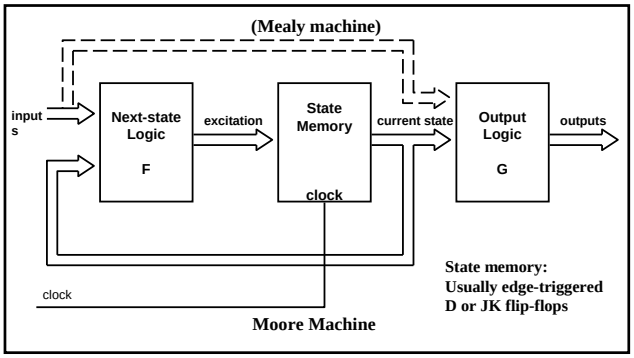
Chapter B
State Machine

VHDL

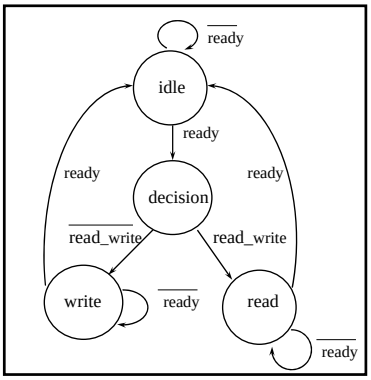
Outline

- Finite State Machine
  - Moore & Mealy FSM
  - Memory Control Example
  - Extended Memory Controller
  - Two & Three Process State Machine
  - One Process State Machine
- Algorithmic State Machine
  - Simple Example (“111…1” Detector)
  - Multiplier Example

Finite State Machine Structure



Memory Controller Example



	outputs	
state	oe	we
idle	0	0
decision	0	0
write	0	1
read	1	0

Memory Controller (Conventional Design)

State	PS: $q_0 q_1$	read_write, ready 0 0	0 1	1 1	1 0	Outputs oe	we
idle	0 0	0 0	0 1	0 1	0 0	0	0
decision	0 1	1 1	1 1	1 0	1 0	0	0
write	1 1	1 1	0 0	0 0	1 1	0	1
read	1 0	1 0	0 0	0 0	1 0	1	0

NS: $Q_0 Q_1$

$$Q_0 = q_0 \overline{q}_1 + q_0 \overline{ready} \qquad we = q_0 q_1$$

$$Q_1 = \overline{q}_0 \overline{q}_1 ready + \overline{q}_0 q_1 \overline{read\_write} + q_0 \overline{q}_1 ready \qquad oe = q_0 \overline{q}_1$$

State Machines In VHDL

- **A given state diagram can be easily translated into a high level VHDL description without having to generate the state transition table.**
- **In VHDL each state can be translated to a case in a CASE-WHEN construct.**
- **The state transitions can then be specified in IF-THEN-ELSE statements.**

Memory Controller Using VHDL (p1)

```
ENTITY example IS PORT (
  read_write, ready, clk :   IN bit;
  oe, we                 :   OUT bit);
END example;

ARCHITECTURE state_machine OF example IS
  TYPE StateType IS (idle, decision, read, write);
  SIGNAL present_state, next_state : StateType;
BEGIN
  state_comb:PROCESS (present_state, read_write, ready)
  BEGIN
    CASE present_state IS

      WHEN idle =>  oe <= '0'; we <= '0';
        IF ready = '1' THEN
          next_state <= decision;
        ELSE
          next_state <= idle;
        END IF;
```


Memory Controller Using VHDL (p2)

```
      WHEN decision =>  oe <= '0'; we <= '0';
        IF (read_write = '1') THEN
          next_state <= read;
        ELSE
          next_state <= write;
        END IF;
      WHEN read =>  oe <= '1'; we <= '0';
        IF (ready = '1') THEN
          next_state <= idle;
        ELSE
          next_state <= read;
        END IF;
       WHEN write =>  oe <= '0'; we <= '1';
        IF (ready = '1') THEN
          next_state <= idle;
        ELSE
          next_state <= write;
        END IF;
    END CASE;
  END PROCESS state_comb;
```


Memory Controller Using VHDL (p3)

```
  state_clocked:PROCESS(clk)
  BEGIN
    IF (clk'event and clk='1') THEN
      present_state <= next_state;
    END IF;
  END PROCESS state_clocked;

END ARCHITECTURE state_machine;
```


