EE-334
Digital System Design

Custom Digital Circuits

Finite State Machines
and their description in VHDL

Andreas Burg

Purpose/Task of Finite State Machines

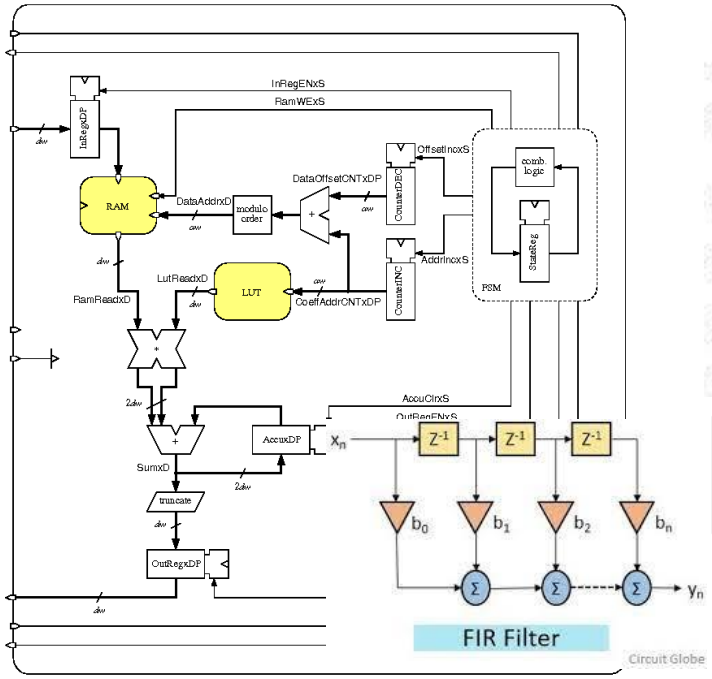
- **Finite state machines (FSMs) are a mathematical model of computation**
- FSMs describe discrete time systems
  - At each point in time the **FSM is in exactly one of its possible states**
  - **State changes** instantly and only **when proceeding from one time instant to the next**
    - Perfect match with synchronous design, where state changes only with clock edge
  - The **outputs and the next state** are **defined by** the **inputs and** the **present state**
- In digital system design, the FSM formalism is particularly useful for
  - Functional specification of control- and protocol-tasks
  - Modeling overall system behavior in an abstract way for simulation
  - Formal verification (with the help of Automata Theory)
  - Describing sequential circuits for synthesis

Purpose/Task of Finite State Machines

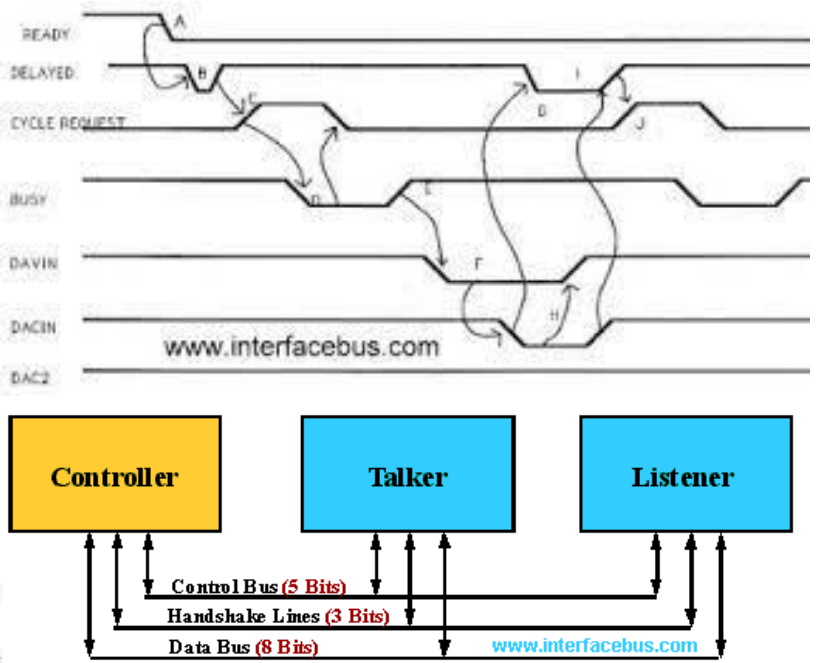
- **Every practical discrete-time system** with a finite number of states and inputs **can be described** completely (datapath and control) **as a finite-state-machine**
  - However, **for datapaths**, the number of states explodes and **FSMs are not a convenient description**.
- In practice, the **FSM formalism is** mostly **used for the control part**
  - **control** the operations and the flow of data in **datapaths**
    - generate sequences of control signals to orchestrate operation over time
    - internal results from a datapath and modify dataflow accordingly
  - **manage the communication between** system **components** through sequential protocols
    - react to inputs depending on the current state of the system
  - **keep track of the "relevant" history** of a system in compressed form (state)

Purpose/Task of Finite State Machines

- Some examples for use of FSMs



Datapath of a dedicated DSP block (here: FIR filter)



Handshake protocol for connected chips/components

Control of traffic lights

Mealy FSM: Formal Specification

- **Mealy-FSMs are the most generic description of an FSM** (can, in principal fully describe any synchronous system on a functional level)

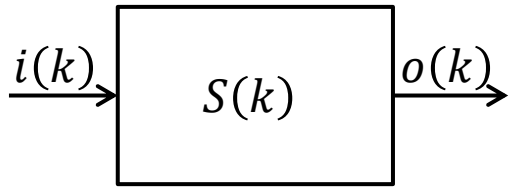
- **Inputs, outputs, and states are abstract quantities** chosen from finite sets
- **Formal specification of Mealy FSM**: Separate functions compute
  - the output from the present state and the input
  - the next state from the present state and the input

