Lesson 4: Algorithmic design

Algorithmic State Machine (ASM)
Control unit and datapath

toc

Outline

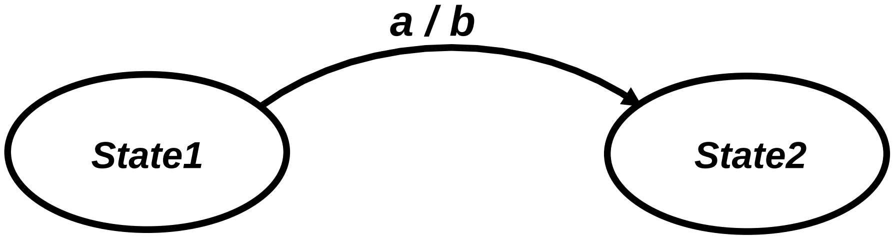
1. Introduction
   - FSMs: Quick review
   - Memory modules: Quick review
2. Algorithmic design:
   - Algorithmic state machine (ASM) diagram, control unit and datapath
3. Algorithmic design in VHDL

Finite-state machine (FSMs)

- **Mealy:** The output depends on the current state AND on the value of the FSM input(s).

  O(t) = F (X(t), S(t))

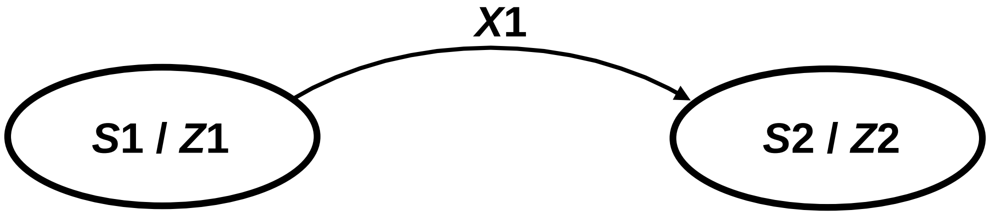
  S(t+1) = G (X(t), S(t))



- **Moore:** The output only depends on the current state.

  Z(t) = F (S(t))

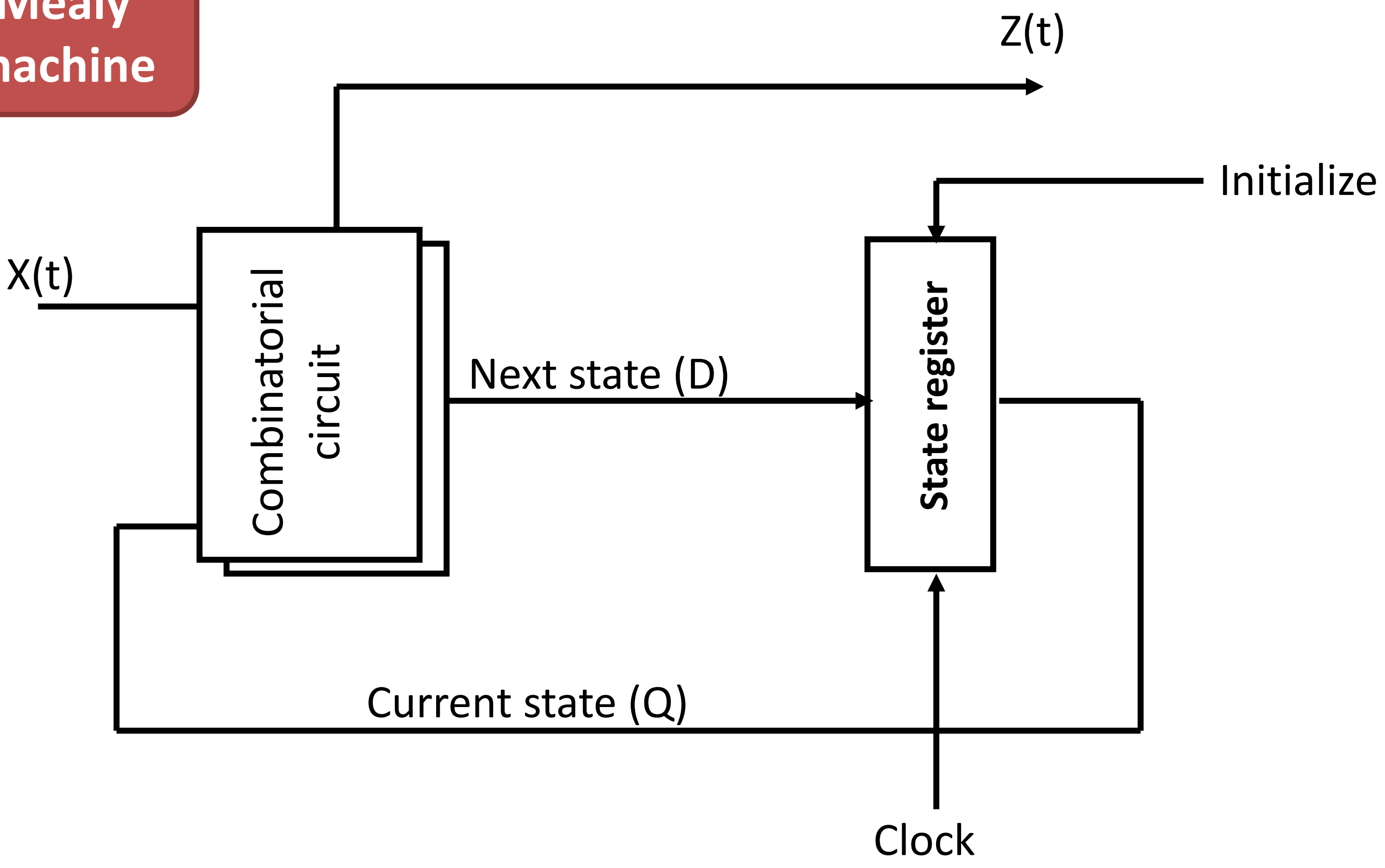
  S(t+1) = G (X(t), S(t))



toc

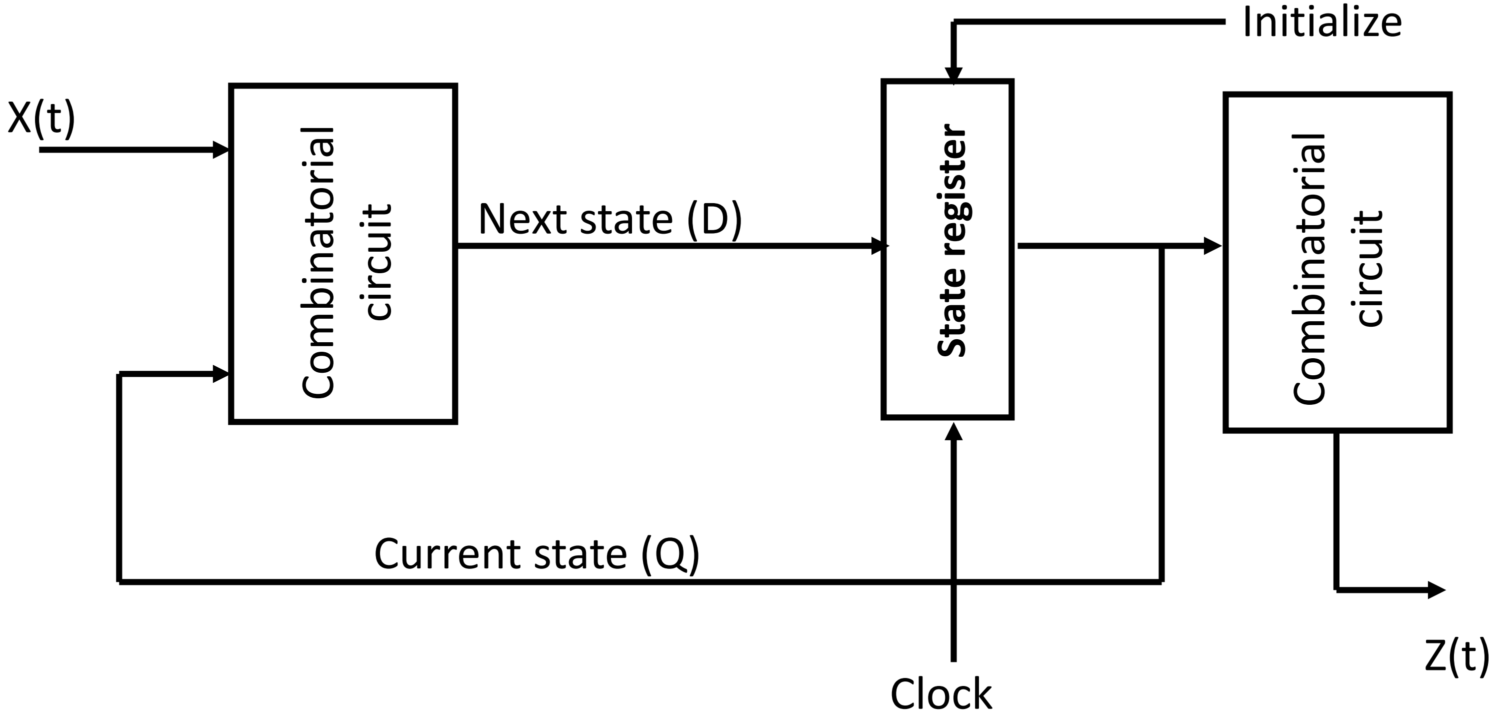
FSMs: Implementation

Mealy machine



FSMs: Implementation

Moore machine



Synchronous operation

- The hypothesis of synchronous operation of sequential systems assumes that:
  - The state only changes once per clock cycle and that transition is simultaneous in all the bits of the state register.
  - After a state transition, the inputs of the state register have enough time to reach a stable state before the following state transition.

