the same as the next state logic of FSM.

Exactly the same as the output logic of FSM.

# A 2D-Iterative Logic --- Booth Multiplier

| $x_i$ | $x_{i-1}$ | add_sub_sel | add_sub_en | Operation |
|---|---|---|---|---|
| 0 | 0 | $\phi$ | 0 | Shift only |
| 0 | 1 | 0 | 1 | Add and shift |
| 1 | 0 | 1 | 1 | Subtract and shift |
| 1 | 1 | $\phi$ | 0 | Shift only |



# A 2D-Iterative Logic --- Booth Multiplier