Synchronous Reset in a Two-Process FSM

- Rather than specifying the reset transition to the idle state in each branch of the case statement, IF-THEN-ELSE statements is used either at the beginning of the process or at the end of the process as its last statement
- Reset in the beginning of the process:

```
state_comb:PROCESS(reset, present_state, read_write, ready)
  BEGIN
    IF (reset = '1') THEN
      oe <= '-'; we <= '-';              -- needed to avoid
                                         -- creating latches
      next_state <= idle;
    ELSE
      CASE present_state IS
        …
      END CASE;
    END IF;
  END PROCESS state_comb;
```

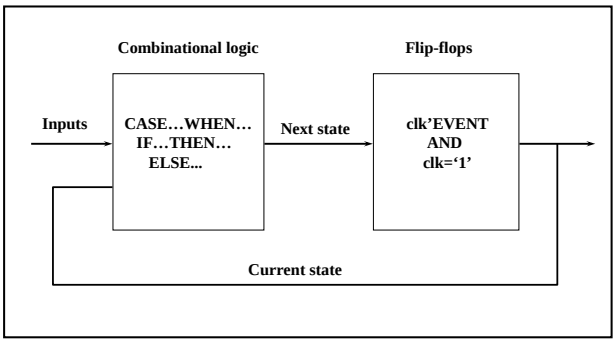

Asynchronous Reset in a Two-Process FSM

- An asynchronous reset is written as follows:

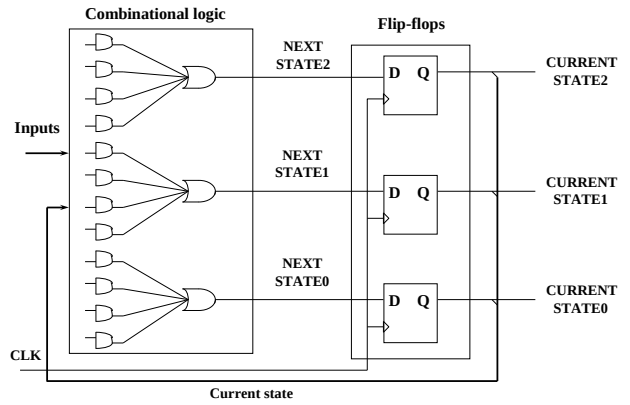
```
state_clocked:PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN
      present_state <= idle;
    ELSIF rising_edge(clk) THEN
      present_state <= next_state;
    END IF;
  END PROCESS state_clocked;
```

- If reset is asserted, then the state machine remains idle regardless of the value of clk.
- If the reset is not asserted and the clock change is from '0' to '1', then the state machine transitions to the next state.
- Asynchronous reset is better than synchronous reset when used only during initialization or system failure, because synchronous reset may require additional device resources (such as product terms)

Code Structure - CPLD Architecture Analogy



Code Structure - CPLD Architecture Analogy



Code Structure - CPLD Architecture Analogy

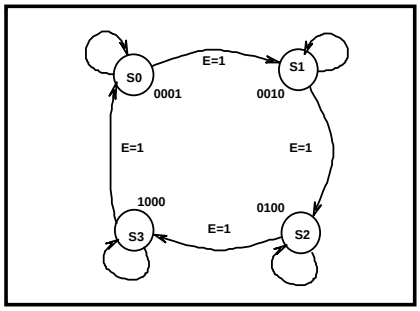
- One process describes the combinational logic and another describes synchronization of state transitions to the clock
- Decoding of present_state and inputs is performed in the combinatorial portion (product-term array) of the logic block just as it is described in the combinatorial process in the code
- Synchronization of next_state is described in the state_clocked process, which describes a bank of registers such as the macrocells in a logic block

One-Process FSM

- One Process FSM
  - State transitions and transition synchronization to the clock are all described in one process.
  - Outputs are described:
    - using concurrent signal assignments (outside the process)
    - using assignments in condition (inside the process)
    - using another process
- One Process FSM code example
  - Functionally equivalent to two-process FSM with the use of an asynchronous reset instead of a synchronous reset

Simple Example – OneHot Counter

- 4 states.
- With enable



FSM Example Entity

```
-- Writren by Flaxer Eli
-- Two Process FSM
library ieee;
use ieee.std_logic_1164.all;

entity FSM is port (
   Clk         :      in std_logic;
   Reset       :      in std_logic;
   Enable      :      in std_logic;
   FsmOut      :      out std_logic_vector(3 downto 0));
end FSM;
```