Memory modules

- In sequential systems design, **one of the most important hardware modules are memory modules**:
  - They keep track of the current state of the machine (control machine).
  - They store intermediate information (data registers).

- The simplest memory modules are **flip-flops**.
  - Flip-flops feature two stable states:
    - Output 0
    - Output 1
  - They can be used to store 1 bit of information.

toc

Kinds of flip-flops

- According to their logical operation:
  - S-R
  - D
  - J-K
  - T

S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	forbidden

D	Q+
0	0
1	1

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

T	Q+
0	Q
1	$\overline{Q}$

- According to their temporal operation:
  - Latch
  - Synchronous latch (level-triggered)
  - Edge-triggered flip-flop
  - Master-slave flip-flop

Characteristic equations

R-S: $Q+ = S + \overline{R}\,Q$

D: $Q+ = D$

J-K: $Q+ = J\,\overline{Q} + \overline{K}\,Q$

T: $Q+ = T\,\overline{Q} + \overline{T}\,Q$

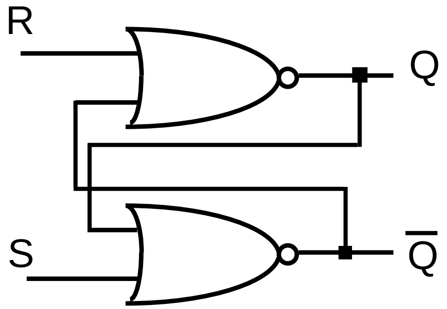
Deduced from the Karnaugh maps
For Q(t+1) = Q+ = ƒ(Inputs, Q)

toc

Asynchronous flip-flop: Latch

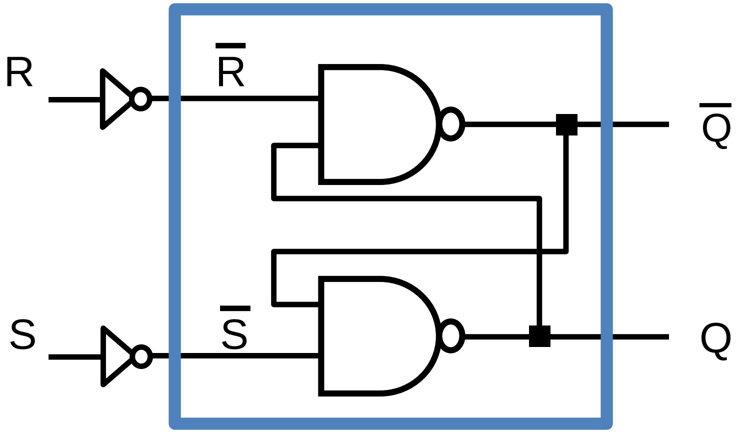
- The output changes with the inputs change
- Example: S-R latch direct logic

S	R	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	forbidden



- Inverted logic (smaller transistors): **Inverted S-R Latch**

S	R	$\overline{S}$	$\overline{R}$	Q(t+1)
0	0	1	1	Q(t)
0	1	1	0	0
1	0	0	1	1
1	1	0	0	forbidden

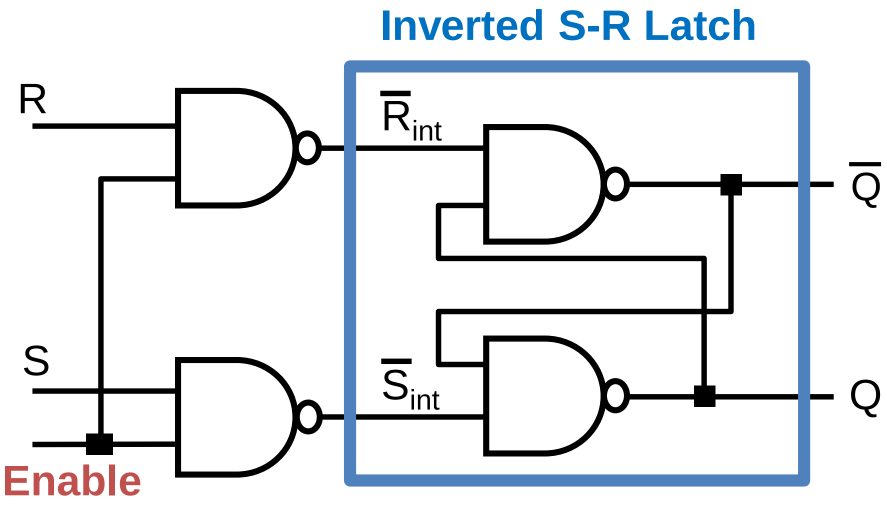


toc

Synchronous level-triggered cell: Synchronous latch

- The output changes when the *enable* input is active.
- Example: S-R with an *enable* input.
  - If Enable = 0 then $\overline{S}_{int}$ = $\overline{R}_{int}$ = 1 and therefore Q(t+1) = Q(t)
  - If Enable = 1 then $\overline{R}_{int}$ = not R and $\overline{S}_{int}$ = not S

$\overline{S}_{int}$	$\overline{R}_{int}$	Q(t+1)
1	1	Q(t)
1	0	0
0	1	1
0	0	forbidden

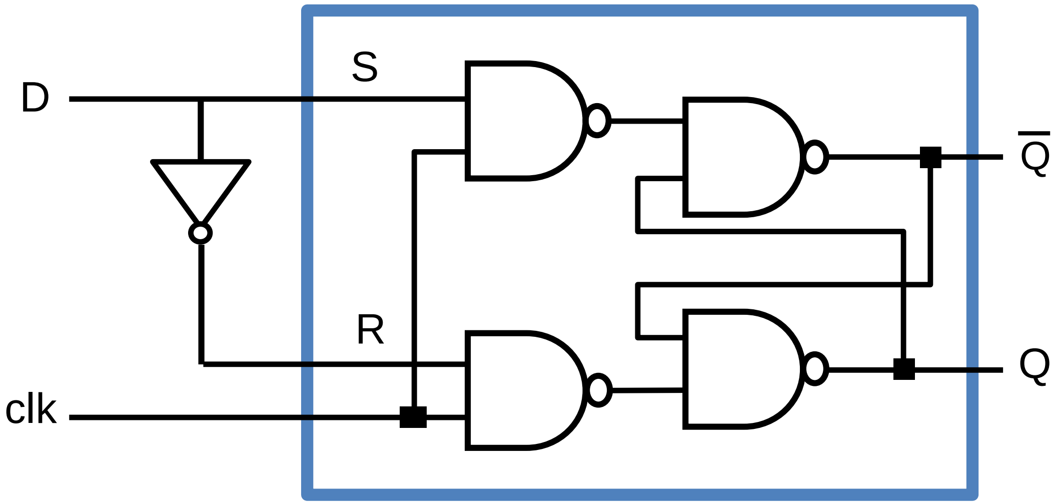


E	S	R	Q(t+1)
0	x	x	Q(t)
1	0	0	Q(t)
1	0	1	0
1	1	0	1
1	1	1	forbidden

Synchronous level-triggered cell: Synchronous latch

- How can we make a D flip-flop out of this?
  - This latch is sensitive to the level of the clock clk.
    In other words, it samples the input ONLY when clk = 1.

Level triggered R-S Latch



clk	D	Q(t+1)
0	0	Q(t)
0	1	Q(t)
1	0	0
1	1	1

Problems of latches

- If we use latches, we have to make sure that:
  - The clock pulse duration is shorter than the latch delay (to prevent the output from oscillating indefinitely in some situations).
  - The inputs are constant during the clock pulse.
  - Rule: **DO NOT USE LATCHES**
- An alternative is to use flip-flops: more reliable.
    - Edge-triggered: the output only changes when a rising or falling clock edge occurs.
    - Master-slave.

