Digital System Design Based on Data Path and Control Unit Partitioning

Dr. Mahdi Abbasi

Computer Engineering Department

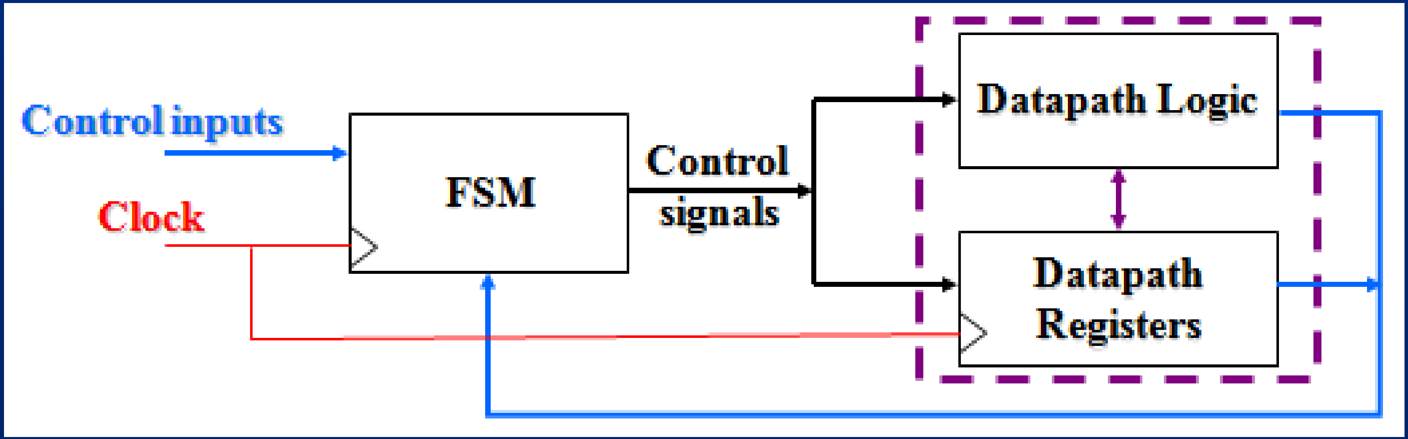
Bu Ali Sina University

Outline

- Data Path & Control Unit Partitioning
- Traffic Light Controller Design
- Algorithmic State Machine (ASM) Chart
- Design Examples

Digital Systems

- **Digital systems**
  - Control-dominated systems :
    being reactive systems responding to external events, such as traffic controllers, elevator controllers, etc.
  - Data-dominated systems :
    requiring high throughput data computation and transport such as telecommunications and signal processing
- **Sequential machines are commonly partitioned into data path units and control units**

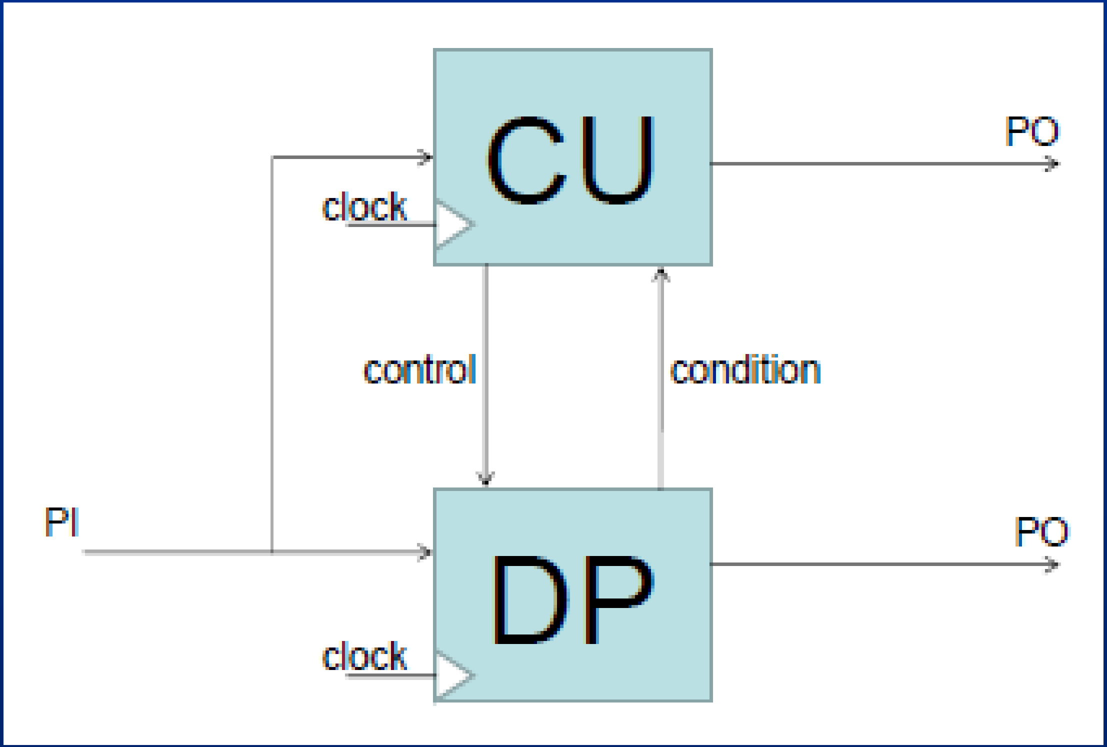


Data Path & Control Unit Partitioning

- **A common design practice decomposes the system in two parts:**
  - A Data Path (DP): a collection of interconnected modules that perform all the relevant computation on the data: it can use both combinational and sequential components
  - A Control Unit (CU) that coordinates the behavior of the Data Path by issuing appropriate control signals that guarantee the correct sequence of operations: it is typically designed as a single or cooperating FSMs
- **DP and CU communicate through 2 types of signals:**
  - Control signals are output of the CU to the DP and correctly synchronize the operations
  - Condition signals (or flags) are sent from the DP to the CU to indicate certain data dependent conditions (that could influence future behavior)

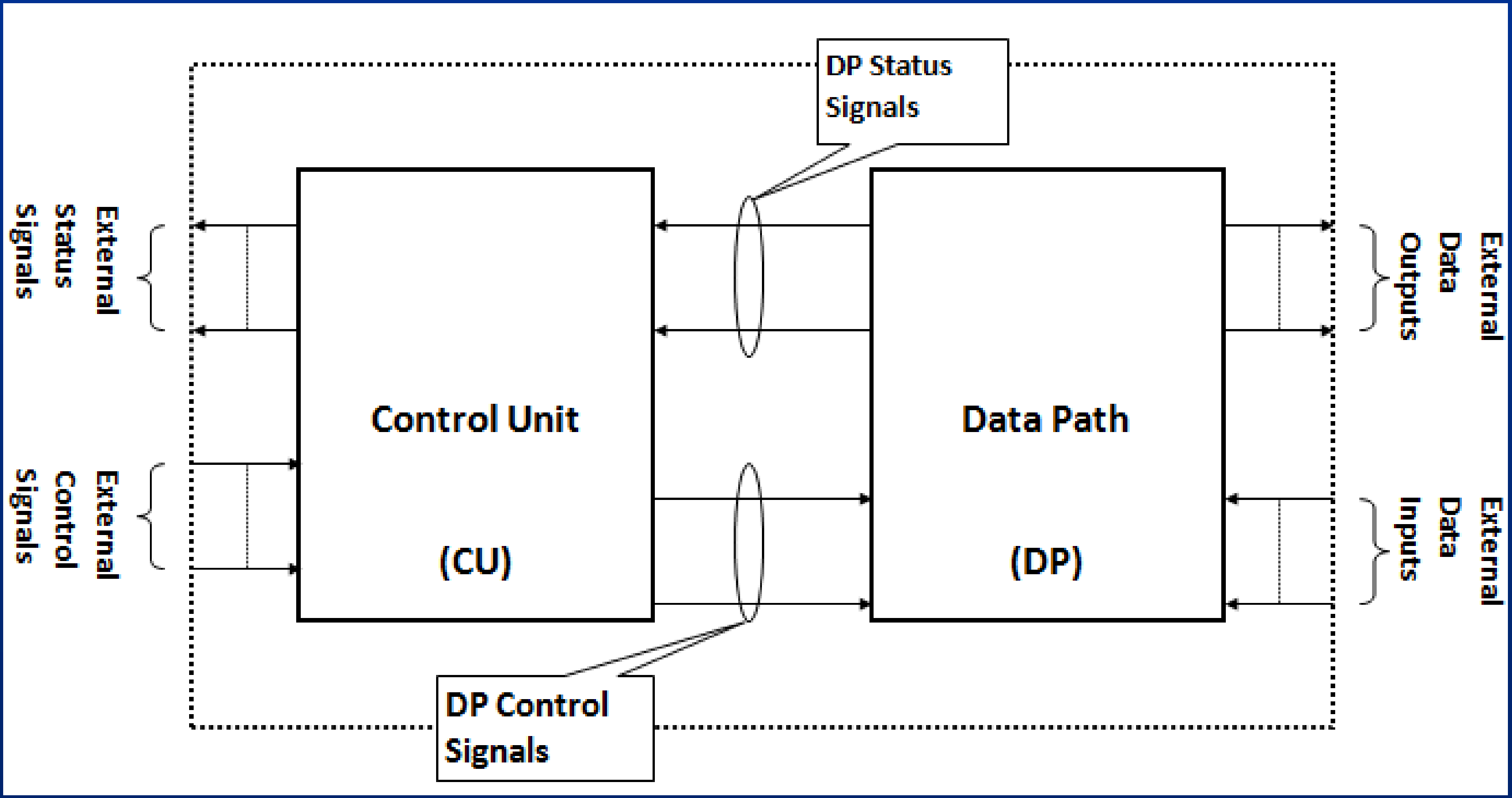
Data Path & Control Unit Partitioning

- Both DP and CU might receive the system's inputs (Primary inputs) and generate its outputs (Primary outputs).



Data Path & Control Unit Partitioning

- The general structure of a digital system that performs a specific task(s) is as follows:



Data Path & Control Unit Partitioning

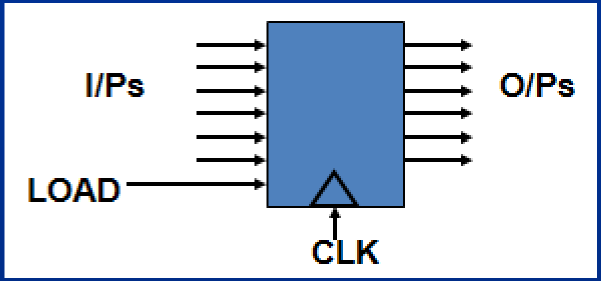
- **External Control Signals**: Specify the task required from the whole circuit (e.g. calculate the average of some integers)
- **External Status Signals**: Indicate the status of the whole circuit (e.g. finished processing, error or overflow ...etc.)
- **External Data Inputs/Outputs**: Data going into the circuit or out of it (e.g. the integers to be averaged and their average)
- **DP Control Signals**: Signals generated by the CU to control different blocks in the DP (e.g. Shift Registers, Counters, MUXs ...etc.)
- **DP Status Signals**: Signals that indicate the status of some blocks in the DP (e.g. when a counter reaches 7 or when an adder produces a carry or an overflow, or when the sign bit of the result is negative ...etc.)

Data Path Design

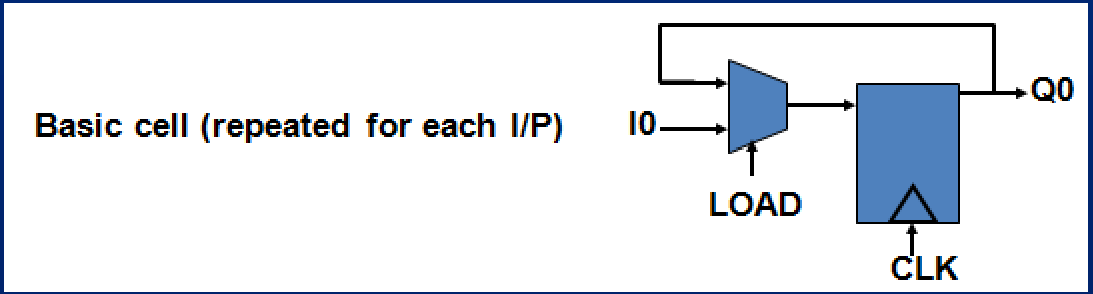
- **The data path contains blocks that only deal with data; they do not provide control to any other blocks and need to be controlled (by the CU).**
- **Data Path blocks can be viewed as the workers that perform certain tasks (on the data) who need to be managed by someone (in this case the CU is the manager that tells every 'worker' in the Data Path what to do).**
- **Examples of Data Path blocks:**
  - Registers, Counters, Multiplexors, Decoders, Logic Circuits (AND, OR, etc)
  - Arithmetic Circuits:
    - Adders, Subtractors, Comparators, Multipliers, Square root, etc.

Registers

- Parallel load registers to read data in parallel

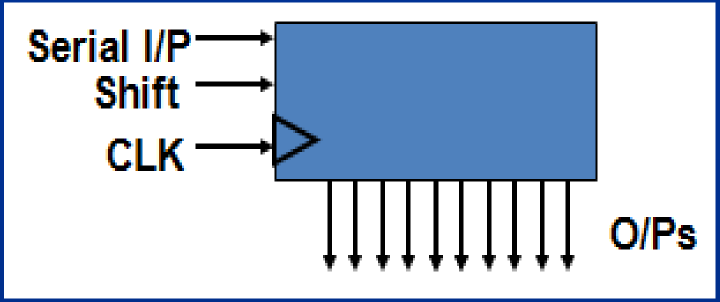


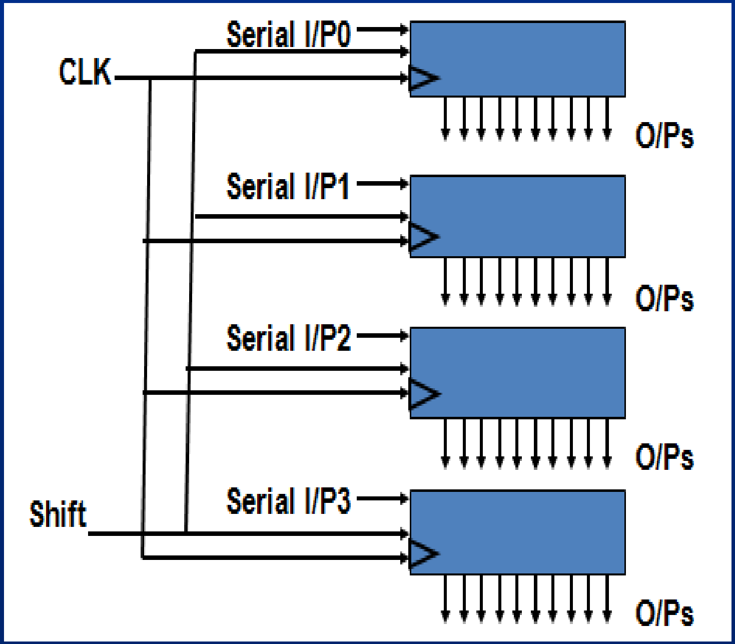
- Load is a synchronous control to control reading the data. When LOAD is inactive, the register keeps the data as is
- Used when we want to read data as fast as possible (in one clock cycle)
- Implemented using D-FFs and MUXs



Shift Registers

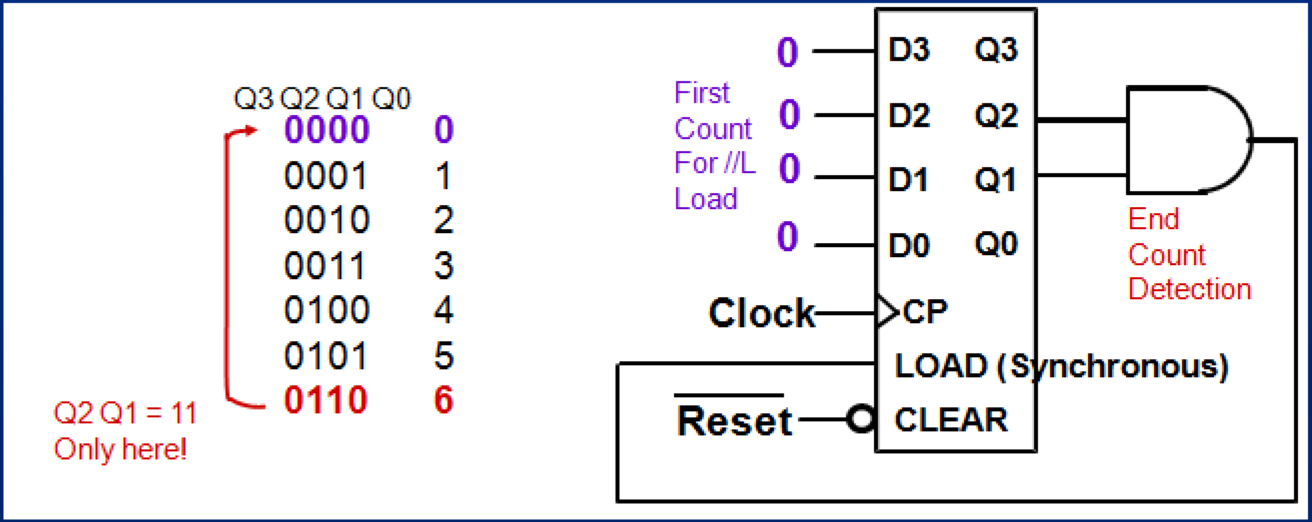
- Shift Registers to read data **serially** one bit at a time
- **Shift** is a synchronous control of shifting (register keeps data as is when **Shift** is not active)
- Digit serial registers that read data serially one digit at a time, where the digit size could be anything (e.g. 4-bits, 8-bits, 16-bits …etc.) → multiple of shift registers in parallel



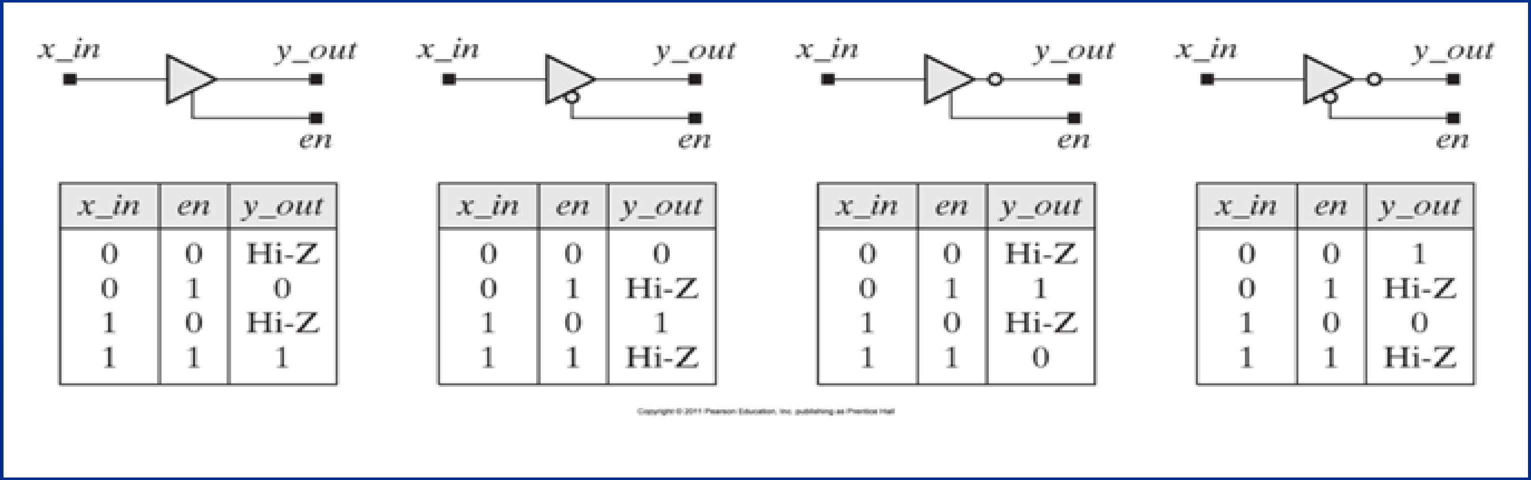


Modulo N (i.e. divide by N) Counters N counting states: 0, 1, 2, …, (N-1)