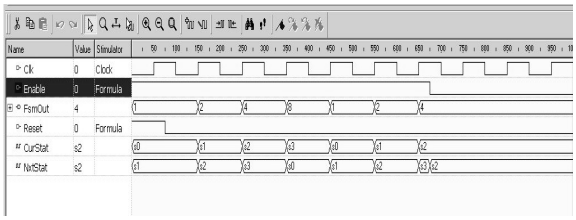

Two-Process FSM Example (p1)

```
architecture Flaxer of FSM is
TYPE    StatusType    IS (S0, S1, S2, S3);
-----------------------------------------------------------------
signal CurStat, NxtStat :StatusType;
begin
-----------------------------------------------------------------
NextState: process(CurStat, Enable)
  begin
   CASE CurStat IS
        WHEN S0 =>     IF Enable = '1' THEN
                              NxtStat <= S1;
                       ELSE
                              NxtStat <= S0;
                       END IF;
                       FsmOut <= "0001";

        WHEN S1 =>     IF Enable = '1' THEN
                              NxtStat <= S2;
                       ELSE
                              NxtStat <= S1;
                       END IF;
                       FsmOut <= "0010";
```

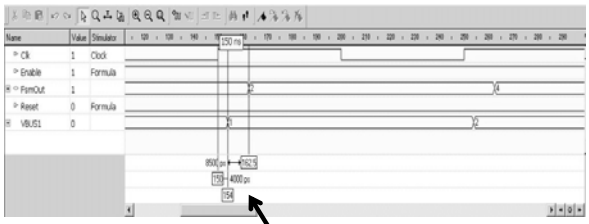
Two-Process FSM Example (p2)

```
        WHEN S2 =>      IF Enable = '1' THEN
                                NxtStat <= S3;
                        ELSE
                                NxtStat <= S2;
                        END IF;
                        FsmOut <= "0100";

        WHEN S3 =>      IF Enable = '1' THEN
                                NxtStat <= S0;
                        ELSE
                                NxtStat <= S3;
                        END IF;
                        FsmOut <= "1000";
   END CASE;
end process;

Update: process(Clk, Reset)
begin
   IF Reset = '1' THEN
        CurStat <= S0;
   ELSIF Rising_Edge(Clk) THEN
        CurStat <= NxtStat;
   END IF;
end process;
end Flaxer;
```


Two-Process FSM Simulation - Pre Syn



Two-Process FSM Simulation - Post Syn



Delay of 4000+8500

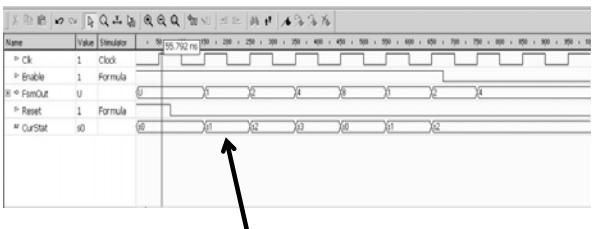
One-Process FSM Example Ver1 (p1)

```
architecture Flaxer of FSM is

TYPE    StatusType    IS (S0, S1, S2, S3);
-----------------------------------------------------------------
signal CurStat :StatusType;
begin
-----------------------------------------------------------------
NextState: process(Clk, Reset)
  begin
   IF Reset = '1' THEN
        CurStat <= S0;
   ELSIF Rising_Edge(Clk) THEN
     CASE CurStat IS
        WHEN S0 =>      IF Enable = '1' THEN
                                CurStat <= S1;
                        ELSE
                                CurStat <= S0;
                        END IF;
                        FsmOut <= "0001";

        WHEN S1 =>      IF Enable = '1' THEN
                                CurStat <= S2;
                        ELSE
                                CurStat <= S1;
                        END IF;
                        FsmOut <= "0010";
```


One-Process FSM Example Ver1 (p2)

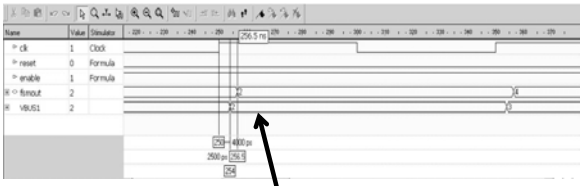
```
        WHEN S2 =>      IF Enable = '1' THEN
                                CurStat <= S3;
                        ELSE
                                CurStat <= S2;
                        END IF;
                        FsmOut <= "0100";

        WHEN S3 =>      IF Enable = '1' THEN
                                CurStat <= S0;
                        ELSE
                                CurStat <= S3;
                        END IF;
                        FsmOut <= "1000";
    END CASE;
   END IF;
end process;
-----------------------------------------------------------------
end Flaxer;
```