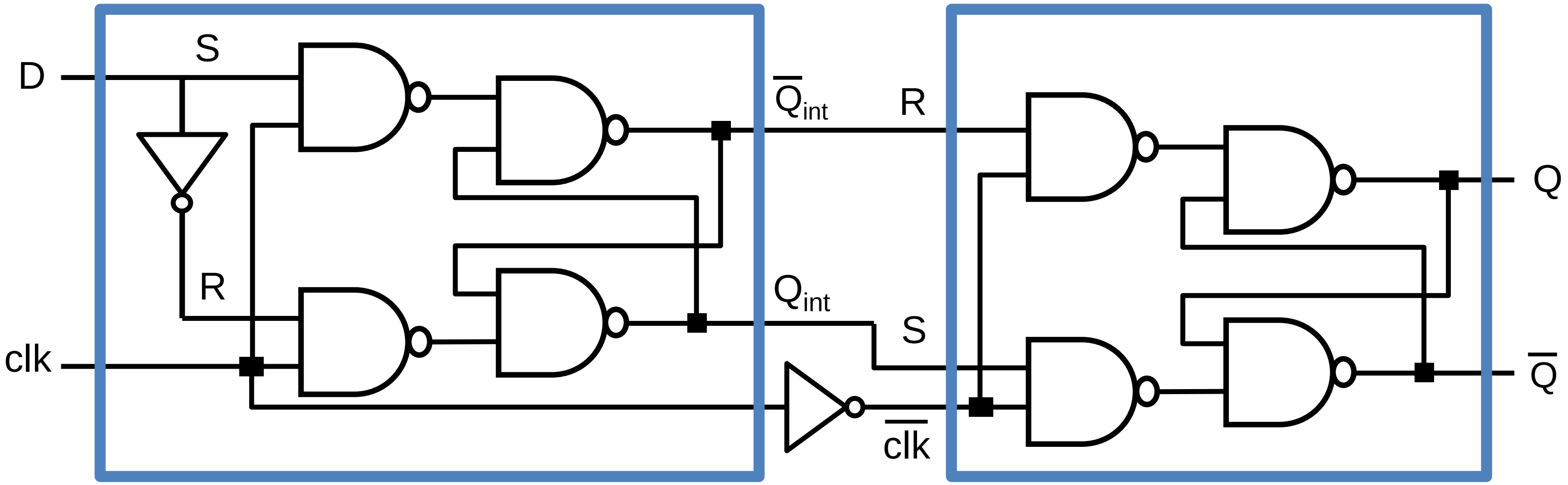
toc

Master-slave D flip-flop

- When the clock is high, *D* is stored in the first latch, but the second latch cannot change state. When *clk* is low, the first latch's output is stored in the second latch, but the first latch cannot change state.
- The result is that *D* is sampled at the falling edge of *clk*.

Level triggered D Latch

Level triggered R-S Latch



Master

If clk=1 then Q_{int}=D

Slave

If clk=0 then Q=S

Temporal features of different memory modules

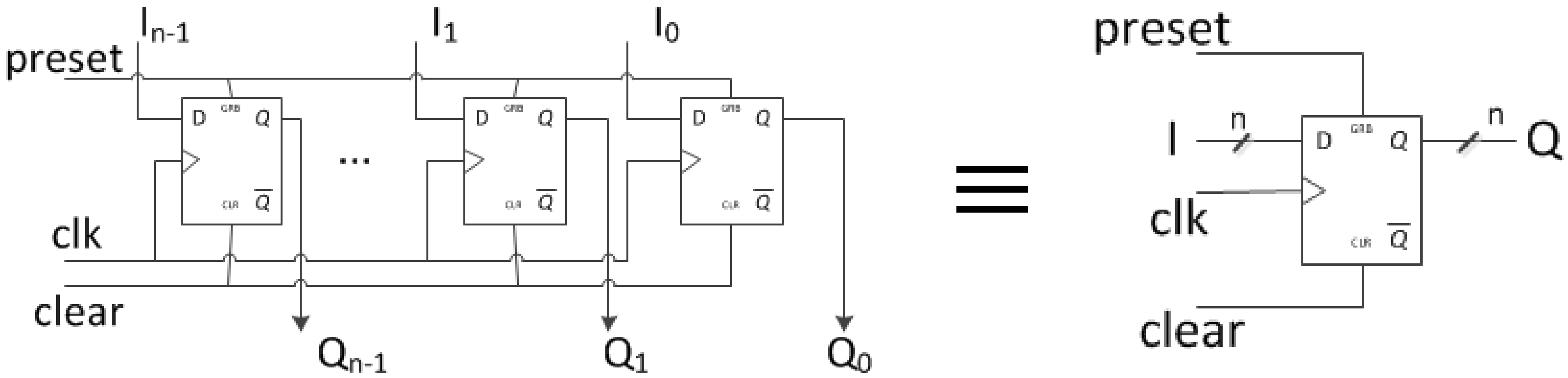
TYPE	When are the inputs sampled?	When are the outputs valid?
Latch without clk	always	There is a propagation delay from the change in the input
Level-triggered latch	When the clock is set to high (T_{setup} and T_{hold} shortly before and after the falling edge)	There is a propagation delay from the change in the input
Rising-edge-triggered flipflop	Clock transition from low to high (T_{setup} and T_{hold} shortly before and after the rising edge)	There is a propagation delay from the clock rising edge
Falling-edge-triggered flipflop (previous slide)	Clock transition from high to low (T_{setup} and T_{hold} shortly before and after the falling edge)	There is a propagation delay from the clock falling edge