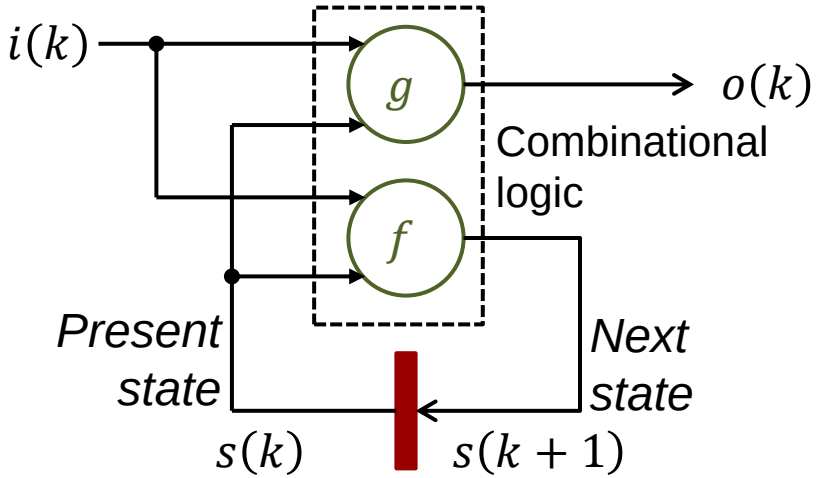
$$o(k) = g\big(i(k), s(k)\big) \quad Output\ function$$
$$s(k+1) = f\big(i(k), s(k)\big) \quad Next\ state\ function$$

$$i(k) \in I \quad Input$$
$$o(k) \in O \quad Output$$
$$s(k) \in S \quad State$$

$$s(0) = s_0 \quad Initial\ state$$

$$Latency = 0$$



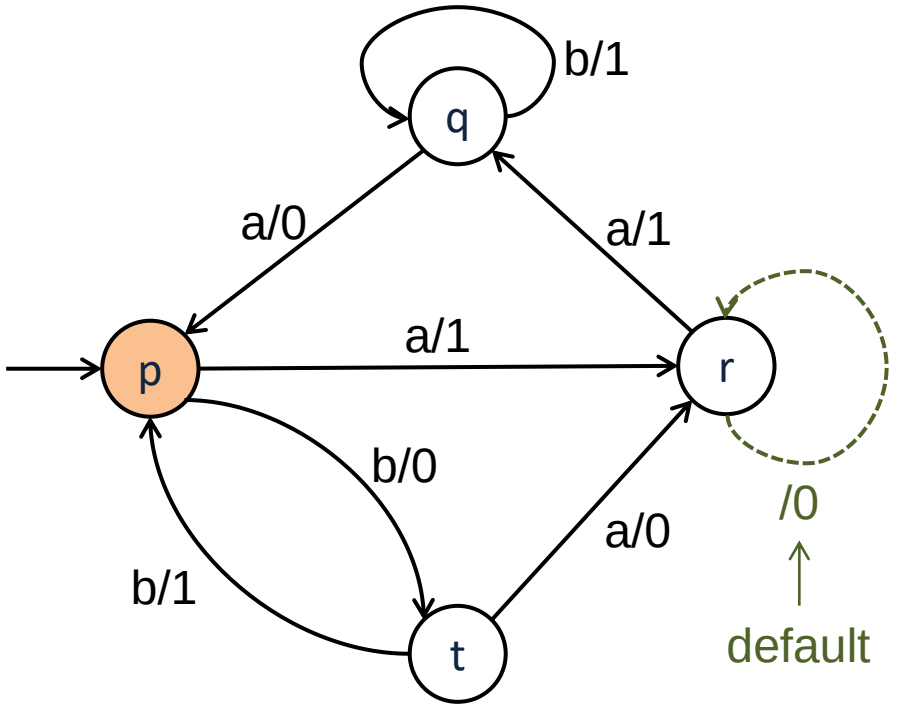


State Diagram Representation

- Formal FSM definition describes the operation, but not the transfer functions
- **State diagrams** provide a graphical means to represent
  - the next state transfer function and
  - the output
- **State diagram formalism**
  - State represented by vertices
  - Transition represented by edges
  - Input (condition) and output: annotated on edges <*input/output*>

    Default transition: no input annotation
    Taken when the condition for no other transition is met
- **Initial** (RESET) **state** defines $s(0)$

$$I = \{a, b\}, S = \{p, q, r, t\}, O = \{0,1\}$$

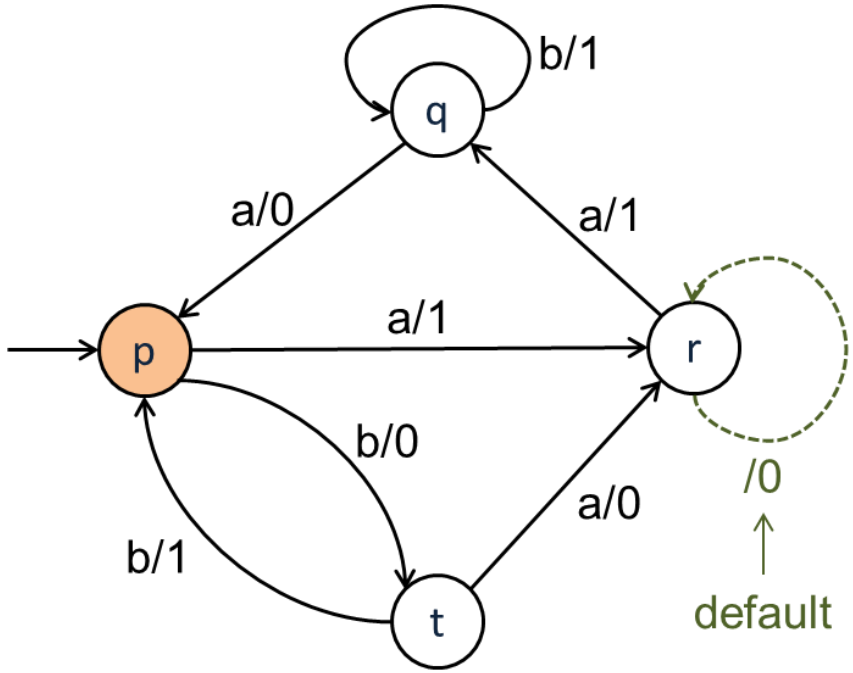


State Table Representation

- State diagrams become difficult to handle for complex FSMs
- **Table representation** of next-state and output functions is then **more convenient**
  - Specifies the next-state-function and the output-function in tabular form
  - Equivalent to the state diagram (except for not containing the reset state)