One-Process FSM Simulation - Pre Syn



Wrong Result – Delay in State

One-Process FSM Simulation - Post Syn



Wrong Result – Delay in State

But better Delay of 4000+2500

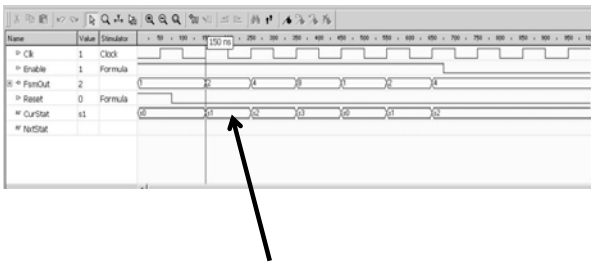
One-Process FSM Example Ver2 (p1)

```
architecture Flaxer of FSM is

TYPE    StatusType    IS (S0, S1, S2, S3);
-------------------------------------------------------------------
signal CurStat :StatusType;
begin
-------------------------------------------------------------------
NextState: process(Clk, Reset)
  begin
   IF Reset = '1' THEN
        CurStat <= S0;
   ELSIF Rising_Edge(Clk) THEN
     CASE CurStat IS
        WHEN S0 =>      IF Enable = '1' THEN
                              CurStat <= S1;
                        ELSE
                              CurStat <= S0;
                        END IF;

        WHEN S1 =>      IF Enable = '1' THEN
                              CurStat <= S2;
                        ELSE
                              CurStat <= S1;
                        END IF;

        WHEN S2 =>      IF Enable = '1' THEN
                              CurStat <= S3;
                        ELSE
                              CurStat <= S2;
                        END IF;
```


One-Process FSM Example Ver2 (p2)

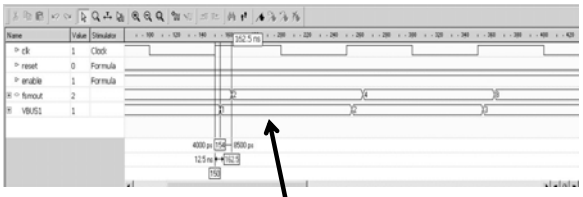
```
        WHEN S3 =>      IF Enable = '1' THEN
                              CurStat <= S0;
                        ELSE
                              CurStat <= S3;
                        END IF;
    END CASE;
   END IF;
end process;
-------------------------------------------------------------------

OutDecode: process(CurStat)
  begin
   CASE CurStat IS
        WHEN S0 =>  FsmOut <= "0001";
        WHEN S1 =>  FsmOut <= "0010";
        WHEN S2 =>  FsmOut <= "0100";
        WHEN S3 =>  FsmOut <= "1000";
     END CASE;
end process;
-------------------------------------------------------------------

end Flaxer;
```


One-Process FSM Simulation - Pre Syn



Right Result – No Delay in State

One-Process FSM Simulation - Post Syn



Right Result – No Delay in State

Delay of 4000+8500

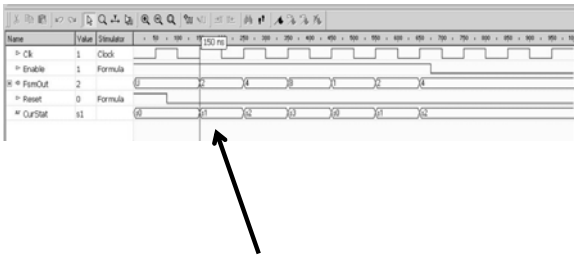
One-Process FSM Example Ver3 (p1)

```
architecture Flaxer of FSM is

TYPE    StatusType    IS (S0, S1, S2, S3);
-------------------------------------------------------------------
signal CurStat :StatusType;
begin
-------------------------------------------------------------------
NextState: process(Clk, Reset)
  begin
   IF Reset = '1' THEN
        CurStat <= S0;
        FsmOut <= "0001";
   ELSIF Rising_Edge(Clk) THEN
     CurStat <= CurStat;
     CASE CurStat IS
        WHEN S0 =>      IF Enable = '1' THEN
                              CurStat <= S1;
                              FsmOut <= "0010";
                        ELSE
                              CurStat <= S0;
                              FsmOut <= "0001";
                        END IF;

        WHEN S1 =>      IF Enable = '1' THEN
                              CurStat <= S2;
                              FsmOut <= "0100";
                        ELSE
                              CurStat <= S1;
                              FsmOut <= "0010";
                        END IF;
```

