DVA494

Programming of Reliable Embedded Systems

Obed Mogaka | Hamid Mousavi | Masoud Daneshtalab

IDT, Malardalens University

January 12, 2026

Today's Agenda

- State Diagram vs. ASM Charts.
- ASM Components
- Register Transfer Methodology
- FSMD and ASMD Models.
- Deriving the ASM Chart:
  - Sequential Binary Multiplier.

State Machine (SM) Charts

- A state machine controls a digital system to carry out a step-by-step algorithm.
- **Problem:** Traditional State Diagrams ("bubbles and arcs") work well for simple logic but become cluttered and hard to read for complex algorithms.

- The **SM Chart** (or ASM Chart).
  - A special type of flowchart used to describe the behavior of a state machine.
  - "ASM" stands for **Algorithmic State Machine**.

- **Why use them?**
  - They resemble software flowcharts (intuitive).
  - They directly represent the hardware timing (Clock Cycles).
  - They serve as a blueprint for VHDL coding.

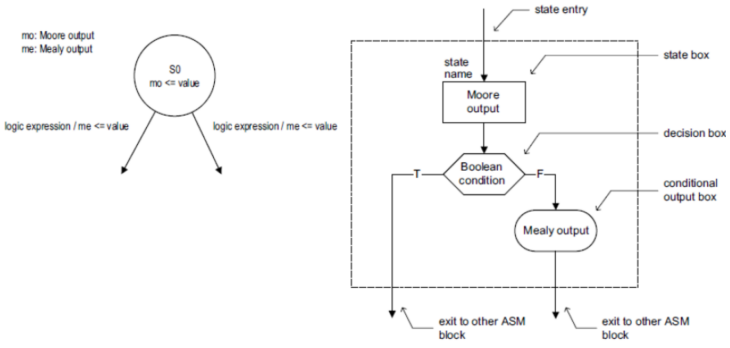
Comparison: State Diagram vs. ASM Chart

1. State Graph

- **Nodes:** Unique states.
- **Arcs:** Transitions labeled with conditions.
- *Drawback:* Hard to visualize complex data operations.

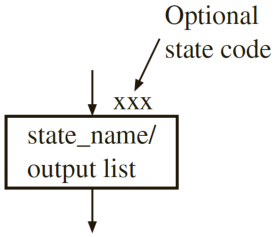
2. ASM Chart

- Composed of a network of **ASM Blocks**.
- More descriptive for applications with complex transitions and actions.



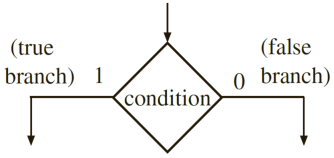
ASM Components: 1. The State Box

- **Function:** Represents a single state in the ASM/FSM.
- The state box contains a *state name* followed by a slash (/) and an optional output list.
  - A *state code* (binary encoding) may be placed above the box.
  - **Output List:** Moore-type outputs (depend only on the present state) and unconditional register-transfer operations that occur whenever the machine is in this state.
- **Timing Rule:** All actions inside the state box occur in parallel and complete within **one clock cycle**.
- Once the next clock edge arrives, the machine transitions to the next state.



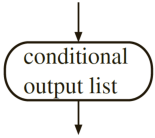
ASM Components: 2. The Decision Box

- **Function:** Implements the control-flow decision used to select the next path in the ASM.
- The decision box is drawn as a *diamond* and represents a Boolean test.
  - Has **one entry point** and **two or more exit paths** (e.g., True/False or multi-way conditions).
  - Contains a **Boolean expression** evaluated based on the current state and input signals (e.g., `Start = '1'` or `Count = Max`).
  - Guides the transition to the appropriate next state or conditional output block.
- **Timing Rule:** The decision box is **purely combinational** and does **not** consume a clock cycle. Its output is evaluated within the same cycle as the state that precedes it.



