



LUND  
UNIVERSITY

# EITF35: Introduction to Structured VLSI Design

Part 3.1.1: FSMD

Liang Liu  
liang.liu@eit.lth.se



# Outline

- **FSMD Overview**
- **Algorithmic state machine with data-path (ASMD)**
- **FSMD design of a repetitive-addition multiplier**
- **Timing analysis of FSMD**



# Why FSMD? Start with algorithm

- **Task:** sums four elements of an array, divides the sum by 8 and rounds the result to the closest integer

```
size = 4
sum = 0;
for i in (0 to size-1) do {
    sum = sum + a(i);}
q = sum / 8;
r = sum rem 8;
if (r > 3) {
    q = q+1;}
outp = q;
```

*Algorithm: a sequence  
steps of actions*

- **Two characteristics of an algorithm:**
  - Use of **variables**  
e.g., sum, or  $n = n + 1$  in C and Matlab
  - **Sequential execution**  
e.g., sum must be finished before division



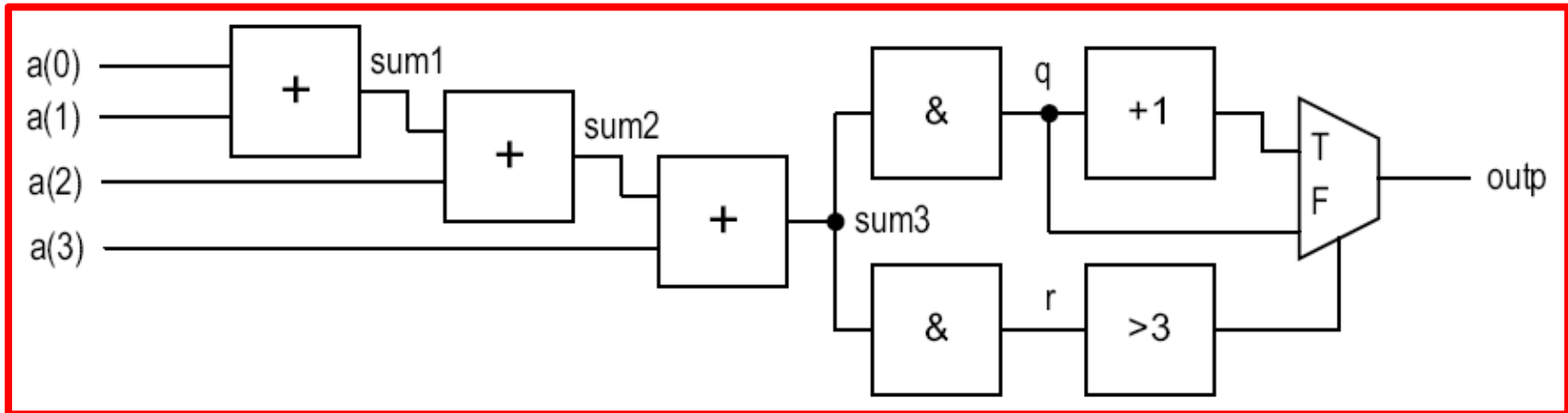
# Converting algorithm to hardware

## □ “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;
```

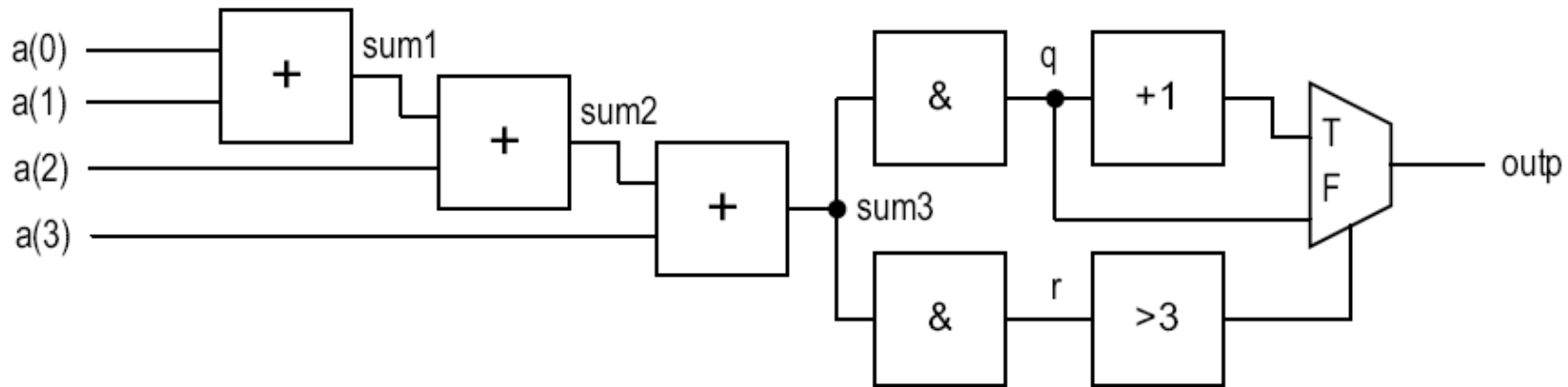
*The "sequential" operations are represented by the data flow from left to right*



# Dataflow Implementation: 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*



# Alternatively?

## □ 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

$$r_{\text{dest}} \leftarrow f(r_{\text{src1}}, \dots, r_{\text{srcn}})$$

- Sequence of data manipulation and transfer among registers (**RTL**)

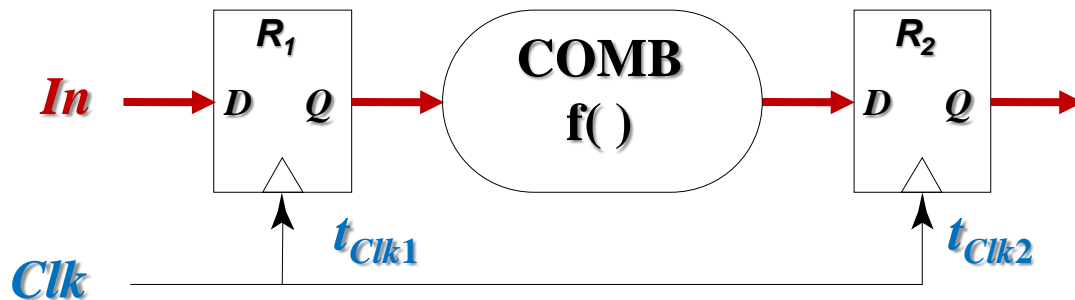


# RT Operation: Timing

$$r_{\text{dest}} \leftarrow f(r_{\text{src1}}, r_{\text{src2}}, \dots, r_{\text{srcn}})$$

## □ Timing:

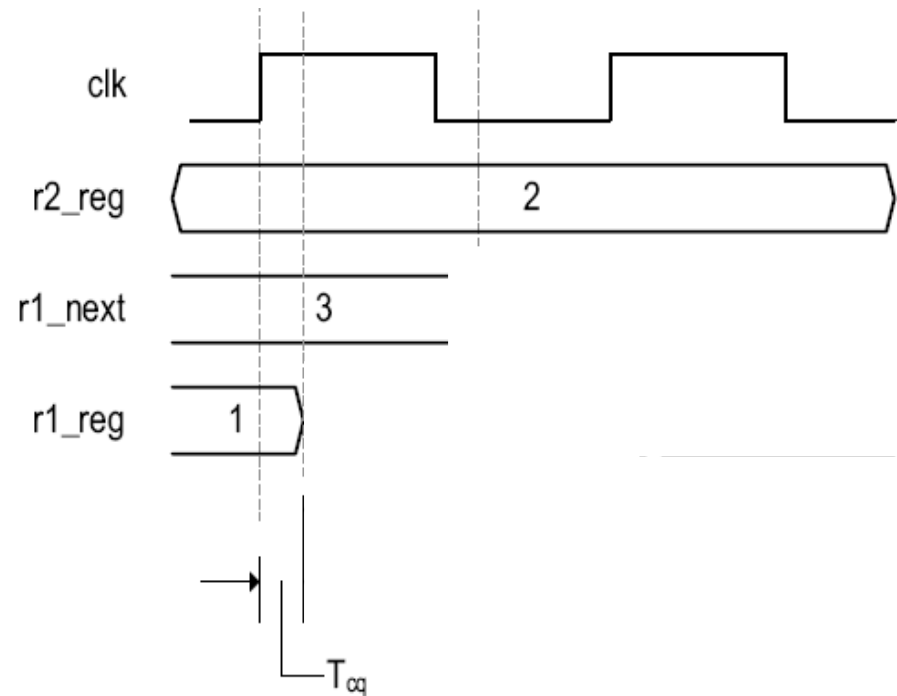
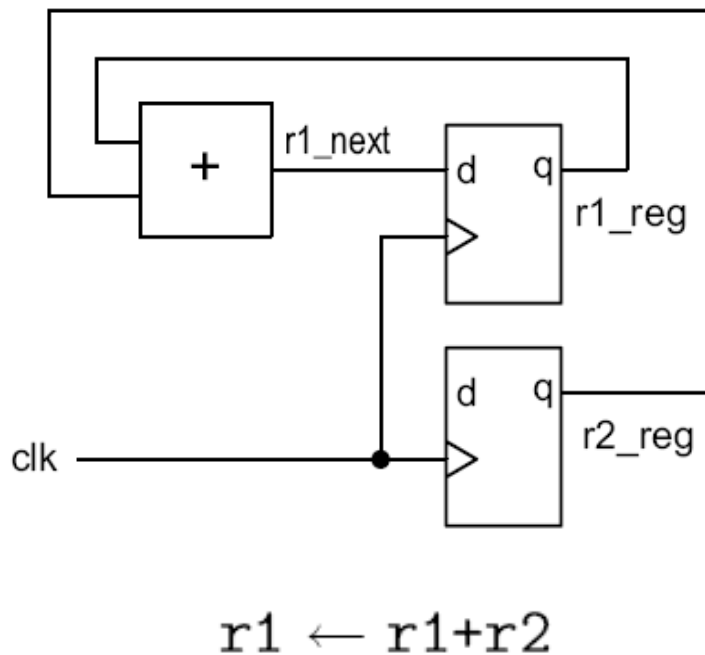
- **Hardware!** major difference between a *variable* and a *register* is that a **clock** is embedded in an RT operation
- Rising edge of clk: outputs of source reg  $r_{\text{src1}}$   $r_{\text{src2}}$  etc. are available
- The output are passed to a combinational circuit that performs  $f()$
- At the **next rising edge** of the clock, the result is stored into  $r_{\text{dest}}$



# Hardware Mapping of RT: Example 1

## □ E.g. $r1 \leftarrow r1+r2$

- C1:  $r1\_next \leftarrow r1\_reg + r2\_reg$
- C2:  $r1\_reg \leftarrow r1\_next$



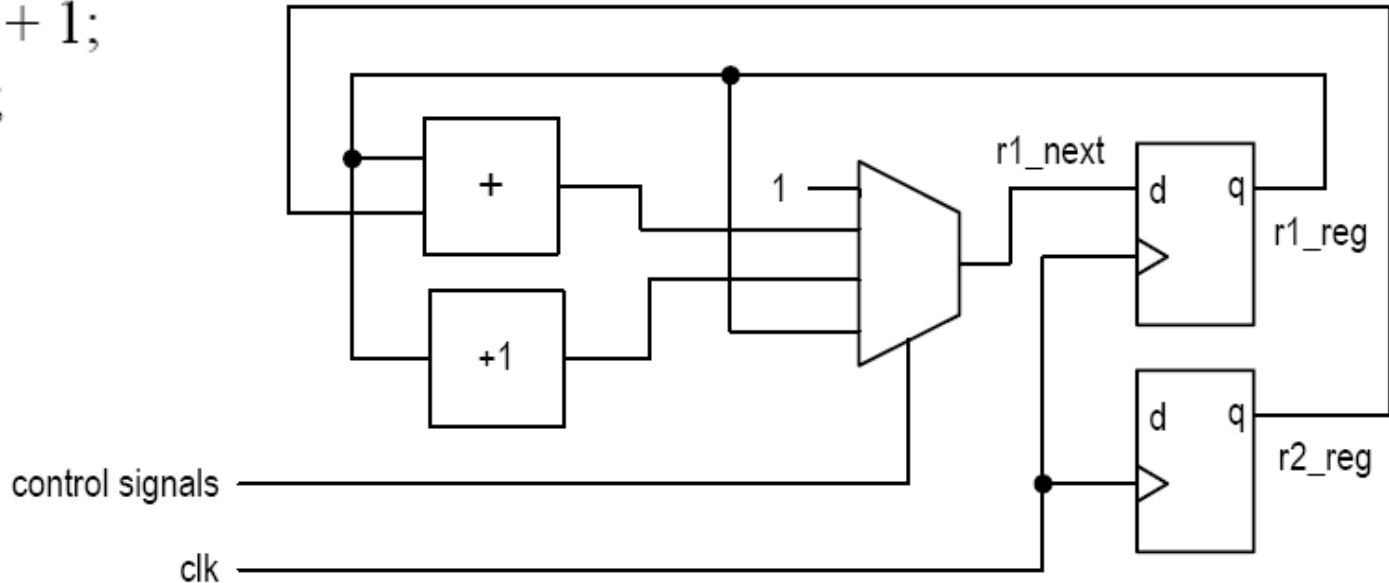


# Hardware Mapping of RT: Example 2

## Multiple RT operations

How can we organize multiple operations on one register? Discuss!