FFs in VHDL

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

entity ff is
    port( clk  : in   std_logic;
          s, r : in   std_logic;
          d    : in   std_logic;
          q    : out std_logic );
end ff;
architecture rtl of ff is
begin
   p_ff: process(clk, r, s, d)
    begin
        if rising_edge(clk) then
           if      r = '1' then  q <= '0';
           elsif   s = '1' then  q <= '1';
           else                  q <= d;
           end if;
        end if;
    end process p_ff;
end rtl;
```

Registers

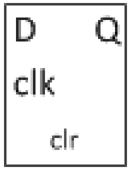
- n-bit flip-flop



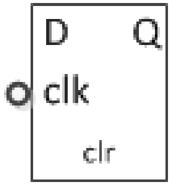
toc

Types of registers

- Based on timing
  - Level-triggered

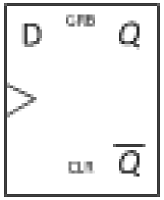


high

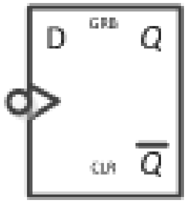


low

  - Edge-triggered



rising

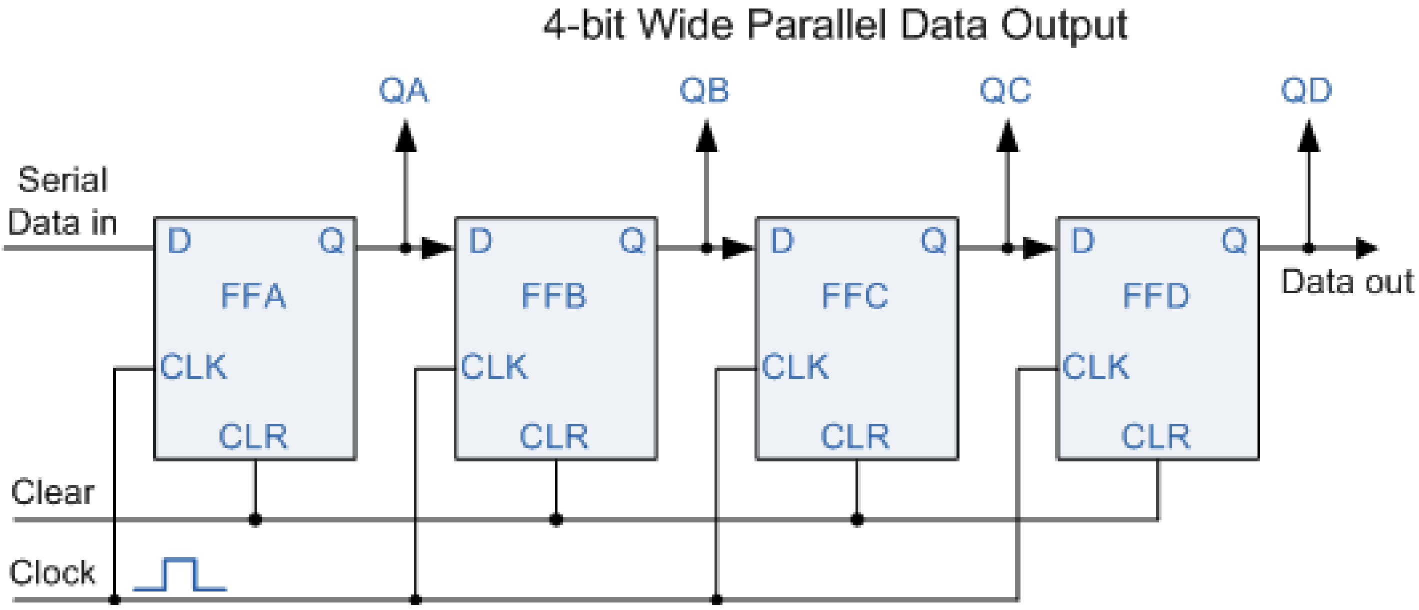


falling

- Based on functionality
  - Parallel input/Parallel output-PIPO
  - Serial input/Parallel output-SIPO
  - Parallel input/Serial output-PISO
  - Serial input/Serial output-SISO

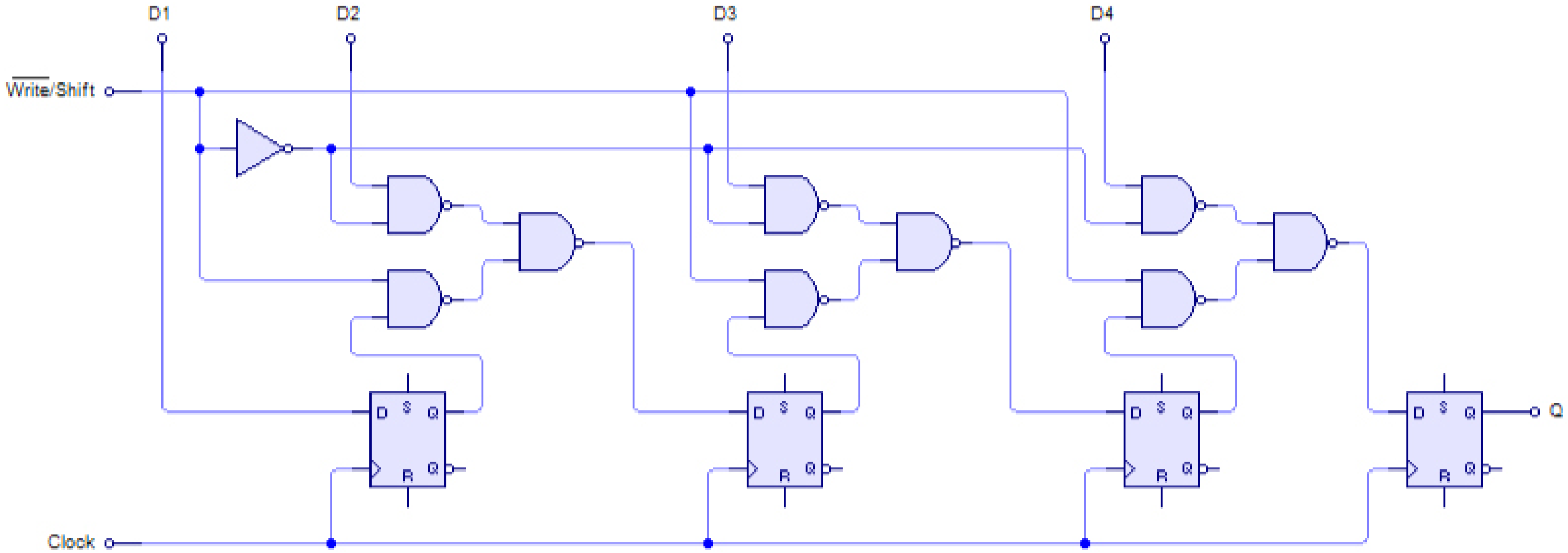
Registers

- Types:
  - PIPO (*Parallel-Input Parallel-Output*). Previously explained
  - SIPO (*Serial-Input Parallel-Output*)



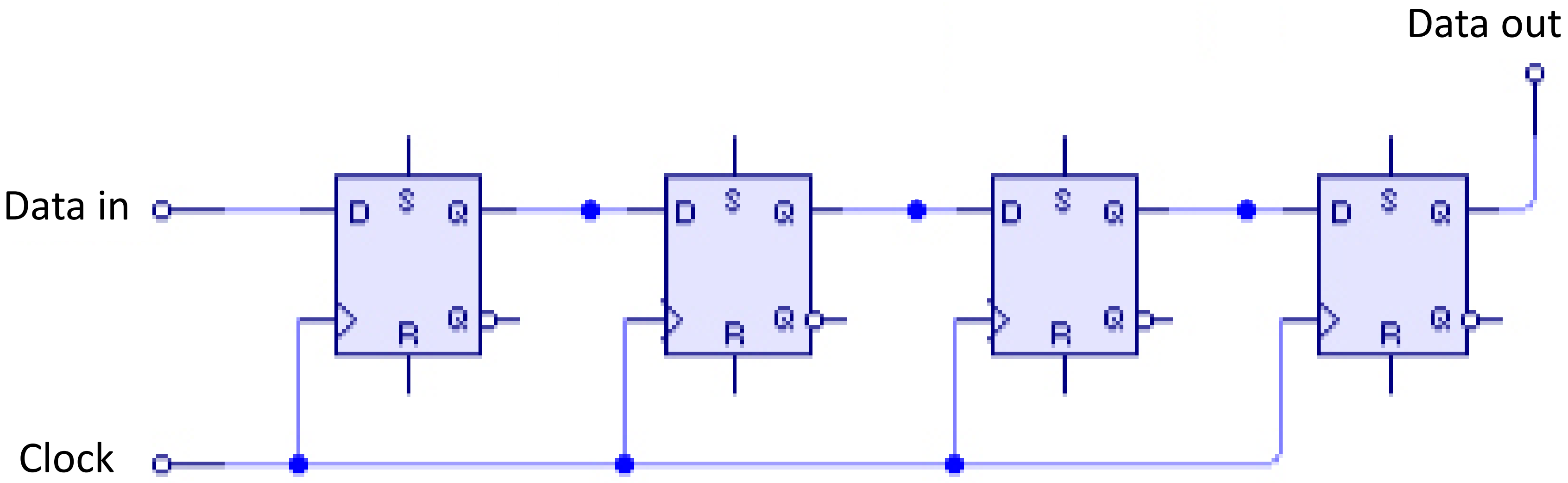
Registers

– PISO (*Parallel-Input Serial-Output*)



Registers

- SISO (*Serial-Input Serial-Output*)



Registers in VHDL

```vhdl
entity register_n is
    generic (n: natural := 8);
    port( clk:  in  std_logic;
          rst:  in  std_logic;
          load: in  std_logic;
          din:  in  std_logic_vector (n-1 downto 0);
          dout: out std_logic_vector (n-1 downto 0) );
end register_n;
```

```vhdl
architecture ARCH1 of register_n is
begin
  process(clk)
  begin
    if rising_edge(clk) then
        if rst=‘1’ then
          dout <= (others => ‘0’);
        elsif load = ‘1’ then
          dout <= din;
        end if;
    end if;
  end process;
end ARCH1;
```

PIPO register with synchronous reset

```vhdl
architecture ARCH2 of register_n is
begin
  process(rst, clk)
  begin
    if rst = ‘1’ then
      dout <= (others => ‘0’);
    elsif rising_edge(clk) then
      if load = ‘1’ then
        dout <= din;
      end if;
    end if;
  end process;
end ARCH2;
```

PIPO register with asynchronous reset

toc

Registers in VHDL

- Serial input/Parallel output - SIPO?
- Parallel input/Serial output - PISO?
- Serial input/Serial output - SISO?

Counters in VHDL

- Up-down counter with parallel load and asynchronous reset

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

entity counter_n is
    generic(n: natural := 8);
    port( clk:        in  std_logic;
          rst:        in  std_logic;
          load:       in  std_logic;
          count_up:   in  std_logic;
          count_down: in  std_logic;
          din:        in  std_logic_vector(n-1 downto 0);
          dout:       out std_logic_vector(n-1 downto 0) );
end counter_n;
```

Counters in VHDL

```
architecture ARCH of counter_n is
    signal aux_output: unsigned (n-1 downto 0);

begin

    process(clk, rst)
    begin
        if rst ='1' then
            aux_output <= (others => '0');
         elsif rising_edge(clk) then
            if load = ‘1’ then
                aux_output <= unsigned(din);
            elsif count_up = ‘1’ then
                aux_output <= aux_output + 1;
            elsif count_down = ‘1’ then
                aux_output <= aux_output – 1;
            end if;
        end if;
    end process;

    dout <= std_logic_vector(aux_output);

end ARCH;
```

Outline

1. Introduction
   - FSMs: Quick review
   - Memory modules: Quick review
2. Algorithmic design:
   - Algorithmic state machine (ASM) diagram, control unit and datapath
3. Algorithmic design in VHDL

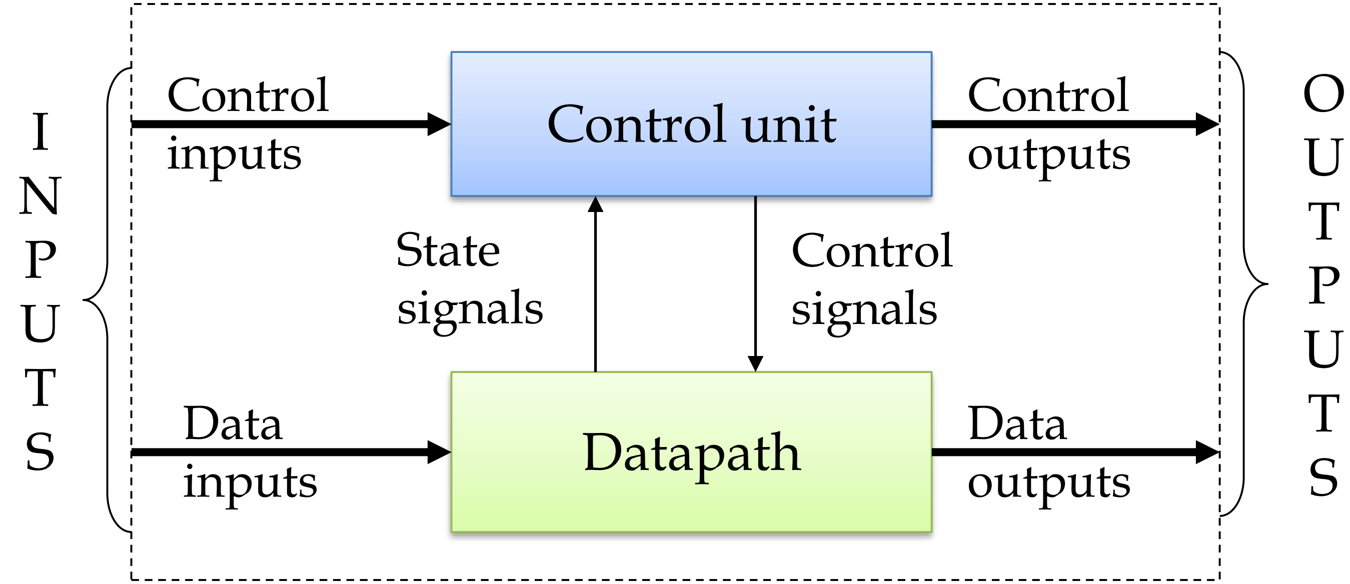
What is an algorithmic design?