ASM Components: 3. The Conditional Output Box

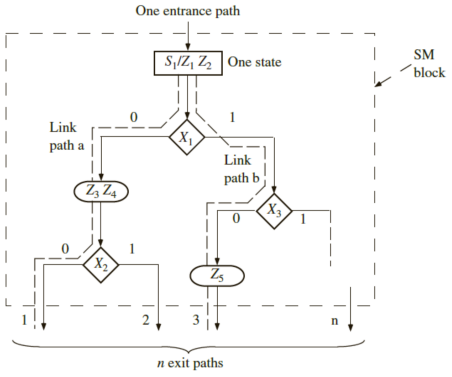
- **Function:** Describes **Mealy-type outputs** or operations that occur only under specific input conditions.
- The conditional output box is drawn as an *oval* (or rounded rectangle).
  - It always appears on a branch coming from a Decision Box.
  - The actions listed inside execute **only** when the Boolean condition leading to this branch is true.
  - Typical uses include: asserting control signals, enabling devices, or performing conditional register-transfer operations.



- **Timing Rule:** This box represents **pure combinational logic**. The outputs respond **immediately** to changes in the inputs and present state— no clock edge is required (asynchronous with respect to the clock).

Putting it Together: The ASM Block

- An ASM chart is formed by connecting multiple **ASM Blocks** in sequence.
- An **ASM Block** consists of:
  - Exactly **one State Box** (the root of the block), and
  - All **Decision Boxes** and **Conditional Output Boxes** that can be reached from that state before entering the next state.
- **Structure:**
  - The block has **one entrance path** (into the state box).
  - It may have **one or many exit paths**, each corresponding to a unique sequence of decisions and conditional outputs.
  - A complete route from entrance to an exit is called a **Link Path**.
- **Purpose:** Encapsulates all possible actions and decisions performed during the **one clock cycle** in which the machine is in that state.



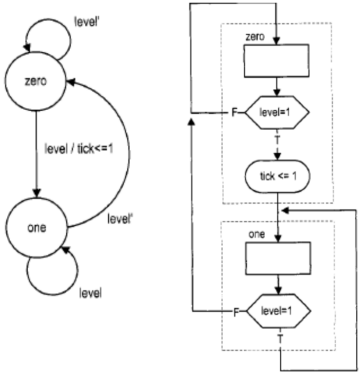
Simple Example: Rising Edge Detector as an ASM Chart

Goal: Detect a rising edge on the input signal `level`. The output `tick` becomes `1` for one clock cycle when `level` changes from `0` to `1`.

State Logic (State Diagram Interpretation):

- **State zero:**
  - Input `level = 0`: stay in zero.
  - Input `level = 1`: – Assert `tick = 1` (rising edge detected) – Go to **state one**.
- **State one:**
  - Input `level = 1`: stay in one.
  - Input `level = 0`: go back to **state zero**.

This is a simple **Mealy machine**: the output `tick` depends on both the current state and the input (`level`).

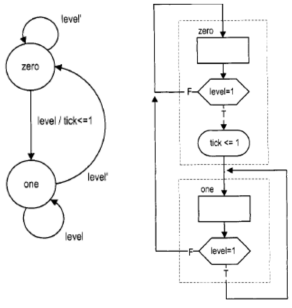


(a) State diagram (b) ASM chart

VHDL Implementation: Rising Edge Detector

Mapping ASM to VHDL:

- The **State Box** corresponds to the `when` clause.
- The **Decision Box** becomes the `if` statement.
- The **Conditional Output** goes *inside* the `if` .