```
r1 ← 1;  
r1 ← r1 + r2;  
r1 ← r1 + 1;  
r1 ← r1;
```



Control signals are needed!



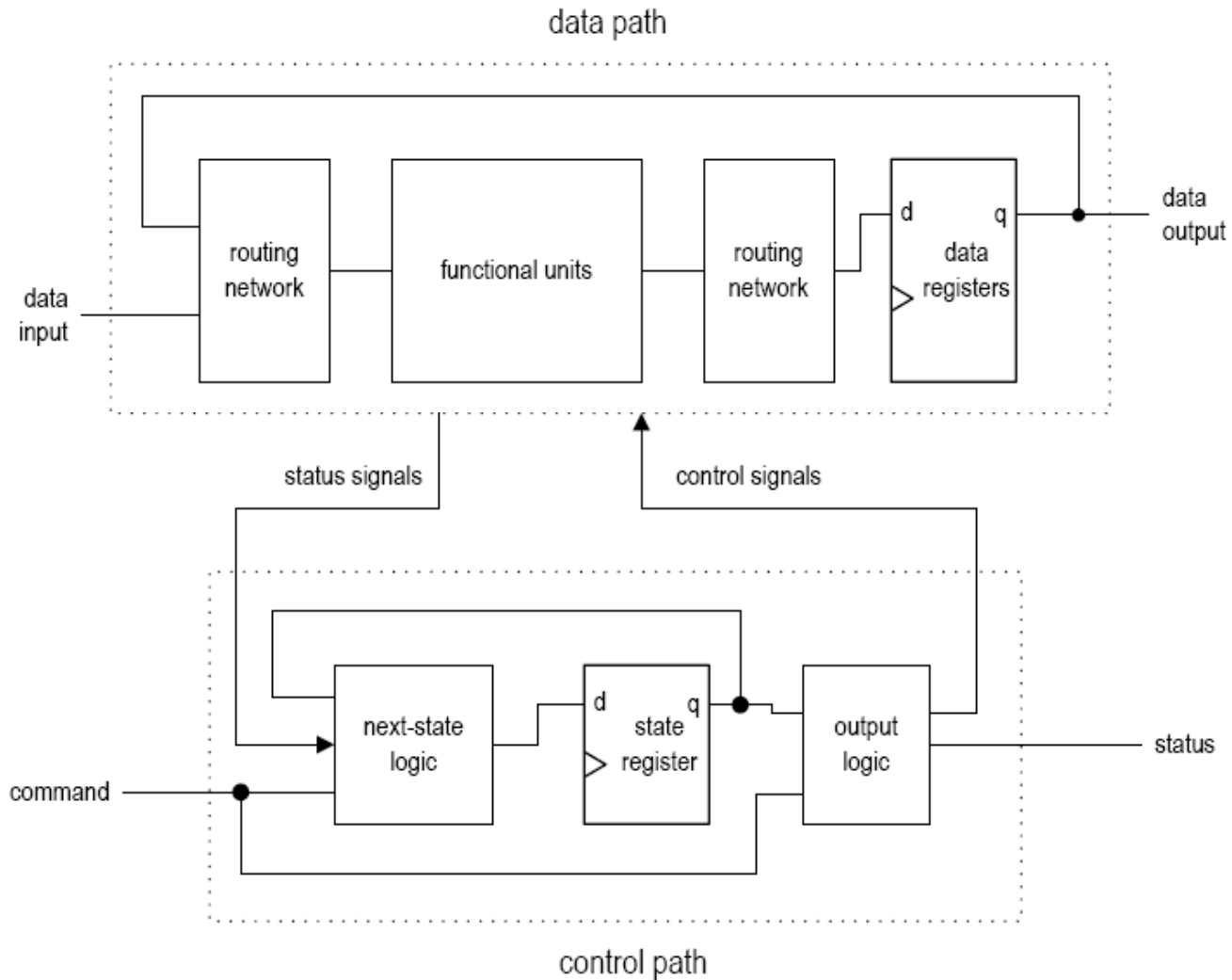
# FSM as control path

## □ FSMD: FSM with data path

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



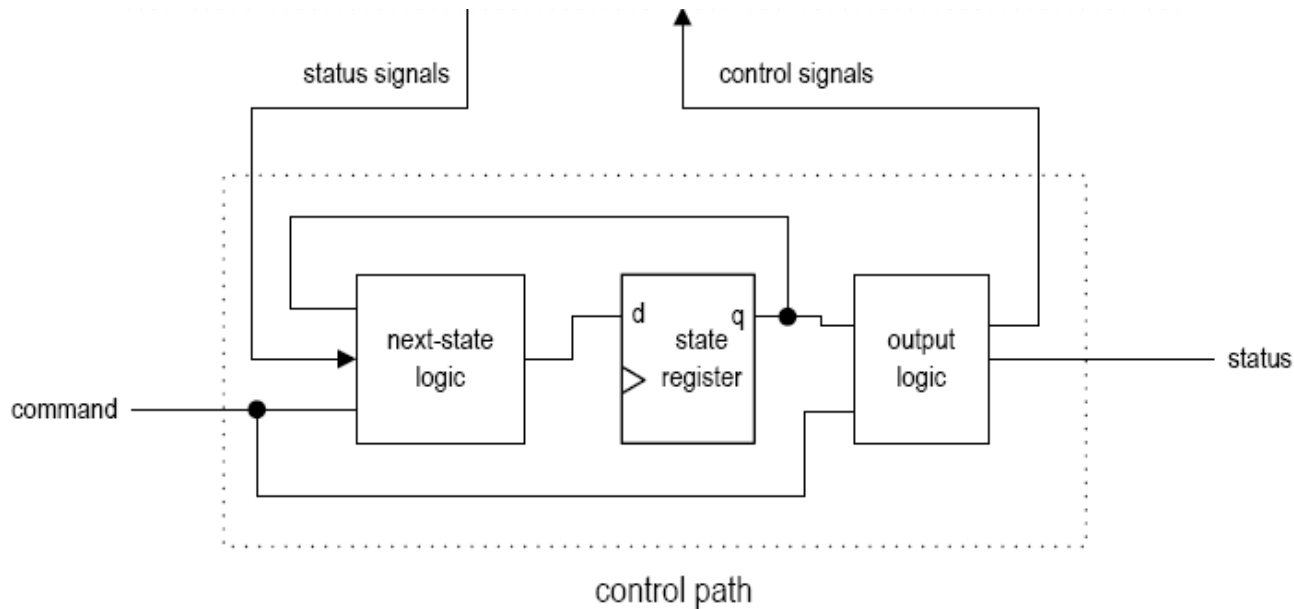
# FSMD (FSM with Date Path)



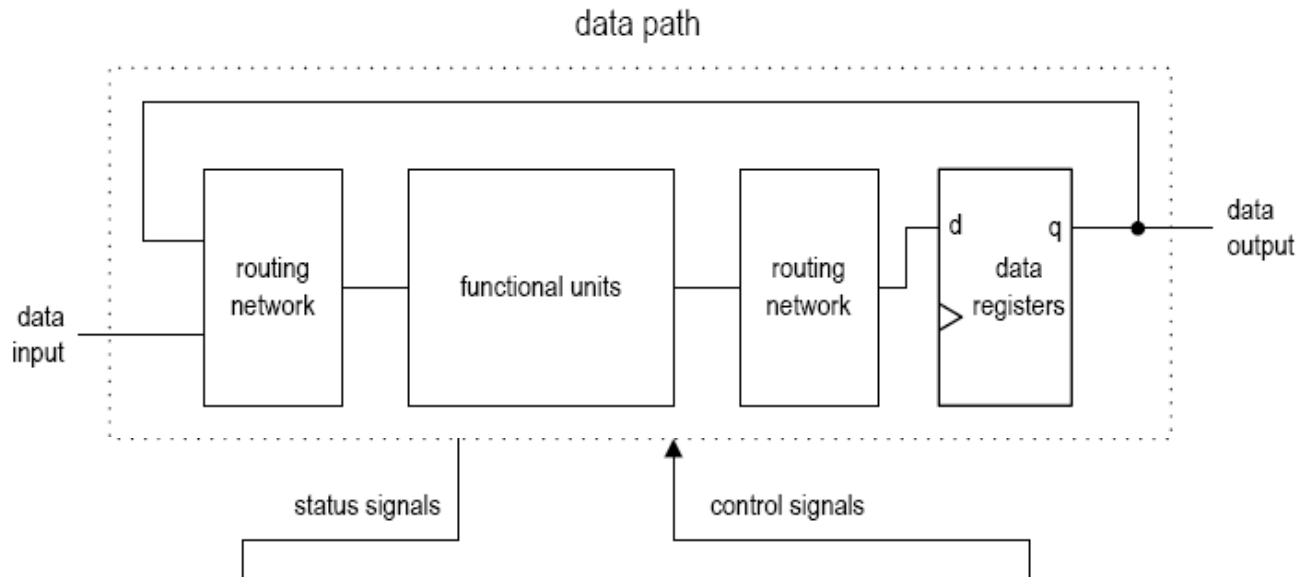
# FSMD (FSM with Data Path)

## □ 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 (FSM with Date Path)



## □ **Data Path:** perform all the required RT operations

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



# Outline

- Overview of FSMD
- **Algorithmic state machine with data-path (ASMD)**
- FSMD design of a repetitive-addition multiplier
- Timing analysis of FSMD



# ASM (algorithmic state machine)

## □ ASM (algorithmic state machine) chart

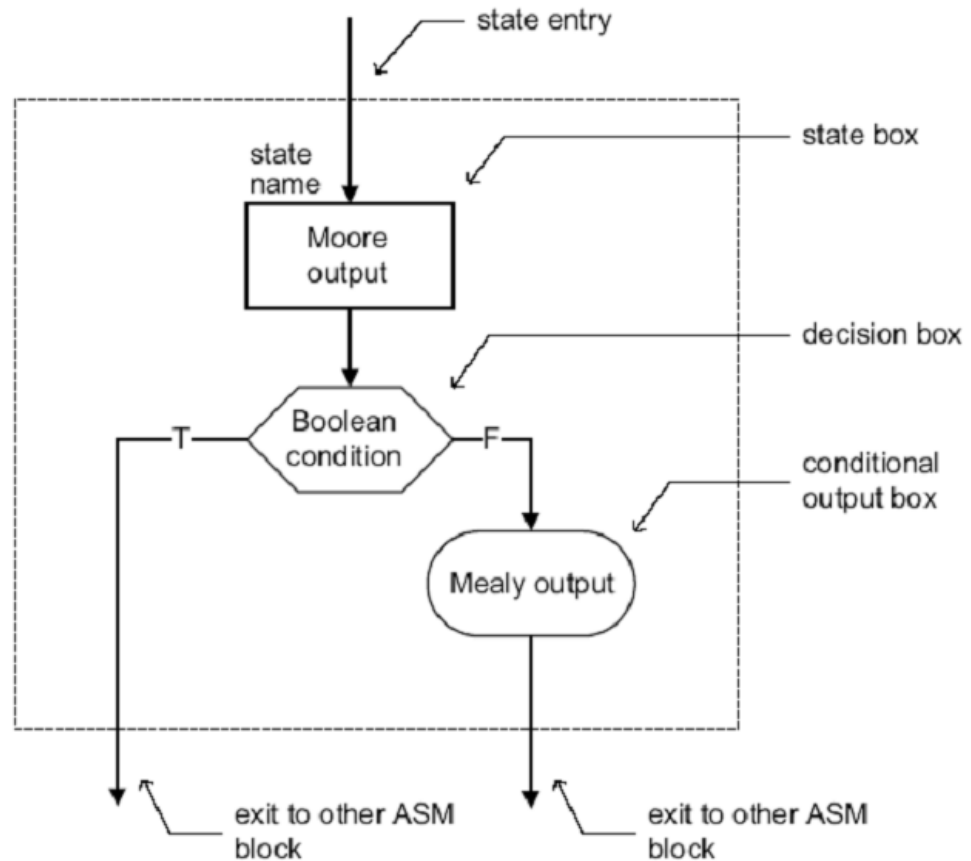
- **Flowchart-like** diagram, provide the same information as an FSM
- More **descriptive**, better for complex algorithm
- Can easily be transformed to **VHDL code**

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

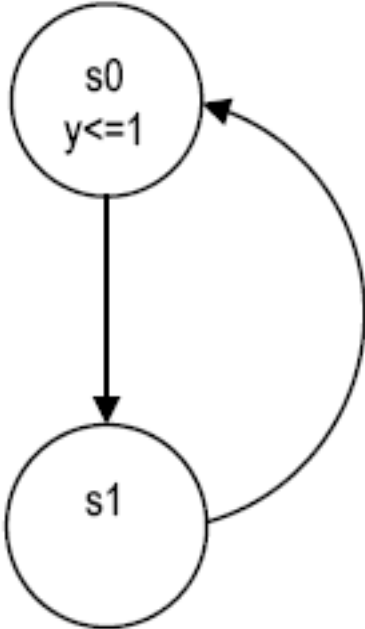
□ **One state box: FSM state**