One-Process FSM Example Ver3 (p2)

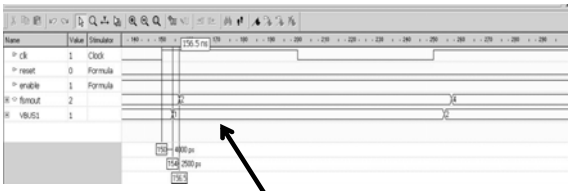
```
        WHEN S2 =>      IF Enable = '1' THEN
                                CurStat <= S3;
                                FsmOut <= "1000";
                        ELSE
                                CurStat <= S2;
                                FsmOut <= "0100";
                        END IF;
        WHEN S3 =>      IF Enable = '1' THEN
                                CurStat <= S0;
                                FsmOut <= "0001";
                        ELSE
                                CurStat <= S3;
                                FsmOut <= "1000";
                        END IF;
    END CASE;
  END IF;
end process;
-------------------------------------------------------------------
end Flaxer;
```


One-Process FSM Simulation - Pre Syn



Right Result – No Delay in State

One-Process FSM Simulation - Post Syn



Right Result – No Delay in State
Delay of 4000+2500

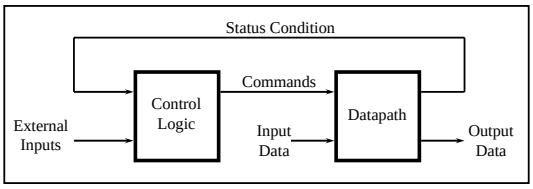
State and Output Assignments in One-Process FSM

- Output Assignments
  - In this example these assignments are made using concurrent signal assignment
  - Recommended because outputs and transitions are easily identified
  - Outputs could be assigned in the CASE state branches instead
  - Not recommended because the code is not as clear to the reader, and the outputs would have to be defined in terms of the next state since they would be registered
- Original two-process FSM recommended generally
  - Easiest to create and maintain, especially for large FSMs
  - Simple clocked process for state register
  - Simple combinational process for state transitions and outputs

Algorithmic State Machine

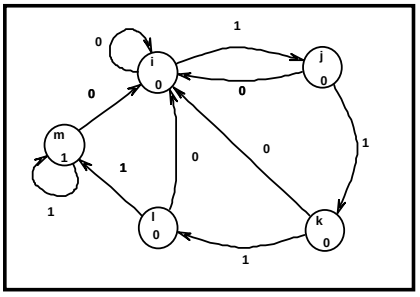
Control and Datapath Interaction

- The relationship between the control logic and the data processing in a digital system is shown below. The data processing path, commonly referred to as the datapath, manipulates data in registers according to the system's requirements. The control logic initiates a sequence of commands to the datapath. The control logic uses status conditions from the datapath to serve as decision variables for determining the sequence of control signals.



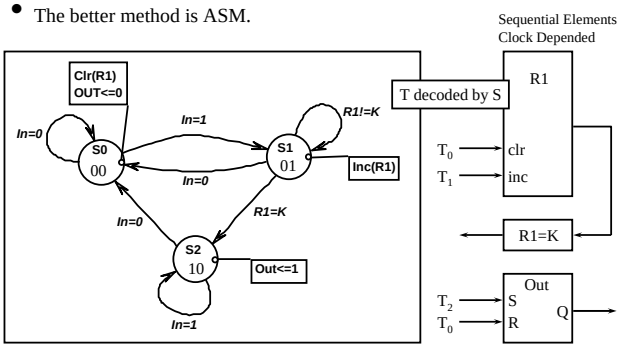
Simple Example

- Machine that detect sequence of K ‘1’ in the input.
- In the exercise we show the solution for K=4:



Simple Example (cont.)

- But if K=1000 it is very uneasy to use this method.
- The better method is ASM.