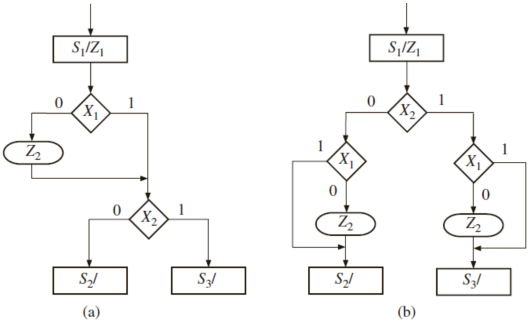
(a) State diagram (b) ASM chart

```vhdl
architecture mealy_arch of edge_detect is
    type state_type is (zero, one);
    signal state_reg, state_next : state_type;
begin
    -- Process 1: State Register (Sequential)
    process(clk, reset)
    begin
        if reset='1' then
            state_reg <= zero;
        elsif rising_edge(clk) then
            state_reg <= state_next;
        end if;
    end process;

    -- Process 2: Next State & Output Logic
    process(state_reg, level)
    begin
        state_next <= state_reg; -- Default
        tick <= '0';             -- Default

        case state_reg is
            when zero =>
                if level = '1' then  -- Decision Box
                    tick <= '1';     -- Conditional Output
                    state_next <= one;
                end if;
            when one =>
                if level = '0' then  -- Decision Box
                    state_next <= zero;
                end if;
        end case;
    end process;
end mealy_arch;
```

Equivalent ASM Charts: Same Logic, Different Forms

An ASM block can be drawn in several different but **functionally equivalent** ways.
The order in which input conditions are tested may change the diagram's shape, **but not the logic it implements**.

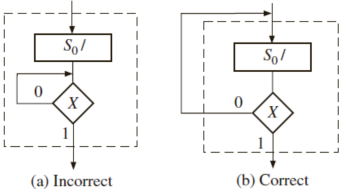
- Two SM blocks may differ in:
  - order of decisions,
  - grouping of conditions,
  - placement of conditional output boxes.
- **They are equivalent if:**
  - all tests lead to the same exit paths, and
  - they generate the same outputs.
- Even combinational circuits can be expressed as ASM charts.



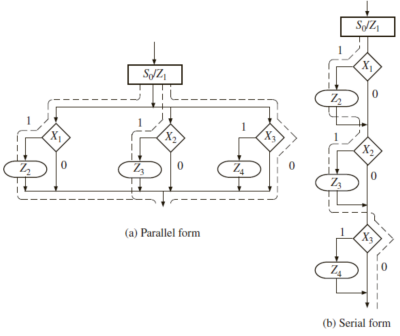
Rules for Constructing an SM Block

Two fundamental rules must always be followed when drawing an ASM (SM) block:

1. **Each valid input combination must produce exactly one exit path.**
   - Ensures deterministic behavior.
   - Multiple exit paths may exist, but only one is taken per cycle.
2. **No internal feedback is allowed inside an SM block.**
   - Feedback must occur **between** SM blocks, never within.
   - Prevents multiple passes through a block in the same clock cycle.



Incorrect vs. correct SM block with feedback.

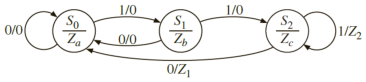


Parallel vs. serial equivalent SM blocks.

From State Graph to SM Chart: The Conversion

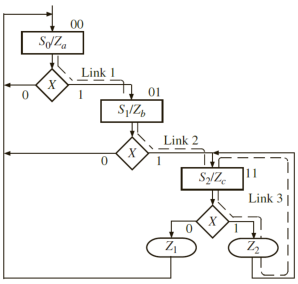
- The conversion process is mechanical and reversible.
- Every State Bubble becomes an **SM Block**.

Source: State Graph



- **Nodes:** S_0, S_1, S_2
- **Moore Out:** Z_a, Z_b, Z_c (Inside node)
- **Mealy Out:** Z_1, Z_2 (On arc)

Target: SM Chart



- **State Box:** Contains Moore outputs (Z_a).
- **Decision Box:** Checks inputs (X).
- **Conditional Box:** Contains Mealy outputs (Z_1).

