DVA494

Programming of Reliable Embedded Systems

Obed Mogaka | Hamid Mousavi | Masoud Daneshtalab

IDT, Malardalens University

January 12, 2026

Today's Agenda

- From Algorithms to Hardware
- The Register Transfer Methodology (RTM)
- Finite State Machines (FSMs)
- The State Diagram
- FSM Modeling in VHDL.

From Algorithms to Hardware: Structural Dataflow

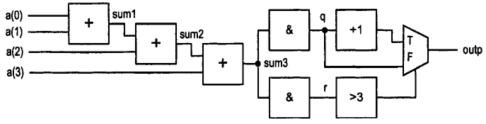
- **The Goal:** Implement an **Algorithm** (a sequence of steps with variables) in hardware.
- **Method 1:** Structural Dataflow (Unrolling)
  - Map every step of the algorithm to a dedicated hardware block.
  - "Variables" become wires; "Sequential execution" becomes cascading logic blocks.

Example: Summation Loop

- **Algorithm:** Loop 4 times to add numbers.
- **Structural Hardware:** Use 3 separate adders and a divider connected in a chain.

Drawbacks:

- **High Area Cost:** Hardware is not shared.
- **Inflexible:** Hard to change the algorithm logic.



Structural Dataflow Implementation

The Register Transfer Methodology (RTM)

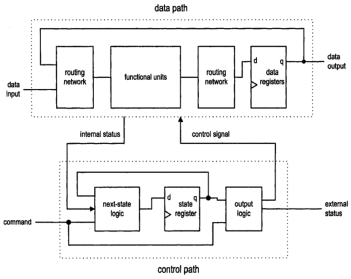
- To realize algorithms efficiently, we separate the system into two parts: storage/processing and sequencing.

1. The Data Path (The "Muscle")

- Contains **Registers** to store intermediate data (variables).
- Contains **Functional Units** (Adders, ALUs) to perform operations.

2. The Control Path (The "Brain")

- Specifies *when* register operations take place.
- **Implemented as a Finite State Machine (FSM).**
- It uses states to enforce the order of steps and decision boxes to imitate loops/branches.



Detailed FSMD Architecture

Key Interaction: The Control Path sends commands based on status flags from the Data Path.

Definition

This combination (FSM controlling a Datapath) is often called an **FSMD** (FSM with Data path).

Finite State Machines (FSMs)

- **Definition:** A mathematical model used to design sequential logic circuits.
- **Why "Finite"?** The system has a limited (finite) number of possible internal conditions, called **states**.
- **Core Function:** An FSM acts as the **Controller** (Control Path) of a larger digital system, managing the operations of the Datapath based on inputs and history.

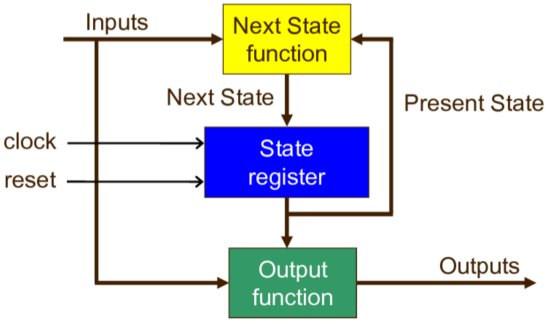
Key Design Steps

1. **Define States:** Identify all unique conditions the system can be in.
2. **Define Transitions:** Rules for moving from one state to another based on inputs.
3. **Define Outputs:** Actions performed in each state or transition.

The Hardware Model of an FSM

Any sequential controller can be modeled using three distinct hardware blocks:

1. **State Register:**
   - A set of Flip-Flops that store the **Present State**.
   - Synchronized by the Clock and Reset.
2. **Next State Logic:**
   - Combinational logic that calculates the **Next State** based on Inputs and Present State.
3. **Output Logic:**
   - Combinational logic that generates **Outputs** based on the State (and optionally Inputs).