next-state-function

$i(k)$	a	b
$s(k)$	$s(k+1)$	
p	r	t
q	p	q
r	q	r
t	r	p



output-function

$i(k)$	a	b
$s(k)$	$o(k)$	
p	1	0
q	0	1
r	1	0
t	0	1

Moore FSM: Formal Specification

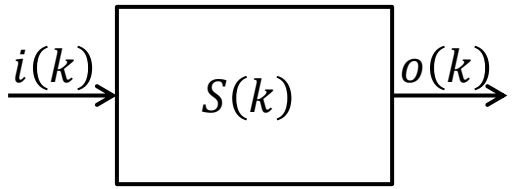
- **Moore-FSMs are a less generic form of FSMs**
  - Output is only a function of the present state

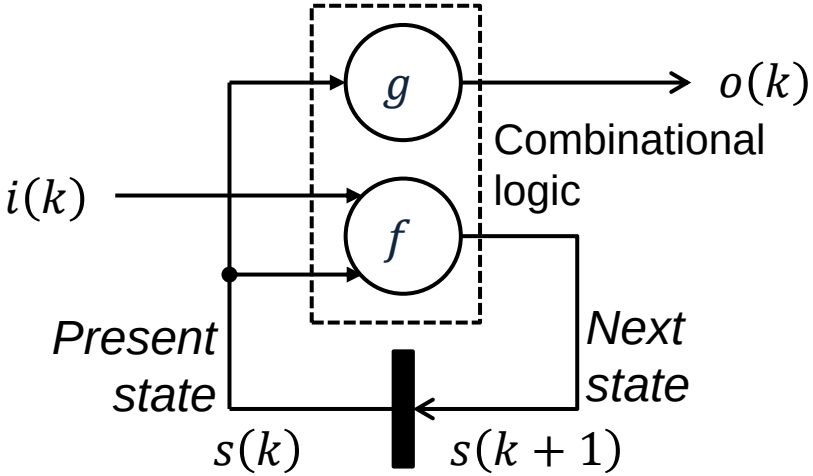
$$o(k) = g(s(k)) \quad \textit{Output function}$$
$$s(k+1) = f(i(k), s(k)) \quad \textit{Next state function}$$

$$i(k) \in I \quad \textit{Input}$$
$$o(k) \in O \quad \textit{Output}$$
$$s(k) \in S \quad \textit{State}$$

$$s(0) = s_0 \quad \textit{Initial state}$$

$$\textit{Latency} = \mathbf{1}$$





- The **Moore structure** mainly **impacts latency**
  - **Outputs can not react directly to an input** in the same state (time step)
  - **No** direct combinational **logic between input and output**

Mealy FSM: Formal Specification and RTL Diagram

- **Mealy-FSMs are the most generic description of an FSM** (can, in principal fully describe any synchronous system on a functional level)

Formal specification

- Outputs and next state are defined based on separate transfer functions
- Initial state required to initialize the iteration

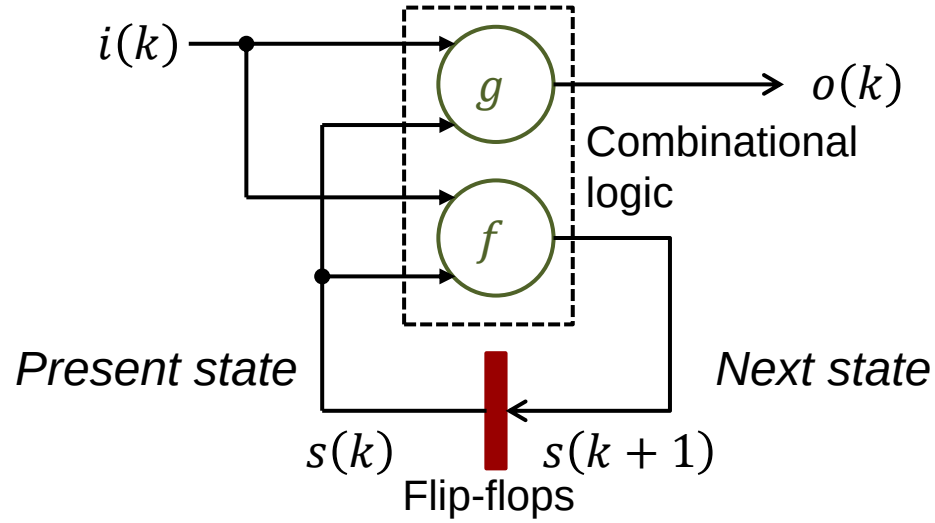
$$o(k) = g\big(i(k), s(k)\big) \quad Output\ function$$
$$s(k+1) = f\big(i(k), s(k)\big) \quad Next\ state\ function$$

$$s(0) = s_0 \quad Initial\ state$$

$$Latency = 0$$

RTL diagram

- **Two blocks of combinational logic to**
  - **Compute the next state**
  - **Compute the output**
- **A register to store the present state**