- **The following techniques use an n-bit ($2^n >= N$) binary counter with synchronous clear or parallel load:**
  - Detect terminal count (N – 1) and use to synchronously Clear the counter to 0 (first count) on next clock pulse
  - Detect terminal count (N – 1) to synchronously Load in the value 0 (first count) on next clock pulse
- **Modulo 7 (0,1,…,6):**



Three-State Devices



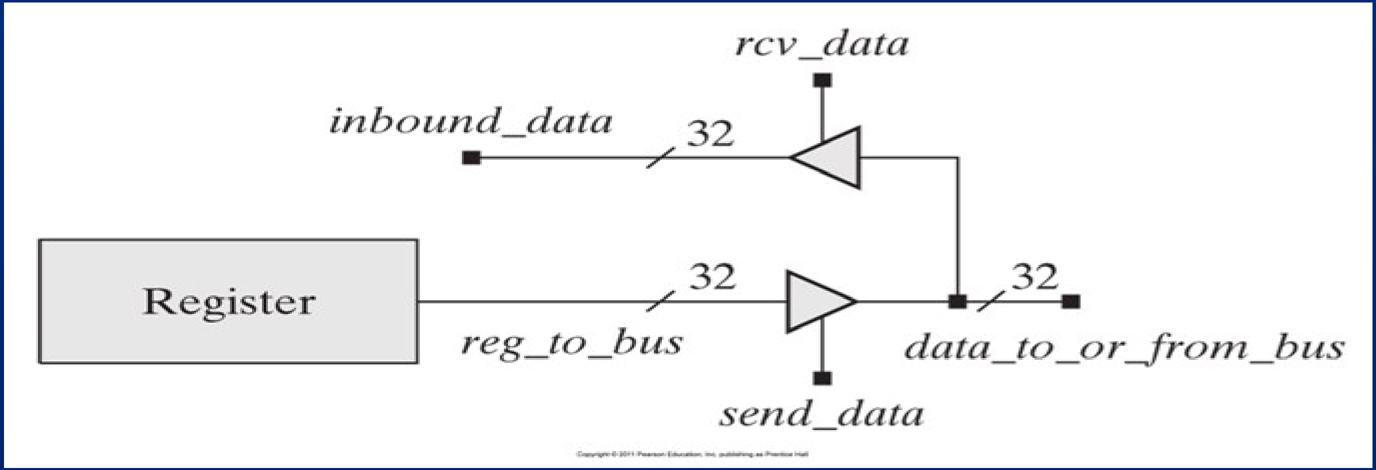
x_in	en	y_out
0	0	Hi-Z
0	1	0
1	0	Hi-Z
1	1	1

x_in	en	y_out
0	0	0
0	1	Hi-Z
1	0	1
1	1	Hi-Z

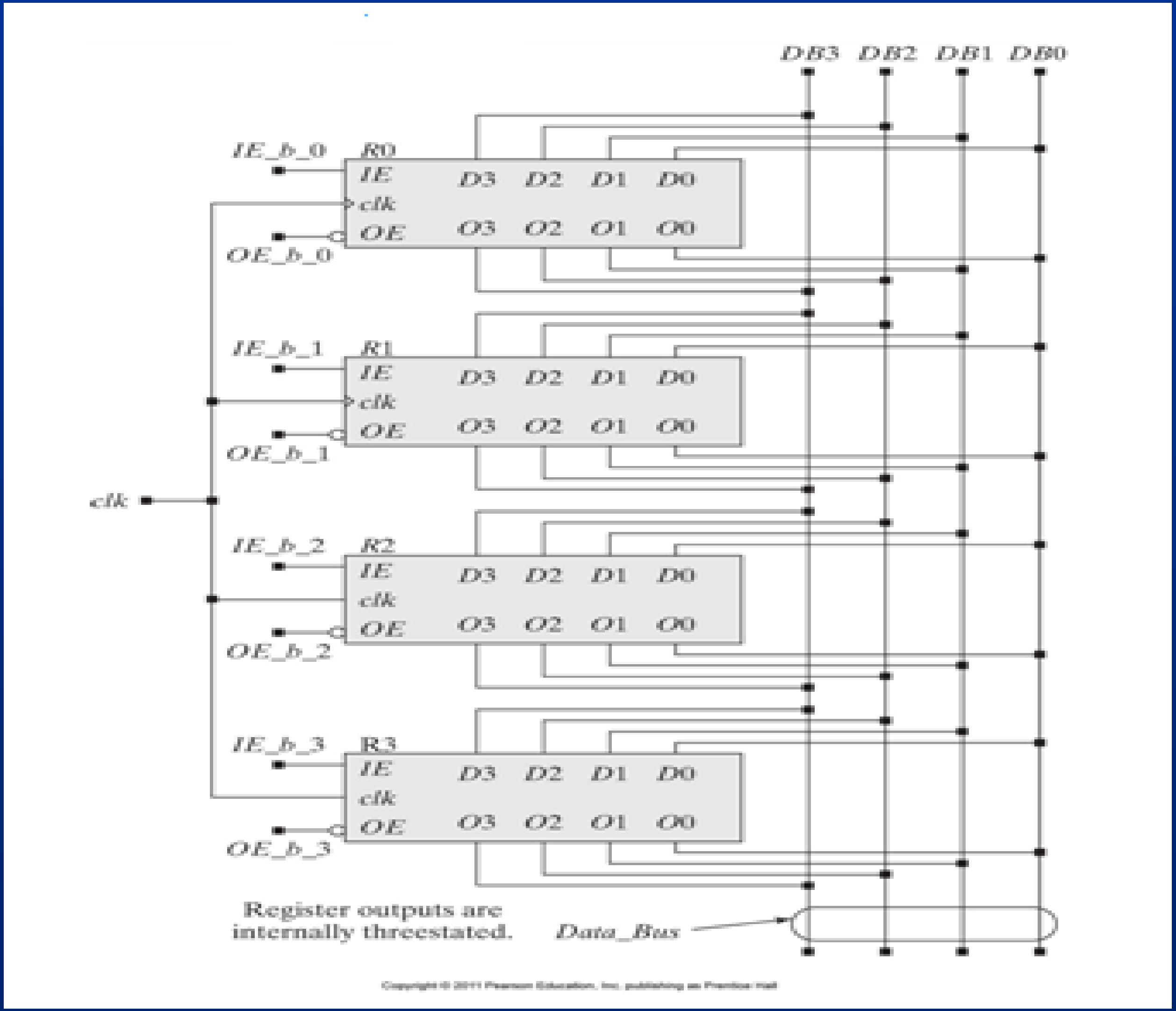
x_in	en	y_out
0	0	Hi-Z
0	1	1
1	0	Hi-Z
1	1	0

x_in	en	y_out
0	0	1
0	1	Hi-Z
1	0	0
1	1	Hi-Z

- **Bus isolation with three-state devices**



A Register Bank with a 4-bit Data Bus

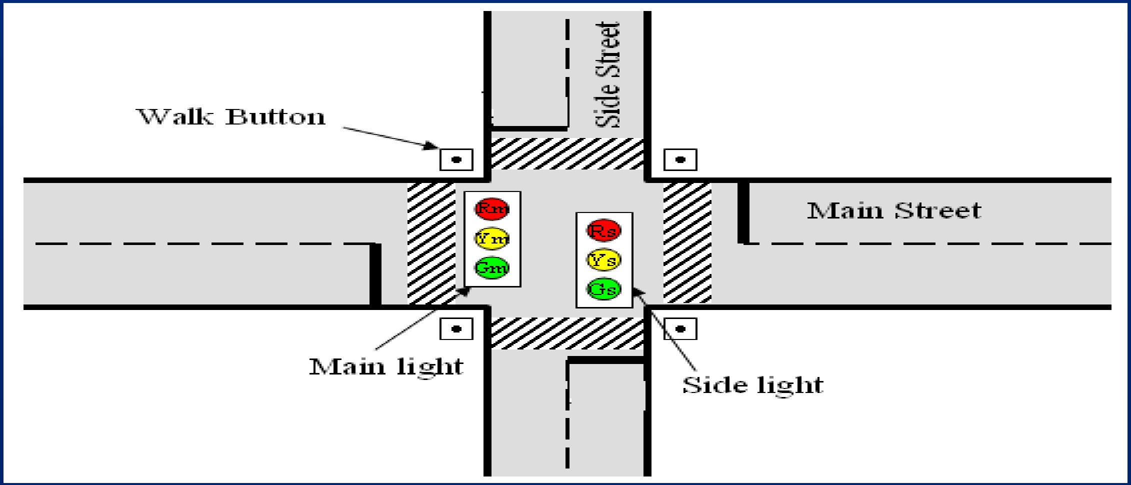


Design Steps

- Identify all inputs and outputs for the whole circuit. Identify, separately, data inputs/outputs and control inputs/outputs (external status).
- Identify the required Data Path blocks and their control signals and design them.
- Identify the input and output signals to the Data Path and Control unit.
- Design the control Unit (start by obtaining the state diagram, then the next state and output equations and finally the logic implementation) and connect it to the DP.

Example: Traffic Light Controller

- **Design a digital system that controls the traffic lights at an intersection:**
  - It receives inputs from all four corners indicating pedestrians that want to cross
  - In absence of crossing requests it should allow each direction 30 seconds of green light, followed by 5 seconds of yellow light while the other traffic light will be red light (i.e. for 35 seconds)
  - In presence of crossing requests at or after 15 seconds, immediately proceed with yellow
  - It is assumed that the clock frequency of the system is 1KHz



Example: Traffic Light Controller

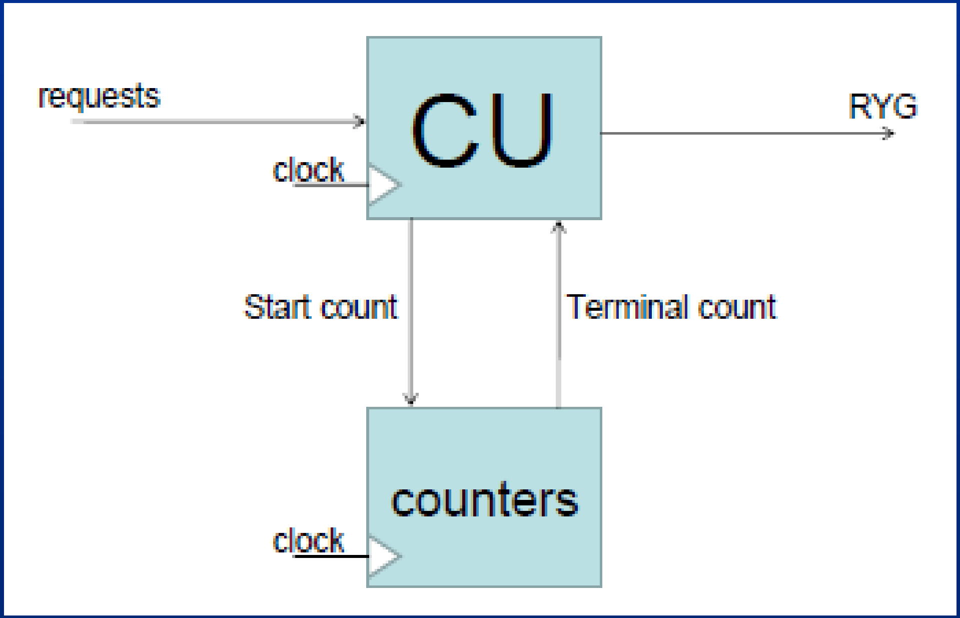
- **Is it possible to design the system as a single combinational circuit or an FSM?**
- **Because of the time delay, a combinational circuit will not work (we need to make sure that we waited 30 seconds, and then 5 seconds etc.)**
- **However, a sequential circuit is perfectly capable to deal with the problem.**
- **So why bother designing a more complex circuit?**
  - Let’s look better at the specification

Example: Traffic Light Controller

- **If we were to design the system as a single FSM, we will have to include in it all the possible conditions.**
- **In particular, we will have to deal with the time delays by introducing delay states**
  - How many? With a 1KHz clock, we need 1000 clock cycles to measure one second, 30,000 to measure 30 seconds.
  - So, we will need more than 30,000 states!
  - The FSM design will be boring to say the least….
- **How can we handle the state explosion problem?**
- **We could use counters to measure the time intervals, and a control unit to coordinate their behavior.**

Example: Traffic Light Controller

- **DP/CU for the TLC:**

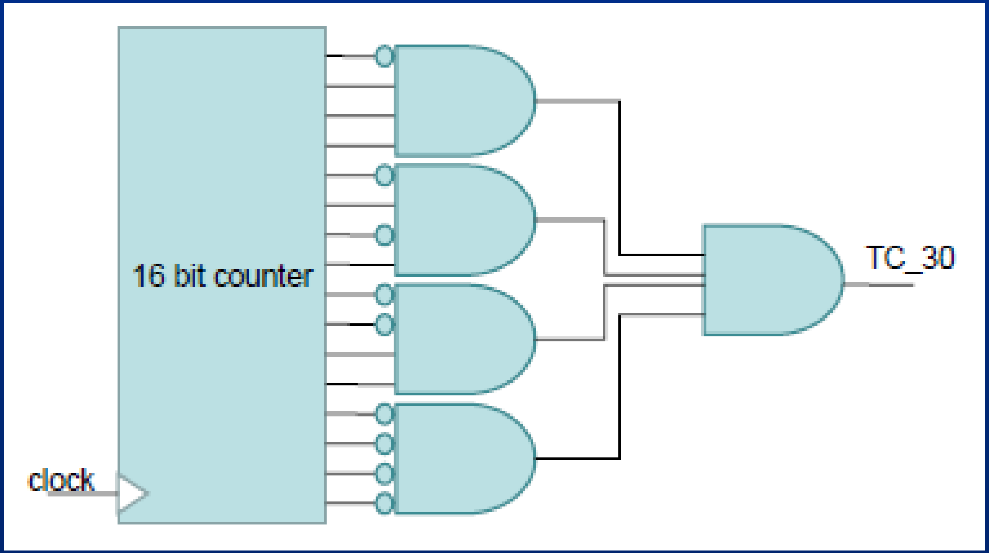


Example: Traffic Light Controller

- **DP design: we need to be able to count:**
  - 30 seconds (30,000 clock cycles) for the “regular green”
  - 15 seconds (15,000 clock cycles) for the “reduced green”
  - 5 seconds (5,000 clock cycles) for the “yellow light”
- **Typical counters have:**
  - A reset/clear input (asynchronous): that sets the count to “’0”
  - A count enable input: if 1 it allows the counter to increment
  - A multiple bit count output: reports the current state of the count
  - A single bit terminal count: indicates that a total number of clock cycles are elapsed and the counter is back to 0

Example: Traffic Light Controller

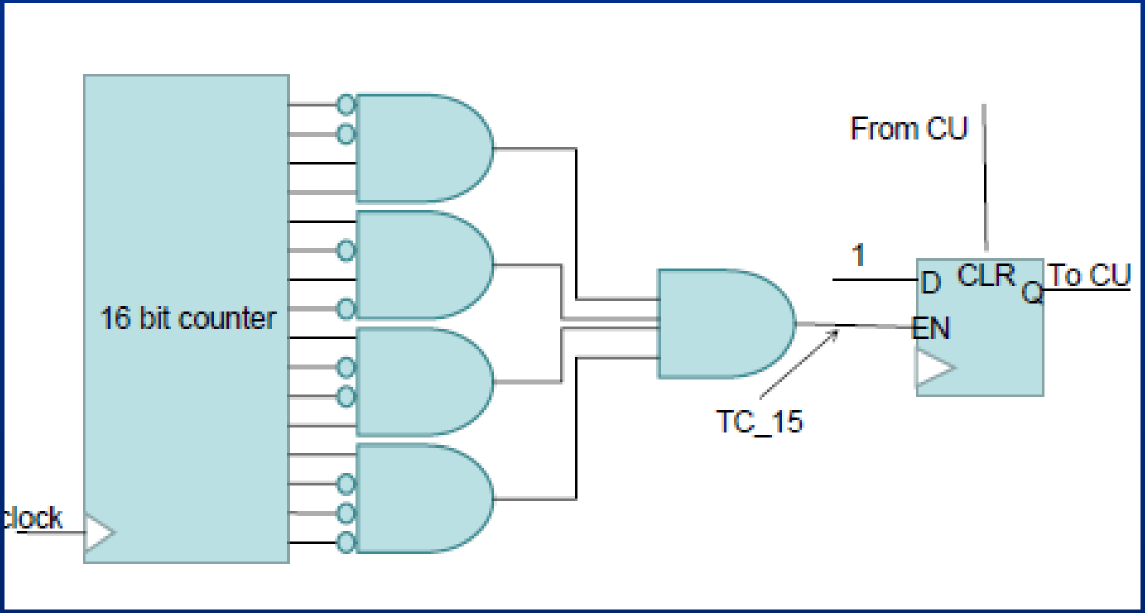
- In order to be able to count 30,000, we need a “power of 2” of at least 32,768 (=2^{15}).
- 30000=$(111\ 0101\ 0011\ 0000)_2$. If the counter output bits are “111010100110000” the terminal count is 1



Example: Traffic Light Controller

- **Requests reduce the time to 15 seconds, but if 15 seconds have already elapsed you should know it**

$150000 = (11\ 1010\ 1001\ 1000)_2$



Example: Traffic Light Controller

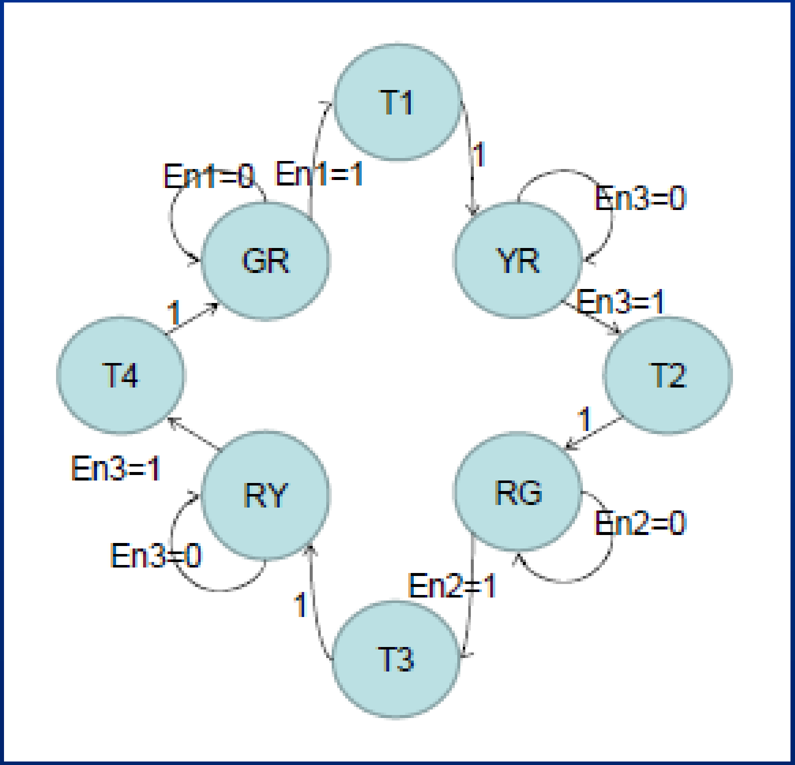
- **We can have three separate blocks in the DP (for the 30 sec, >=15 sec and 5 sec):**
  - DP composed of 3 16-bit counters, and a bunch of AND and NOT gates
- **However, we can do better:**
  - If you look at the specification, the intervals of 5, 15 or 30 seconds are either used in separate times (5 is only used during the “yellow” phase) or they can use the same starting point (the 30 and 15 – they actually have to use the same starting point)
  - Use just one counter, with three circuits for 5, 30 and >=15 sec.