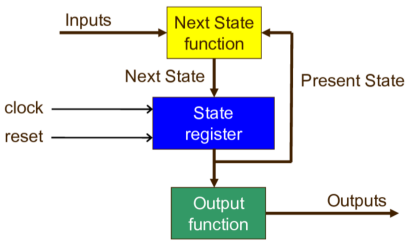
General FSM Architecture

Classification: Moore vs. Mealy Machines

The distinction lies entirely in how the **Outputs** are generated.

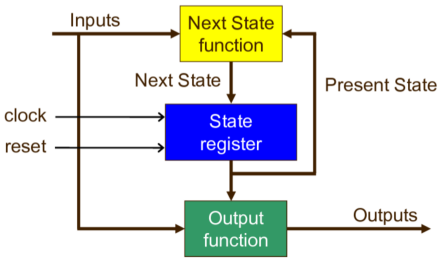
1. Moore Machine

- Outputs depend **ONLY** on the **Present State**.
- $Output = f(State)$
- **Pros:** Safer timing, outputs change synchronously with state transitions.



2. Mealy Machine

- Outputs depend on **Present State AND Inputs**.
- $Output = f(State, Inputs)$
- **Pros:** Faster response (output changes immediately when input changes), fewer states.



Decision Time: Moore vs. Mealy

Which architecture should you choose?

Mealy Machine
"Do more with less."

- **Fewer States:** Often requires fewer states to implement the same logic.
- **Lower Area:** Smaller state register and next-state logic.
- **Faster Response:** Output reacts immediately to input changes (responds 1 clock cycle earlier).

Moore Machine
"Safety First."

- **Glitch-Free:** Outputs are synchronized with the state, filtering out input noise.
- **Timing Safety:** Less likely to affect the critical path of the next stage.
- **Simpler Design:** Easier to visualize and debug.

Rule of Thumb:

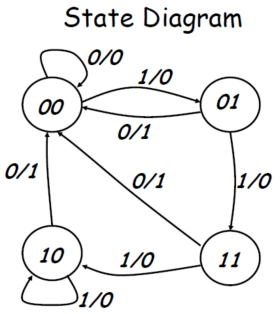
Use **Moore** for safety and simplicity. Use **Mealy** when area or immediate response time is critical.

Visual Representation I: The State Diagram

A graphical graph where nodes are states and edges are transitions.

Key Components:

- **Circles (States):** Each circle represents a unique state (e.g., IDLE, READ).
- **Arcs (Transitions):** Directed arrows showing movement from Present State to Next State.
- **Transition Labels:** The input conditions required to take a specific path.
- **Output Labels:**
  - **Moore:** Written inside the circle (depends only on state).
  - **Mealy:** Written on the arc (depends on transition/input).



Reads as:
When at state ***s1*** and apply input ***I***, we get output ***O*** and proceed to state ***s2***.

Visual Representation II: The State Table

The State Table is essentially the **Truth Table** for a sequential circuit.
It maps every possible combination of Inputs and Present State to the Next State and Outputs.

Structure:

1. **Present State:** Current value of flip-flops.
2. **Inputs:** External signals affecting the system.
3. **Next State:** The calculated state for the next clock cycle ($t + 1$).
4. **Output:** The control signals generated.

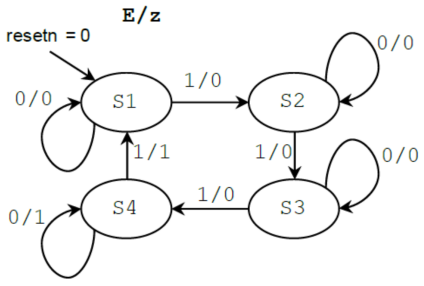
Present State		Input	Next State		Output
A(t)	B(t)	x(t)	A(t+1)	B(t+1)	y(t)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Manual Design Step 1: Capturing Behavior

The Problem: Design a 2-bit counter that counts 00, 01, 10, 11... but only when Enable (E) is high. The output $z = 1$ only when the count is '11'.

1. Draw State Diagram

- **States:** S1(00), S2(01), S3(10), S4(11).
- **Transitions:** If $E = 1$, move to next state. If $E = 0$, hold state (self-loop).