- Specification and implementation approach for digital systems, which greatly allows automating their design.
- Its starting point is always an algorithmic description of its behavior:
  - How to calculate the output of a circuit from its inputs.
- Implementation:
  - Control unit and datapath.

toc

What is an algorithmic state machine?

- Synchronous sequential systems.
- The behavior is defined IMPLICITLY
  - **The value of the output(s) is not directly specified, but how it must be obtained: the algorithm.**
- Model:



Design flow: outline

1. Study of the specification:
   - Sequential steps to be done (algorithm).
   - Hardware to use (complex combinatorial and sequential modules).
2. Extraction of an ASM diagram: from the output of Step 1, create a diagram that fulfills the requirements.
3. Design of the control unit:
   - Code each one of the states of the ASM diagram.
   - Code the transition state function.
     - By means of internal control signals.
     - By means of external control signals.
   - Code the control signals that go to the datapath: each state has associated a set of values of ALL the signals that control the complex modules (for instance, *load* signal of the registers).
4. Design of the datapath:
   - Connect the modules with the external and internal data signals.
   - Connect the control signals (obtained in Step 3) to the modules of the datapath.

toc

Step 1: Specification

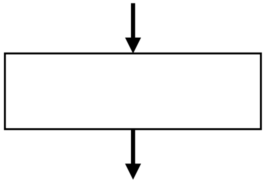
- Thorough study of the specification:
  - Do I understand the formulation of the problem?
- Sequential steps to follow (algorithm)
  - Different valid solutions (simple and efficient)
  - Does it fulfill the requirements? → Prepare a test!
- Hardware used (complex hardware modules)
  - Which HW is available and convenient?
  - Can the algorithm be implemented using the HW?
  - Does the available HW require a rethinking of the previous point?

Step 2: ASM specification

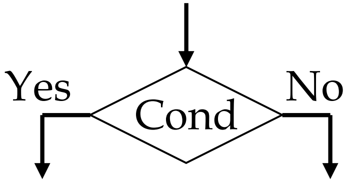
- About the extraction of an ASM diagram:
  - Number of states
  - Do I need more or less states?
  - When and how the control signals are activated:
    - **When some information must be loaded into a register**
    - **When a counter must stop counting**
    - **When a comparison is correct**
    - **…**
  - Should I design a Moore or Mealy machine, or maybe a combination of both?

Step 2: ASM specification

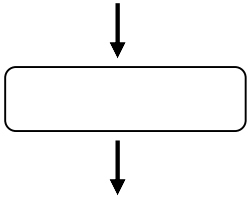
- Graphical way of representing the algorithm. Elements:
  - State box: assignments and operations that are carried out simultaneously.



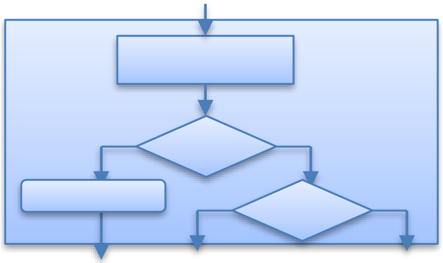
  - Decision box: Conditional fork with 2 branches.



  - Conditional output box: assignments that are made when a condition is met (Mealy).



  - ASM block: A state box with a network of decision boxes and conditional output boxes.