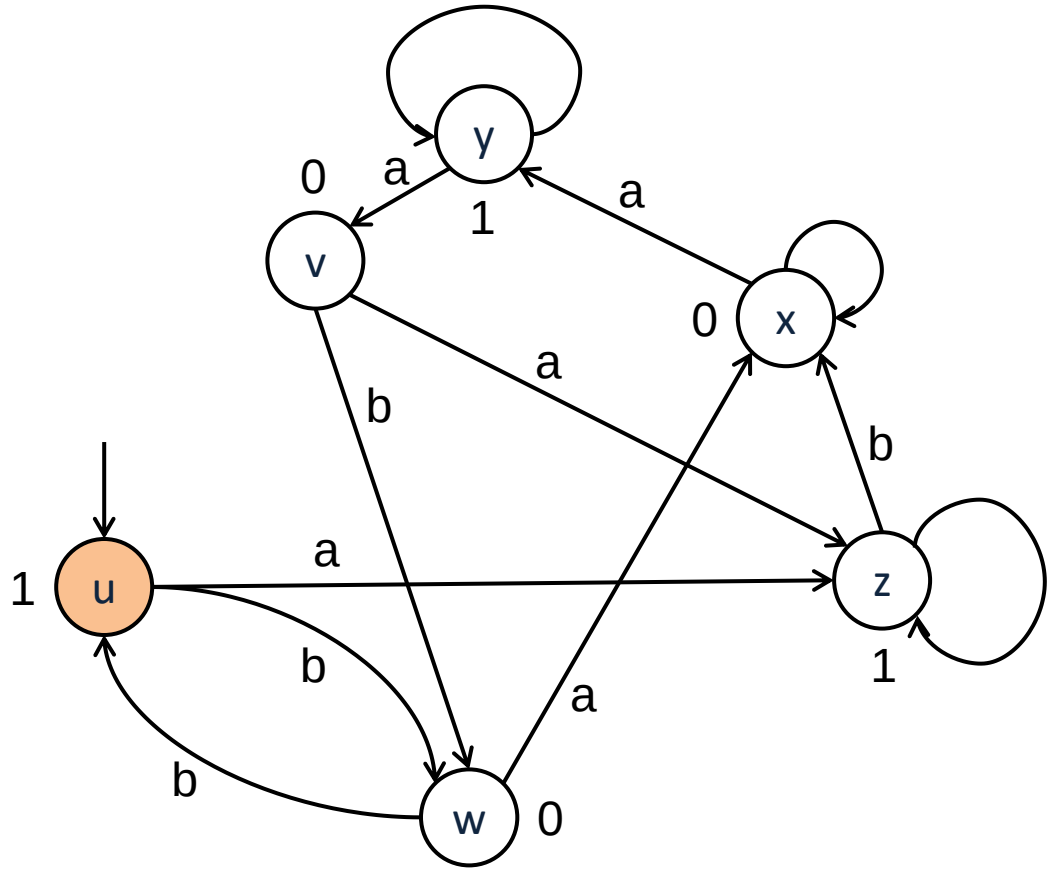


Descriptions of Moore FSM

- **Moore FSMs simplify the state diagram** and the table representation
  - **Outputs are associated (annotated) with states**
  - Transitions without annotation correspond to the default transitions (only one default transition per state!)

next-state- and output-function

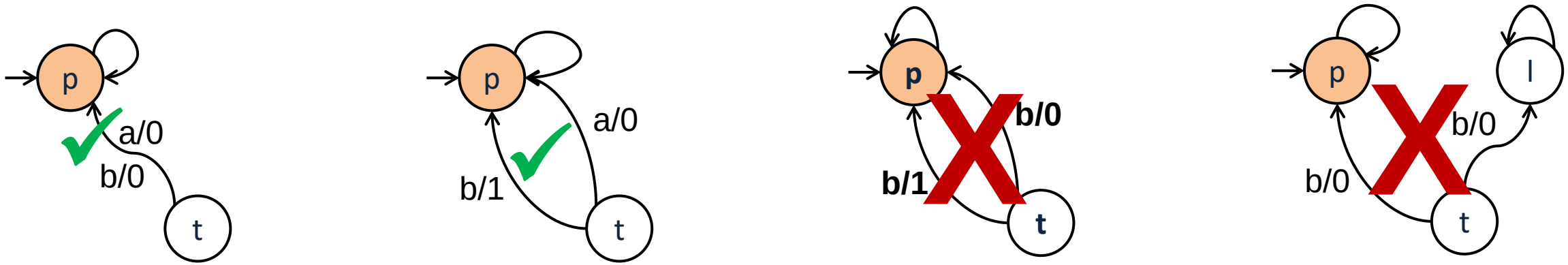
$i(k)$	a	b	
$s(k)$	$s(k+1)$		$o(k)$
u	z	w	1
v	z	w	0
w	x	u	0
x	y	x	0
y	v	y	1
z	z	x	1



Some Rules for Valid FSMs

- **Completeness**: Next-state-function and output-function must be specified for all states and for all possible inputs (Mealy)
  - Default transition collapses all so far unspecified transitions into one
  - Multiple identical transitions can be collapsed with OR, if they end up in the same state and produce the same output

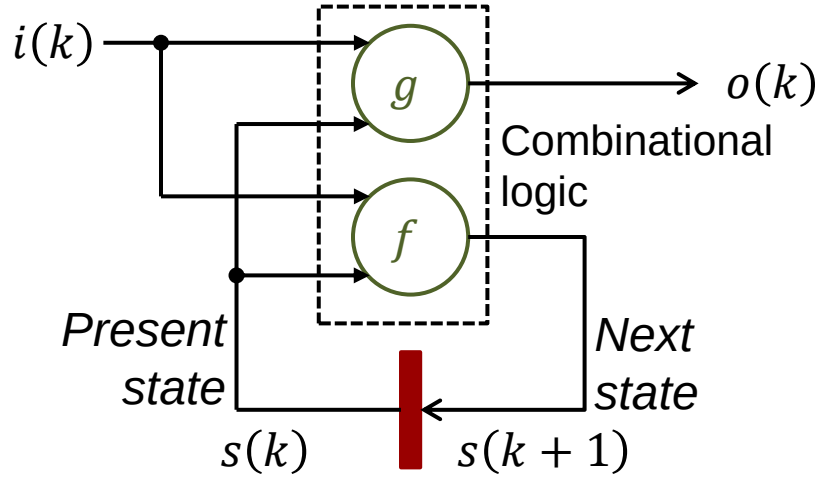
- **Uniqueness**: Next-state-function and output-function must be unique
  - the same input/state combination can never result in two conflicting next-states and/or outputs



RTL Architecture of FSMs

- **Formal definition of FSMs provides** immediately the **RTL architecture diagram** for the digital implementation