Example: Traffic Light Controller

- **Now, we have a complete DP. Its interface with the CU:**
  - One counter reset input (asynchronous reset)
  - One FF reset input (asynchronous reset)
  - Three outputs (TC_30, TC_5, GE_15)
  - In this case, DP does not use PI nor produces PO (but it is just a special case)
- **CU design**
  - Once the DP is finished, the CU design can proceed, as in a usual FSM design
  - In this case, the CU has to:
    - Provide regular switching between green, yellow and red lights
    - Observe the request inputs and use them in combination with GE_15 to shorten green lights

Example: Traffic Light Controller

- **Possible Moore machine implementation**
  - EN1=TC_30 + GE_15 . cross1
  - EN2=TC_30 + GE_15 . cross2
  - EN3=TC_5
- **CU outputs (if not shown =0):**
  - State GR: G1=1, R2=1
  - State T1: Y1=1, R2=1
    - CNT_RES=1
  - State YR: Y1=1, R2=1
  - State T2: R1=1, G2=1
    - CNT_RES=1, FF_RES=1
  - State RG: R1=1, G2=1
  - State T3: R1=1, Y2=1
    - CNT_RES=1
  - State RY: R1=1, Y2=1
  - State T4: G1=1, R2=1
    - CNT_RES=1, FF_RES=1



Example: Traffic Light Controller

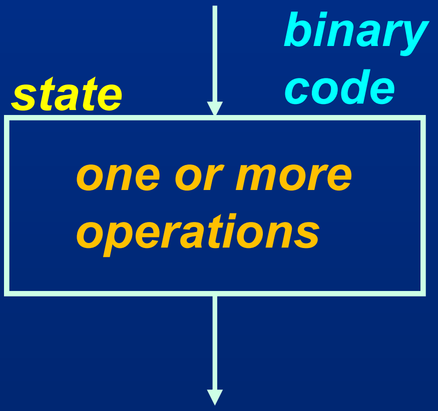
- **One more issue to verify:**
  - Are we actually keeping the lights 5, 15 or 30 seconds?
  - The counters act exactly, but the entire system has a certain delay that we need to take into account:
    - The counters start counting only the clock cycle after the reset (in this case, when the CU enters the states RG, GR YR and RY)
    - When they are done counting, the state changes the following clock tick, and the outputs with them - MOORE machine
  - In conclusion, the lights stay on for 1 more clock cycle: we need to take that into account by reducing the terminal count to 29,999, 4,999 and 14,999…

Algorithmic State Machine (ASM) Chart

- **Algorithmic State Machine (ASM) Chart is a high-level flowchart-like notation to specify the hardware algorithms in digital systems.**
- **Major differences from flowcharts are:**
  - uses 3 types of boxes: state box (similar to operation box), decision box and conditional box
  - contains exact (or precise) timing information; flowcharts impose a relative timing order for the operations.
- **From the ASM chart it is possible to obtain**
  - the control
  - the architecture (data processor)

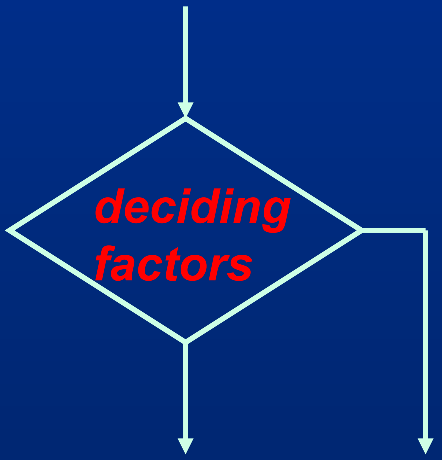
Components of ASM Charts

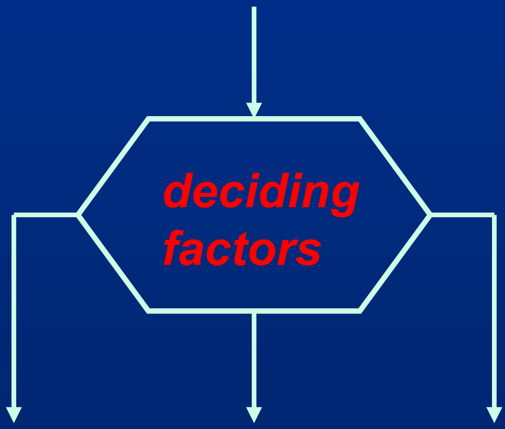
- The **state box** is rectangular in shape. It has at most one entry point and one exit point and is used to specify one or more operations which could be simultaneously completed in one clock cycle.



Components of ASM Charts

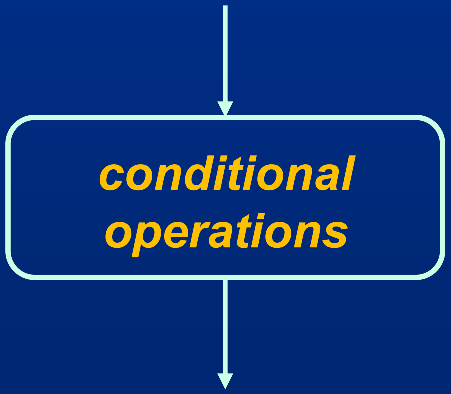
- The **decision box** is diamond in shape. It has one entry point but multiple exit points and is used to specify a number of alternative paths that can be followed.





Components of ASM Charts

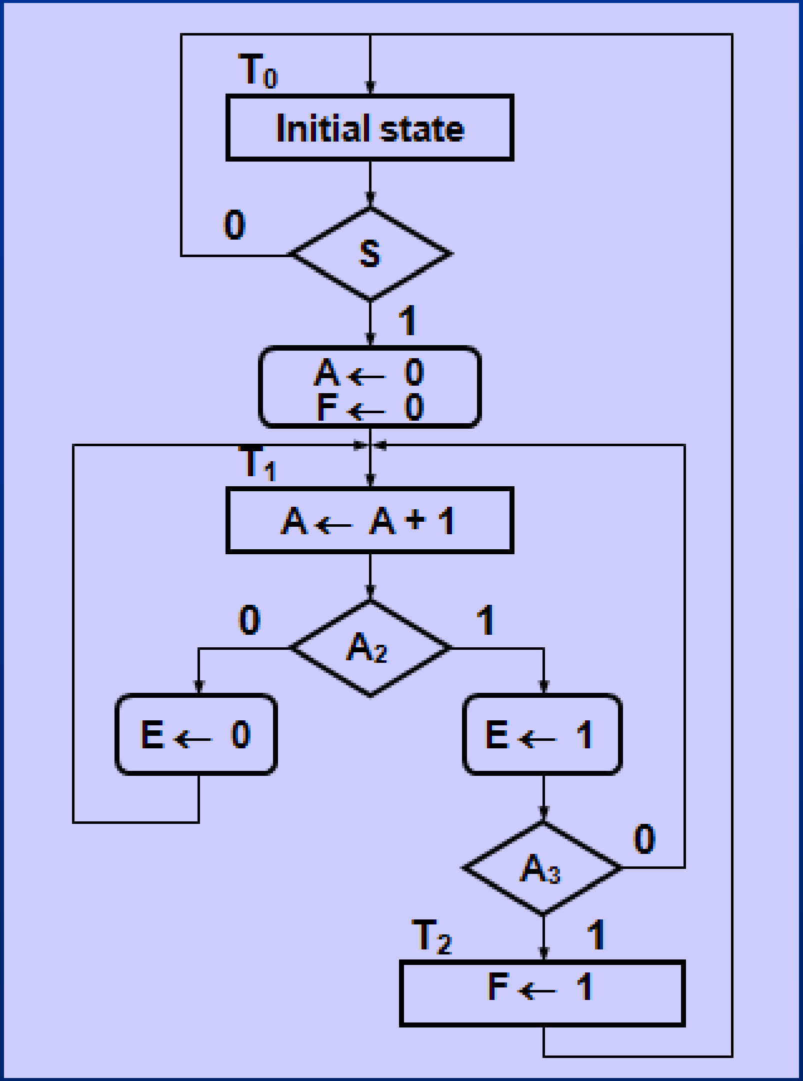
- The conditional box is represented by a rectangle with rounded corners. It always follows a decision box and contains one or more *conditional operations* that are only invoked when the path containing the conditional box is selected by the decision box.



ASM Charts: An Example

- A is a register;
- A_i stands for ith bit of the A register.

  $A = A_4A_3A_2A_1$
- E and F are single-bit flip-flops.



Register Operations

- **Registers are present in the data processor for storing and processing data. Flip-flops (1-bit registers) and memories (set of registers) are also considered as registers.**
- **The register operations are specified in either the state and/or conditional boxes, and are written in the form:**

destination register $\leftarrow$ *function*(*other registers*)

where the LHS contains a destination register (or part of one) and the RHS is some function over one or more of the available registers.

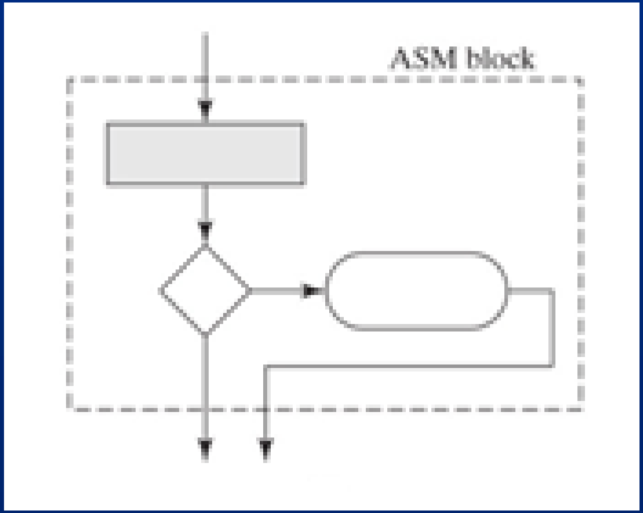
Register Operations

- **Examples of register operations:**

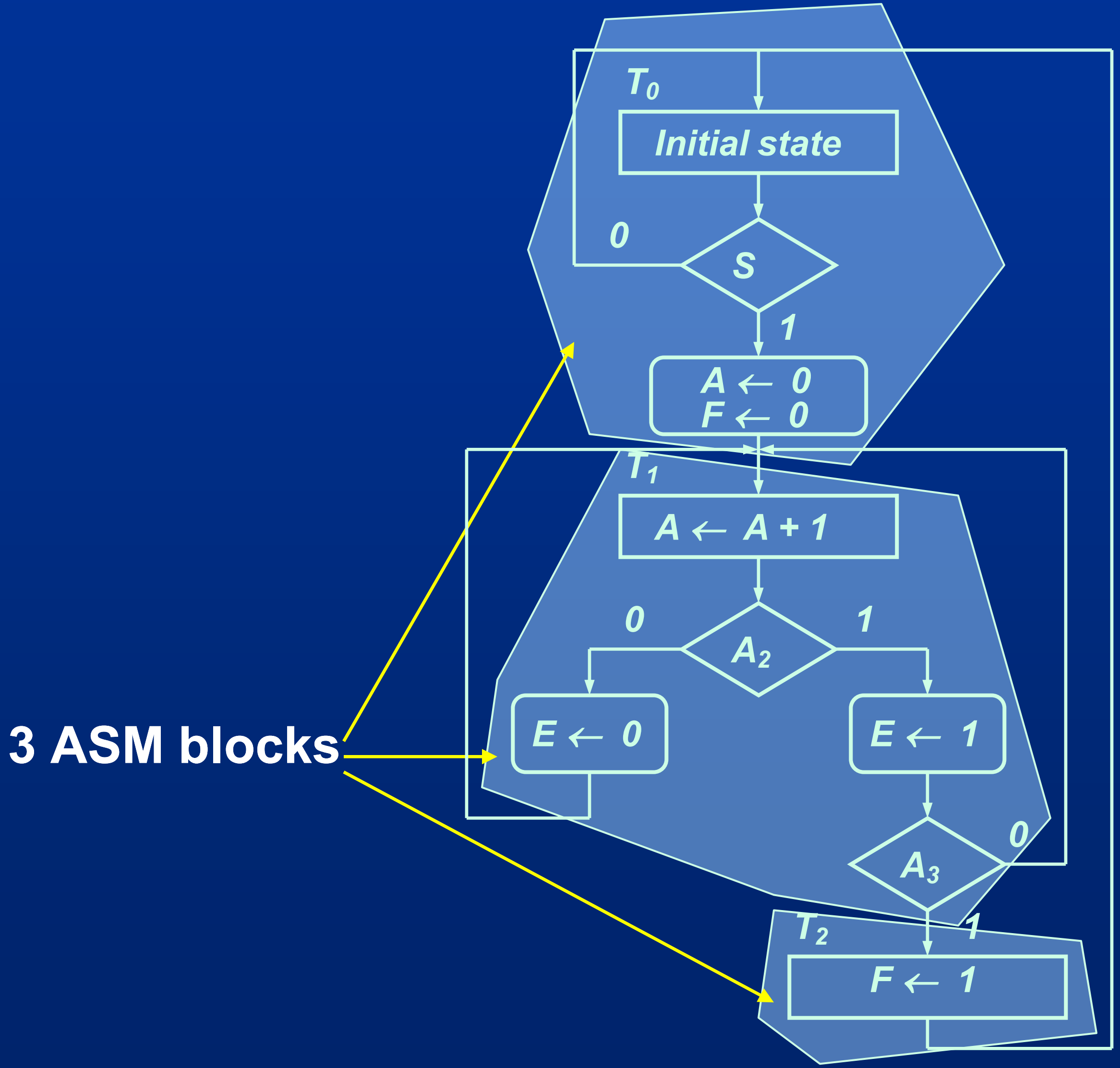
$A \leftarrow B$	Transfer contents of register B into register A.
$A \leftarrow 0$	Clear register A.
$A \leftarrow A - 1$	Decrement register A by 1.

Timing in ASM Charts

- **Precise timing is implicitly present in ASM charts.**
- **Each *state box*, together with its immediately following *decision* and *conditional boxes*, occur within one clock cycle.**
- **A group of boxes which occur within a single clock cycle is called an ASM block.**



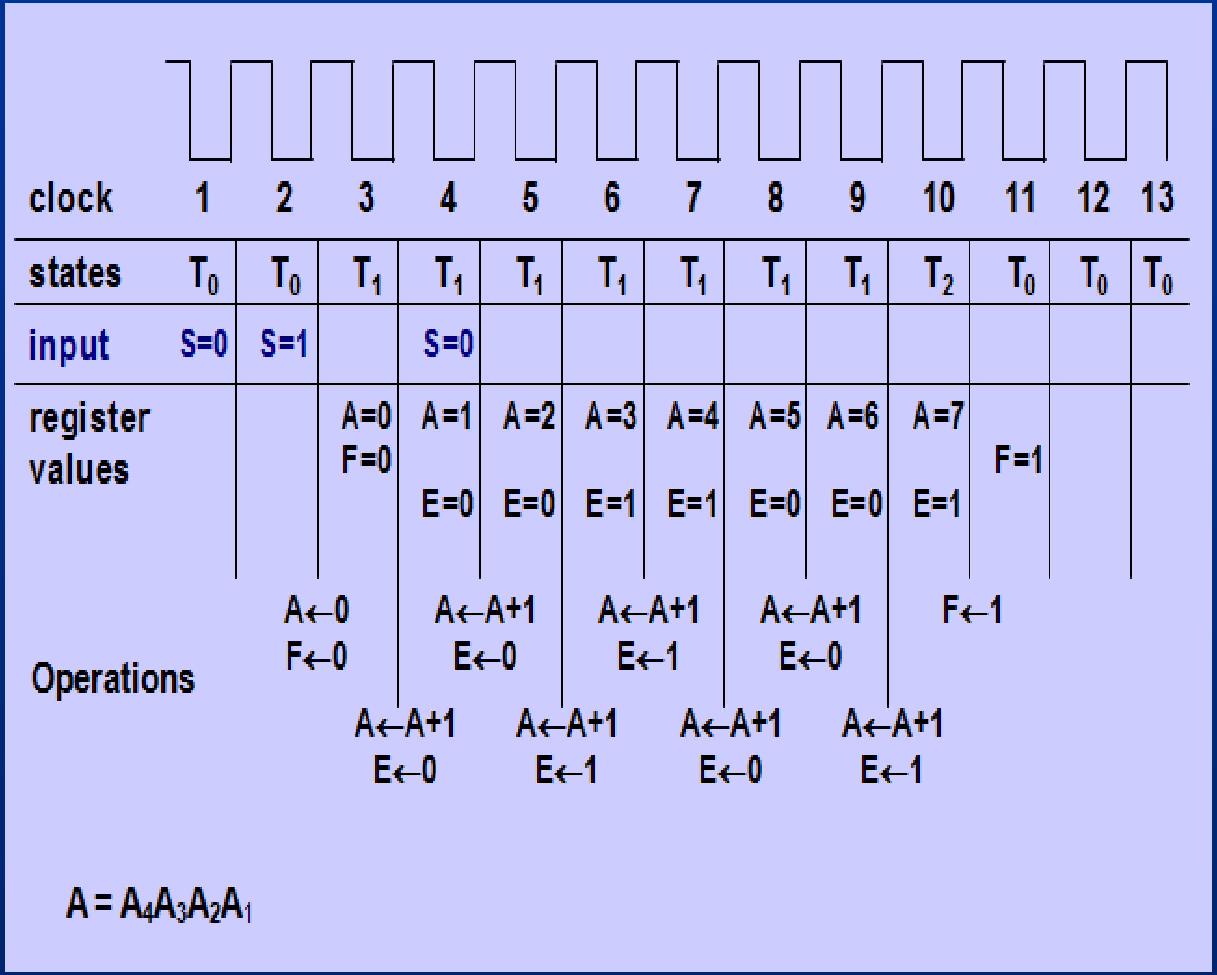
Timing in ASM Charts



Timing in ASM Charts

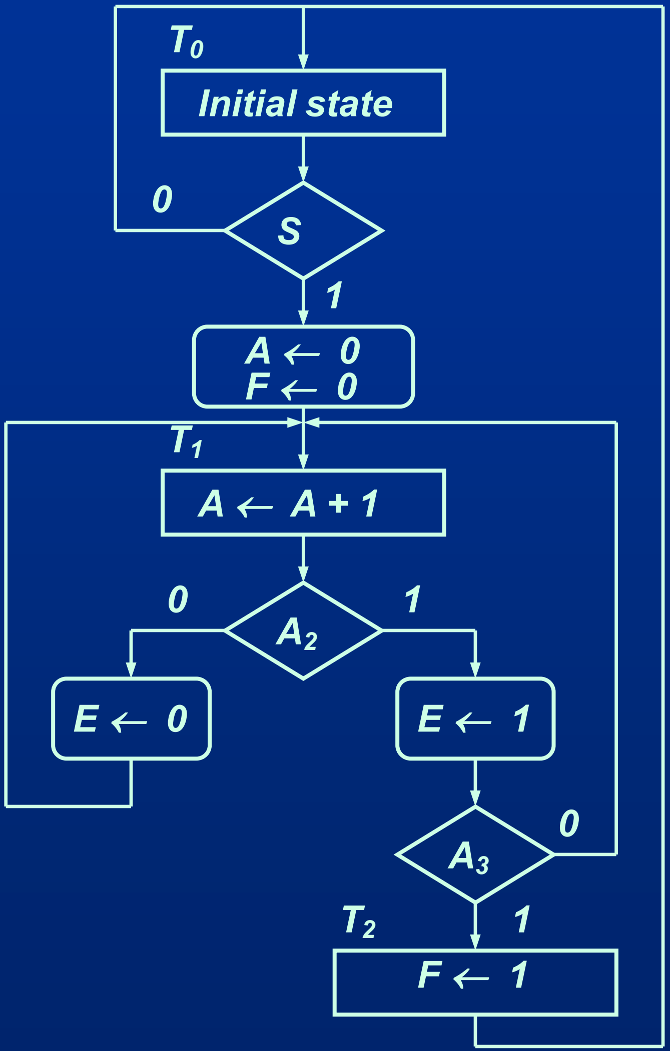
- **Operations of ASM can be illustrated through a timing diagram.**
- **Two factors which must be considered are**
  - operations in an ASM block occur at the same time in *one clock cycle*
  - decision boxes are dependent on the status of the *previous clock cycle* (that is, they do not depend on operations of current block)