2. Create State Table

- A tabular representation of the diagram.
- Lists Present State, Inputs, Next State, and Outputs.

Input E	Present State	Next State	Outputs C_1	C_0	z
0	S1	S1	0	0	0
0	S2	S2	0	1	0
0	S3	S3	1	0	0
0	S4	S4	1	1	1
1	S1	S2	0	1	0
1	S2	S3	1	0	0
1	S3	S4	1	1	0
1	S4	S1	0	0	1

Manual Design Step 2: State Assignment

- **State Assignment:** We must assign specific binary codes to our abstract states (S1..S4) to map them to Flip-Flops (Q_1, Q_0).
- **Choice:** We choose a binary count mapping: $S1 = 00, S2 = 01, S3 = 10, S4 = 11$.

The Excitation Table

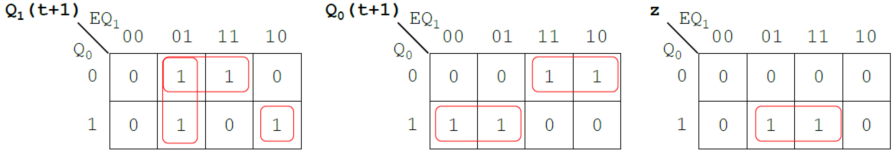
- We replace "S1, S2..." with "00, 01...".
- The "Next State" columns become the required inputs for our Flip-Flops (D-Inputs) for the next clock cycle ($t + 1$).

Input	Present State		Next State		Outputs		
E	Q_1 (t)	Q_0 (t)	Q_1 (t+1)	Q_0 (t+1)	C_1	C_0	z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	0	1	0
1	0	1	1	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	0	0	0	1

Excitation Table (Q_t vs Q_{t+1})

Manual Design Step 3: Logic Minimization

Goal: Find the simplest Boolean equations to drive the Flip-Flop inputs (D_1, D_0) and Output (z).



Karnaugh Maps (K-Maps)

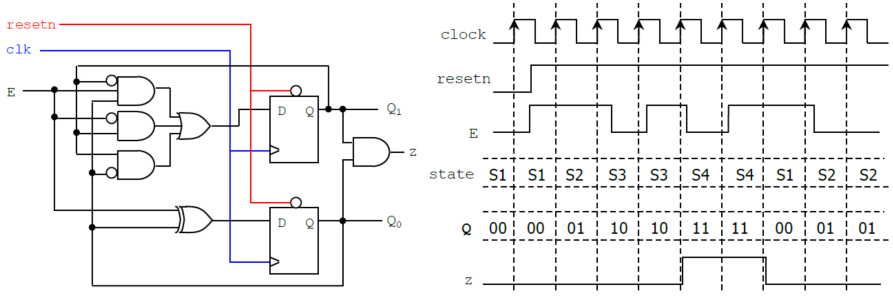
- We map the "Next State" bits from Step 2 into K-Maps.
- Inputs to the map: Q_1, Q_0, E.

Resulting Equations:

- $Q_1(t+1) \leftarrow Q_1\overline{Q_0} + \overline{E}Q_1 + E\overline{Q_1}Q_0$
- $Q_0(t+1) \leftarrow E\overline{Q_0} + \overline{E}Q_0$
- $z = Q_1Q_0$

Manual Design Step 4: Circuit Implementation

- We translate the Boolean equations directly into Logic Gates (AND, OR, XOR).
- These gates drive the D-Flip-Flops, which store the state.



Timing Diagram: Note ***z*** is valid for the whole clock cycle (Moore)
Reflection: This manual process is error-prone and slow for large designs.
Next, we will see how VHDL automates steps 2, 3, and 4!

FSM Modeling in VHDL: Moving to Abstraction

- **The VHDL Advantage:** We skip State Assignment, Excitation Tables, and K-Maps.
- We describe the **behavior** directly. The synthesis tool handles the binary implementation.