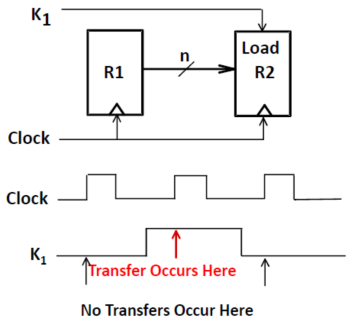
***Note**: Link paths (dashed lines) show the flow of control through one clock cycle.*

Register Transfer Methodology (RTM)

- **Concept:** Complex digital systems are best described by the movement and processing of data between registers.
- **Micro-operations:** Elementary operations performed on data stored in registers during one clock cycle.
  - *Transfer:* `R1 <- R2`
  - *Arithmetic:* `R1 <- R1 + R2`
  - *Logic:* `R1 <- R1 xor R2`
  - *Shift:* `R1 <- R1 « 1`

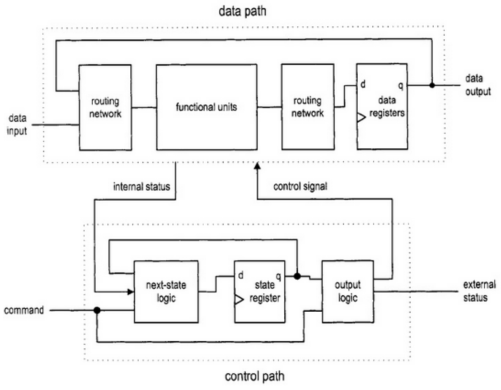
- **Conditional Transfer:** Operations can be controlled by a logic condition (K).

$$\text{If } (K = 1) \text{ then } (R2 \leftarrow R1)$$



The FSMD Model (FSM + Datapath)

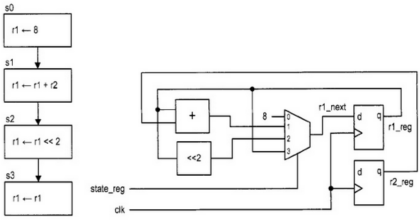
- To implement an algorithm, we separate the circuit into two distinct units:
- **1. The Datapath (Data Processor):**
  - Contains registers, functional units (ALUs), and routing (muxes).
  - Performs the actual data manipulation.
  - Generates **Status Signals** (flags) for the controller.
- **2. The Control Unit (Controller):**
  - An FSM that sequences the operations.
  - Generates **Control Signals** to tell the datapath what to do.



The ASMD Chart (ASM with Datapath)

- This is an extension of the ASM chart that incorporates Register Transfer (RT) operations directly into the boxes.
- **Key Feature:** It links the *timing* (FSM state) with the *action* (Datapath operation).

- **Syntax Update:**
  - **State Box:** Lists register updates that happen *at the next clock edge* if the machine is in this state.
  - **Conditional Box:** Lists register updates that happen *only* if the condition path is taken.



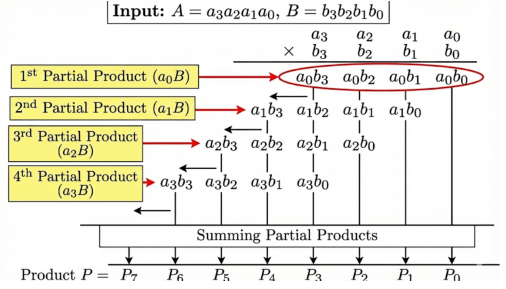
Deriving the ASM Chart: Sequential Binary Multiplier

Sequential Binary Multiplier – Idea and Motivation

Problem Description:
The multiplication of two binary numbers can be done with paper and pencil by successive (i.e., sequential) additions and shifting.

- A **sequential** hardware multiplier mimics this process:
  - processes **one bit** of the multiplier per clock cycle
  - reuses a **single adder**, some registers and a shifter
  - cheaper and smaller than a fully parallel multiplier.

- **Our goal in this section:**
  - understand the algorithm,
  - design a suitable datapath,
  - derive an FSM,
  - and finally express the controller as an **ASM chart**.