□ **Decision boxes: with *T* or *F* exit path: next state logic**

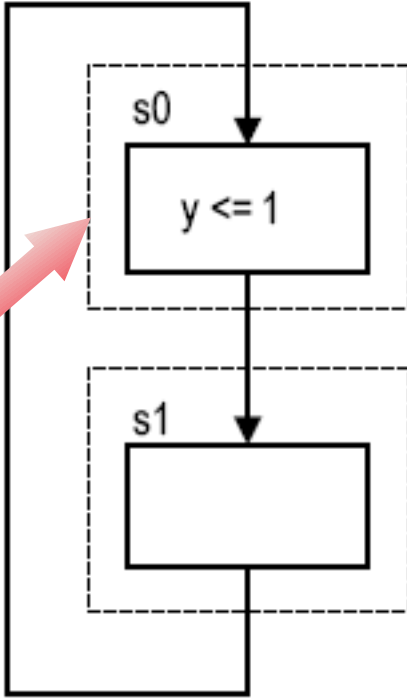
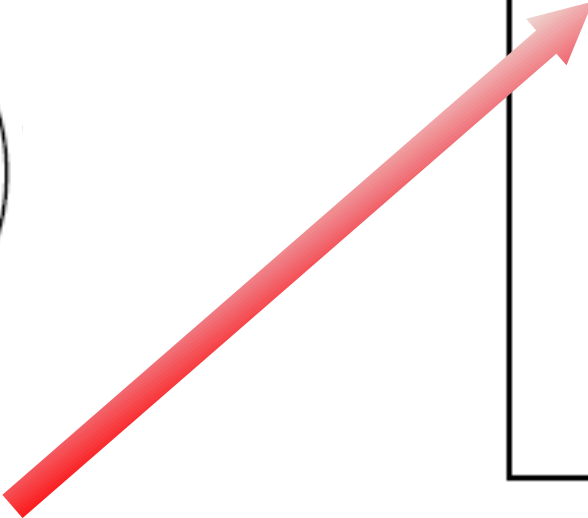
□ **Conditional output boxes: for Mealy output**



# State Diagram and ASM Chart: Example 1

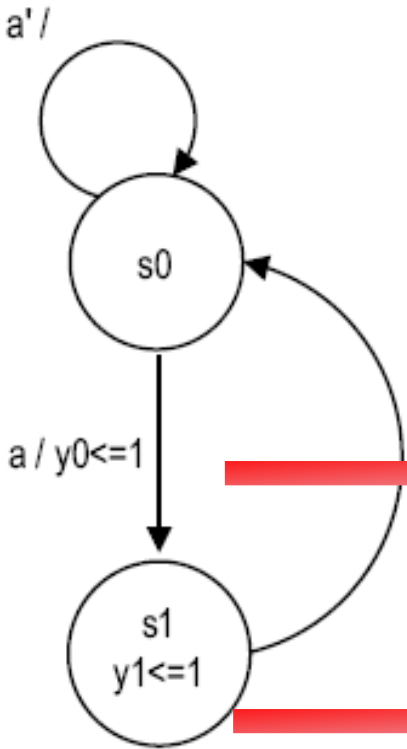


Moore FSM

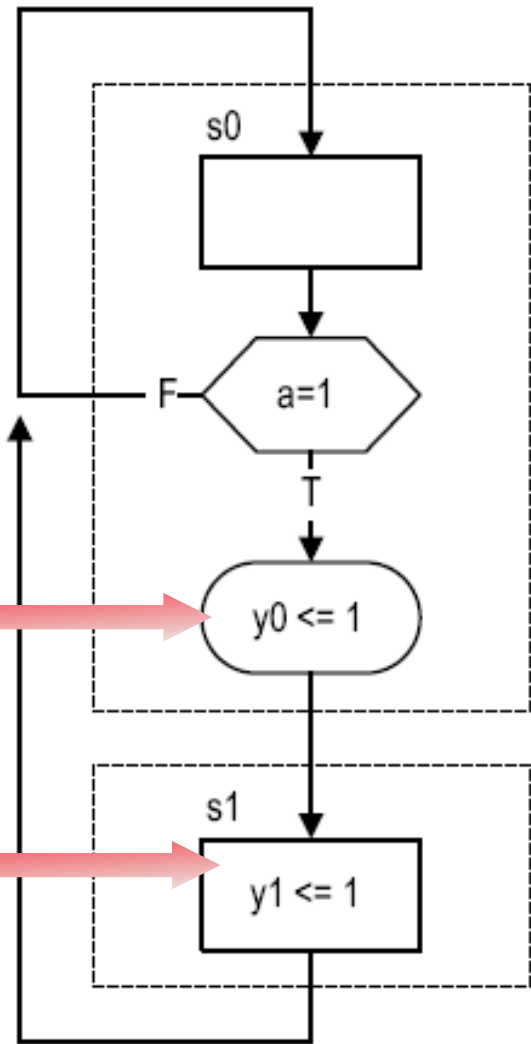




# State Diagram and ASM Chart: Example 2



Moore and Mealy



# ASMD

## ASMD:

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

**S0:**

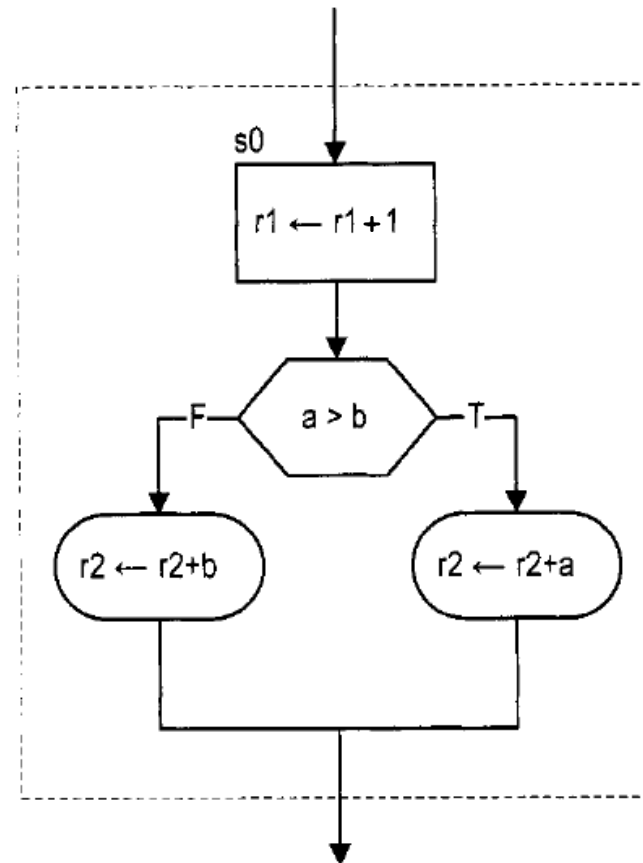
$r1 \leftarrow r1 + 1$

if  $a > b$

$r2 \leftarrow r2 + a$

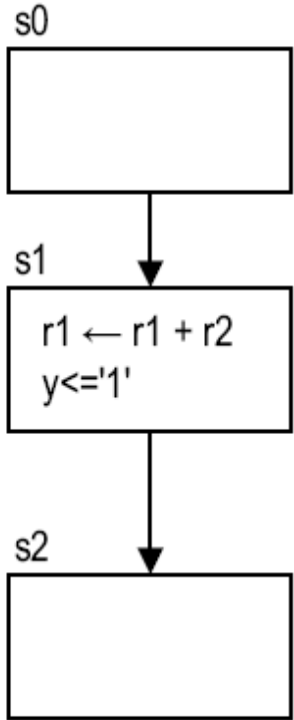
else

$r2 \leftarrow r2 + b$

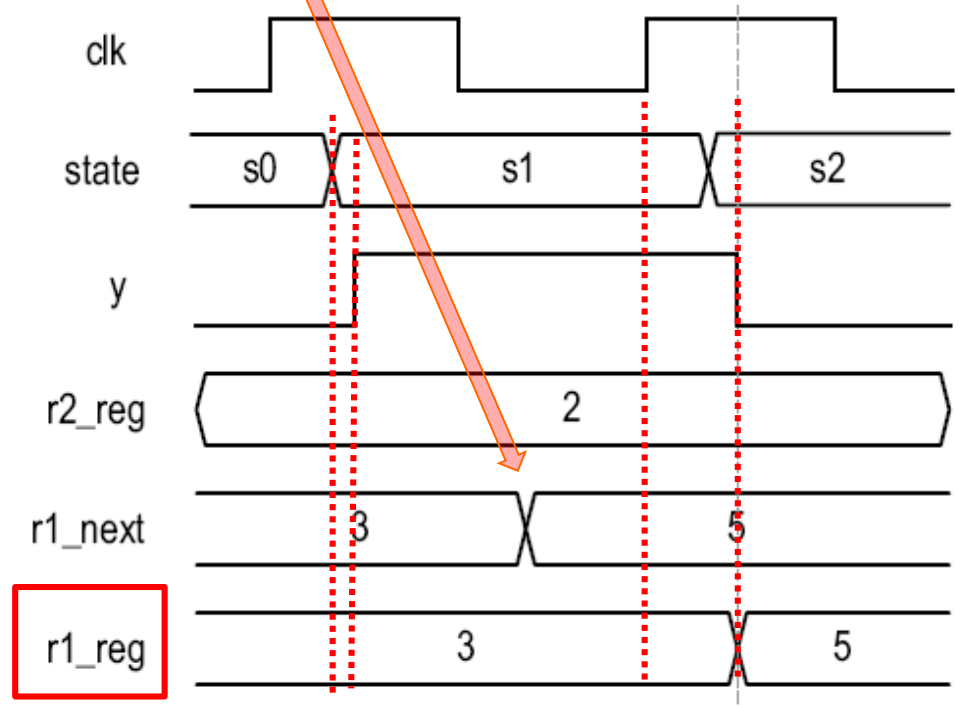


# ASMD: Timing

Value is available at the register input before the next clock tick



(a)



(c) Timing diagram

**Suggestion: use meaningful names for your signals (direction\_function\_type)**



# Outline

- Overview of FSMD
- Algorithmic state machine with data-path (ASMD)
- **FSMD design of a repetitive-addition multiplier**
- Timing analysis of FSMD



# Map Algorithm to FSMD

## Example: Repetitive addition multiplier

□ 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)
```