Timing in ASM Charts



clock	1	2	3	4	5	6	7	8	9	10	11	12	13
states	T_0	T_0	T_1	T_1	T_1	T_1	T_1	T_1	T_1	T_2	T_0	T_0	T_0
input	S=0	S=1		S=0									
register values			A=0 F=0	A=1 E=0	A=2 E=0	A=3 E=1	A=4 E=1	A=5 E=0	A=6 E=0	A=7 E=1	F=1		
Operations		A←0 F←0	A←A+1 E←0	A←A+1 E←0	A←A+1 E←1	A←A+1 E←1	A←A+1 E←0	A←A+1 E←0	A←A+1 E←1	F←1			

$A = A_4A_3A_2A_1$



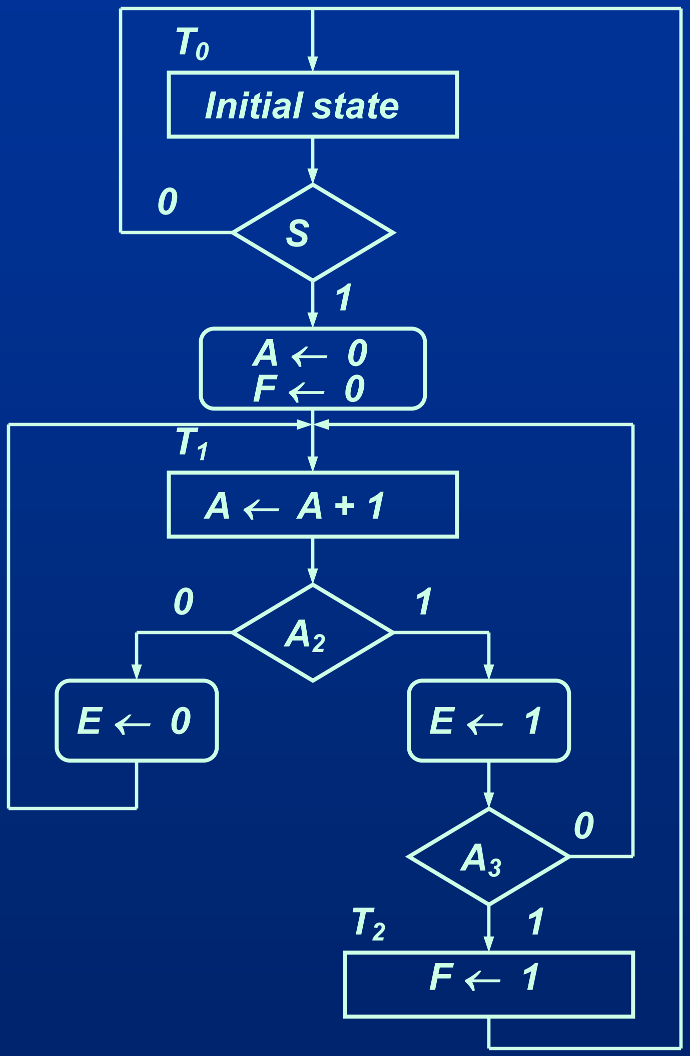
ASM Chart => Digital System

- **ASM chart describes a digital system. From ASM chart, we may obtain:**
  - Controller logic (via State Table/Diagram)
  - Architecture/Data Processor
- **Design of controller is determined from the decision boxes and the required state transitions.**
- **Design requirements of data processor can be obtained from the operations specified with the state and conditional boxes.**

ASM Chart => Controller

- **Procedure:**
  - Step 1: Identify all states and assign suitable codes.
  - Step 2: Formulate state table using
    State from state boxes
    Inputs from decision boxes
    Outputs from operations of state/conditional boxes.
  - Step 3: Obtain state/output equations and draw circuit.

ASM Chart => Controller



Assign codes to states:

T_0 = 00
T_1 = 01
T_2 = 11

Present state		inputs			Next state		outputs		
G_1	G_0	S	A_2	A_3	G_1^+	G_0^+	T_0	T_1	T_2
0	0	0	X	X	0	0	1	0	0
0	0	1	X	X	0	1	1	0	0
0	1	X	0	X	0	1	0	1	0
0	1	X	1	0	0	1	0	1	0
0	1	X	1	1	1	1	0	1	0
1	1	X	X	X	0	0	0	0	1

Inputs from conditions in decision boxes.
Outputs = present state of controller.

ASM Chart => Architecture/Data Processor

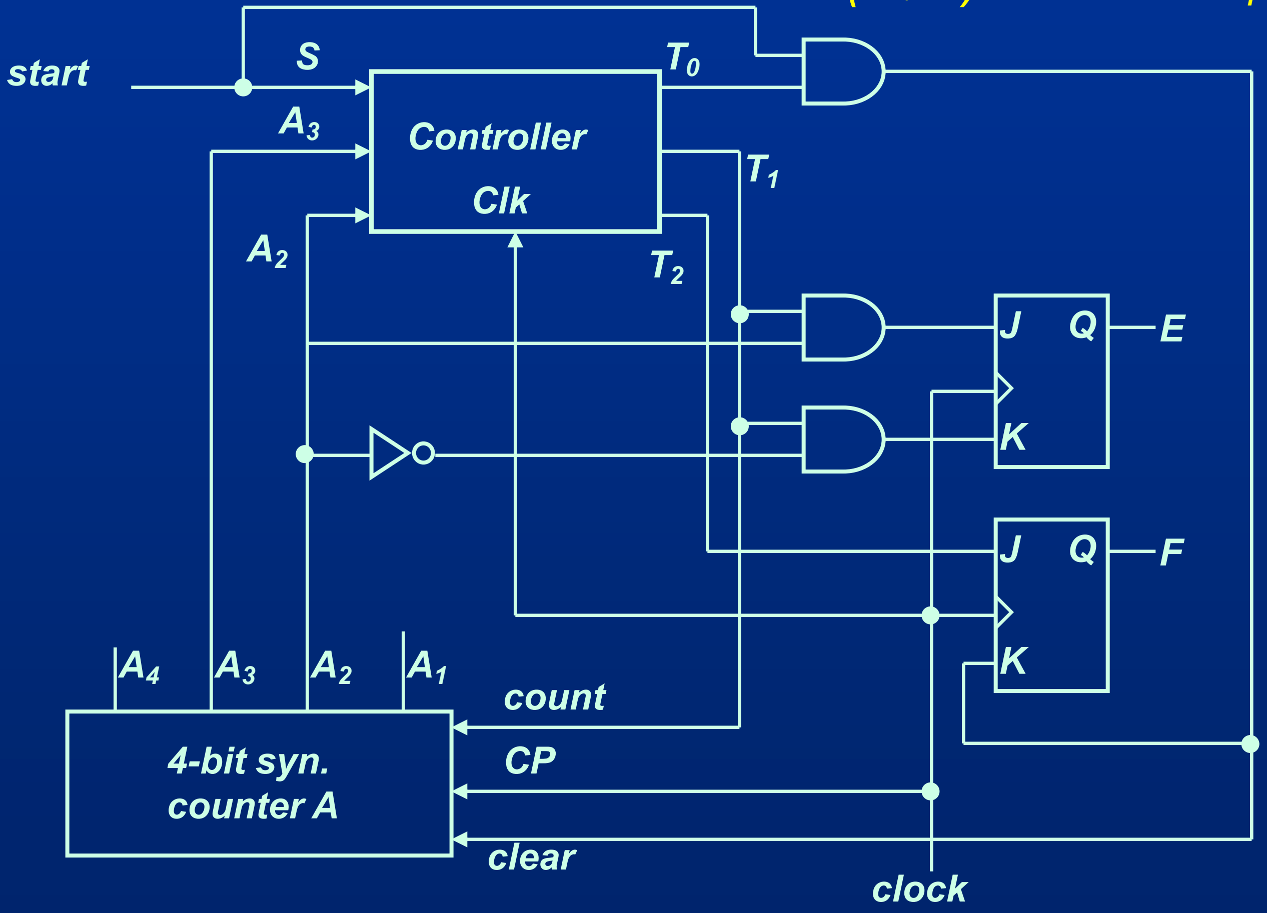
- **Architecture is more difficult to design than controller.**
- **Nevertheless, it can be deduced from the ASM chart. In particular, the operations from the ASM chart determine:**
  - What registers to use
  - How they can be connected
  - What operations to support
  - How these operations are activated.
- **Guidelines:**
  - always use high-level units
  - simplest architecture possible.

ASM Chart => Architecture/Data Processor

- **Various operations are:**
  - Counter incremented ($A \leftarrow A + 1$) when state = T_1.
  - Counter cleared ($A \leftarrow 0$) when state = T_0 and S = 1.
  - E is set ($E \leftarrow 1$) when state = T_1 and $A_2 = 1$.
  - E is cleared ($E \leftarrow 0$) when state = T_1 and $A_2 = 0$.
  - F is set ($F \leftarrow 0$) when state = T_2.
  - F is cleared ($F \leftarrow 0$) when state = T_0 and S = 1.
- **Deduce:**
  - One 4-bit register A (e.g.: 4-bit synchronous counter with clear/increment).
  - Two flip-flops needed for E and F (e.g.: JK flip-flops).

ASM Chart => Architecture/Data Processor

($A \leftarrow A + 1$) when state = T_1.
($A \leftarrow 0$) when state = T_0 and $S = 1$.
($E \leftarrow 1$) when state = T_1 and $A_2 = 1$.

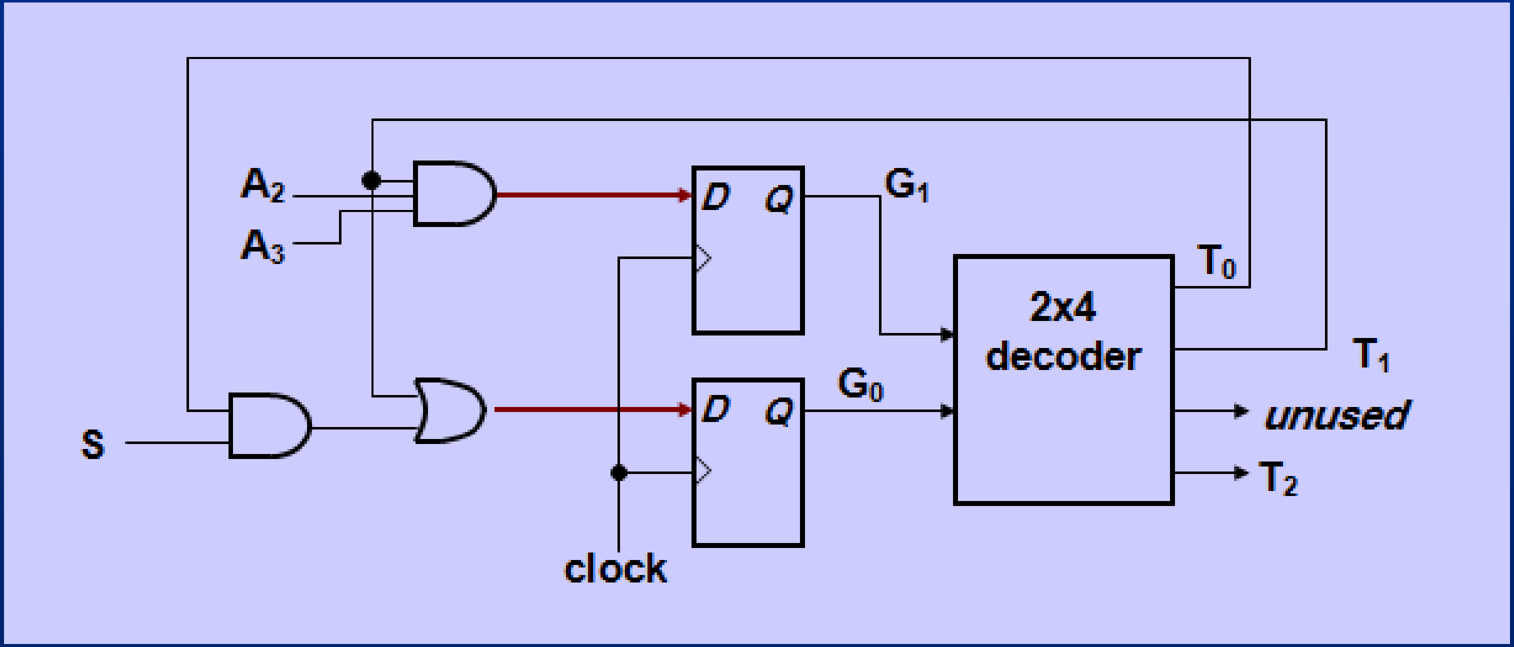


Implementing Controller: Decoder + D Flip-flops

- **Flip-flop input functions:**

$$DG_1 = T_1.A_2.A_3$$
$$DG_0 = T_0.S + T_1$$

- **Circuit:**

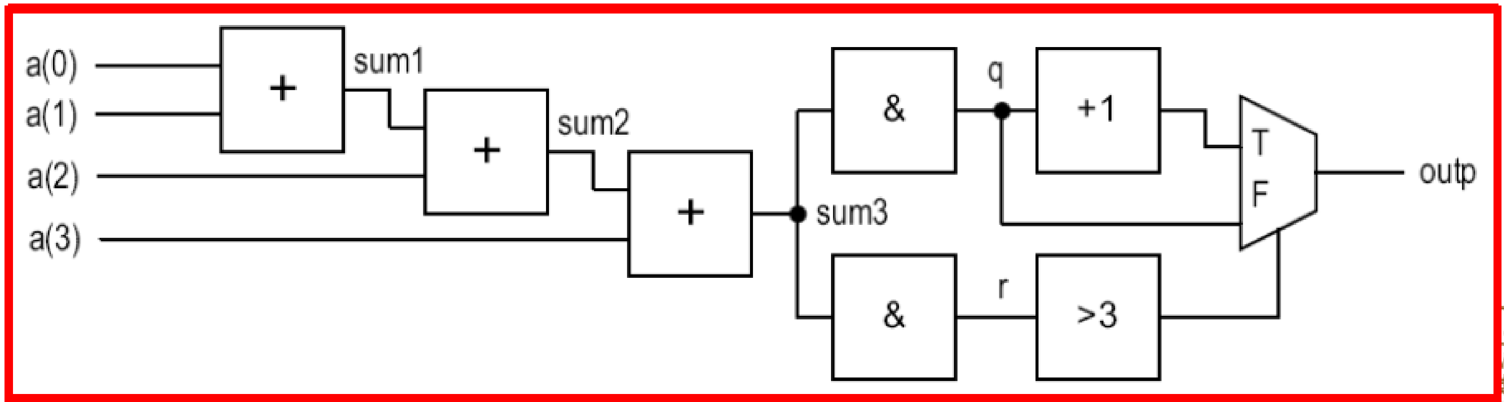


Data Flow Design

- "Dataflow" implementation in VHDL
  - Convert the algorithm in to **combinational circuit**

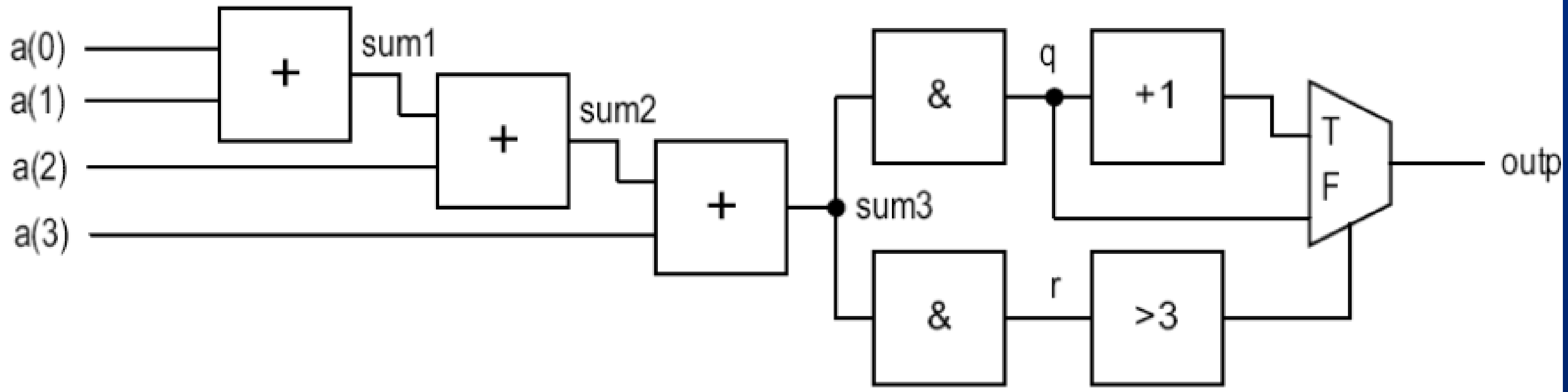
```
sum  <= 0;
sum0  <= a(0);
sum1  <= sum0 + a(1);
sum2  <= sum1 + a(2);
sum3  <= sum2 + a(3);
q  <= "000" & sum3(8 downto 3);
r  <= "00000" & sum3(2 downto 0);
outp  <= q + 1 when (r > 3) else q;
```

The "sequential" operations are represented by the data flow from left to right



Data Flow Design- Drawbacks

- **Problems with dataflow implementation:**
  - Can only be applied to simple trivial algorithm
  - **Not flexible**
    - ***What if size=10, 100, 1000 …***
    - ***or size = n, i.e., size is determined by an external input***
    - ***or changing operation depending on instructions***

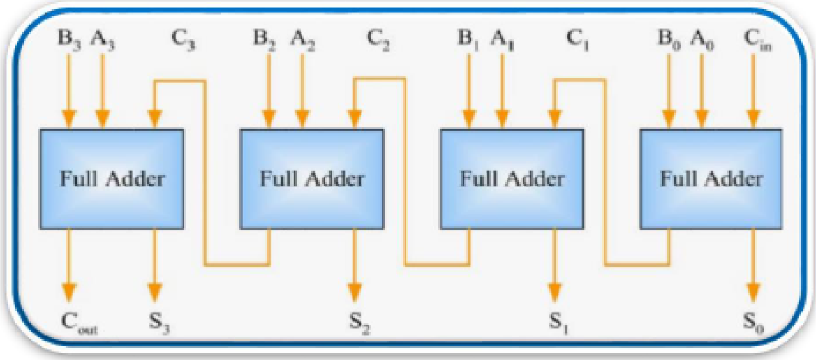


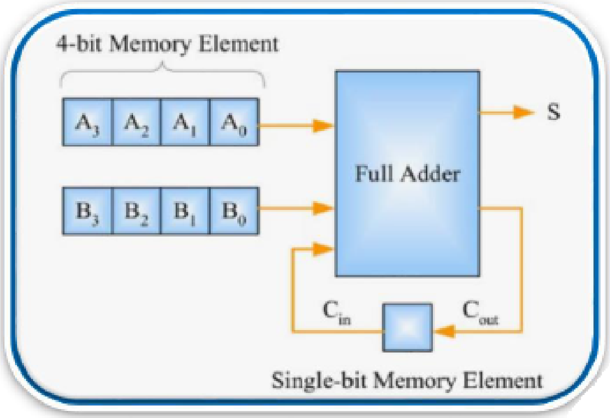
Data Flow Design- Alternative?