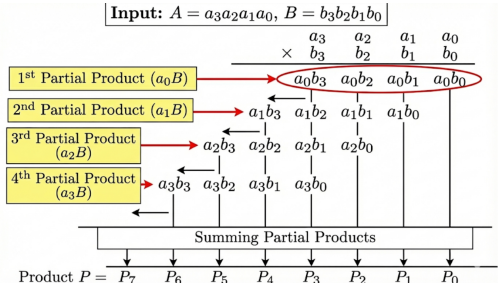
From Paper-and-Pencil Multiplication to Hardware

Paper-and-pencil binary multiplication:

1. Examine each bit of the multiplier, starting from the **LSB**.
2. If the bit is 1, add a suitably shifted copy of the multiplicand to the partial sum.
3. Shift and move to the next bit.

Sequential hardware will:

- inspect **one multiplier bit per clock cycle**,
- optionally perform an **add**, then a **shift**,
- repeat these steps until all bits have been processed.



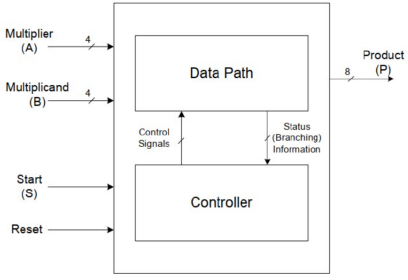
Sequential Binary Multiplier – Interface

External interface:

- Inputs:
  - **A** (multiplicand), **B** (multiplier)
  - **S** (start) – begins a new multiplication
  - **Reset** – clears the internal state
- Output:
  - **P** – final product (up to $2n$ bits for n-bit operands)

Architecture overview:

- **Datapath** performs all arithmetic and shifting.
- **Controller** is an FSM that:
  - sequences the operations in time,
  - generates the datapath **control signals**,
  - receives **status signals** (e.g., LSB, counter zero) from the datapath.



Sequential Binary Multiplication – Algorithm

Informal pseudocode for n-bit operands:

Algorithm 1 Sequential Binary Multiplication

Require: Two n-bit binary numbers: A (multiplicand) and B (multiplier)
Ensure: $P = A \times B$

```
1: P ← 0                                          initial product
2: for i ← 0 to n − 1 do
3:     if B[i] = 1 then
4:         P ← P + (A ≪ i)                        add shifted multiplicand
5:     end if
6: end for
7: return P
```

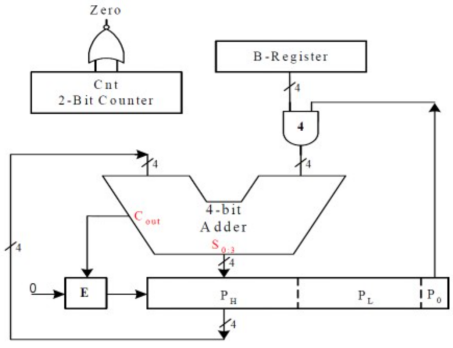
Observation:

- The loop body processes **one bit** of the multiplier.
- In hardware we will:
  - avoid shifting A by i in one step,
  - instead shift the *partial product* and multiplier registers by 1 each cycle.

Multiplier Datapath – Registers and Units

Key registers:

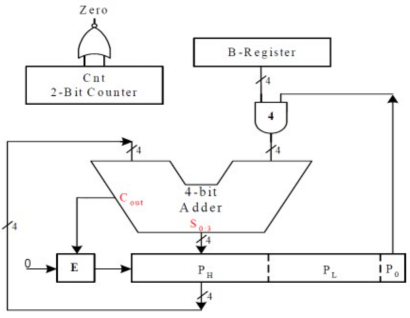
- **B-register** (multiplicand):
  - Stores B for the entire operation.
- **P-register** (partial product):
  - Split into P_H and P_L.
  - Initially: $P_H = 0$, $P_L = A$ (multiplier).
  - Final product is (E, P_H, P_L).
- **E-register**:
  - 1-bit register for the adder carry-out,
  - behaves as the MSB of the product.
- **Cnt**:
  - down-counter initialised to n,
  - determines when all multiplier bits are processed.