**Watch Out!**  
**No loop in ASMD**  
**Un-synthesizable**

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

**Pseudo code**

**ASMD-friendly code**



# ASMD Chart

## Input:

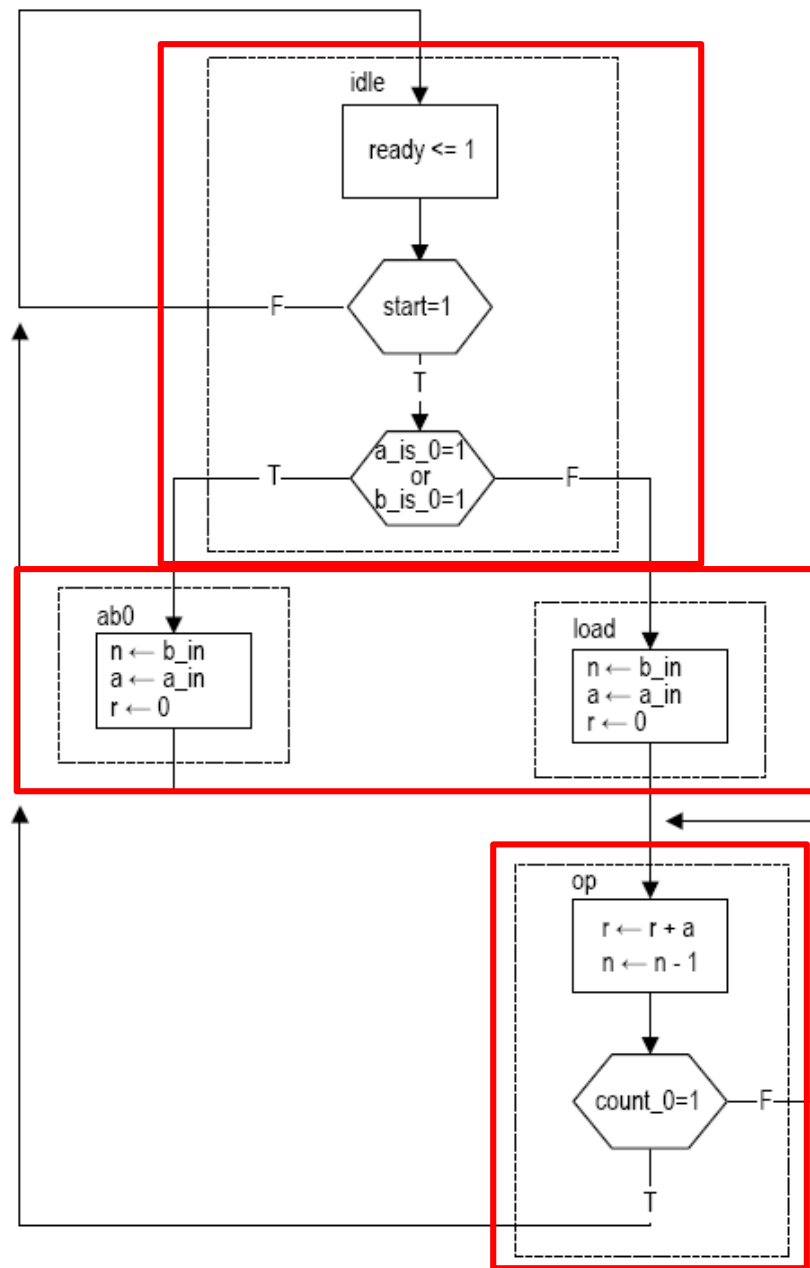
- a\_in, b\_in: 8-bit unsigned
- **clk, reset**
- start: command

## 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



# Construction of FSMD

## □ Construction of the **data path**

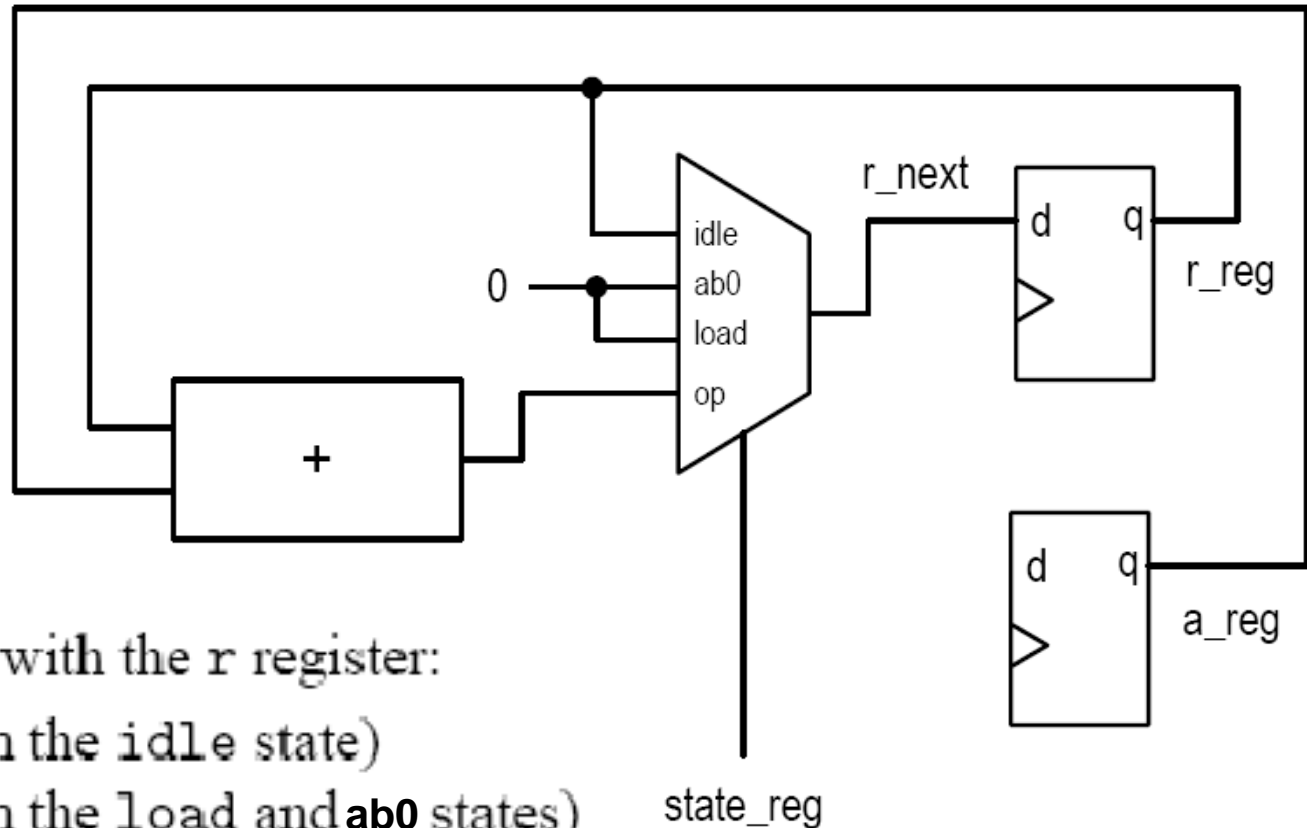
- List all possible RT operations
  - Group RT operation according to the **destination register**
  - Add combinational circuit/mux
- RT operations with the **r** register:
    - $r \leftarrow r$  (in the **idle** state)
    - $r \leftarrow 0$  (in the **load** and **ab0** states)
    - $r \leftarrow r + \mathbf{a}$  (in the **op** state)
  - RT operations with the **n** register:
    - $n \leftarrow n$  (in the **idle** state)
    - $n \leftarrow \mathbf{b\_in}$  (in the **load** and **ab0** states)
    - $n \leftarrow n - 1$  (in the **op** state)
  - RT operations with the **a** register:
    - $\mathbf{a} \leftarrow \mathbf{a}$  (in the **idle** and **op** states)
    - $\mathbf{a} \leftarrow \mathbf{a\_in}$  (in the **load** and **ab0** states)