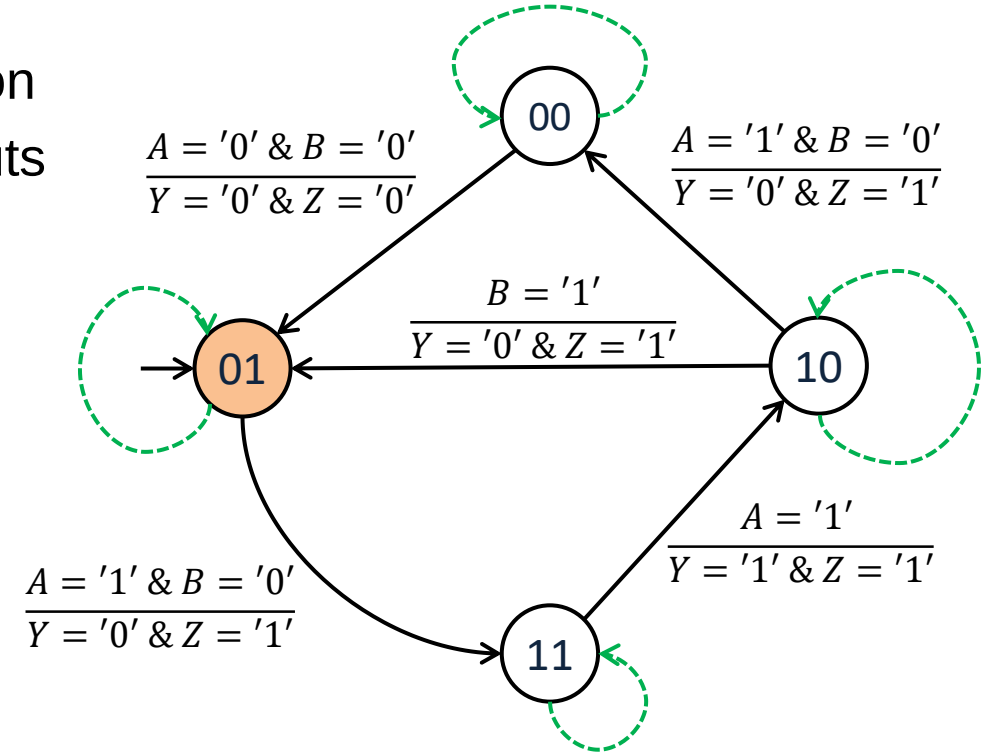
- Required **hardware components**
  - A combinational logic block that computes the output $o(k)$ using the function g
  - A combinational logic block that computes the next state $s(k+1)$ using the function f
  - A register (Flip-Flop) that stores the current state $s(k)$ and updates it to $s(k+1)$ on each clock edge



Specifying FSMs for Digital (RTL) Circuits

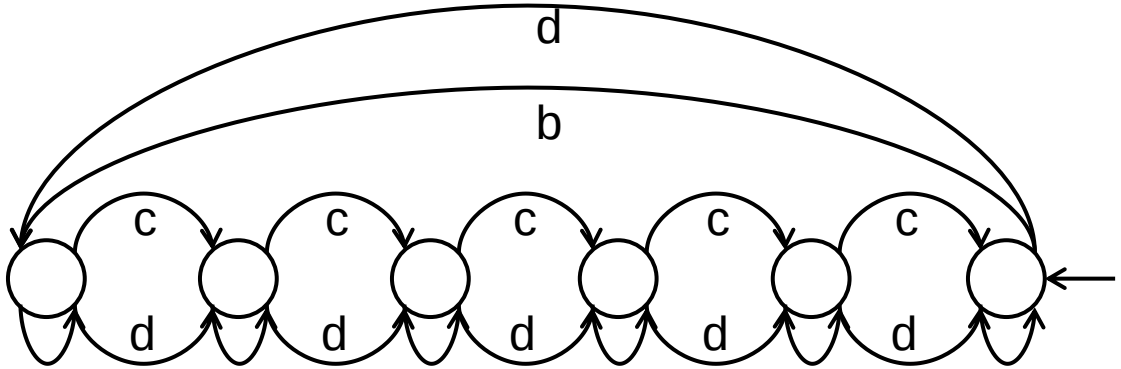
- Abstract **FSM specification relies on abstract** (finite) **sets for states, inputs, and outputs** which do not provide insight how these materialize in a real system

- **Impact of digital circuit implementation**
  - states, inputs, and outputs have a binary representation
  - FSMs typically have multiple inputs and multiple outputs
    - Each output has its own transfer function
    - State transitions (and outputs) are defined by the value of multiple inputs



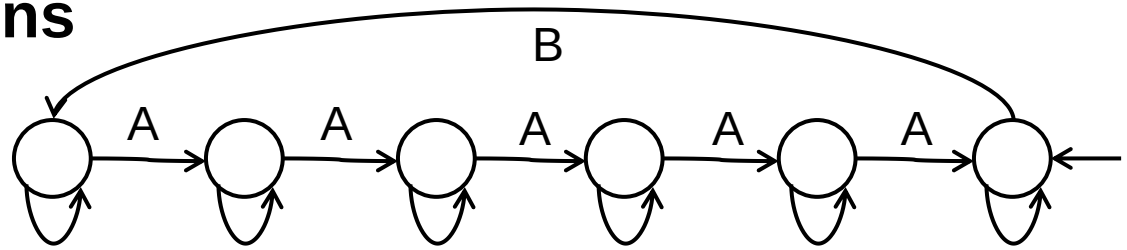
Specifying FSMs for Digital Circuits

- **Many transitions do not depend on all input signals**
  - **Example**: go through 6 states when an enable signal (*En='1'*) is set. In the last state, wait until *Clr='1'* before returning to the first state, regardless of *En*



i	*En*	*Clr*
a	0	0
b	0	1
c	1	0
d	1	1

- **Boolean expressions and** the use of **"don't cares"** ('-') **merge equivalent transitions**



	En	*Clr*
A	1	-
B	-	1

A : En=='1'
B: Clr=='1'

Specifying FSMs for Digital Circuits

- **Caveat**: **Implicit merging of state transitions**
  - Conditions based on boolean functions are not necessarily mutually exclusive
  - Incomplete conditions: using “don’t care” also leads to non-mutually-exclusive conditions