Step 1: Defining Symbolic States (Enumerated Types)

- Instead of thinking in bits (e.g., "00", "01"), we use readable symbolic names.

```vhdl
architecture Behavioral of Counter_FSM is
    -- 1. Define the collection of states
    type state_type is (S1, S2, S3, S4);

    -- 2. Declare signals for current and next state
    -- Synthesis tool determines the number of flip-flops
    signal current_state, next_state : state_type;
begin
    -- Processes go here...
```

The Standard VHDL FSM Template

To ensure robust and synthesizable code, we typically split the hardware model into three separate VHDL processes.

Process 1: State Register
(Sequential)

- Registers the state.
- Handles Clock and Reset.
- Updates `current_state`.

Process 2: Next State Logic
(Combinational)

- Looks at `current_state` and `inputs` to decide `next_state`.

Process 3: Output Logic
(Combinational)

- Looks at `current_state` (Moore) or inputs (Mealy) to drive outputs.

This structure directly maps to the theoretical hardware model we discussed earlier.

VHDL Implementation: 1. The State Register Process

- **Purpose:** Infer the D-Flip-Flops that store the state.
- **Type: Sequential Process**. Must be sensitive to Clock (`clk`) and Reset (`rst`).

```vhdl
-- Synchronous Process
state_reg_proc : process(clk, rst)
begin
    if(rst = '1') then
        -- Asynchronous Reset to initial state
        current_state <= S1;
    elsif(rising_edge(clk)) then
        -- Synchronous update on clock edge
        current_state <= next_state;
    end if;
end process;
```

Key Points:

- Usually includes an asynchronous reset to put the FSM in a known starting state (e.g., S1).
- The assignment `<=` happens only on the rising edge of the clock.

VHDL Implementation: 2. Next State Logic Process

- **Purpose:** Calculate where to go next based on input conditions.
- **Type: Combinational Process**. Sensitive to `current_state` and inputs (`E`).

```vhdl
next_state_proc : process(current_state, E)
begin
    -- Default assignment prevents latches
    next_state <= current_state;

    case current_state is
        when S1 =>  -- We are here (00)
            if E = '1' then
                next_state <= S2; -- Move
            end if; -- else stay in S1 (default)

        when S2 =>  -- We are here (01)
            if E = '1' then
                next_state <= S3;
            end if;

        -- ... (Similar for S3, S4) ...
        when others => next_state <= S1;
    end case;
end process;
```

The Translation Method:

1. The `case` represents standing in a specific "Bubble".
2. The `if` statements represent the "Arrows" leaving that bubble.

VHDL Implementation: 3. Output Logic Process

- **Purpose:** Define outputs based on state (Moore) or state+inputs (Mealy).
- **Type: Combinational Process**.
- **Our Example:** A Moore machine. Output ***z*** is '1' only in state S4.

```
output_proc : process(current_state)
begin
    -- Default output value
    z <= '0';

    case current_state is
        when S1 => z <= '0';
        when S2 => z <= '0';
        when S3 => z <= '0';
        when S4 => z <= '1'; -- Output High
    end case;
end process;
```

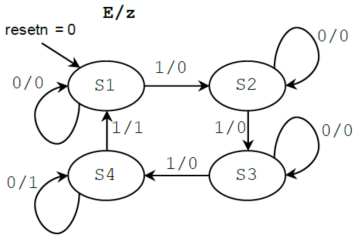
Alternative concise form:

```
z <= '1' when (current_state = S4) else '0';
```

Moore vs. Mealy in Code:

- **Moore (shown):** Sensitivity list has only `current_state`. The `case` checks only the state.
- **Mealy:** Sensitivity list includes inputs (e.g., `E`). Inside the `case`, you use `if E='1' then...` to drive the output.