## **Grouping RT Operations**



# Construction of the Date Path

Circuit associated with **r register**



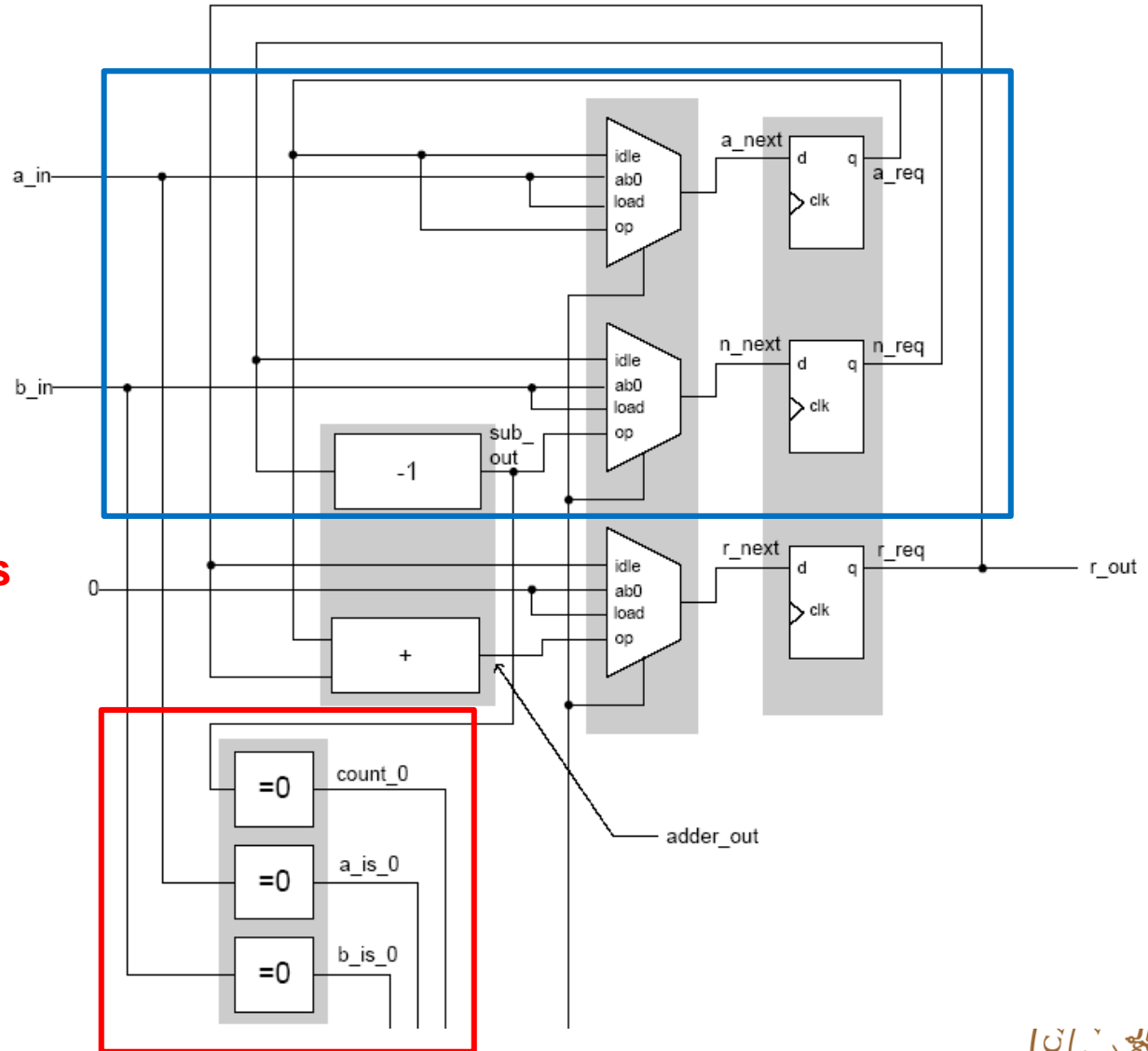
- RT operations with the  $r$  register:
  - $r \leftarrow r$  (in the `idle` state)
  - $r \leftarrow 0$  (in the `load` and `ab0` states)
  - $r \leftarrow r + a$  (in the `op` state)





# Construction of the Date Path

Continue with  
*n*-register  
*a*-register  
Add status circuits

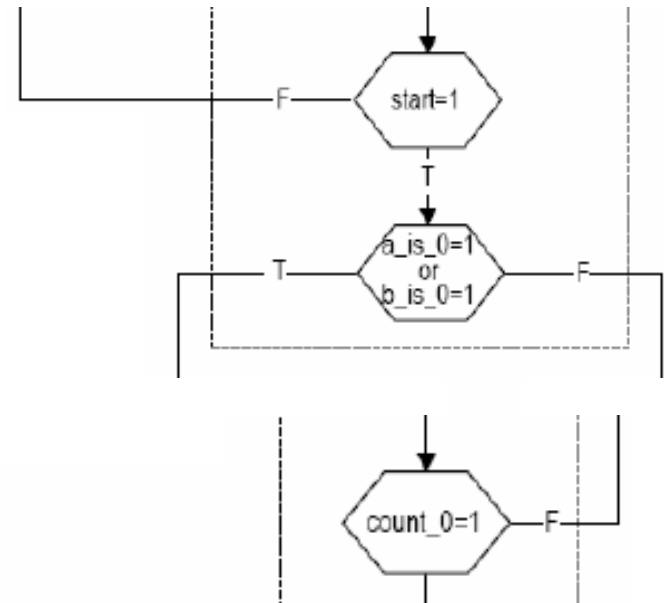
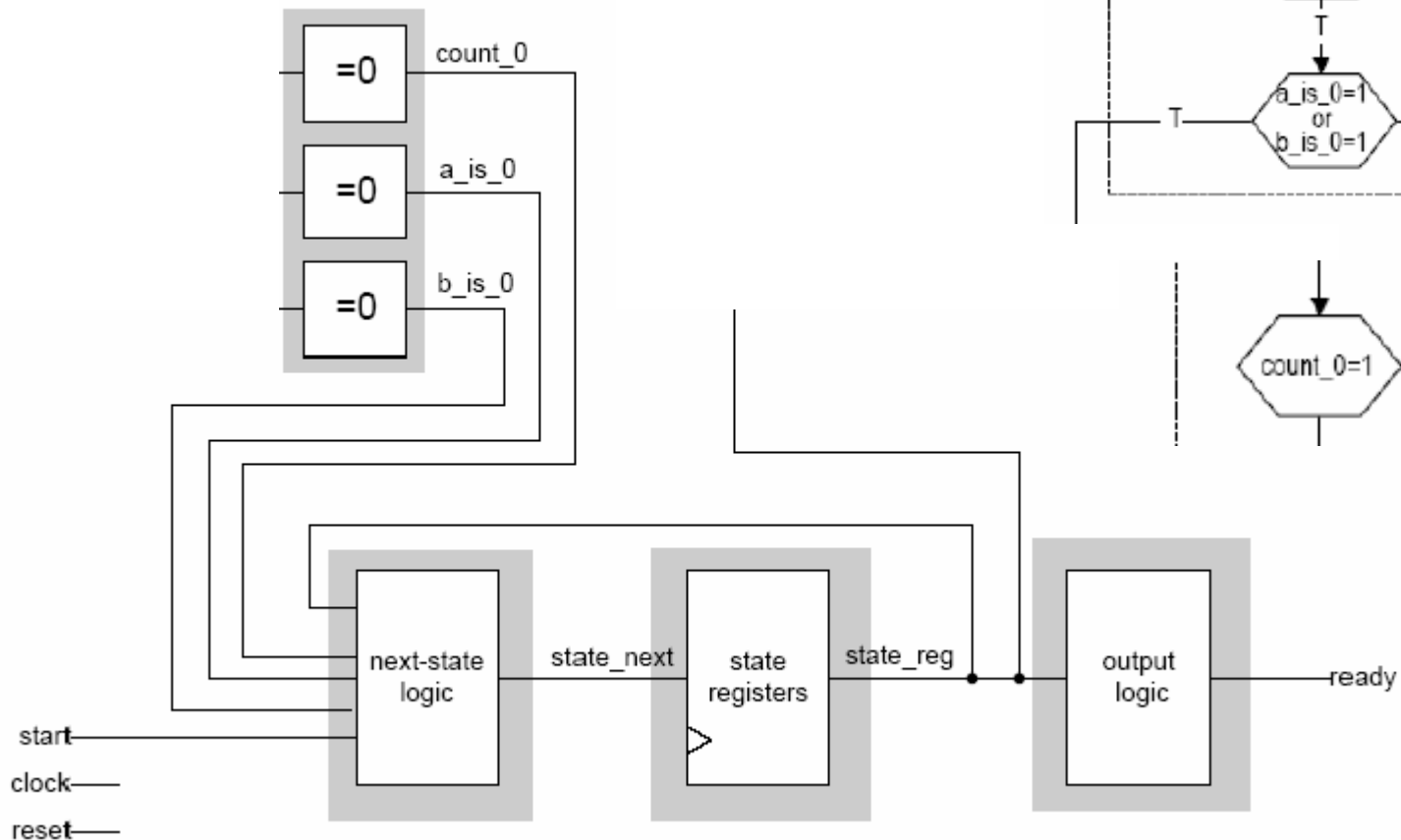