Datapath Operation – One Iteration

Per clock cycle (iteration):

1. Look at the least significant bit of P_L (the current multiplier bit).
2. **If that bit is 1:**
   - add B to P_H using the adder,
   - store sum in P_H and carry-out in E.
3. **Shift** the concatenated register (E, P_H, P_L) one bit to the right.
4. **Decrement** the counter.
5. If the counter is not zero, repeat from step 1 on the next cycle.



Important:

- The multiplicand B never changes.
- The multiplier bits are *consumed* as P_L shifts right.
- After n iterations, (E, P_H, P_L) holds the final $2n$-bit product.

Example Walkthrough (Conceptual)

Consider $A = 1011_2$, $B = 1101_2$ (4-bit example).

Initial state:

- $P_H = 0000$, $P_L = A = 1011$, $E = 0$
- Cnt $= 4$

For each cycle:

1. Check LSB of P_L:
   - if $1 \Rightarrow$ add B into P_H,
   - if $0 \Rightarrow$ skip the addition.
2. Shift (E, P_H, P_L) right by one bit.
3. Decrement Cnt.

After 4 cycles:

- Cnt $= 0$ and (E, P_H, P_L) equals the product $A \times B$.
- This cycle-by-cycle behaviour is exactly what our FSM will control.

Datapath Control Signals

Load and clear operations:

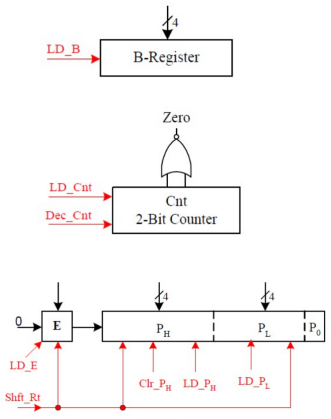
- **Clr_PH** – clear the upper half of the product.
- **Ld_PL** – load the multiplier into P_L.
- **Ld_B** – load the multiplicand into the B-register.
- **Ld_Cnt** – load the loop counter with n.

Runtime operations:

- **Ld_PH** – load adder sum into P_H.
- **Ld_E** – load adder carry-out into E.
- **Shft_Rt** – shift (E, P_H, P_L) right by one bit.
- **Dec_Cnt** – decrement counter by 1.

Status (branching) signals back to controller:

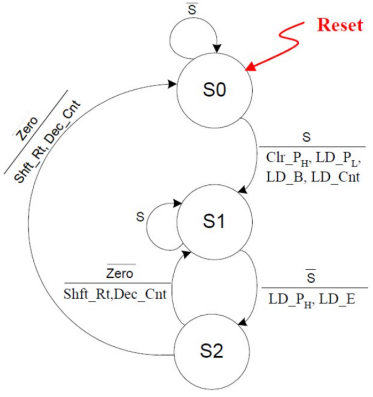
- P_0 – current LSB of P_L (multiplier bit).
- **Zero** – counter has reached zero.



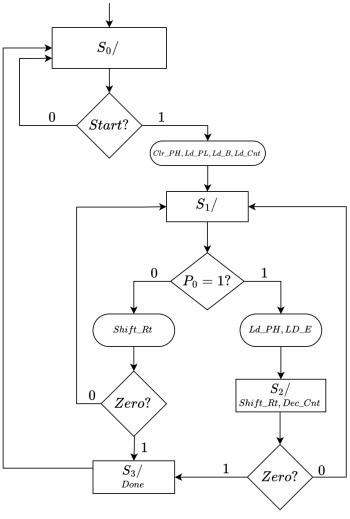
Multiplier Controller – FSM

States (informal meaning):

- S_0 **(Idle/Init):**
  - wait for `Start`,
  - when asserted: load datapath registers and counter.
- S_1 **(Test/Add):**
  - inspect LSB of P_L,
  - if Po=1: perform addition.
- S_2 **(Shift/Loop):**
  - shift and decrement counter,
  - if counter$\neq$ 0: go back to S_1,
  - otherwise: go to **Done**.
- **Done (optional state or flag):**
  - product is ready at (E, P_H, P_L).