Complete VHDL Implementation: 2-bit Counter

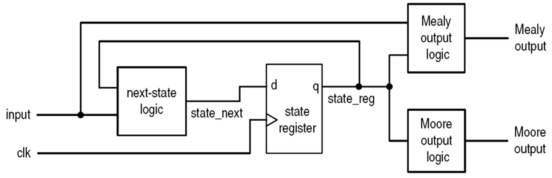



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity counter_fsm is
    port(
        clk, rst : in std_logic;
        E        : in std_logic;
        z        : out std_logic
    );
end entity;
-- Note we use an ACTIVE HIGH rst in the code
```

```vhdl
architecture rtl of counter_fsm is
    type state_type is (S1, S2, S3, S4);
    signal current_state, next_state : state_type;
begin
    -- Process 1: State Register
    process(clk, rst)
    begin
        if rst='1' then
            current_state <= S1;
        elsif rising_edge(clk) then
            current_state <= next_state;
        end if;
    end process;
    -- Process 2: Next State Logic
    process(current_state, E)
    begin
        -- Default prevents latches
        next_state <= current_state;

        case current_state is
            when S1 =>
                if E='1' then next_state <= S2; end if;
            when S2 =>
                if E='1' then next_state <= S3; end if;
            when S3 =>
                if E='1' then next_state <= S4; end if;
            when S4 =>
                if E='1' then next_state <= S1; end if;
        end case;
    end process;

    -- Process 3: Output Logic (Moore)
    -- (Using concurrent assignment for brevity)
    z <= '1' when (current_state = S4) else '0';

end architecture rtl;
```

The Generalized FSM Model

Most real-world FSMs are a hybrid, often featuring both Moore and Mealy outputs.



- **Next-State Logic:** Calculates $State_{next}$ based on *Input* and $State_{reg}$.
- **State Register:** Stores the current state.
- **Moore Output Logic:** Depends only on $State_{reg}$.
- **Mealy Output Logic:** Depends on *Input* AND $State_{reg}$ (creates a direct combinational path from input to output).

State Encoding: Bit Patterns & Trade-offs

How are our symbolic states (e.g., S0, S1) mapped to physical flip-flops?

1. Binary (Sequential)

- Uses minimum FFs ($\lceil \log_2 N \rceil$).
- Good for CPLDs.

2. One-Hot (FPGA Preferred)

- Uses 1 FF per state.
- Logic is faster (comparators are smaller), and FPGAs have abundant registers.

3. Gray Code

- Only 1 bit changes at a time.
- Reduces switching noise and power.

Visual Comparison (4 States):

```
-- Symbolic   |  Binary  |  One-Hot   |  Gray
-- Name       | (2 FFs)  |  (4 FFs)   | (2 FFs)
-- ------------------------------------------
-- S0 (Idle)  |   00     |   0001     |   00
--            |          |            |
-- S1 (Start) |   01     |   0010     |   01
--            |          |            |
-- S2 (Run)   |   10     |   0100     |   11
--            |          |            |
-- S3 (Stop)  |   11     |   1000     |   10
-- ------------------------------------------
-- * One-Hot uses more FFs but simpler logic.
-- * Gray Code minimizes bit toggling.
```

Design Example!

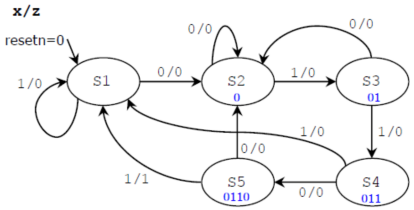
Sequence Detector

Example: Sequence Detector ("01101")

Problem: Design a circuit that detects the sequence **"01101"** in a serial stream `x`. Output `z` goes High when detected. Overlapping is allowed.

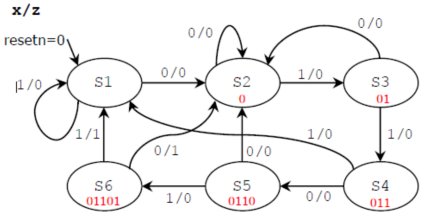
A. Mealy Design (5 States)

- Output depends on State + Input.
- S0(Rst), S1(0), S2(01), S3(011), S4(0110).
- **Transition S4 → S0:** If input is '1', pattern found ("01101"), output '1', next state S0.



B. Moore Design (6 States)

- Output depends on State only.
- Needs extra state **S5** to represent "Sequence Detected".
- S5 outputs '1'.



Implementation A: Mealy FSM ("01101")

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mealy_01101 is
    port ( clk, rst : in std_logic;
           x        : in std_logic;
           z        : out std_logic );
end Mealy_01101;

architecture RTL of Mealy_01101 is
    type state_t is (S0, S1, S2, S3, S4);
    signal curr_s, next_s : state_t;

begin
    -- Process 1: State Register
    process(clk, rst)
    begin
        if rst = '1' then
            curr_s <= S0;
        elsif rising_edge(clk) then
            curr_s <= next_s;
        end if;
    end process;
```

```vhdl
    -- Process 2: Logic (State + Output)
    process(curr_s, x)
    begin
        next_s <= curr_s;  -- Default
        z <= '0';          -- Default

        case curr_s is
            when S0 =>
                if x='0' then next_s <= S1; end if;
            when S1 => -- Got 0
                if x='1' then next_s <= S2; end if;
            when S2 => -- Got 01
                if x='1' then next_s <= S3;
                else next_s <= S1; end if;
            when S3 => -- Got 011
                if x='0' then next_s <= S4;
                else next_s <= S0; end if;
            when S4 => -- Got 0110
                if x='1' then
                    z <= '1'; -- Found!
                    next_s <= S0;
                else
                    next_s <= S1;
                end if;
        end case;
    end process;
end RTL;
```