# Construction of the Control Path

## Input of FSM

- External: start, clock, reset
- Internal: decision box in ASMD

## Output of FSM



# VHDL Follow the Block Diagram

## □ Entity

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

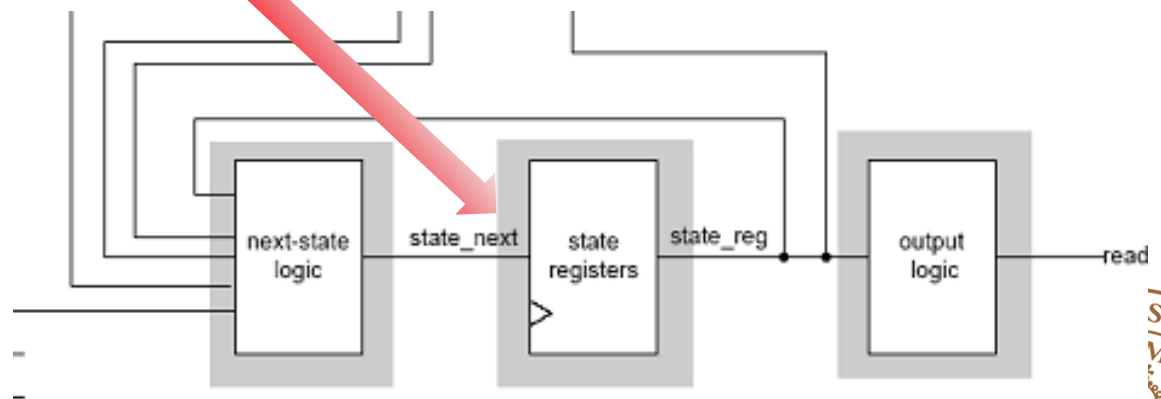
entity seq_mult is
  port (
    clk, reset: in std_logic;
    start: in std_logic;
    a_in, b_in: in std_logic_vector(7 downto 0);
    ready: out std_logic;
    r: out std_logic_vector(15 downto 0)
  );
end seq_mult;
```



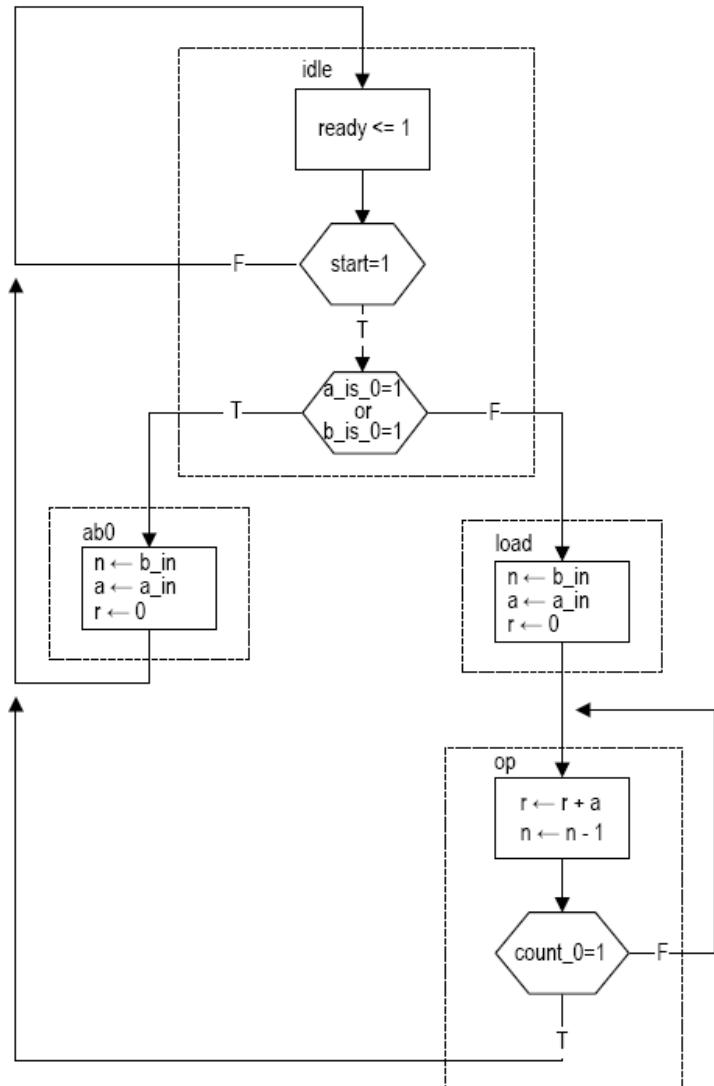
# FSM (state registers)

— *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') then
          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';
  
```

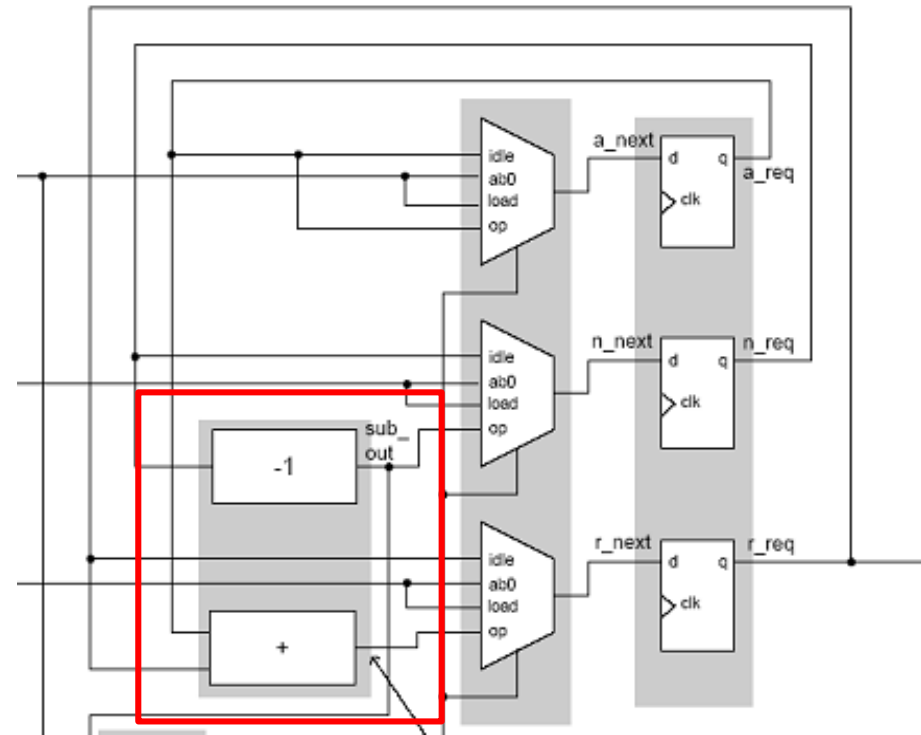
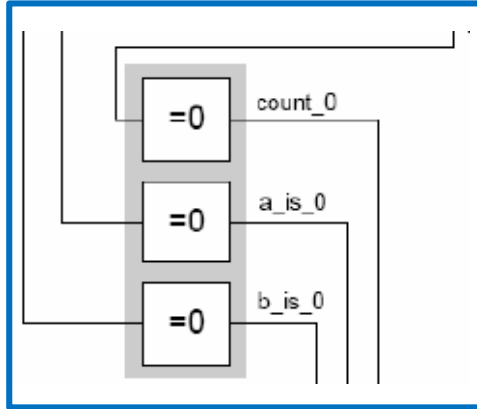


# Data Path (Data Registers)

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

```
-- 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 (Multiplexer 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 =>
```

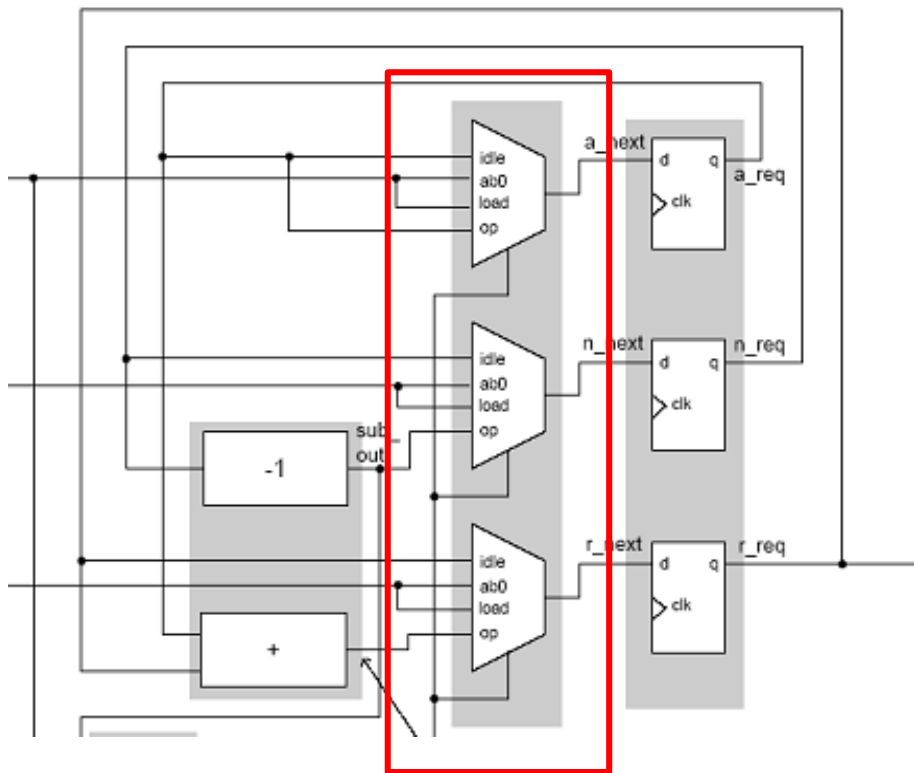
```
      a_next <= a_reg;
```

```
      n_next <= sub_out;
```

```
      r_next <= adder_out;
```

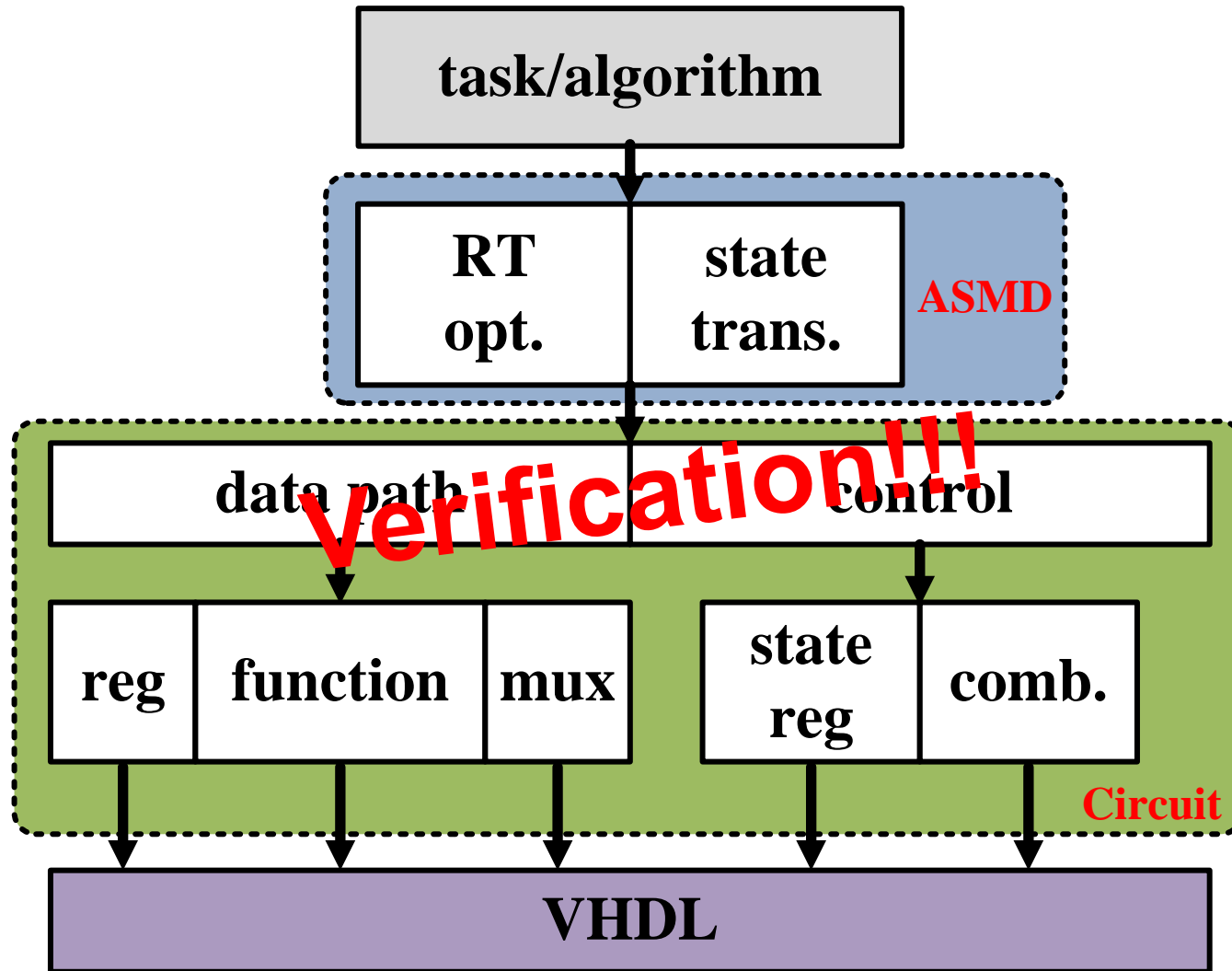
```
  end case;
```

```
end process;
```