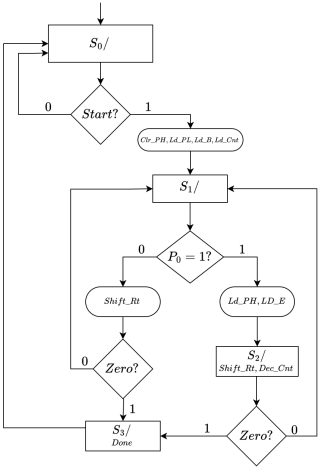


ASM Chart for the Sequential Multiplier



The ASM Chart (Controller)

The controller sequences the operations based on the Multiplier bit (`P0`) and the Counter status (`Zero?`).



State Walkthrough:

- **So (Idle):** Waits for `Start` . If 1, loads all registers (`Ld_B` , `Ld_PL` , etc.).
- **S1 (Test/Process):** Checks LSB of Multiplier (`P0`).
  - **If 1 (Add needed):** Triggers Add (`Ld_PH`) and moves to **S2** to shift later.
  - **If 0 (No Add):** Shifts immediately (`Shift_Rt`). Checks Counter (`Zero?`). If done, go to S3; else loop back to S1.
- **S2 (Shift):** Only entered after an Add. Performs the Shift and Decrement. Checks `Zero?` to decide loop termination.
- **S3 (Done):** Asserts `Done` flag. Returns to So.

VHDL Implementation: The Controller (FSM)

The controller acts as the brain, sequencing operations based on status flags (`start` , `zero_flag`) and driving datapath control signals.

1. State Register & Transitions

```vhdl
    -- FSM State Register (Sequential)
    process (clk, resetn)
    begin
        if resetn = '0' then
            state <= S0;
        elsif rising_edge(clk) then
            state <= next_state;
        end if;
    end process;

    -- Next State Logic (Combinational)
    process (state, start, zero_flag)
    begin
        case state is
            when S0 => -- Idle
                if start = '1' then next_state <= S1;
                else next_state <= S0; end if;

            when S1 => -- Load / Setup
                if start = '0' then next_state <= S2;
                else next_state <= S1; end if;

            when S2 => -- Shift / Compute
                if zero_flag = '0' then next_state <= S1; --
                    Loop
                else next_state <= S0; end if; -- Done

            when others => next_state <= S0;
        end case;
    end process;
```

2. Output Logic (Datapath Control)

```vhdl
    -- Output Logic (Combinational)
    process (state, start)
    begin
        -- Default values (prevent latches)
        load_PH <= '0'; load_PL <= '0'; load_E <= '0';
        load_Cnt <= '0'; shift_Rt <= '0'; clr_PH <= '0';
        dec_Cnt <= '0'; load_B <= '0';

        case state is
            when S0 => -- Initialization
                if start = '1' then
                    clr_PH   <= '1';
                    load_PL  <= '1';
                    load_B   <= '1';
                    load_Cnt <= '1';
                end if;

            when S1 => -- Prepare for Op
                if start = '0' then
                    load_PH <= '1'; -- Add/Load
                    load_E  <= '1';
                end if;

            when S2 => -- Shift & Count
                shift_Rt <= '1';
                dec_Cnt  <= '1';

        end case;
    end process;
```

Summary: From Algorithm to ASM to VHDL

- Began with the **manual** binary multiplication procedure.
- Derived a structured **pseudocode** algorithm.
- Designed a **datapath** with registers, adder, shifter, counter.
- Identified the necessary **control signals** and status signals.
- Built an **FSM** that sequences:
  - initialisation,
  - conditional add,
  - shift/decrement and loop,
  - termination.
- Expressed the FSM as an **ASM chart**:
  - one state box per clock cycle,
  - decision boxes for PLo and Zero,
  - conditional output box for Mealy-style add operation.

This approach generalises: *start from the algorithm, define the datapath, design the FSM, and finally capture it as an ASM chart.*