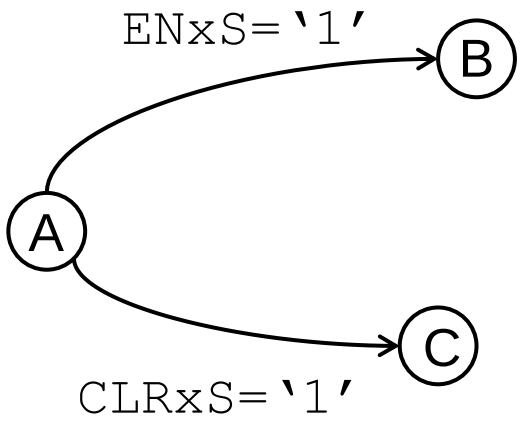
- Remember, next-state- and output-functions must be unambiguous

Specifying state transitions with incomplete conditions must be done with great care

- **Risk of mismatch between** formal **specification and implementation**:
  - HDL descriptions often resolve ambiguities through sequential formulation of the conditions
  - Formal specification of FSMs does not imply precedence of multiple conditions!!

Specifying FSMs for Digital Circuits

- Example for ambiguous state transition specifications, resolved “accidentally” in different VHDL specifications



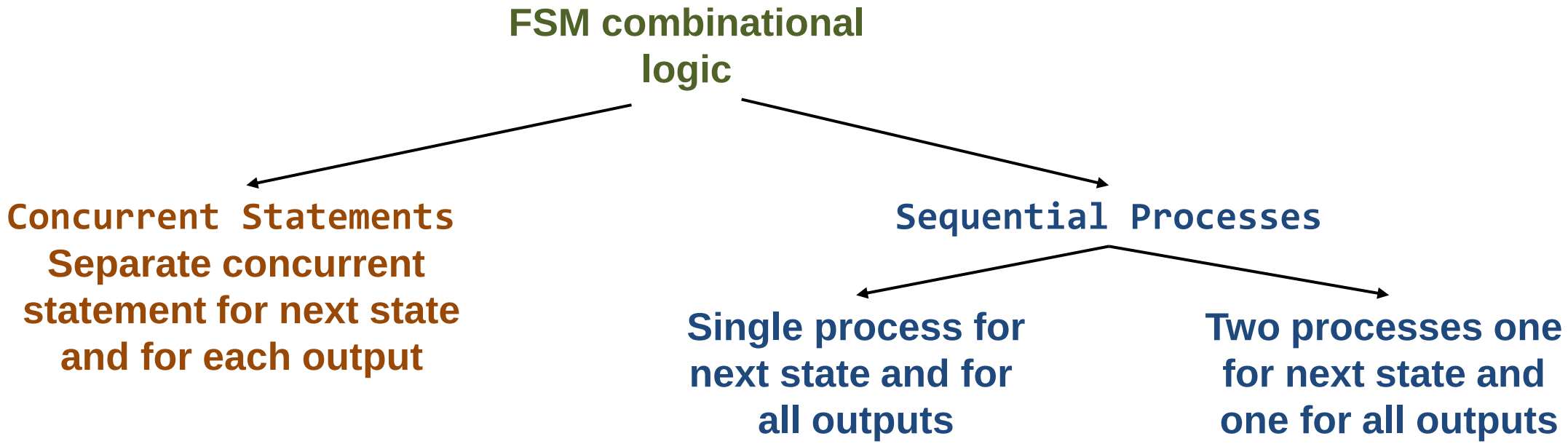

```
case STATExDP is
  -- Initial state
  when A =>
    if ENxS = '1' then
      STATExDN <= B;
    end if;
    if CLRxS = '1' then
      STATExDN <= C;
    end if;
```

```
case STATExDP is
  -- Initial state
  when A =>
    if CLRxS = '1' then
      STATExDN <= C;
    end if;
    if ENxS = '1' then
      STATExDN <= B;
    end if;
```

What happens if `ENxS='1'and CLRxS='1'???`

VHDL Implementation of FSMs

- To implement an FSM in VHDL we directly describe its RTL diagram components
  - The **state register** corresponds to a **clocked process** (register)
  - The two **transfer functions g and f** correspond Boolean logic that **can be realized as**
    - **Concurrent statements:** one for each output and for the next state
    - **Sequential process(es):** different options
- **Implementation of transfer functions based on processes** is often preferred



State Register Description

- **Clocked process only assigns next state to the present state**
  - Next state defined in separate processes

```
-- Clocked Process, generating a FlipFlop behavior
p_seq: PROCESS (clock_signal, async_reset_signal) IS
BEGIN  -- process name: p_seq
  IF async_reset_signal = ‘0|1’ THEN
    present_state_signal <= constant;
  ELSIF clock_signal’EVENT AND clock_signal = '1’ THEN
    present_state_signal <= next_state_signal;
  END IF;
END PROCESS p_seq;
```

Output and Next State Logic in a Single Process

- **Logic with** a **single combinational process** **is** often the **most readable**
  - **Start with a CASE statement to identify the present state**
  - **IF statements** for each state **represent conditions for the next state and outputs as a function of the FSM inputs**

```
-- FSM Logic Process (for next state and/or outputs)
process_name: PROCESS (present_state_signal,
                       FSM_input_signals) IS
BEGIN
  -- Default assignments
  next_state_signal   <= present_state_signal;
  FSM_output_signal_1 <= default_output_expression;

  CASE present_state_signal IS
    WHEN state_1 =>
        -- Conditional statements based on inputs
        IF condition_1_on_FSM_input_signals THEN
          next_state_signal <= next_state_specification;
          FSM_output_signal_1  <= output_expression;
        END IF;
        …
    WHEN state_2 =>
        …
    WHEN OTHERS =>
    END CASE;
END PROCESS process_name;
```