“1111…1” Detector Using VHDL (p1)

```
ENTITY example IS
 GENERIC (LEN : integer := 1000);
 PORT (Din, clk     :IN bit;
       Find         :OUT bit);
END example;

ARCHITECTURE state_machine OF example IS
  TYPE StateType IS (S0, S1, S2);
  SIGNAL present_state, next_state : StateType;
  SIGNAL n: integer;
BEGIN
  nextStateProc: PROCESS (present_state, Din, n)
  BEGIN
    CASE present_state IS
      WHEN S0 =>
        IF Din = '1' THEN
          next_state <= S1;
        ELSE
          next_state <= S0;
        END IF;
```

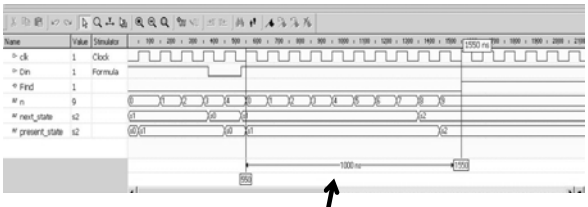

“1111…1” Detector Using VHDL (p2)

```
      WHEN S1 =>
        IF (Din = ’0’) THEN
          next_state <= S0;
        ELSE
              IF (n < LEN-2) THEN
                    next_state <= S1;
              ELSE
                    next_state <= S2;
              END IF;
        END IF;
      WHEN S2 =>
        IF (Din = '1') THEN
          next_state <= S2;
        ELSE
          next_state <= S0;
        END IF;
    END CASE;
  END PROCESS nextStateProc;
```


“1111…1” Detector Using VHDL (p3)

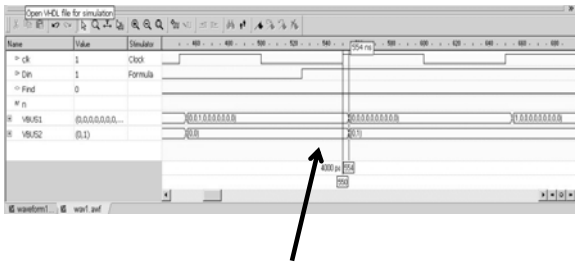
```
updateState: PROCESS(clk)
 BEGIN
   IF (clk'event and clk='1') THEN
      present_state <= next_state;
      CASE present_state IS
            WHEN S0 =>  Find <= '0'; n <= 0;
            WHEN S1 =>  n <= n+1;
            WHEN S2 =>  Find <= '1';
      END CASE;
   END IF;
 END PROCESS updateState;
```


ASM Simulation - Pre Syn



Right Result – Only for LEN-2

ASM Simulation - Post Syn



Reference to last state

Binary Multiplier

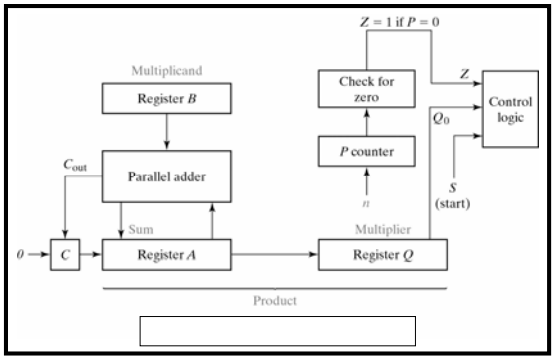
- The multiplication of two binary numbers is done with paper and pencil by successive additions and shifting. This process is best illustrated with a numerical example. Let us multiply the two binary numbers 10111 and 10011:

23	10111	Multiplicand
19	10011	Multiplier
	10111	
	10111	
	00000	
	00000	
	10111	
437	110110101	Product

Binary Multiplier - Hardware

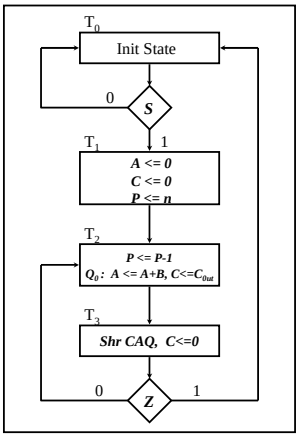
- When the multiplication procedure is implemented with digital hardware, it is convenient to change the process slightly.
- First, instead of providing digital circuits to store and add simultaneously as many binary numbers as there are 1‘s in the multiplier, it is less expensive to provide circuits for the summation of only two binary numbers and successively accumulate the partial products in a register.
- Second, instead of shifting the multiplicand to the left, the partial product being formed is shifted to the right. This leaves the partial product and the multiplicand in the required relative positions.
- Third, when the corresponding bit of the multiplier is 0, there is no need to add all 0’s to the partial product, since this will not alter its resulting value.