- **Hardware resembles the *variable* and *sequential* execution model**
  - Use **register** to store ***intermediate data*** and imitate ***variable***

    e.g. sum=sum+a => sum_reg+a_reg->sum_reg
  - Basic format of **RT operation**

    $$r_{dest} \leftarrow f\,(r_{src1}, \ldots, r_{srcn})$$
  - Sequence of data manipulation and transfer among registers (**RTL**)

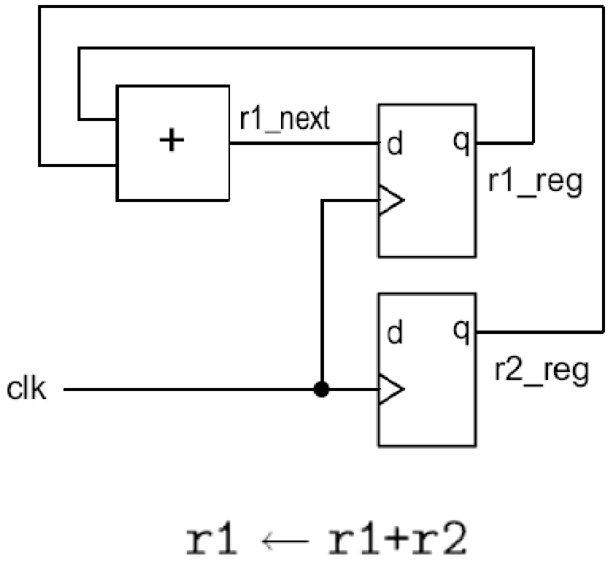


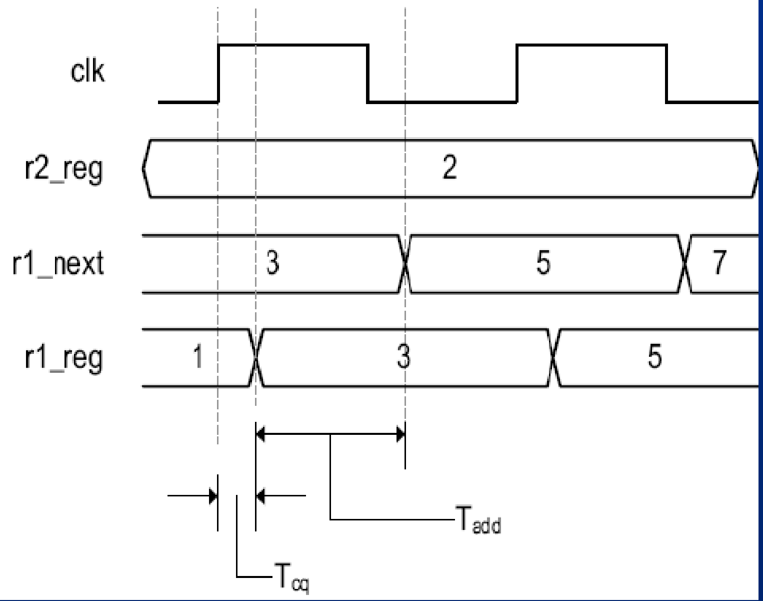


RT- Hardware mapping (example 1)

E.g. r1← r1+r2

- C1: r1_next<=r1_reg+r2_reg
- C2: r1_reg<=r1_next



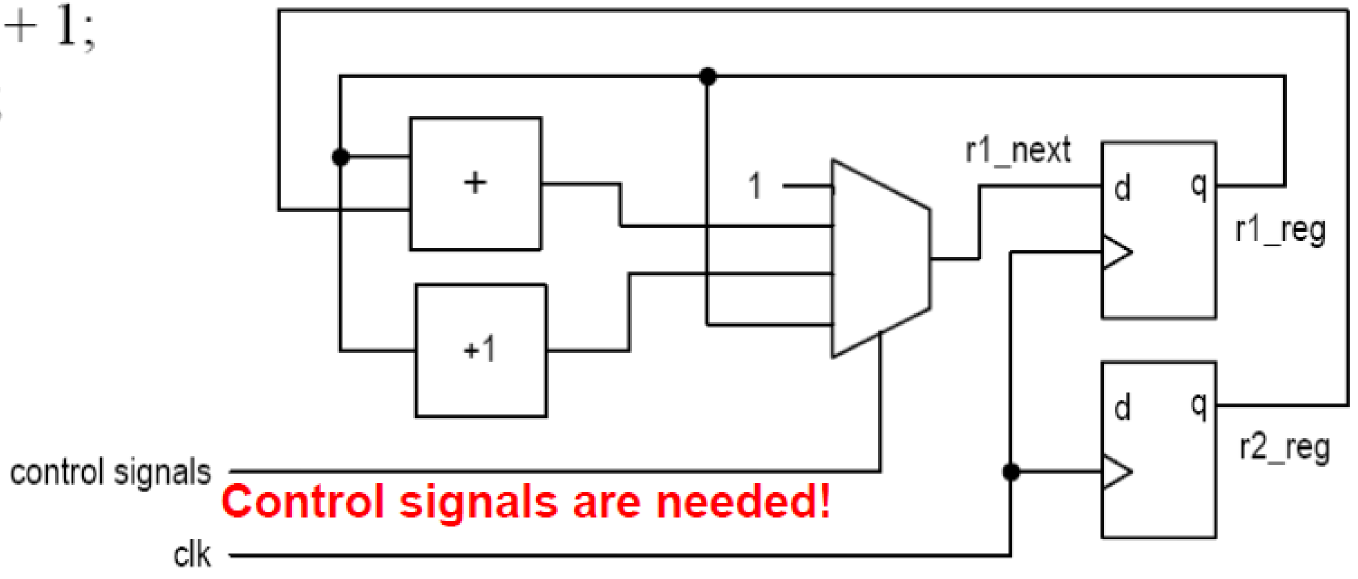


RT- Hardware mapping (example 2)

- Multiple RT operations

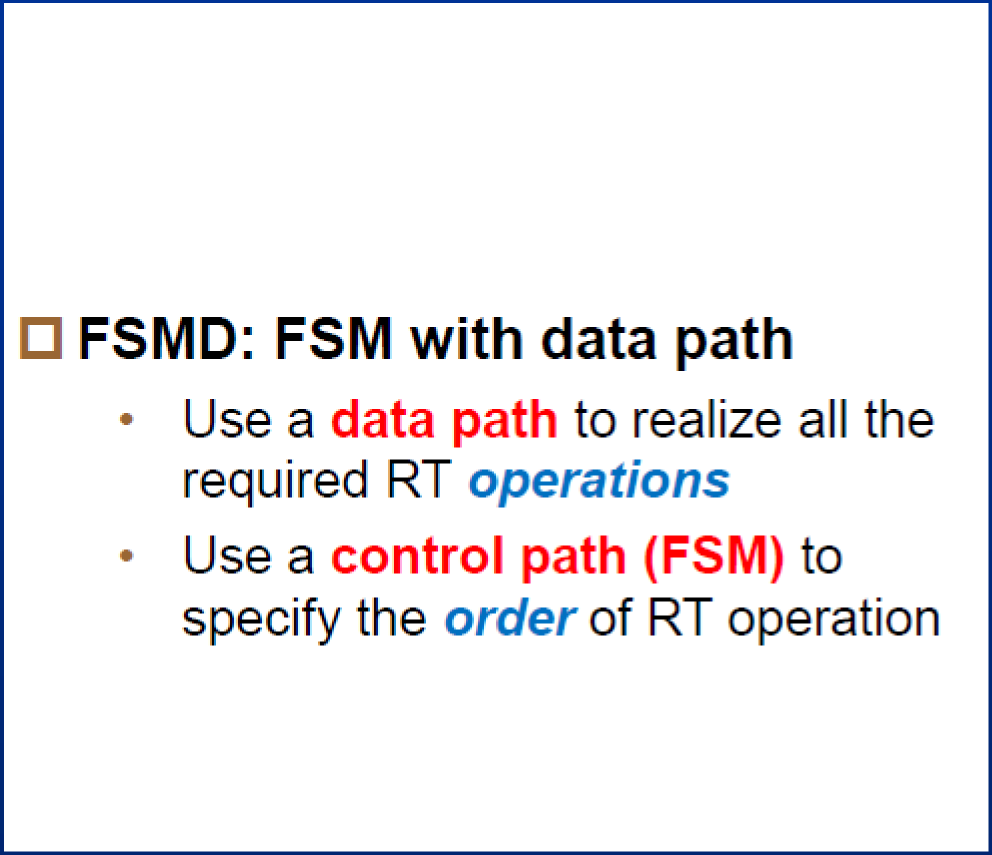
$r1 \leftarrow 1;$
$r1 \leftarrow r1 + r2;$
$r1 \leftarrow r1 + 1;$
$r1 \leftarrow r1;$

How can we organize multiple operations on one register (in a time-multiplexing way)?



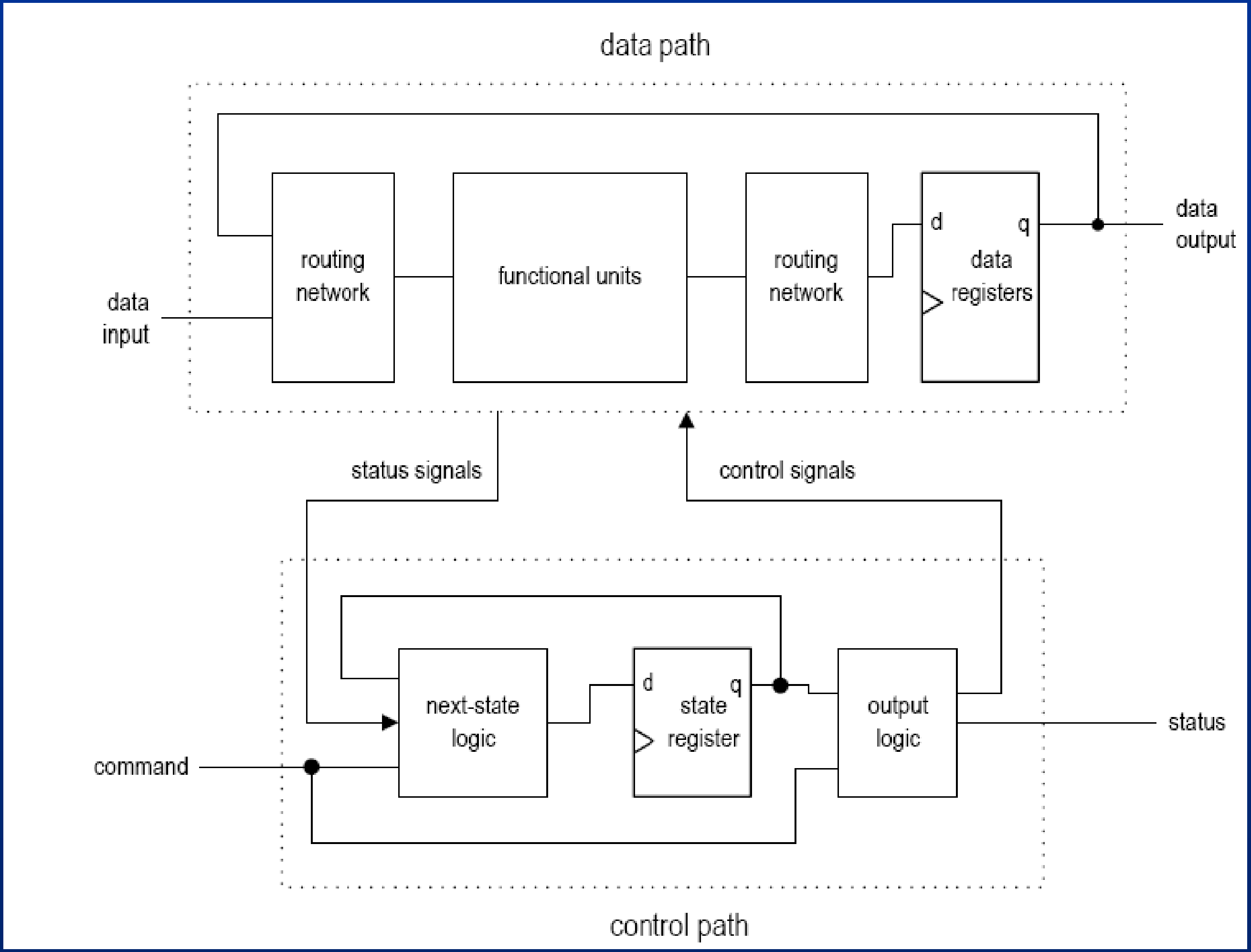
Control signals are needed!

FSMD- you saw previously!

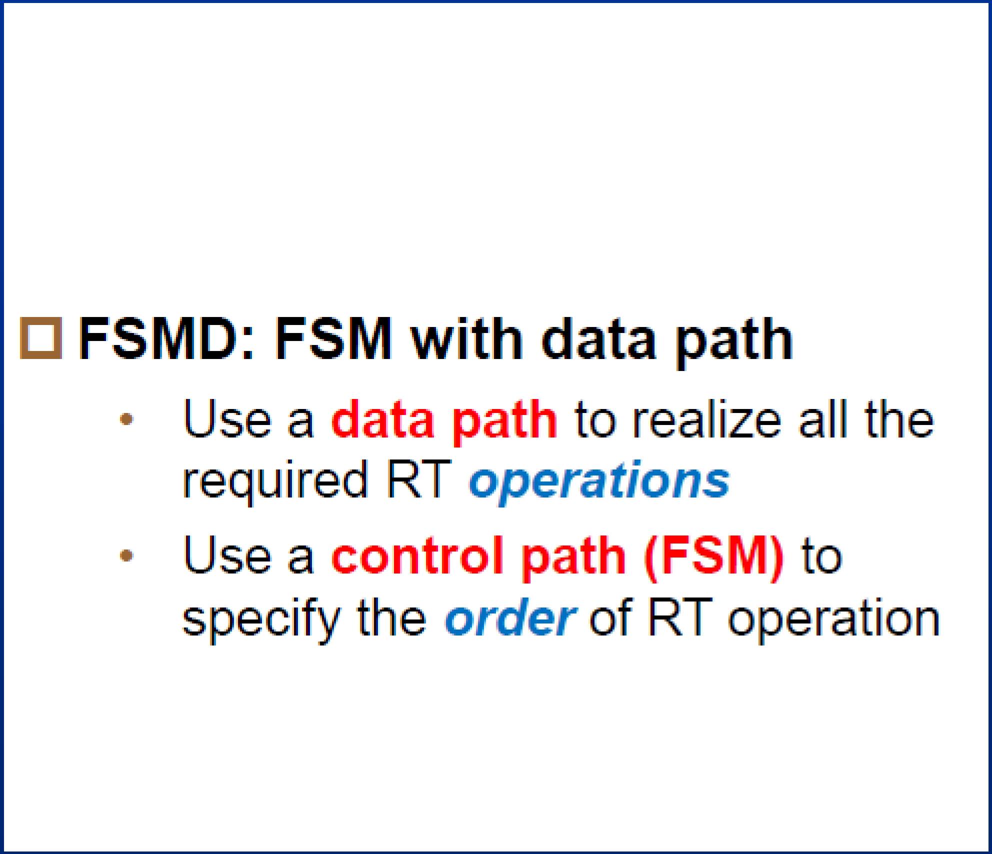


- **FSMD: FSM with data path**
  - Use a **data path** to realize all the required RT ***operations***
  - Use a **control path (FSM)** to specify the ***order*** of RT operation

FSMD- you saw previously!



FSMD- you saw previously!

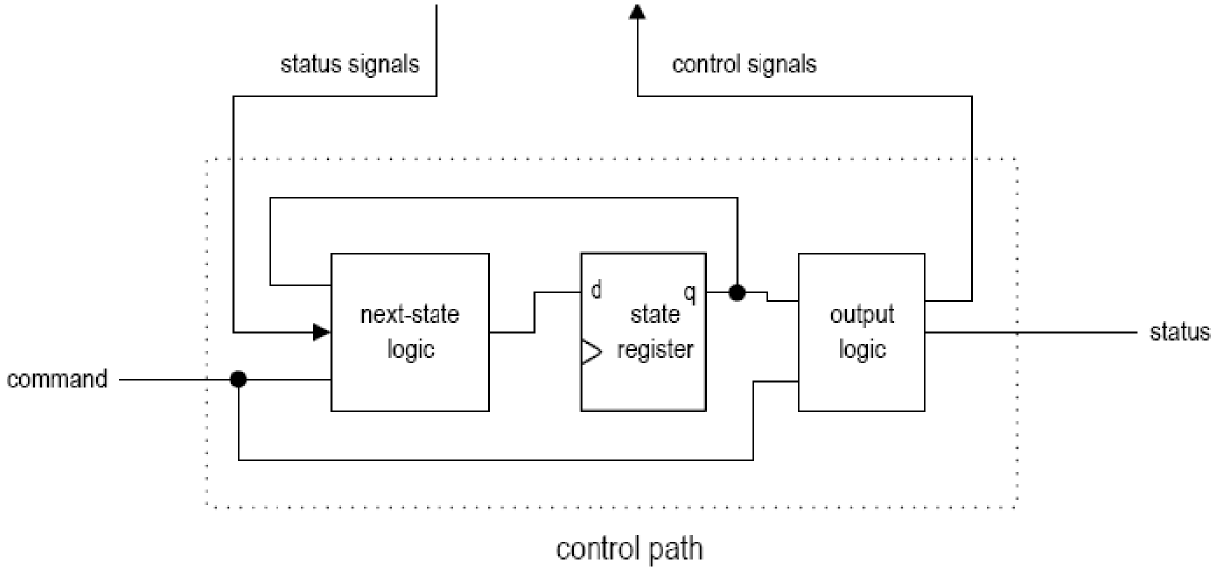


- **FSMD: FSM with data path**
  - Use a **data path** to realize all the required RT ***operations***
  - Use a **control path (FSM)** to specify the *order* of RT operation

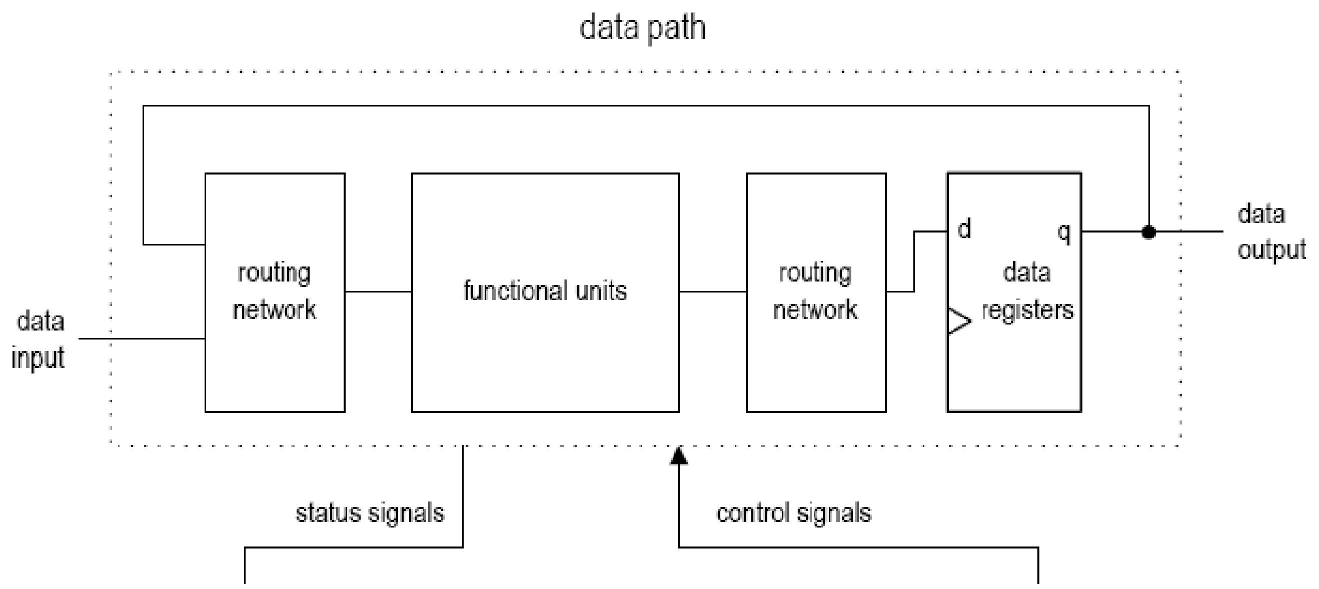
FSMD- you saw previously!