FSM State Encoding

- **State representation is irrelevant from a functional perspective**

- **For implementation**: abstract **states must be represented with multiple bits**
  - Each state must be assigned a unique binary representation
  - Minimum bit encoding

$$K \text{ states} \quad \Rightarrow \quad \lceil \log_2 k \rceil \text{ bits}$$

  - More bits than necessary can be used to represent the state of a system
  - For representing K states with B_K bits the number of possible state encodings is

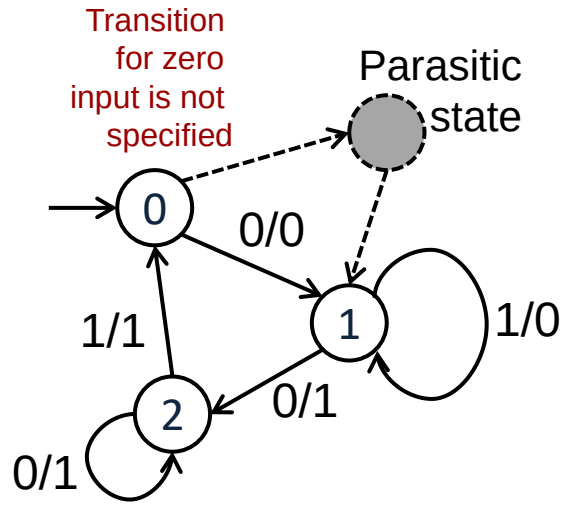
$$\frac{(2^{B_K} - 1)!}{(2^{B_K} - K)!\, B_K!}$$

Parasitic States

- Consider an FSM with K states encoded with $B_K \geq \lceil \log_2 K \rceil$ bits

# of physical states	# of logical states
$\mathbf{2}^{B_K}$	K

- In some cases, $\mathbf{2}^{B_K} > K$
- Parasitic states: unused physical states
  - For parasitic states, the *behavior of an FSM implementation* is *deterministic* and *defined* by the circuit implementation, but *not specified*
  - Transition into a parasitic state is triggered by unspecified input => State-transitions must be specified completely (possibly using default transitions)



- Impact varies from *erroneous output* to *unspecified behavior for some cycles* to *complete lock-up*

Defining States as Enumerated Types

- **Manually encoding states** to define state variables often
  - **is inconvenient** overhead during VHDL implementation
  - **limits readability** of the code: calling states by a name contributes to readability
  - **is inflexible** since state representation can not be optimized

- VHDL allows for **enumeration types** : ideal for state variables
  - Values can be arbitrary strings (ideal for state names)

```
TYPE enum_type_name IS (value_1, value_2, …);
SIGNAL signal_name : enum_type_name;
```

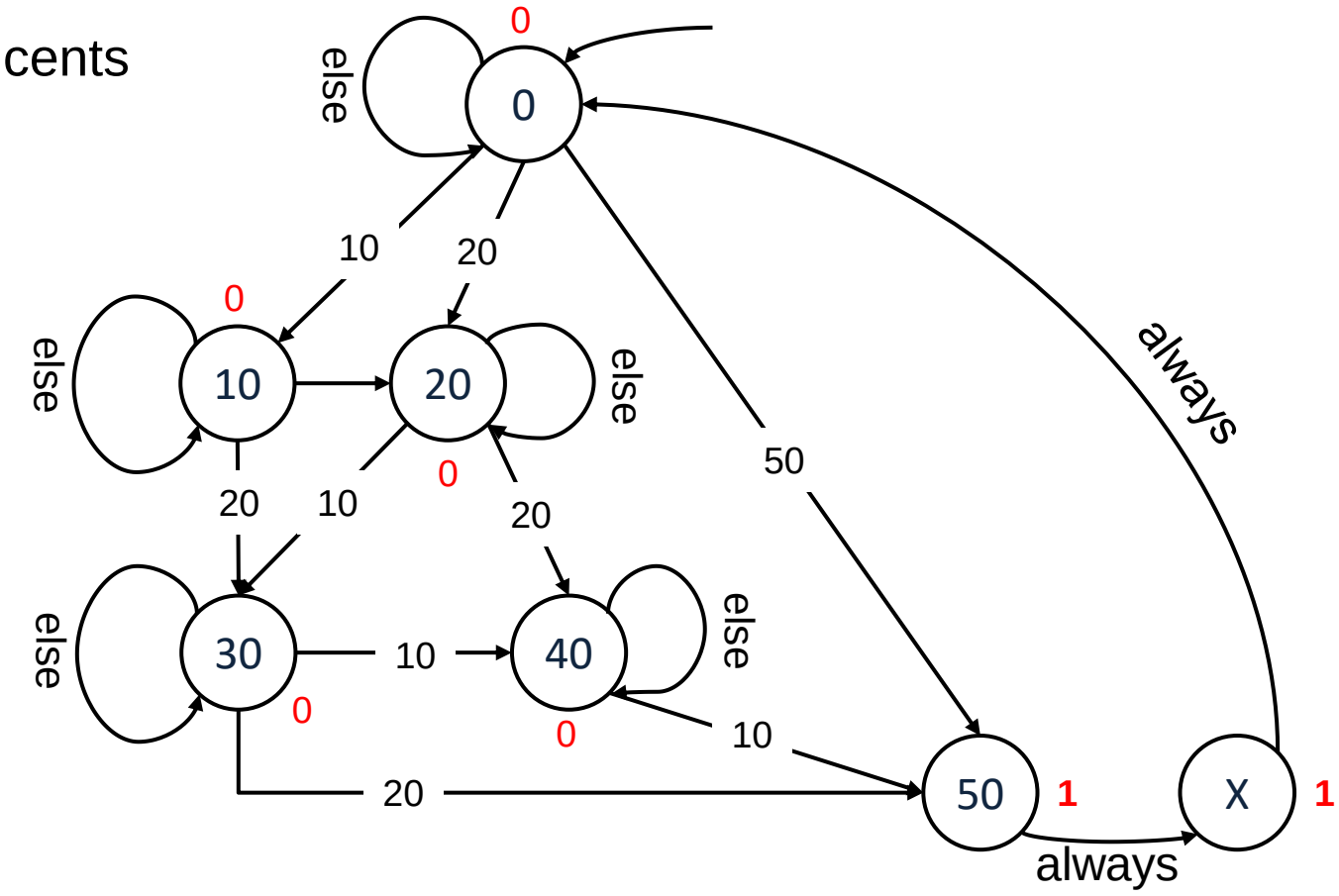
  - Abstract type without pre-defined binary representation: chosen by the synthesis tool during optimization
  - Every FSM requires its own type matched to the states required

```
ARCHITECTURE rtl OF my_fsm_states IS
          -- type declarations for FSM states
          TYPE state_type IS (StateA, StateB, StateC);

          -- signal declaration
          SIGNAL STATExDN, STATExDP : state_type;
BEGIN
…
```