# Design Flow



# Outline

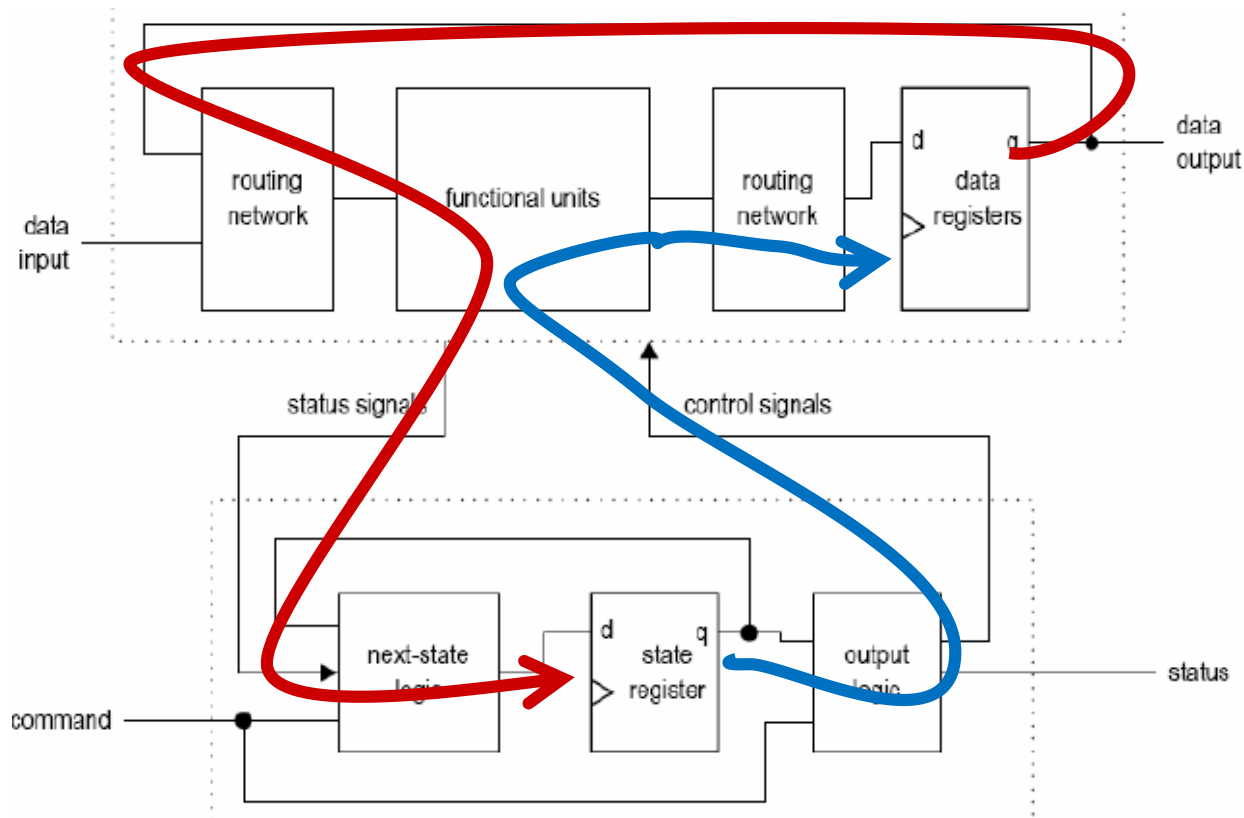
- Overview of FSMD
- Algorithmic state machine with data-path (ASMD)
- FSMD design of a repetitive-addition multiplier
- **Timing analysis of FSMD**



# Timing and Performance of FSMD

## □ Maximal clock rate

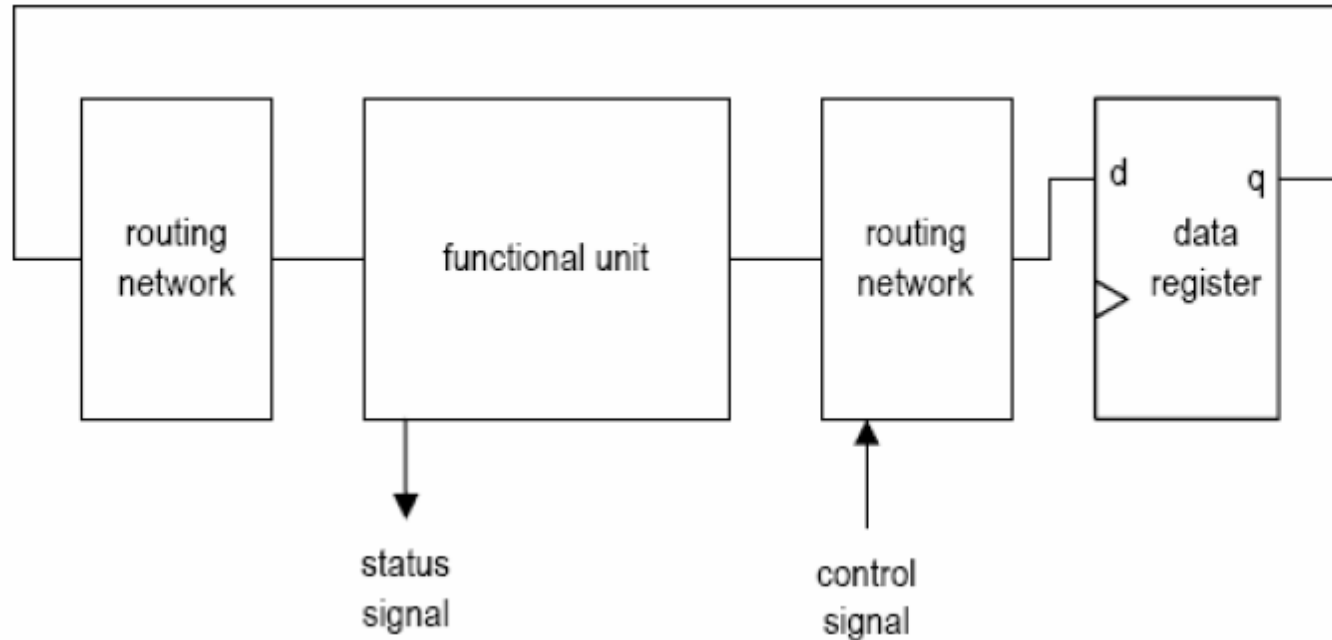
- More difficult to analyze because of *two interactive loops*, depend on the specified design
- The boundary of the clock rate can be found, *best/worse-case*



# Clock-Rate Boundary: Best-Case

## □ Best-case scenario:

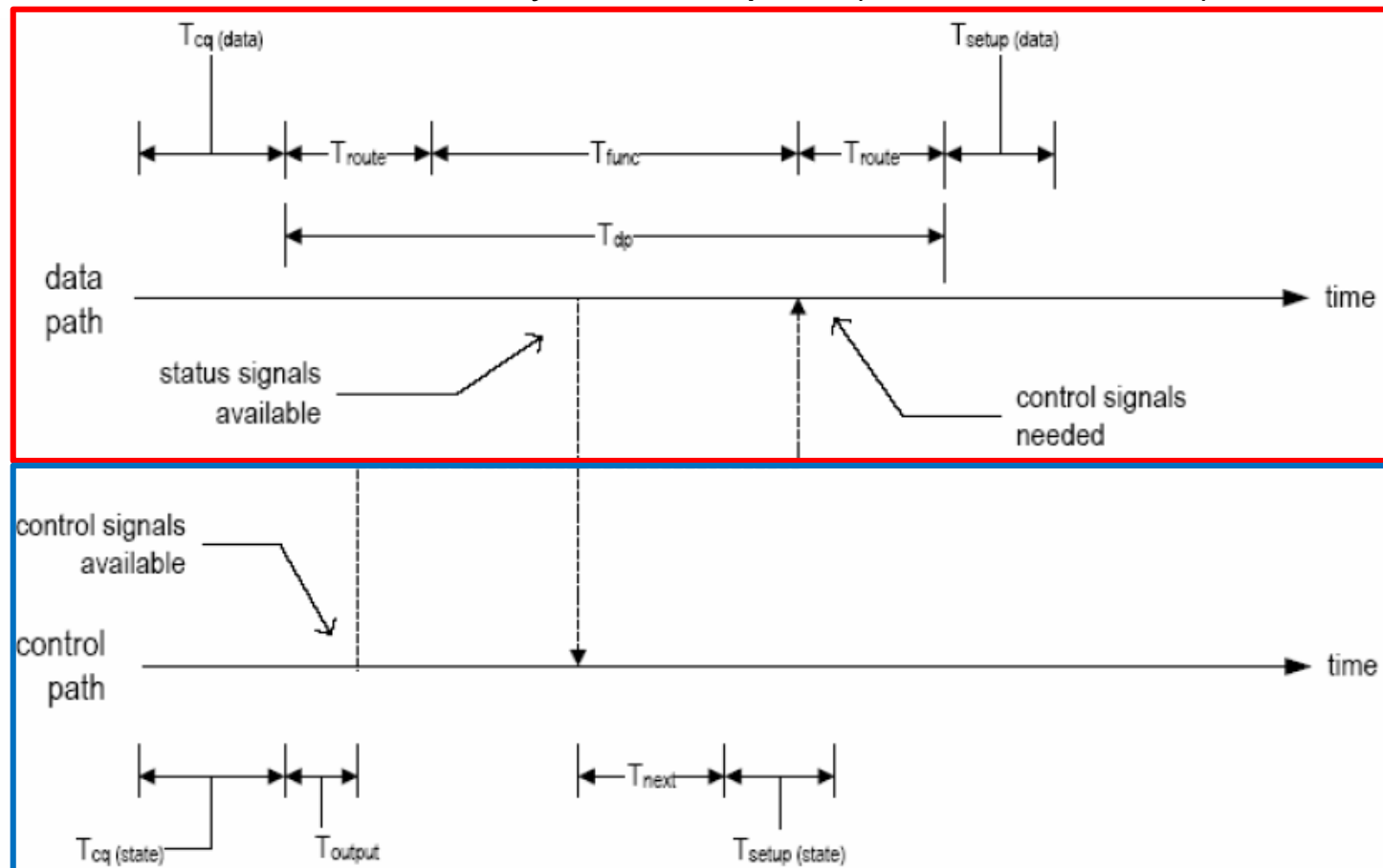
- Control signals needed at late stage
- Status signal available at early stage