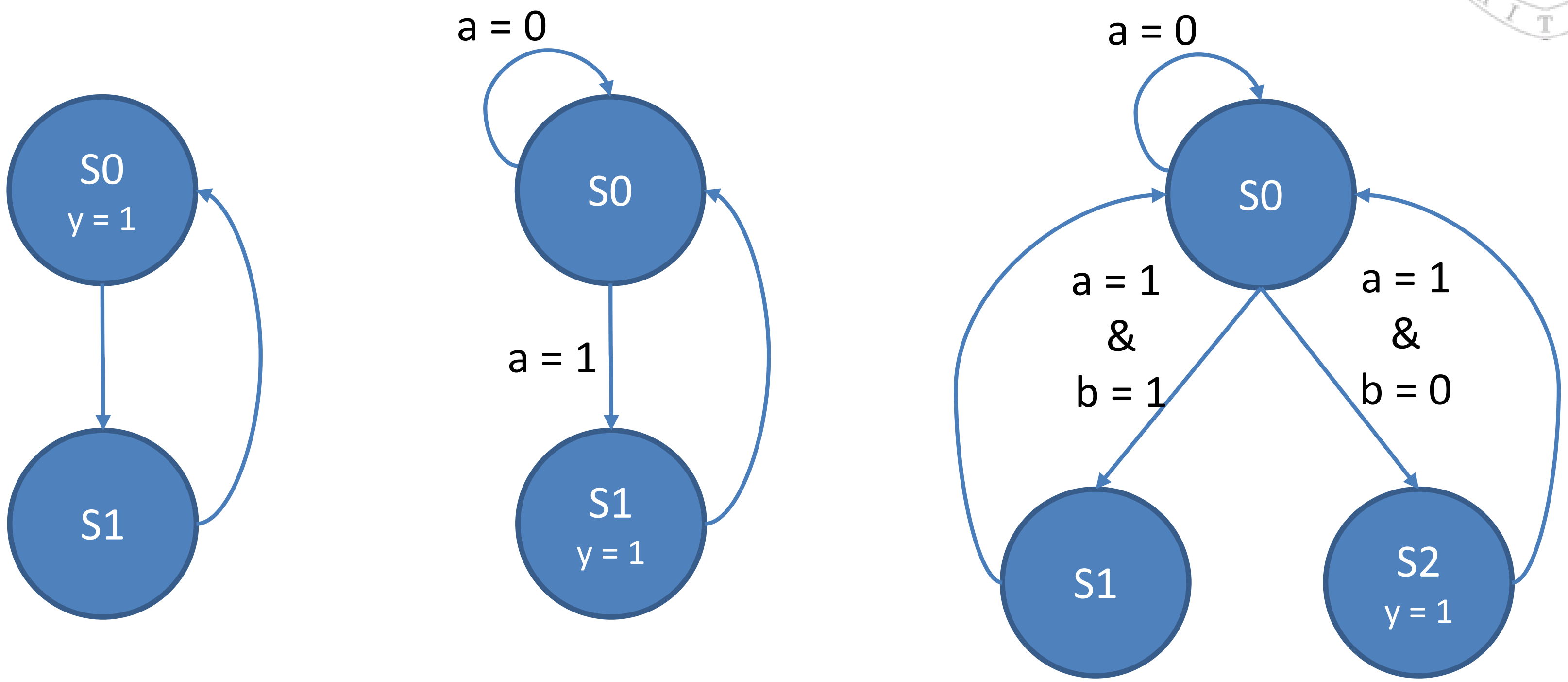
toc

Step 2: ASM specification

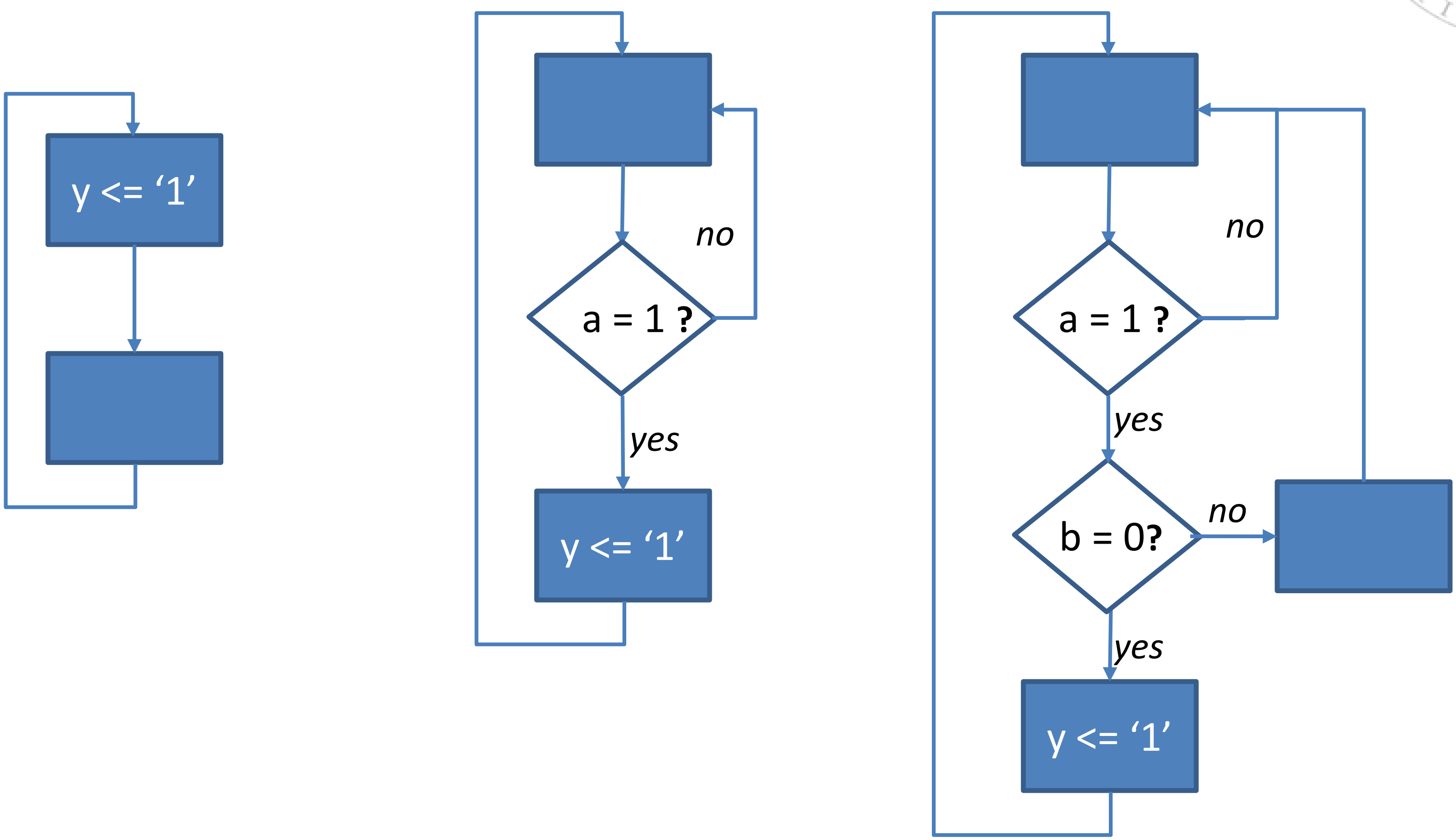
1. Each state box can have several entries but only one exit
2. A decision box can only be reached from the state box contained in its ASM block
3. All the operations of an ASM block (state box, decision boxes and conditional outputs) are concurrent
4. An ASM block can only contain one state box and any number of decision boxes (0, 1, 2, ...) and conditional outputs
5. The first element of an ASM block is always a state box. An ASM block can not contain only decision boxes.
6. Always enter an ASM block in its state box
7. All ASM blocks must be labeled