Control Path: FSM

- ***Command***: the external command signal to the FSMD
- ***Internal status***: signal from the data path.
- ***Control signal***: output, used to control data path operation.
- ***External status***: output, used to indicate the status of the FSMD



FSMD- you saw previously!



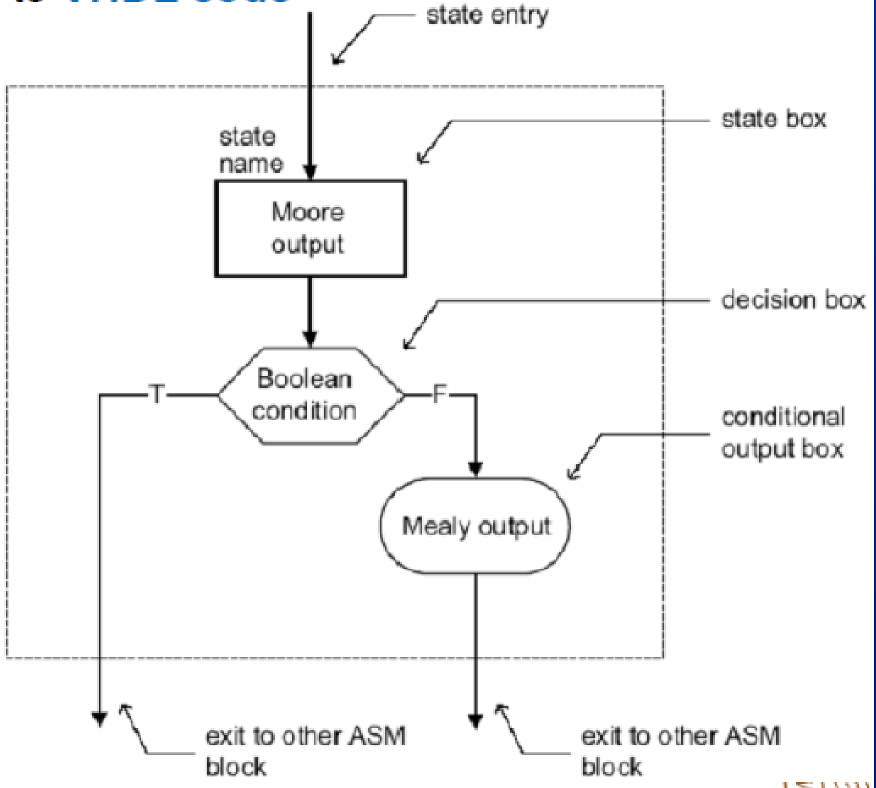
- **Data Path: perform all the required RT operations**
  - ***Data registers***: store the intermediate results.
  - ***Functional units***: perform RT operations
  - ***Routing circuit***: connection, selection (multiplexers)

ASM

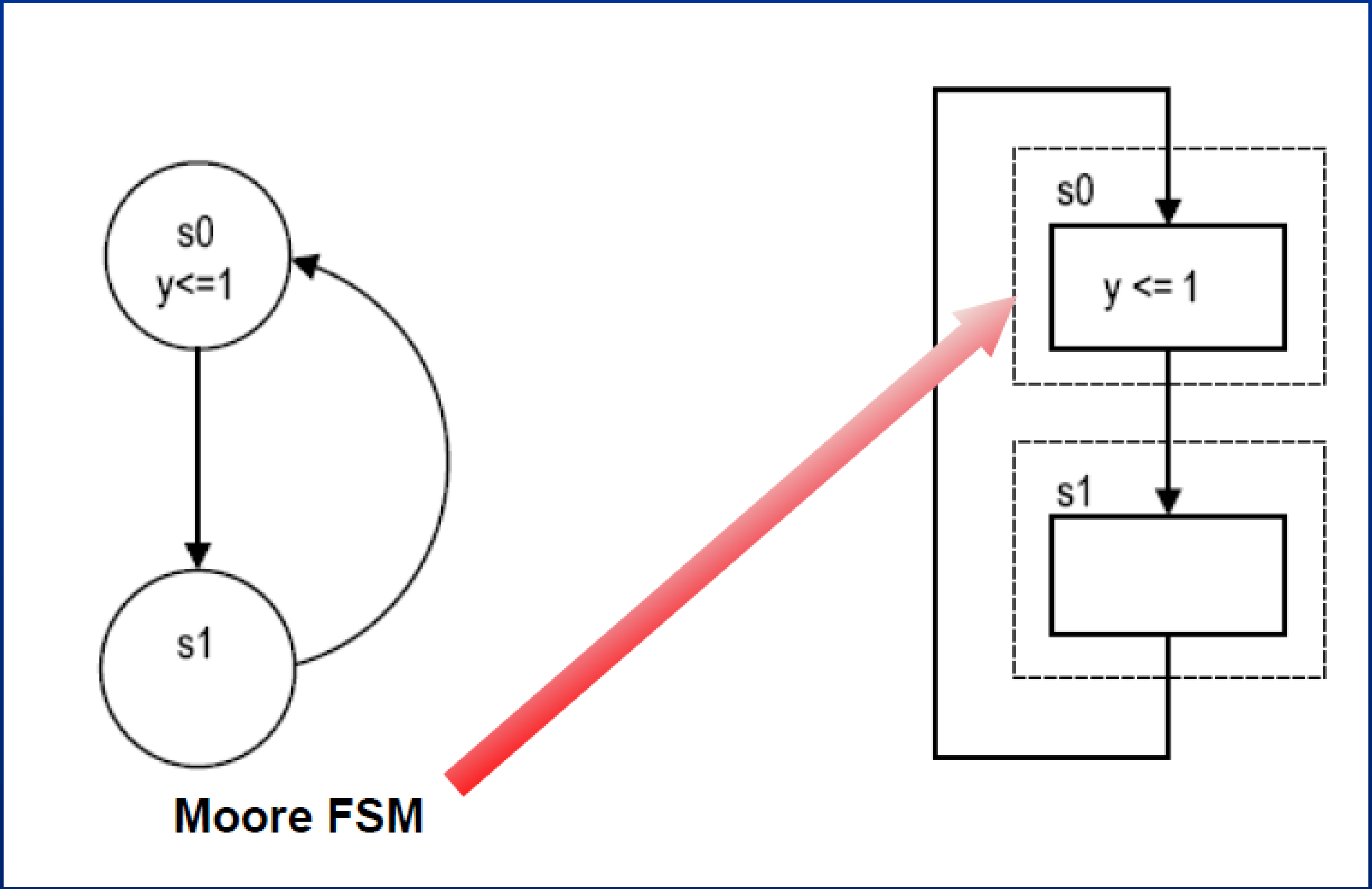
- **ASM (algorithmic state machine) chart**
  - **Flowchart-like** diagram, provide the same information as an FSM
  - More **descriptive**, better for complex algorithm (both condition and un-condition operations)
  - Can easily be transformed to **VHDL code**

An ASM chart is a network of **ASM blocks**

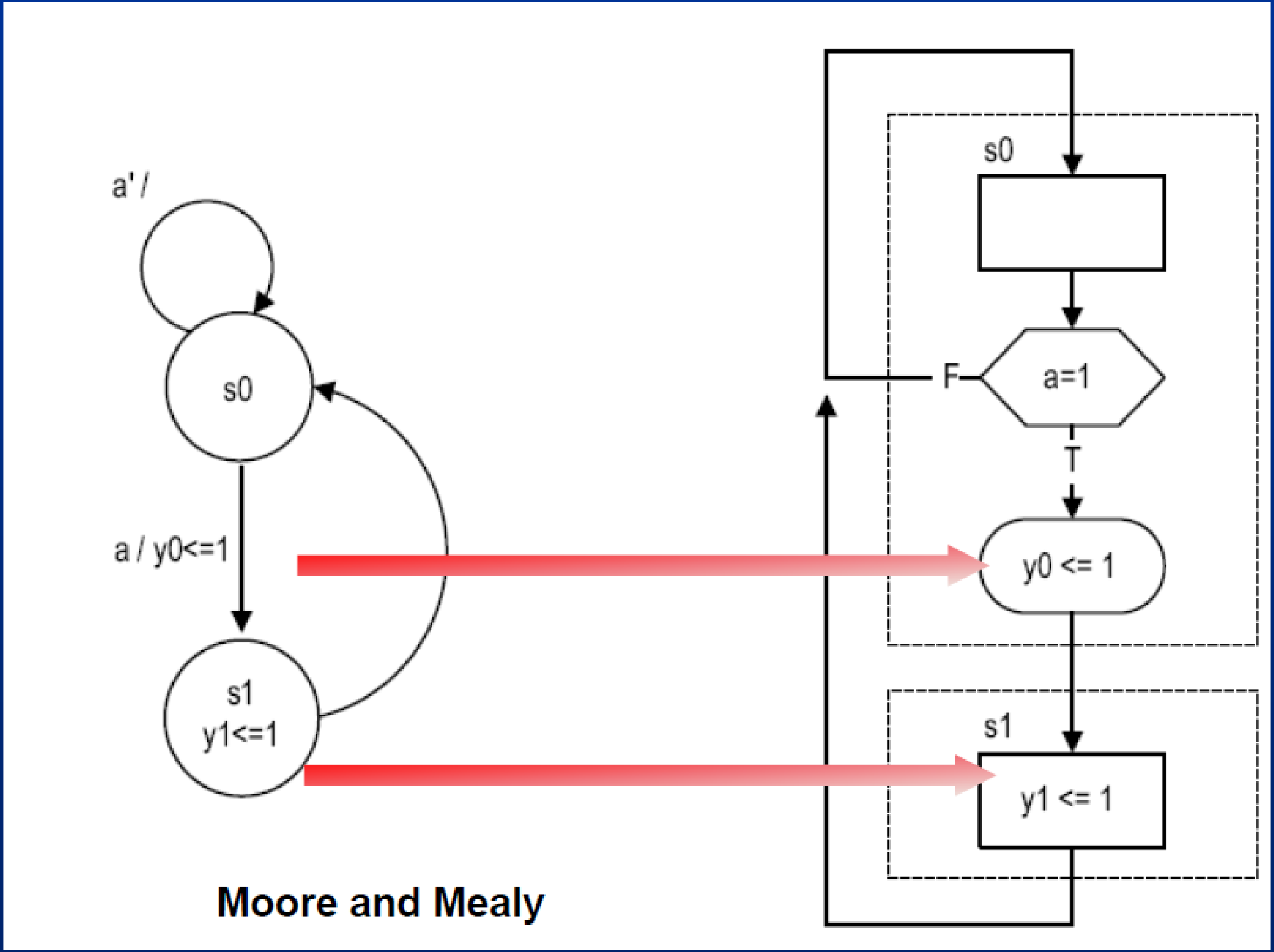
- ***One state box: FSM state***
- ***Decision boxes: with T or F exit path: next state logic***
- ***Conditional output (operation) boxes: for Mealy output***



ASM- cont’d



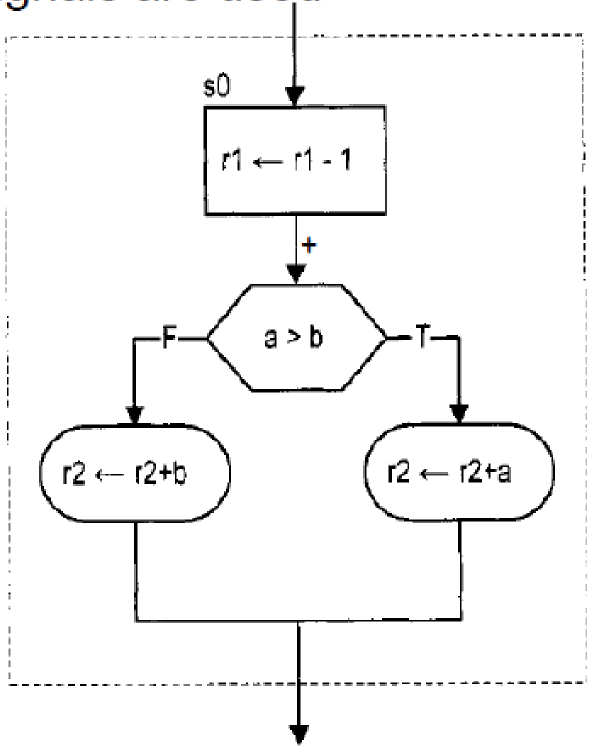
ASM- cont'd



ASMD

- ASMD:
  - Extend ASM chart to incorporate RT operations
  - RT operations are treated as another type of activity and placed where the output signals are used

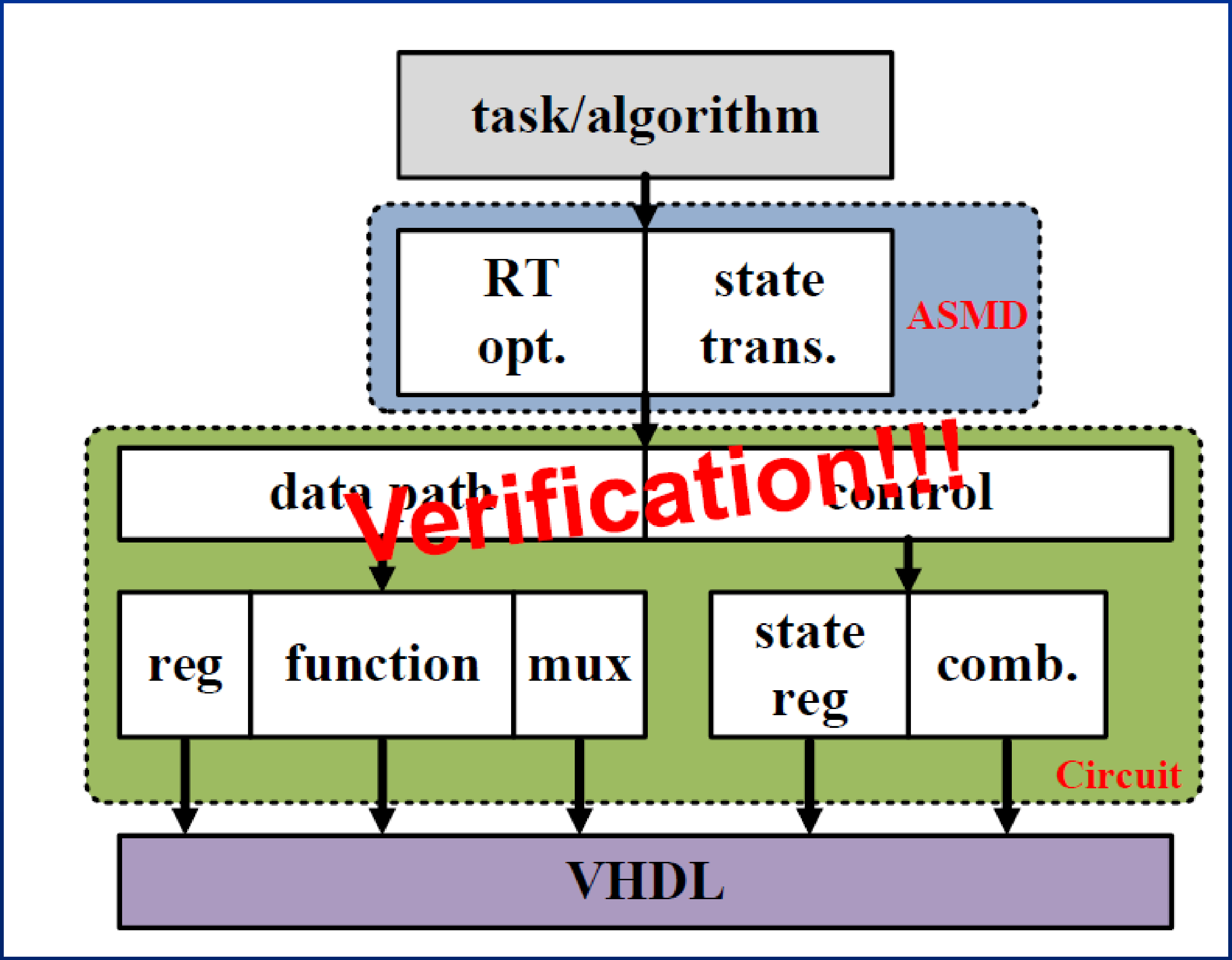
```
S0:
r1←r1+1
if a>b
   r2←r2+a
else
   r2←r2+b
```



← :RT operation, infer a register

= or <=:infer combinational logic

Design Flow


task/algorithm
RT opt.
state trans.
ASMD
data path
control
Verification!!!
reg
function
mux
state reg
comb.
Circuit
VHDL

ASMD- Map algorithm to ASMD

- Basic algorithm: 7*5 = 7+7+7+7+7

```
if (a_in=0 or b_in=0) then {
   r = 0;}
else{
   a = a_in;
   n = b_in;
   r = 0;
   while (n != 0 ){
      r = r + a;
      n = n-1;}
}
return(r)
```

Pseudo code

```
      if (a_in=0 or b_in=0) then {
         r = 0;}
      else{
         a = a_in;
         n = b_in;
         r = 0;
op:      r = r + a;
         n = n-1;
         if (n = 0) then{
            goto stop;}
         else{
           goto op;}
      }
stop: return(r);
```

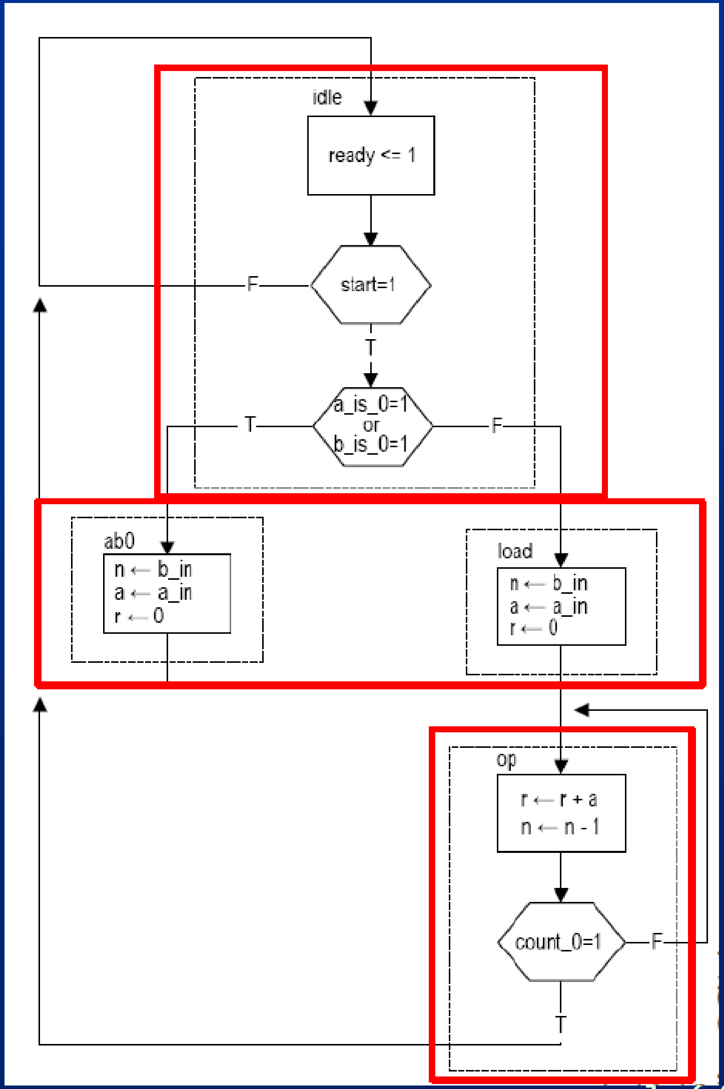
ASMD-friendly code

ASMD chart

- **Input:**
  - a_in, b_in: 8-bit unsigned
  - clk, reset
  - start: command
- **Output:**
  - r: 16-bit unsigned
  - ready: ready for new input
- **ASMD chart**
  - 3 registers (n,a,r)
  - 4 states
  - Data-path: RT operations
  - FSM: state transition

```
        if (a_in=0 or b_in=0) then {
           r = 0;}
        else{
           a = a_in;
           n = b_in;
           r = 0;
op:        r = r + a;
           n = n-1;
           if (n = 0) then{
              goto stop;}
           else{
             goto op;}
        }
stop:   return(r);
```