Block Diagram of the Multiplier



ASM Chart for the Multiplier

n = width of register

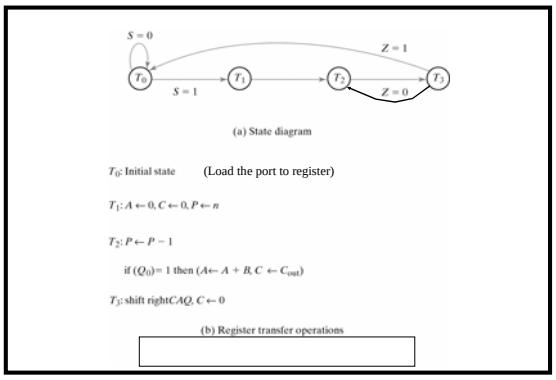


A <= shr A, A_{n-1} <= C
Q <= shr Q, Q_{n-1} <= A_0
C <= 0

Control Logic Design

- The design of a digital system can be divided into two parts: the design of the register transfers in the datapath and the design of the **control logic**.
- The control logic design is a sequential circuit design problem. As such, it may be convenient to formulate the state diagram of the sequential control.
- An ASM chart is similar to a state diagram. The rectangular blocks that designate state boxes are the states of the sequential circuit. The diamond shaped blocks that designate decision boxes determine the conditions for the next state transition in the state diagram.

State Diagram of Control



(a) State diagram

T_0: Initial state (Load the port to register)

T_1: $A \leftarrow 0, C \leftarrow 0, P \leftarrow n$

T_2: $P \leftarrow P - 1$

if $(Q_0) = 1$ then $(A \leftarrow A + B, C \leftarrow C_{out})$

T_3: shift right CAQ, $C \leftarrow 0$

(b) Register transfer operations

Multiplier Using VHDL (p1)

```
ENTITY example IS PORT (
  Start, clk :   IN  std_logic;
  X1, X2     :   IN  unsigned(7 DOWNTO 0);
  Rez        :   OUT unsigned(15 DOWNTO 0));
END example;

ARCHITECTURE state_machine OF example IS
  TYPE StateType IS (S0, S1, S2, S3);
  SIGNAL present_state, next_state : StateType;
  SIGNAL P    : integer;
  SIGNAL RCAQ : unsigned(16 DOWNTO 0); -- C, A and Q
  SIGNAL RB   : unsigned(7 DOWNTO 0); -- B

  ALIAS  RCA: unsigned(8 DOWNTO 0) IS RCAQ(16 DOWNTO 8);
  ALIAS  RA: unsigned(7 DOWNTO 0) IS RCAQ(15 DOWNTO 8);
  ALIAS  RQ: unsigned(7 DOWNTO 0) IS RCAQ(7 DOWNTO 0);

BEGIN
```


Multiplier Using VHDL (p2)

```
nextState: PROCESS (present_state, S, X1, X2)
BEGIN
  CASE present_state IS
    WHEN S0 =>
      IF Start = '1' THEN
        next_state <= S1;
      ELSE
        next_state <= S0;
      END IF;
    WHEN S1 =>
      next_state <= S2;
    WHEN S2 =>
      next_state <= S3;
    WHEN S3 =>
      IF P = 0 THEN
        next_state <= S0;
      ELSE
        next_state <= S2;
      END IF;
END PROCESS nextState;
```


Multiplier Using VHDL (p3)

```
updateReg: PROCESS (clk)
BEGIN
  IF (clk'event and clk='1') THEN
    CASE present_state IS

      WHEN S0 =>  RB <= X1; RQ <= X2;

      WHEN S1 =>  RCA <= (OTHERS => '0'); P <= 8;

      WHEN S2 =>  P <= P-1;
        IF (RQ(0) = '1') THEN
          RCA <= ('0'& RA) + ('0'& RB);
        END IF;

      WHEN S3 => RCAQ <= '0' & RCAQ(16 DOWNTO 1);
    END CASE;
  END IF;
END PROCESS updateReg;
```


Multiplier Using VHDL (p4)

```
  PROCESS(clk)
  BEGIN
    IF (clk'event and clk='1') THEN
      present_state <= next_state;
    END IF;
  END PROCESS;

END ARCHITECTURE state_machine;
```