toc

Step 2: ASM specification

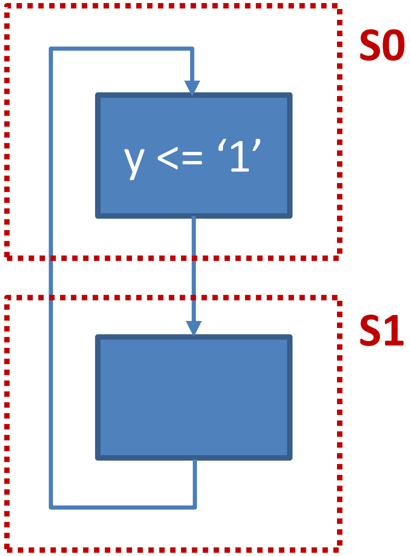


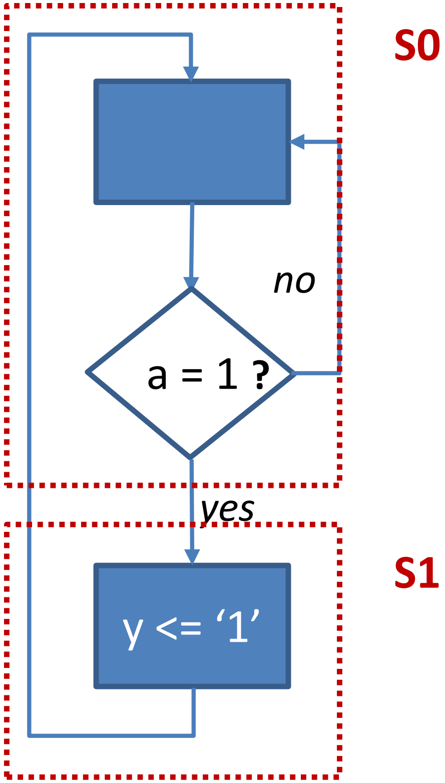
Step 2: ASM specification

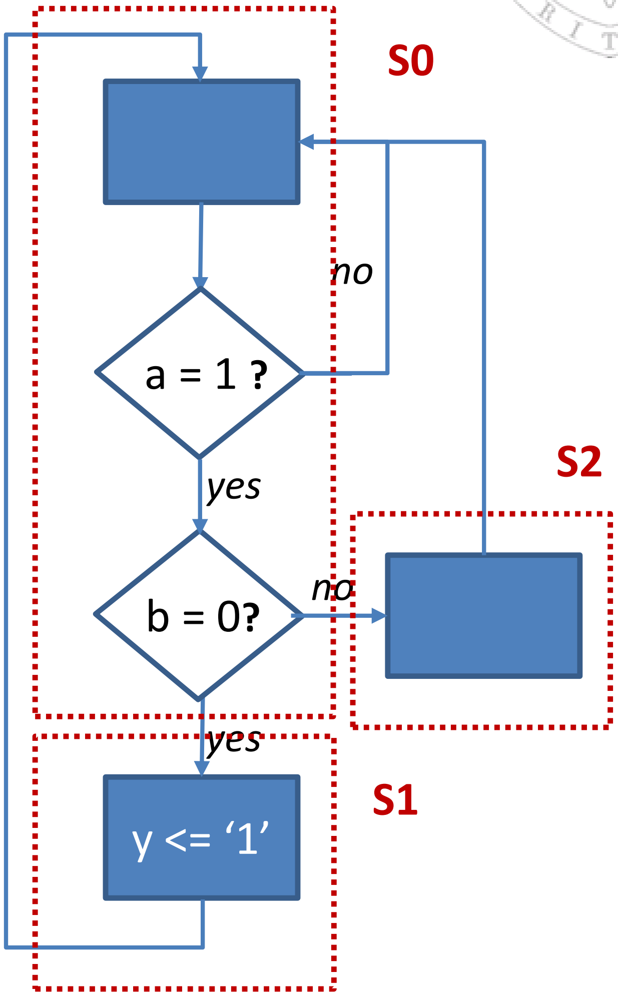


toc

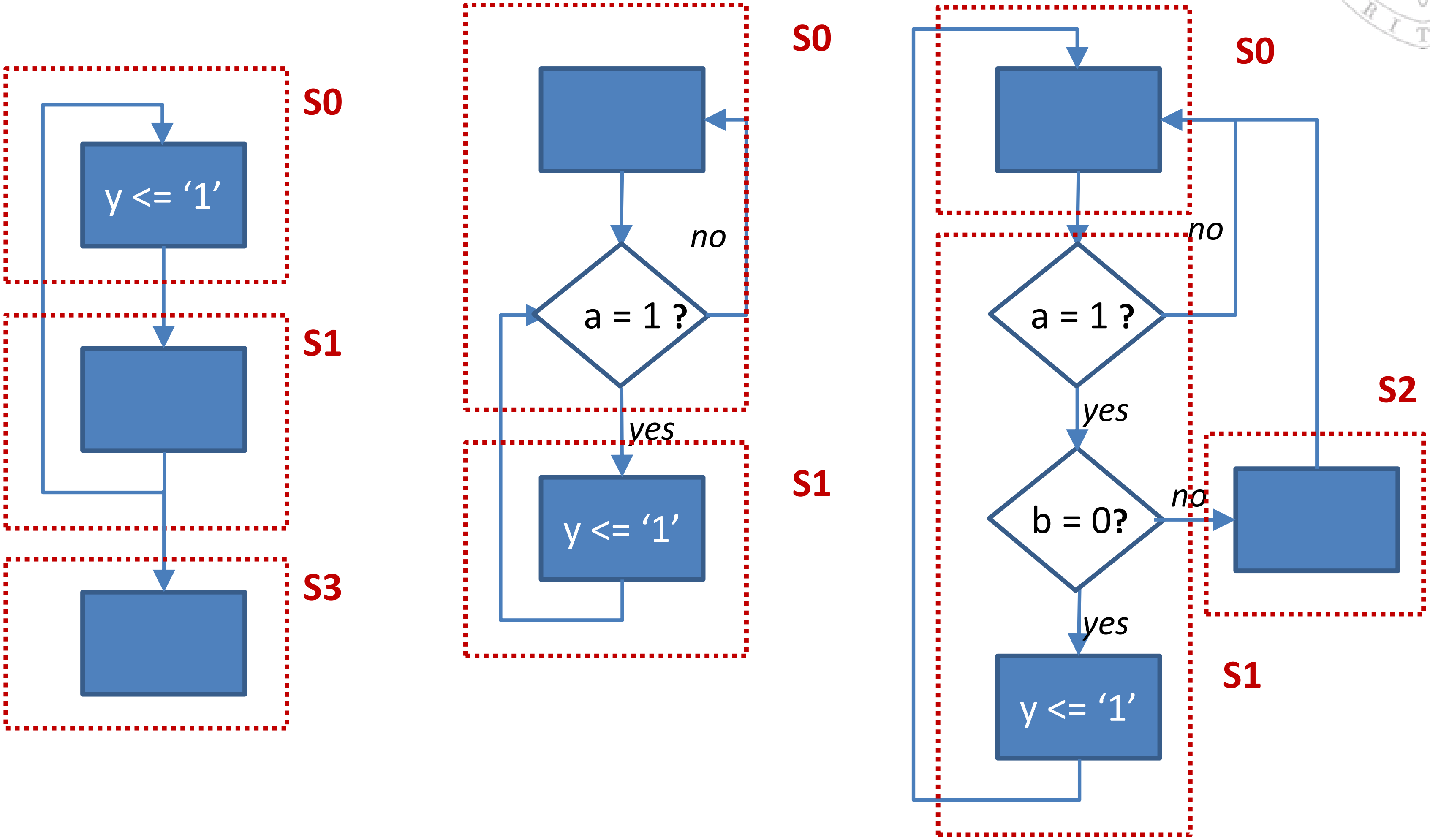
Step 2: ASM specification







ASM specification with errors



ASM Diagram: Data Path

- Identify the HW modules from the operations described in the ASM
- Interconnect the external and internal data signals to the modules
  - When the same module receives different inputs through the same port ⇒ multiplexer
- Identify the control signals of the HW modules
- Interconnect the control and the status signals with the Control Unit
  - Status signals: outputs of the Data Path that the Control Unit needs to make decisions

ASM Diagram: Control Unit

1. The Control Unit is implemented as a FSM
2. Each ASM block becomes a state of the FSM with the same name
3. Transitions between ASM blocks become transitions between the equivalent states of the FSM
4. The outputs of the Control Unit are the control signals of the Data Path (and the external control outputs)
5. The inputs of the Control Unit are the status signals (and the external control inputs)

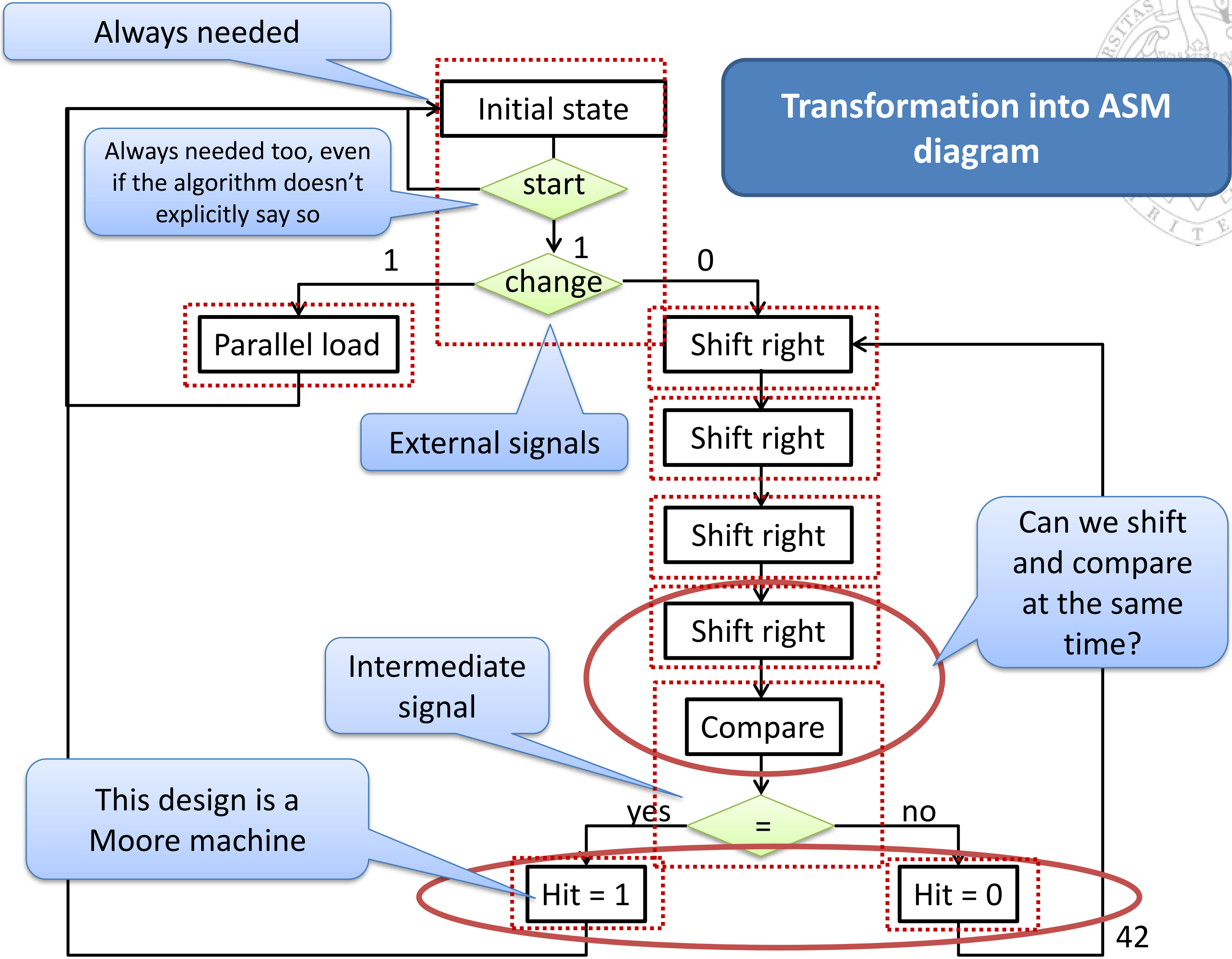
Example of an algorithmic system

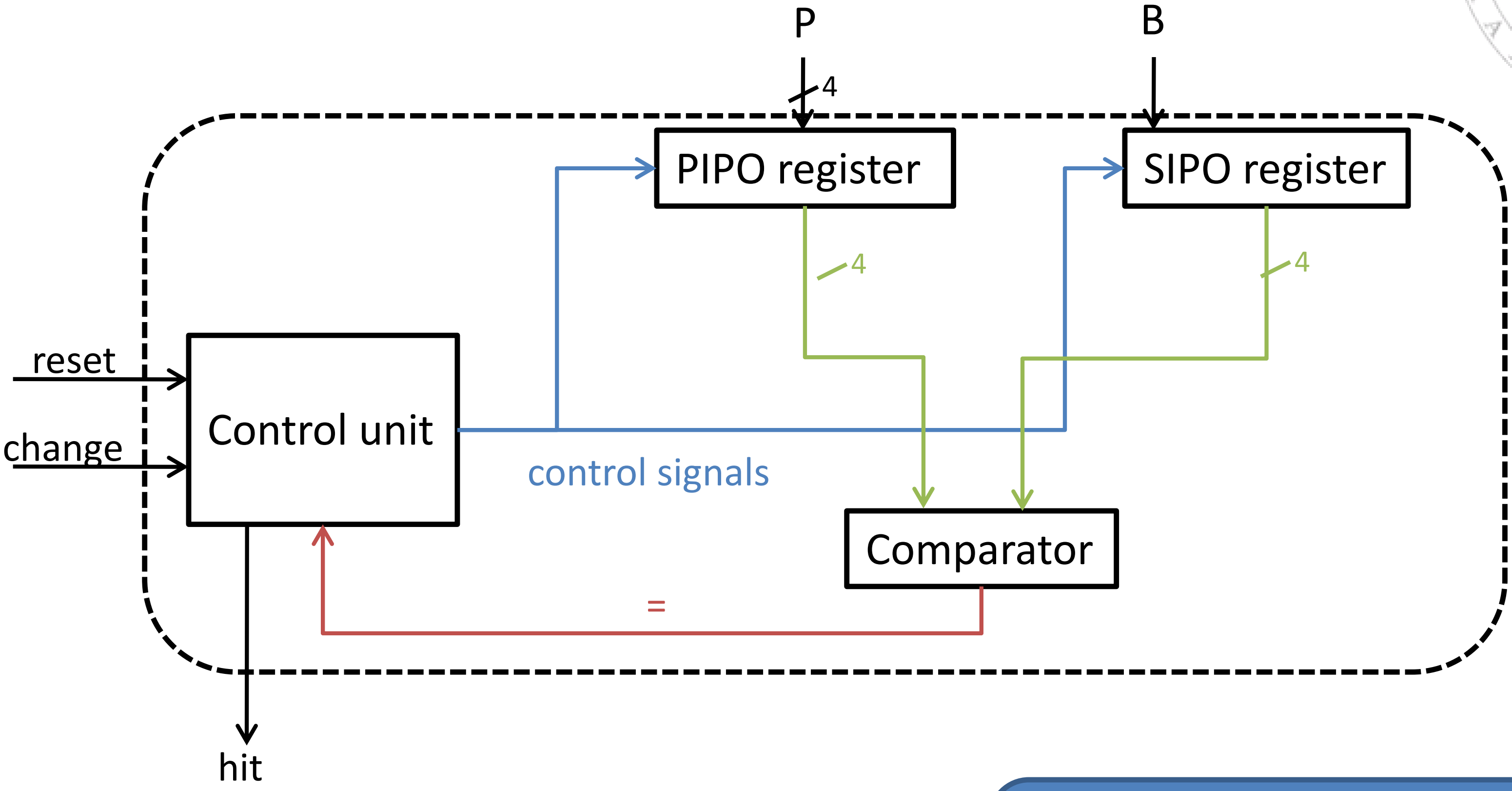
- Using the concepts that we have learnt about algorithmic design, design a system that is able to recognize a key.
- The system has an initial state that offers the user to change the key or to guess it:
  - Change key: The user changes the key of the device (4 bits) in parallel.
  - Introduce key: The user introduces an input key (through a serial input) and the system compares it with the key that has been stored:
    - If the input key is correct, the *hit* signal is set to ‘1’ and the system comes back to the initial state.
    - If the input key is incorrect, the *hit* signal is set to ‘0’ and the system asks again for a new input key to the user (through a serial input).