# Clock-Rate Boundary: Best-Case

## □ Best-case scenario

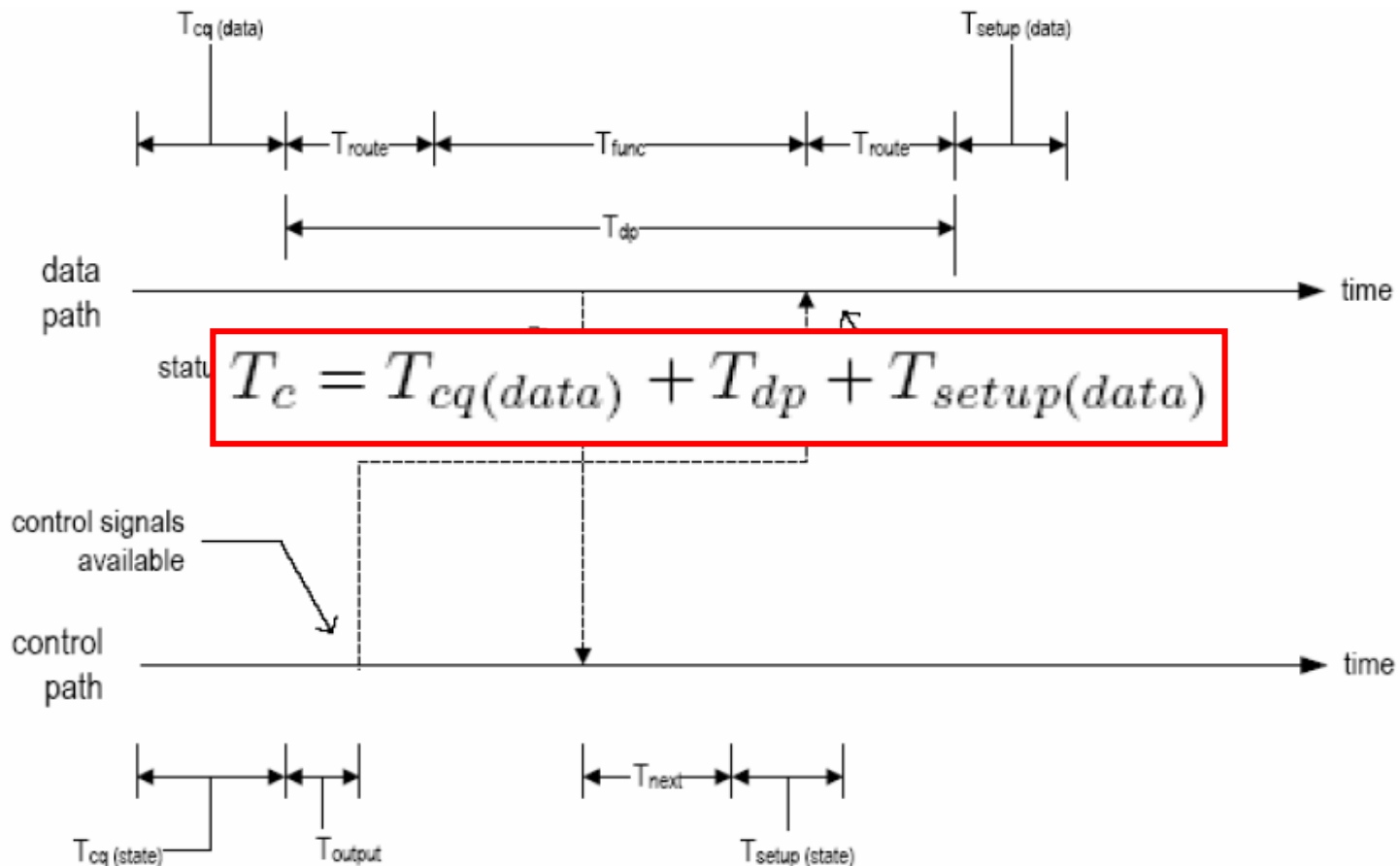
- Output logic overlaps with the data path, no extra delay
- Next-state logic of the control path and the data path are in parallel
- Clock rate dominates by the data-path (most of the case)



# Clock-Rate Boundary: Best-Case

## □ Best-case scenario

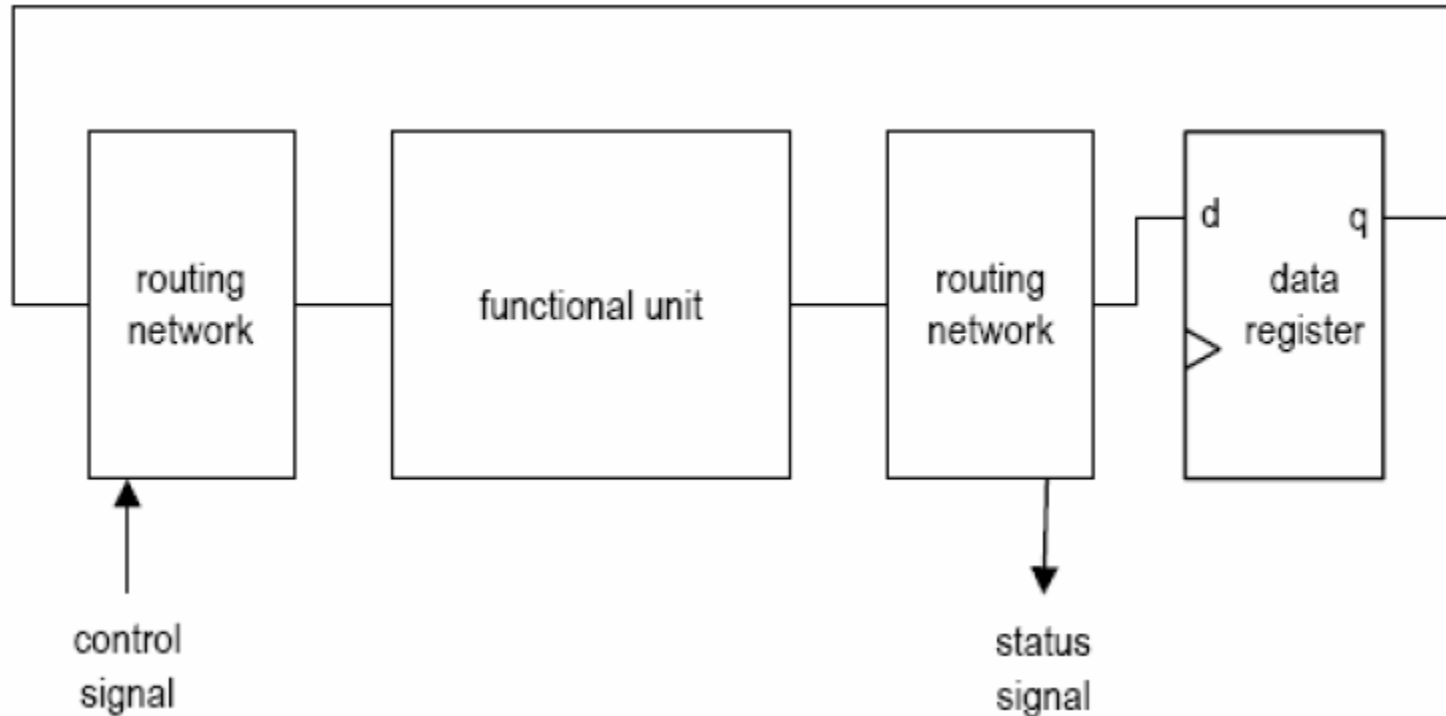
- Output logic overlaps with the data path, no extra delay
- Next-state logic of the control path and the data path are in parallel
- Clock rate dominates by the data-path (most of the case)



# Clock-Rate Boundary: Worst-Case

## □ Worst-case scenario:

- Control signals needed at early stage
- Status signal available at late stage



# Thanks!

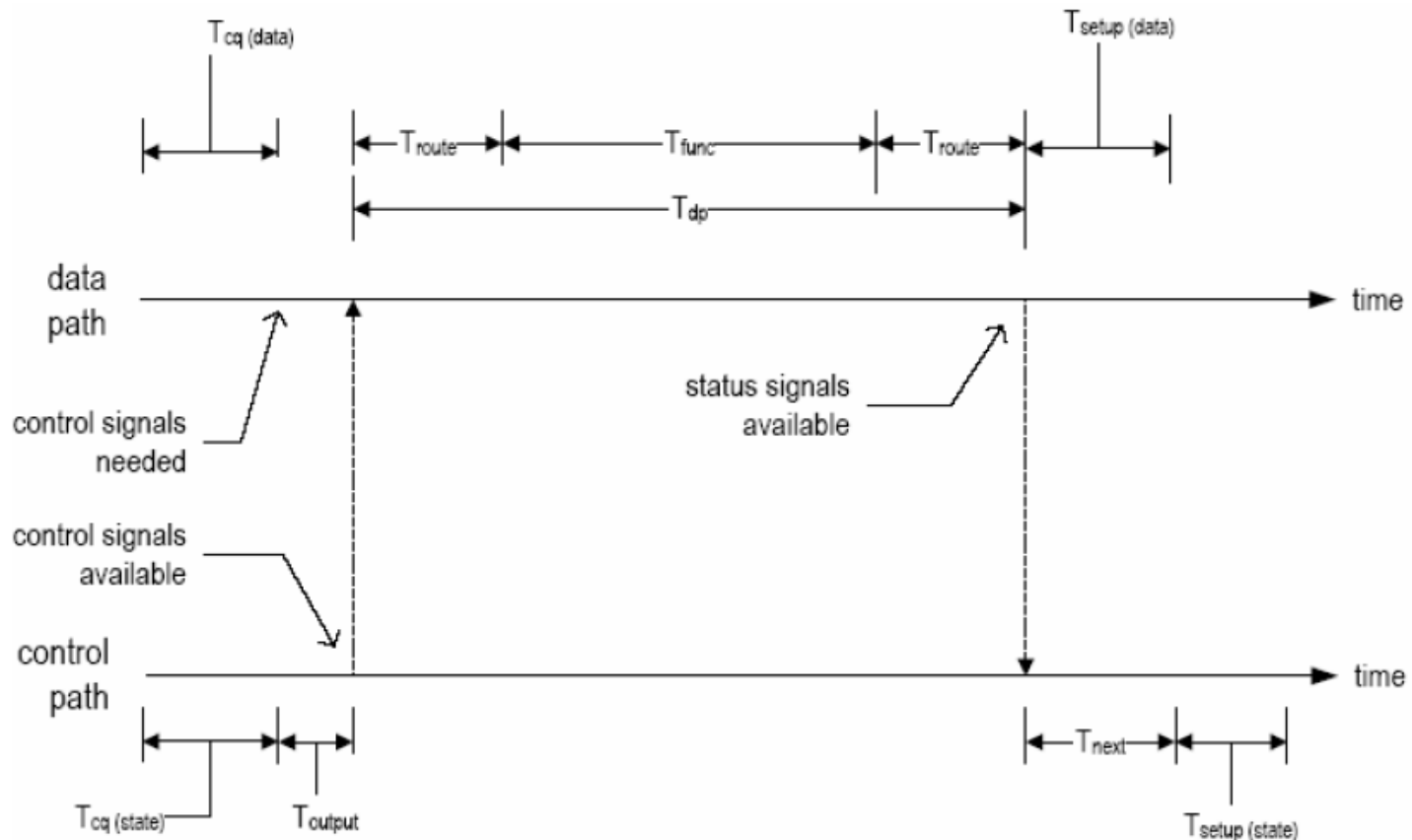




# Clock-Rate Boundary: Worst-Case

## Worst-case scenario

- The data path must wait for the FSM to generate the output signals
- The control path must wait for status signals to generate the next-state
- Clock period includes the delays of all combinational components



# Clock-Rate Boundary: Worst-Case

## Worst-case scenario

- The data path must wait for the FSM to generate the output signals
- The control path must wait for status signals to generate the next-state
- Clock period includes the delays of all combinational components