ASMD-friendly code

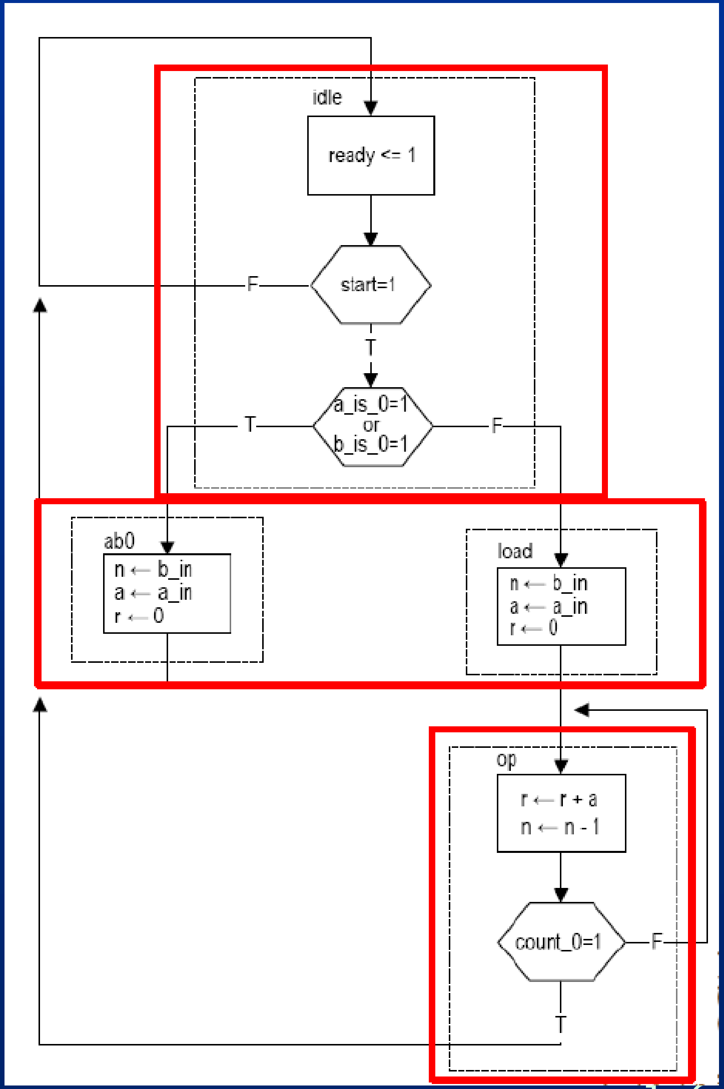


ASMD Translate to Hardware: FSMD Construction

- Construction of the data path
  - List all possible RT operations
  - Group RT operation according to the destination register
  - Add combinational circuit/mux

Grouping RT Operations

- RT operations with the r register:
  - $r \leftarrow r$ (in the idle state)
  - $r \leftarrow 0$ (in the load and ab0 states)
  - $r \leftarrow r + a$ (in the op state)
- RT operations with the n register:
  - $n \leftarrow n$ (in the idle state)
  - $n \leftarrow b\_in$ (in the load and ab0 states)
  - $n \leftarrow n - 1$ (in the op state)
- RT operations with the a register:
  - $a \leftarrow a$ (in the idle and op states)
  - $a \leftarrow a\_in$ (in the load and ab0 states)



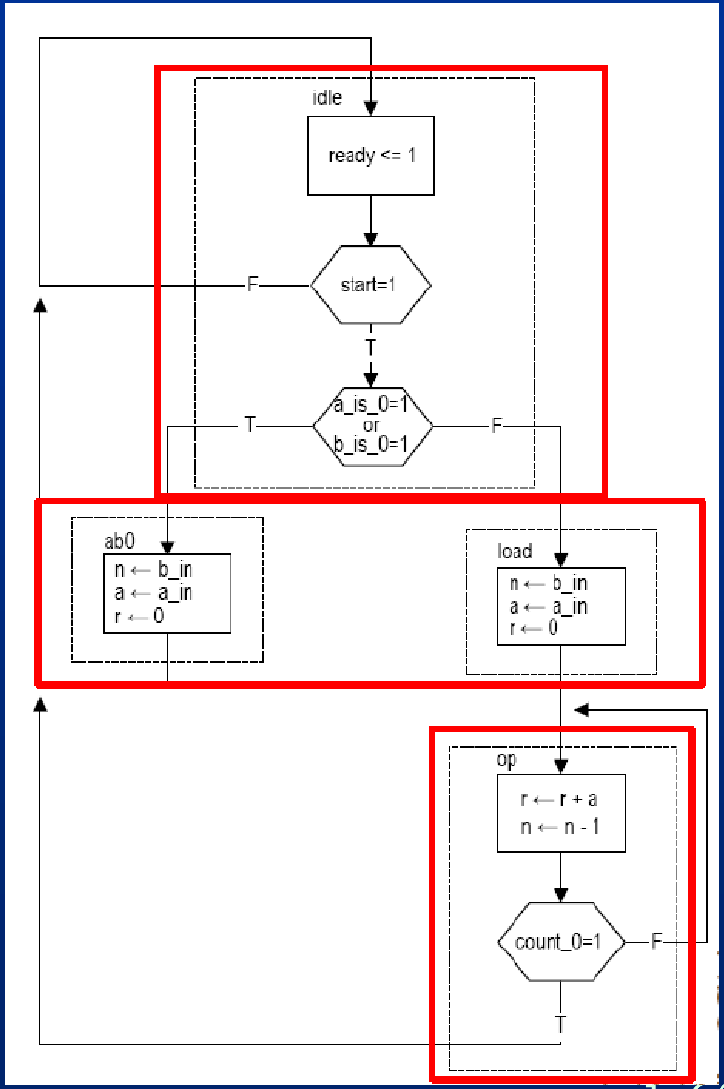
Construction of data path

Construction of the data path

- List all possible RT operations
- Group RT operation according to the **destination register**
- Add combinational circuit/mux

Grouping RT Operations

- RT operations with the r register:
  - r ← r (in the idle state)
  - r ← 0 (in the load and ab0 states)
  - r ← r + a (in the op state)
- RT operations with the n register:
  - n ← n (in the idle state)
  - n ← b_in (in the load and ab0 states)
  - n ← n - 1 (in the op state)
- RT operations with the a register:
  - a ← a (in the idle and op states)
  - a ← a_in (in the load and ab0 states)

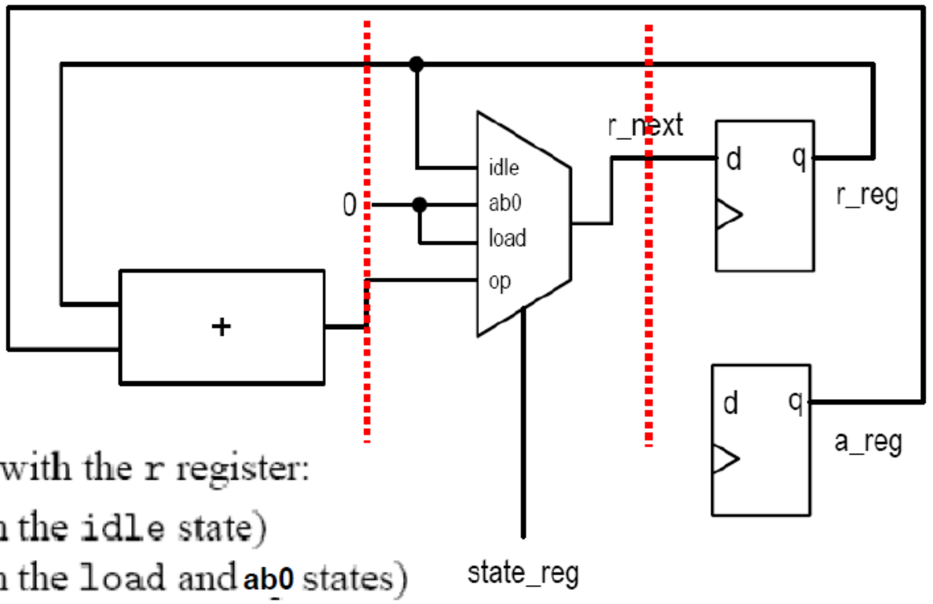


Construction of data path- first consider ‘r’

Grouping RT Operations

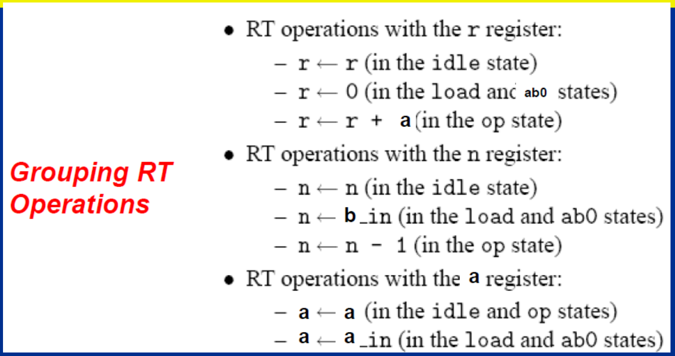
- RT operations with the r register:
  - r ← r (in the idle state)
  - r ← 0 (in the load and ab0 states)
  - r ← r + a (in the op state)
- RT operations with the n register:
  - n ← n (in the idle state)
  - n ← b_in (in the load and ab0 states)
  - n ← n - 1 (in the op state)
- RT operations with the a register:
  - a ← a (in the idle and op states)
  - a ← a_in (in the load and ab0 states)

Circuit associated with **r register**



- RT operations with the r register:
  - r ← r (in the idle state)
  - r ← 0 (in the load and ab0 states)
  - r ← r + a (in the op state)

Construction of data path- similarly for other destined regs.



Grouping RT Operations

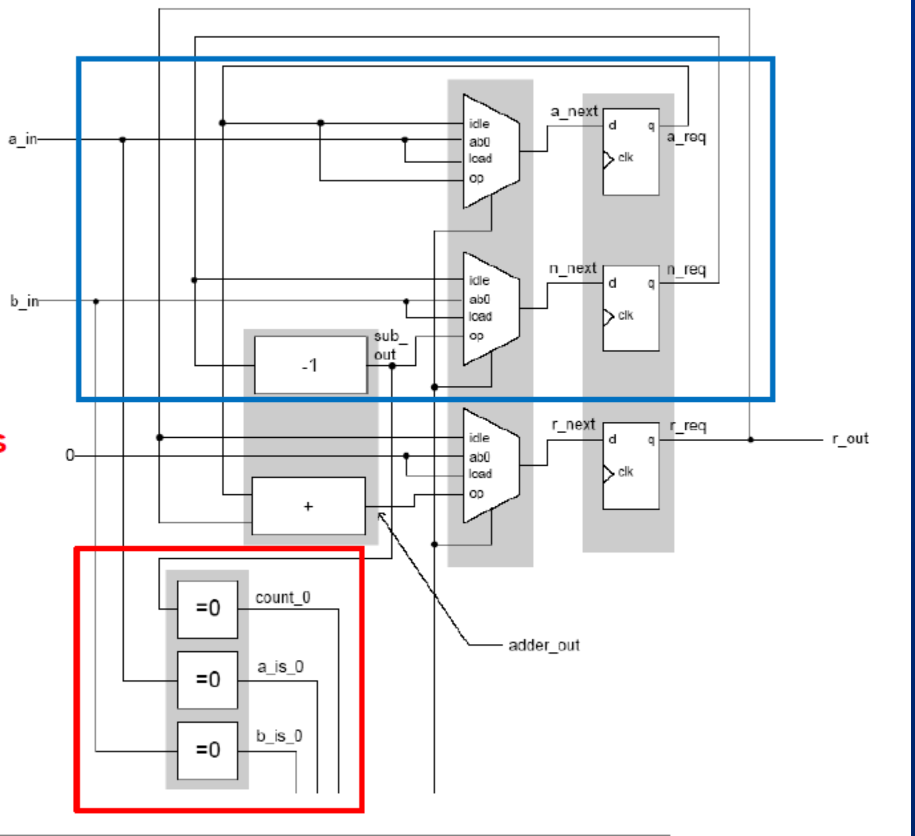
- RT operations with the r register:
  - r ← r (in the idle state)
  - r ← 0 (in the load and ab0 states)
  - r ← r + a (in the op state)
- RT operations with the n register:
  - n ← n (in the idle state)
  - n ← b_in (in the load and ab0 states)
  - n ← n - 1 (in the op state)
- RT operations with the a register:
  - a ← a (in the idle and op states)
  - a ← a_in (in the load and ab0 states)

Continue with

n-register

a-register

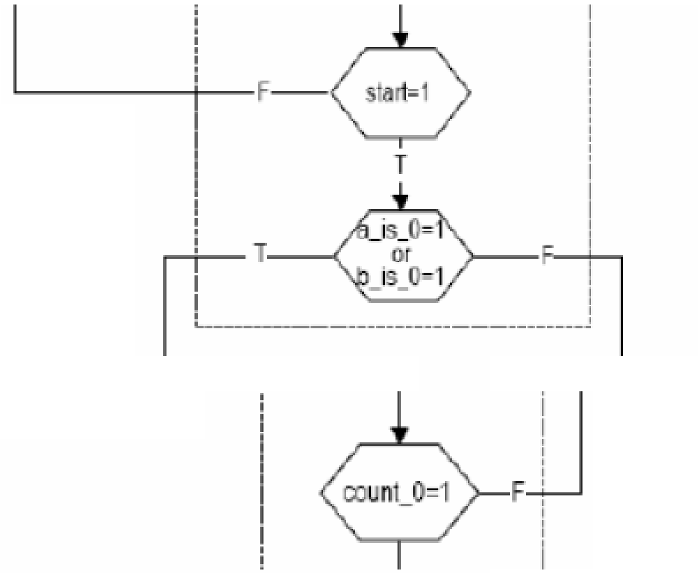
Add status circuits

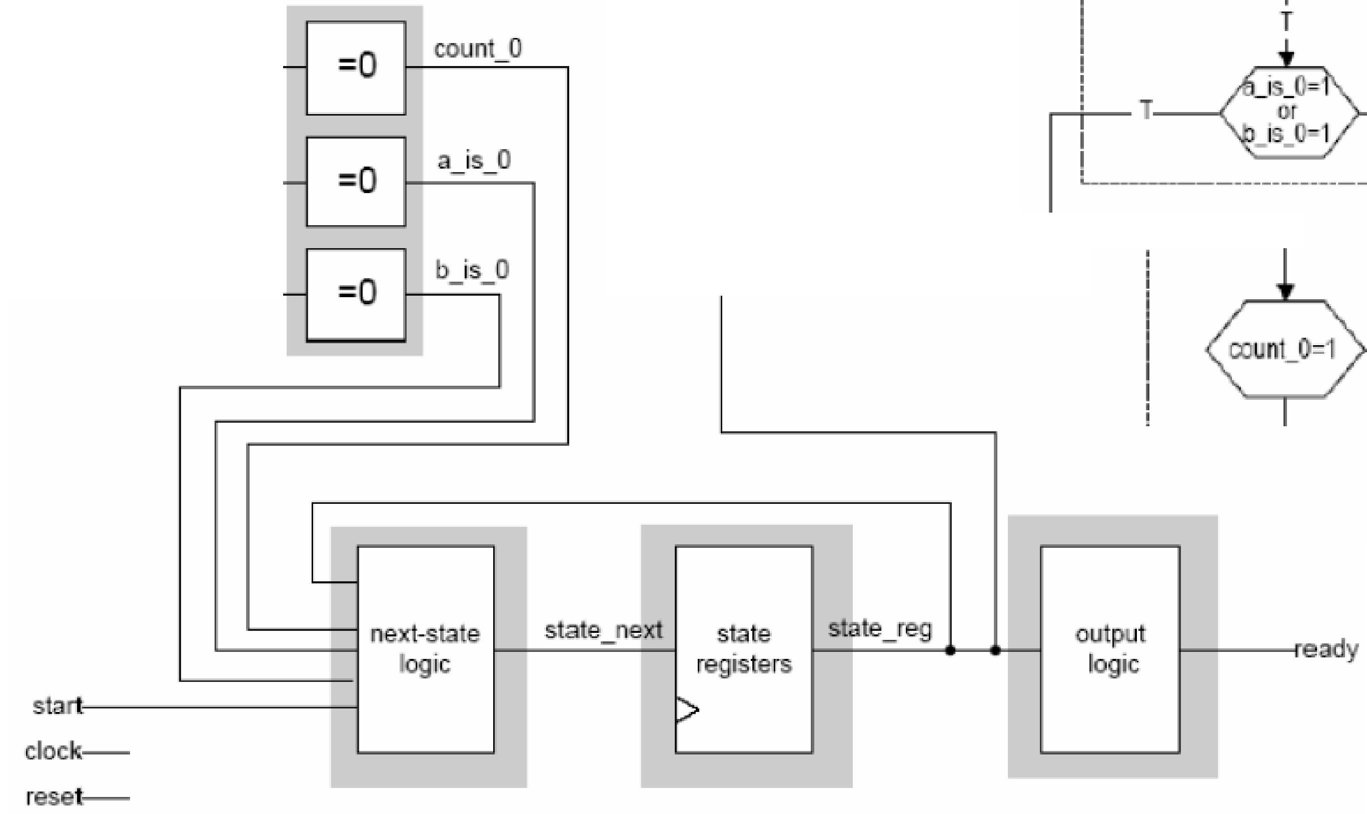


Construction of control unit

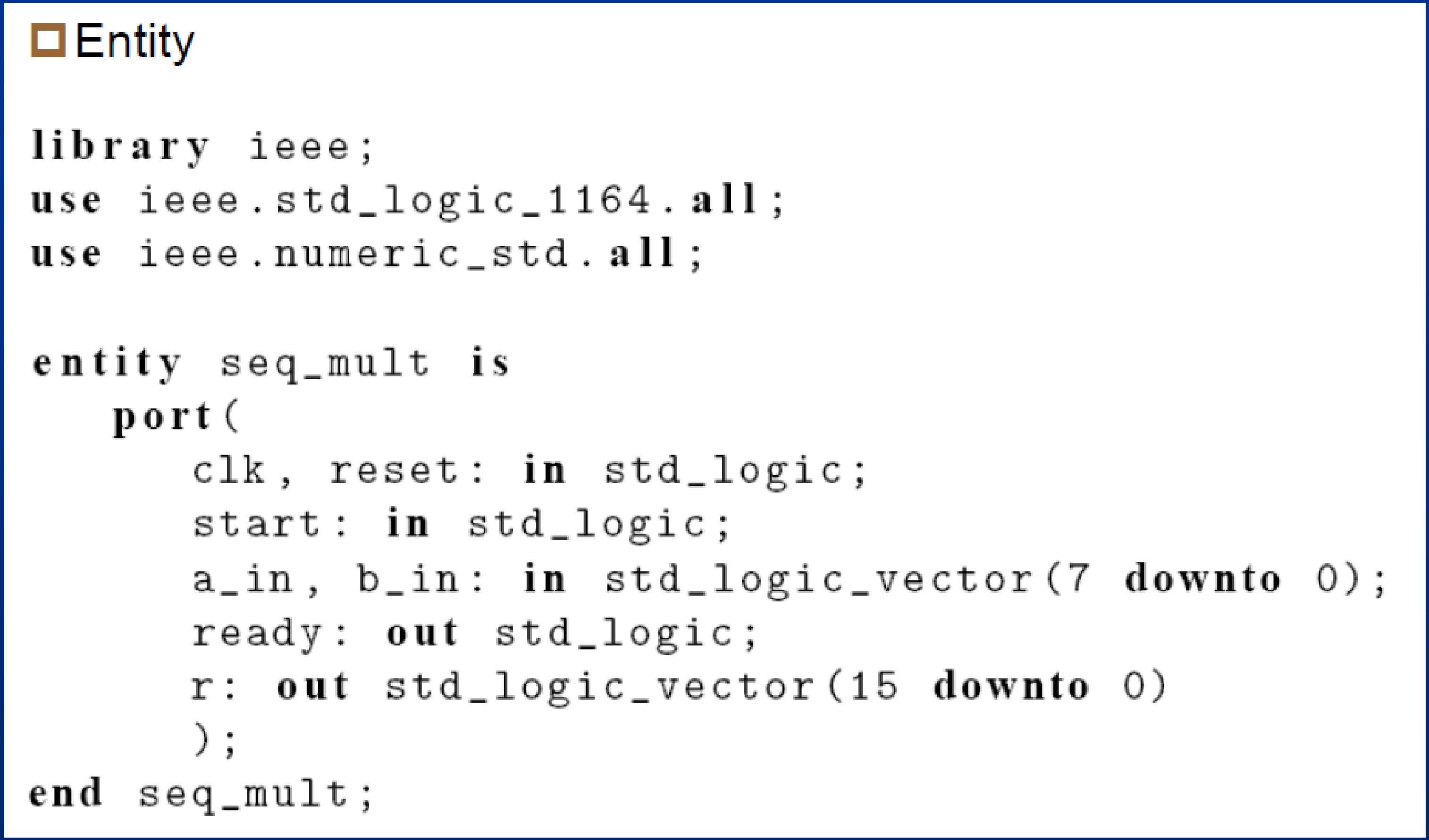
- **Input of FSM**
  - External: start, clock, reset
  - Internal: decision box in ASMD
- **Output of FSM**