FSM Example

- Vending machine waiting to enter 50 cents
  - Coins: 10, 20, 50 cents
  - Output = 1 for 2 entire cycles after 50 cents have been entered
  - Overflows are handled by returning the entered coin and remaining on the current amount



Timing Diagrams

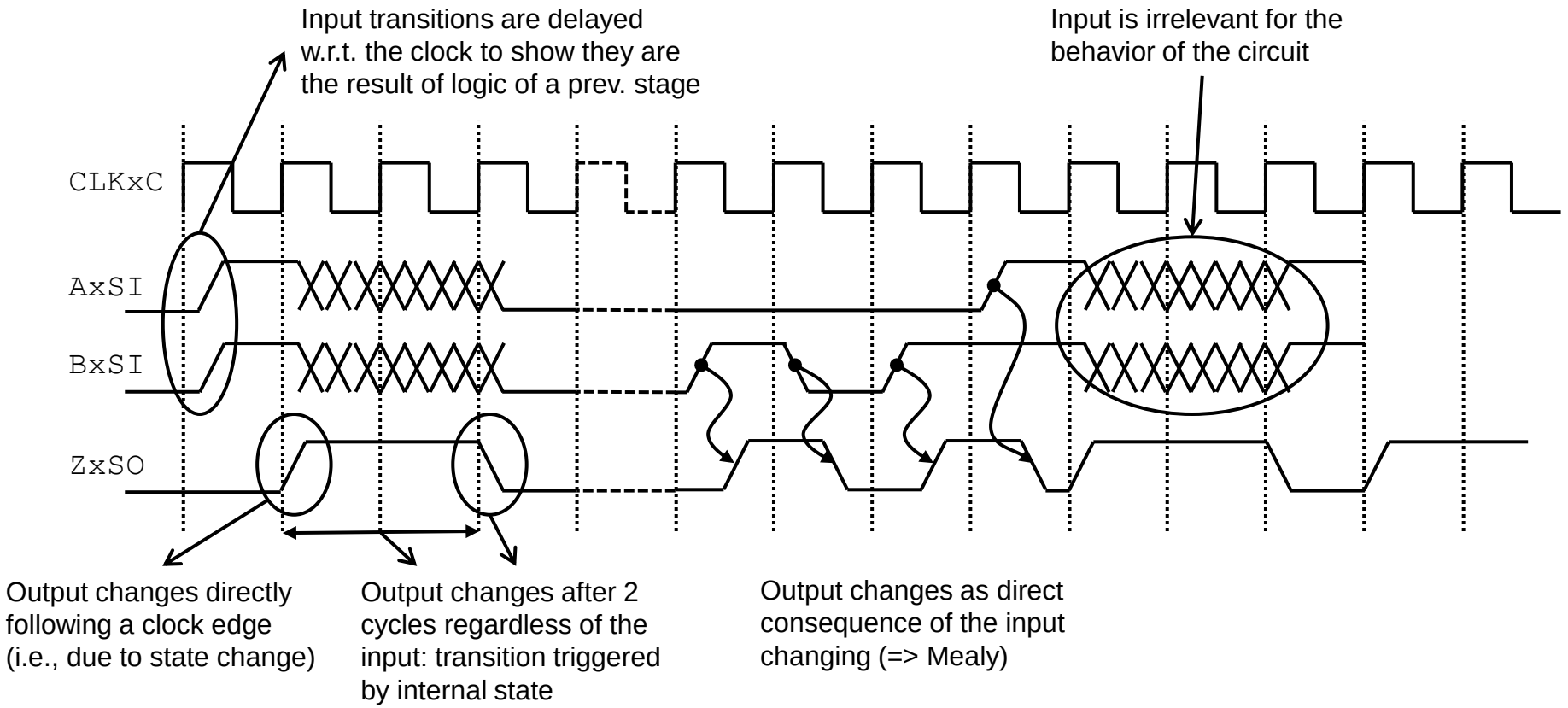
- State diagrams provide a complete and unambiguous FSM specification
  - However, state diagrams do not illustrate the evolution of (input and output) signals over time

- **Timing diagrams show waveforms examples** produced by an FSM

- **Timing diagrams** are very useful to illustrate system behavior, but have also several issues:
  - **Complete specification requires enumeration of all possible input/state combinations**
  - A **single** timing **diagram is often incomplete**
  - **Causality** is often **not clearly visible** from a timing diagram
  - Often **no explicit definition of states:** must be reconstructed

Timing Diagrams

- Timing diagrams often use a particular graphical notation to clarify causalities and dependencies



From Timing Diagrams to FSMs

- To reconstruct an FSM compatible with a given set of timing diagrams **check each clock cycle of all timing diagrams to identify**
  - **States:** assign separate states to each clock cycle in which
    - Outputs are different for the same input
    - Next state is different for the same input
  - **State-transitions:** for each state, find all the next states in the timing diagram
    - Annotate transitions with the inputs causing/triggering the transition
  - **Outputs:** distinguish between Moore and Mealy outputs
    - Moore: for each state, identify outputs that are independent of the inputs (i.e., change only on state transition)
    - Mealy: for each state and each input combination annotate a state transition with the corresponding input with the output (different transitions may have the same start- and end-point, but different outputs)

Example

- **Specify a state machine that complies with the given timing diagram**
  - **Note:** Specification is incomplete since AxSI=1 and BxSI=0 is not specified!!!