Implementation B: Moore FSM ("01101")

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Moore_01101 is
    port ( clk, rst : in std_logic;
           x        : in std_logic;
           z        : out std_logic );
end Moore_01101;

architecture RTL of Moore_01101 is
    type state_t is (S0,S1,S2,S3,S4,S5);
    signal curr_s, next_s : state_t;

begin
    -- Process 1: State Register
    process(clk, rst)
    begin
        if rst = '1' then
            curr_s <= S0;
        elsif rising_edge(clk) then
            curr_s <= next_s;
        end if;
    end process;
```

```vhdl
    -- Process 2: Next State Logic
    process(curr_s, x)
    begin
        next_s <= curr_s;

        case curr_s is
          when S0 => if x='0' then next_s<=S1; end if;
          when S1 => if x='1' then next_s<=S2; end if;
          when S2 => if x='1' then next_s<=S3;
                     else next_s<=S1; end if;
          when S3 => if x='0' then next_s<=S4;
                     else next_s<=S0; end if;
          when S4 => if x='1' then next_s<=S5;
                     else next_s<=S1; end if;
          when S5 => -- Found (01101)
                     if x='0' then next_s<=S1;
                     else next_s<=S0; end if;
        end case;
    end process;

    -- Process 3: Output (Moore)
    z <= '1' when (curr_s = S5) else '0';
end RTL;
```

Analysis: Timing Differences ("01101")

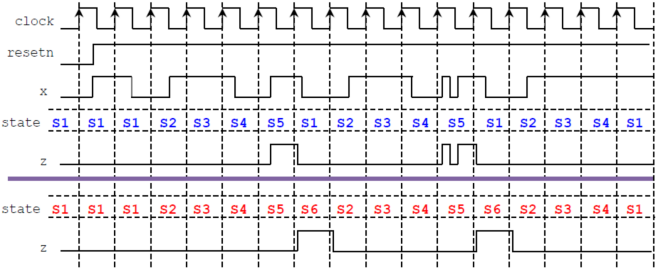
When does `z` actually go high?

Mealy Timing

- **Trigger:** Goes high as soon as `x` becomes '1' while in state S4.
- **Duration:** High as long as `x` is '1' (and clock hasn't ticked yet to change state).
- **Response:** Immediate (Combinational path from X to Z).

Moore Timing

- **Trigger:** Goes high on the **clock edge** that moves the system into state S5.
- **Duration:** High for exactly one clock cycle (until S5 transitions to S1 or S0).
- **Response:** Delayed by 1 clock cycle relative to the input change.



Note

Mealy outputs can "glitch" if the input *x* is noisy.

Happy Design!