Decision Box





VHDL realization



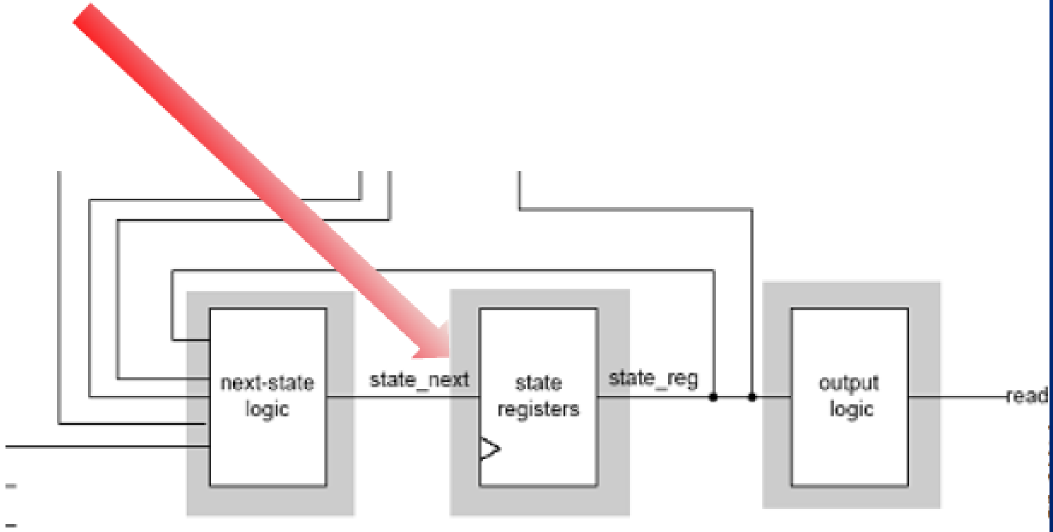
- Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity seq_mult is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(15 downto 0)
      );
end seq_mult;
```

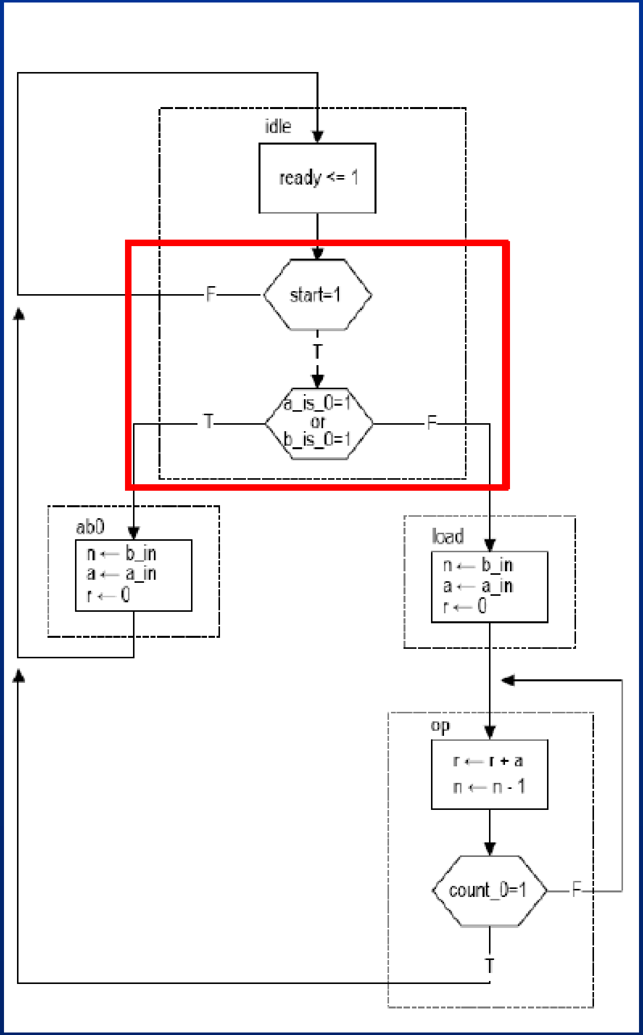
FSM: state registers

```
-- control path: state register
process(clk,reset)
begin
   if reset='1' then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
end process;
```

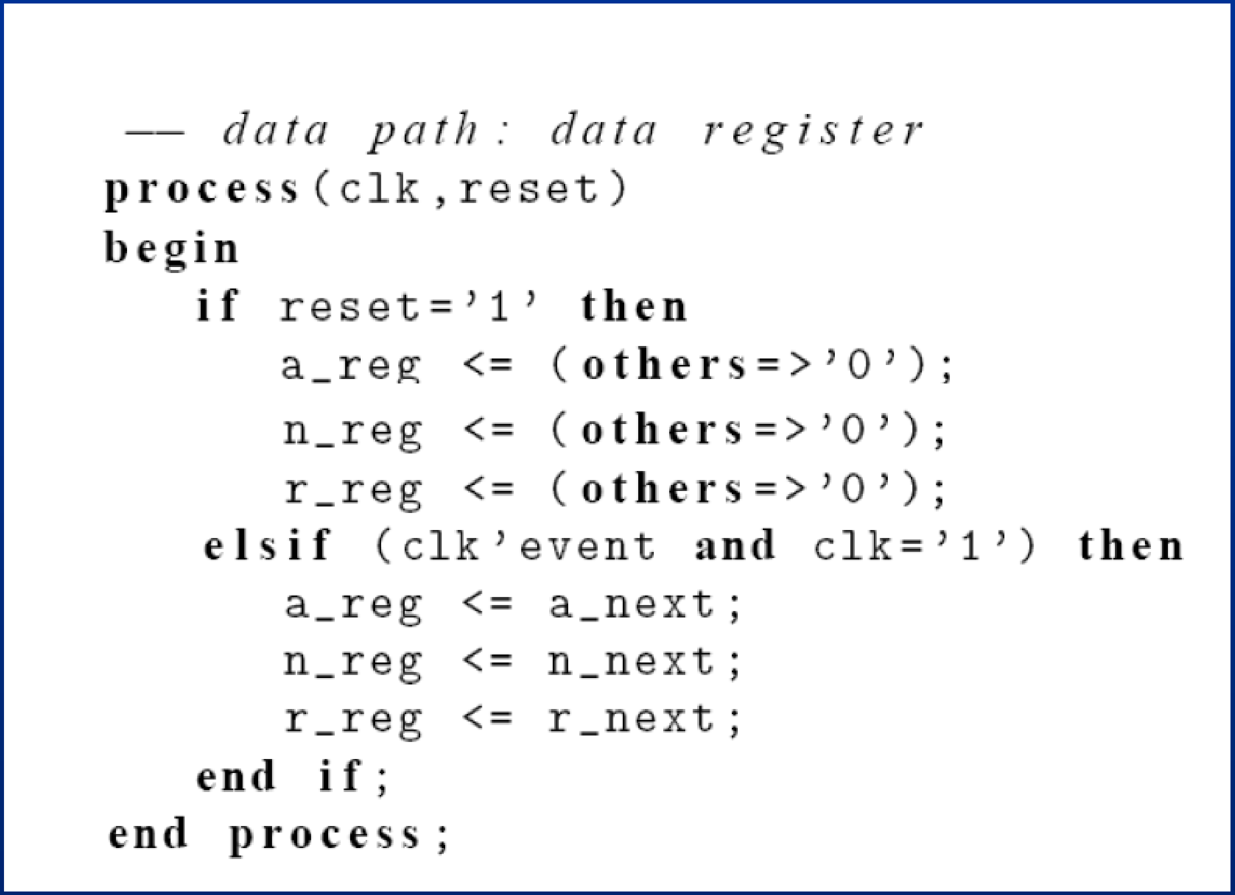


FSM: Next State & output logic

```
process(state_reg,start,a_is_0,b_is_0,count_0
begin
   case state_reg is
      when idle =>
         if start='1' then
            if (a_is_0='1' or b_is_0='1') the
               state_next <= ab0;
            else
               state_next <= load;
            end if;
         else
            state_next <= idle;
         end if;
      when ab0 =>
         state_next <= idle;
      when load =>
         state_next <= op;
      when op =>
         if count_0='1' then
            state_next <= idle;
         else
            state_next <= op;
         end if;
   end case;
end process;
ready <= '1' when state_reg=idle else '0';
```

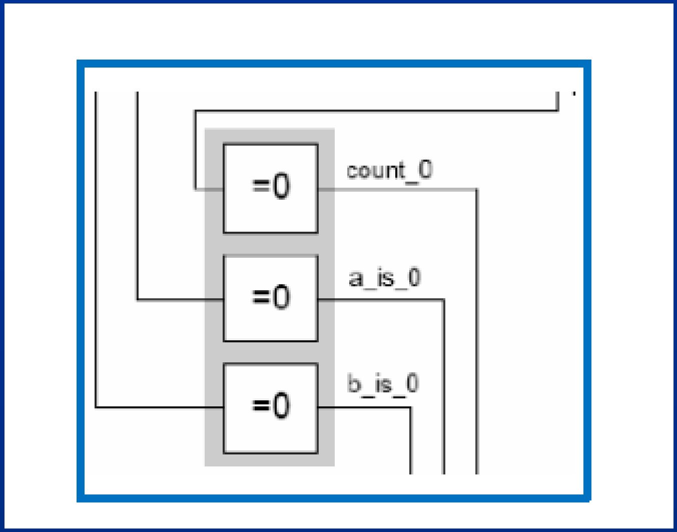


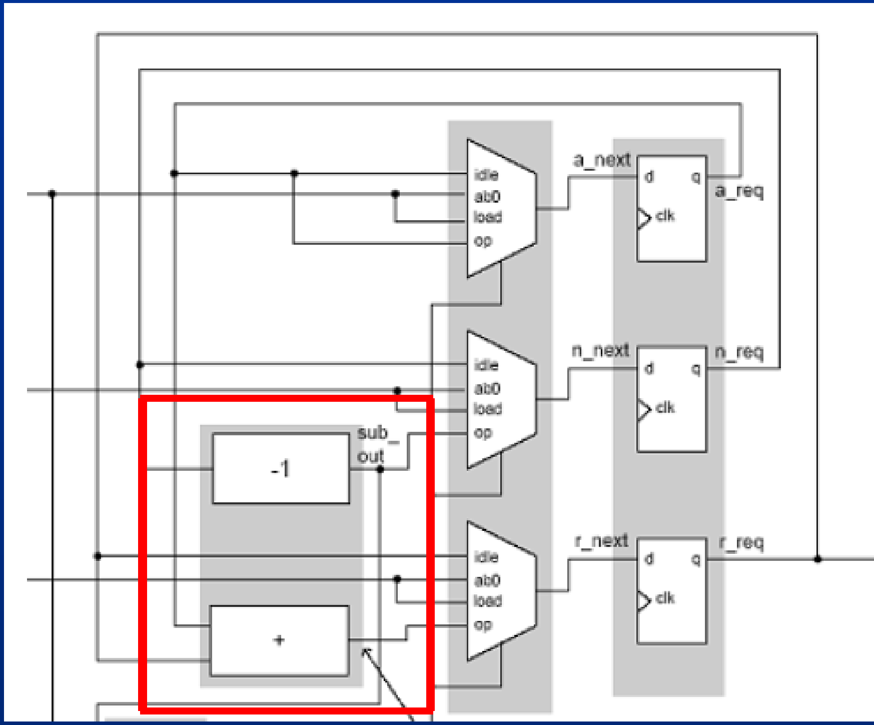
Data path (Destination Registers)



```
-- data path: data register
process(clk,reset)
begin
   if reset='1' then
      a_reg <= (others=>'0');
      n_reg <= (others=>'0');
      r_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      a_reg <= a_next;
      n_reg <= n_next;
      r_reg <= r_next;
   end if;
end process;
```

Data path (Function Unit)

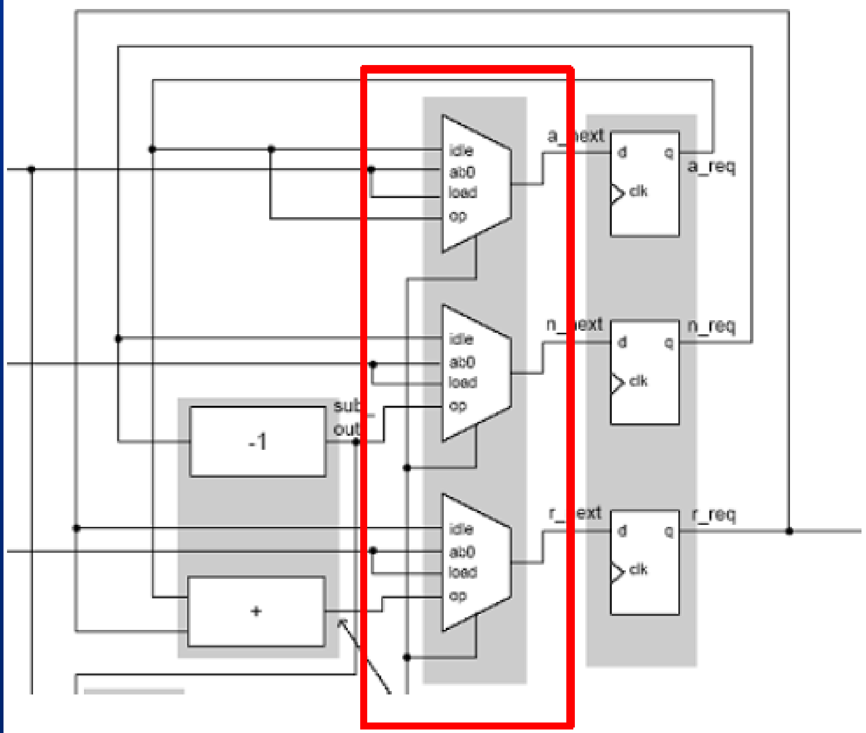




```
-- data path: status
a_is_0 <= '1' when a_in="00000000" else '0';
b_is_0 <= '1' when b_in="00000000" else '0';
count_0 <= '1' when n_next="00000000" else '0'
```

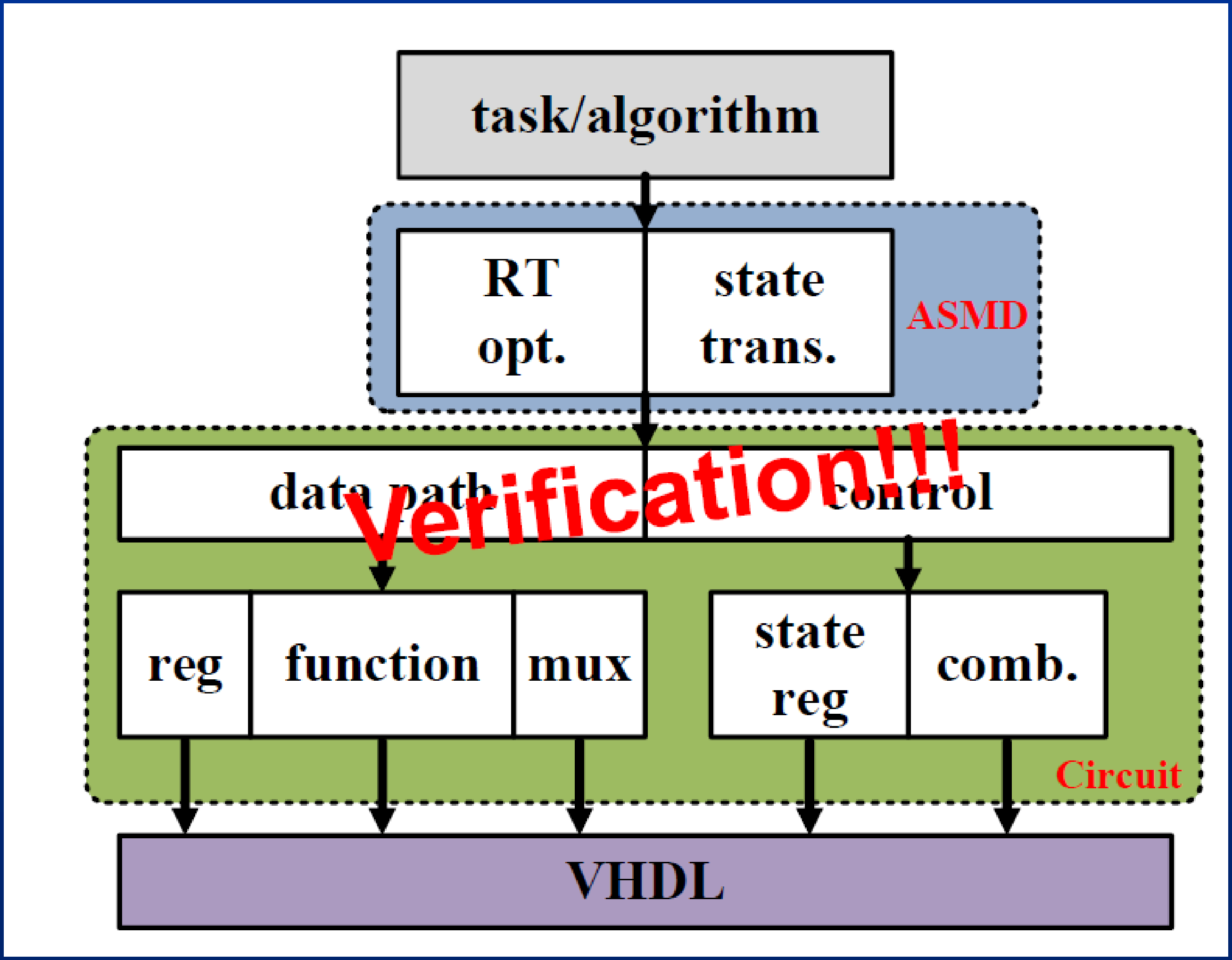
```
-- data path: functional units
adder_out <= ("00000000" & a_reg) + r_reg;
sub_out <= n_reg - 1;
```

Data path (Mux Routing)



```
process(state_reg,a_reg,n_reg,r_reg,
        a_in,b_in,adder_out,sub_out)
begin
   case state_reg is
      when idle =>
         a_next <= a_reg;
         n_next <= n_reg;
         r_next <= r_reg;
      when ab0 =>
         a_next <= unsigned(a_in);
         n_next <= unsigned(b_in);
         r_next <= (others=>'0');
      when load =>
         a_next <= unsigned(a_in);
         n_next <= unsigned(b_in);
         r_next <= (others=>'0');
      when op =>
         a_next <= a_reg;
         n_next <= sub_out;
         r_next <= adder_out;
      end case;
end process;
```

Design Flow


task/algorithm
RT opt.
state trans.
ASMD
data path
control
Verification!!!
reg
function
mux
state reg
comb.
Circuit
VHDL