Example of an algorithmic system

- Scheme of the steps that must be carried out (algorithm):
  1. Initial state:
     - If *change* = 1 go to State 2
     - If *change* = 0 go to State 3
  2. State 2: Change key
     - Writes the key in parallel.
     - Back to the initial state.
  3. State 3: Guess key
     - Carry out 4 right shifts (4 cycles).
     - If the key is correct, then *hit* = 1. Back to State 1.
     - If the key is incorrect, then *hit* = 0. Back to State 3.

Transformation into ASM diagram





Datapath
+
Control unit

Outline

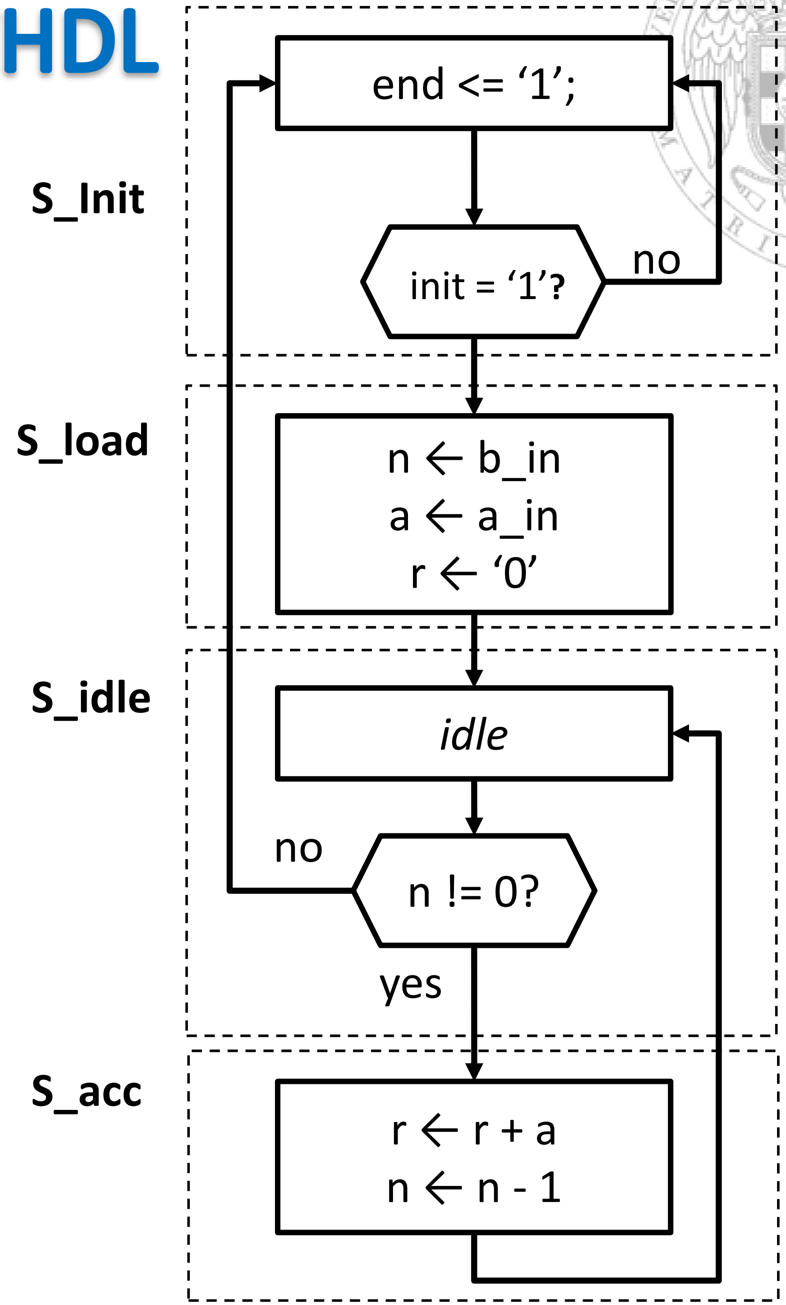
1. Introduction
   - FSMs: Quick review
   - Memory modules: Quick review
2. Algorithmic design:
   - Algorithmic state machine (ASM) diagram, control unit and datapath
3. Algorithmic design in VHDL

toc

Algorithmic design in VHDL
Example I

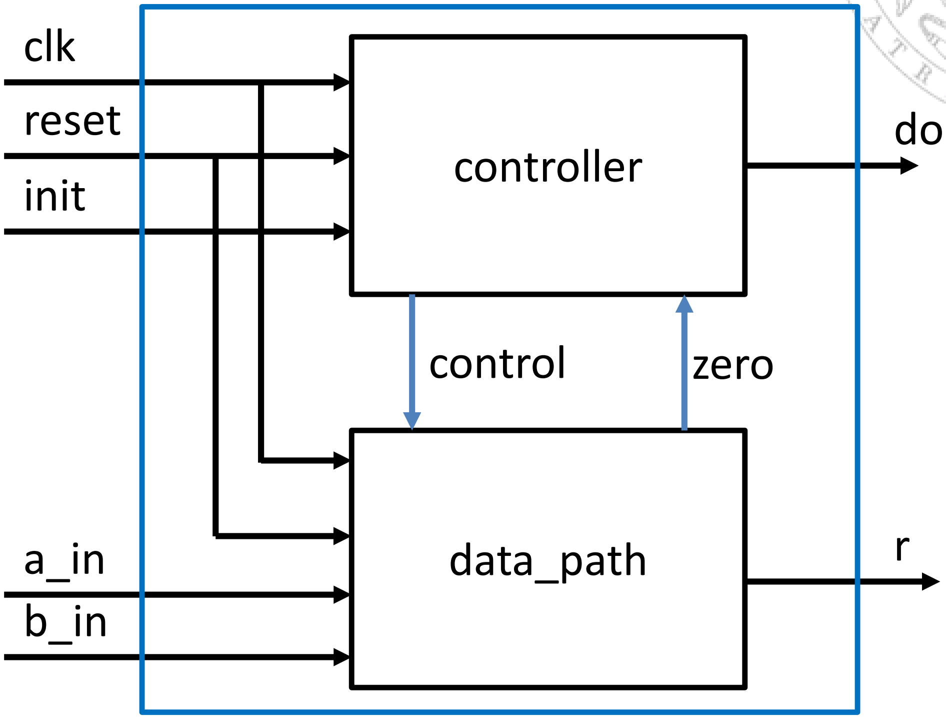
- ASM multiplier (add a `n` times)

```
a = a_in;
n = b_in;
r = 0;
while (n!=0) {
    r = r + a;
    n = n - 1;
}
```



Entity

- We will define a package for constants:
  - Widths of `a_in`, `b_in` and `r`
  - Control and Status indexes (names)



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.definitions.all;

entity ASM_multiplier is
port( reset, clk, init : in  std_logic;
      a_in, b_in       : in  std_logic_vector(W_FACTORS-1 downto 0);
      done             : out std_logic;
      r                : out std_logic_vector(W_RESULT-1 downto 0) );
end ASM_multiplier;
```

Definitions

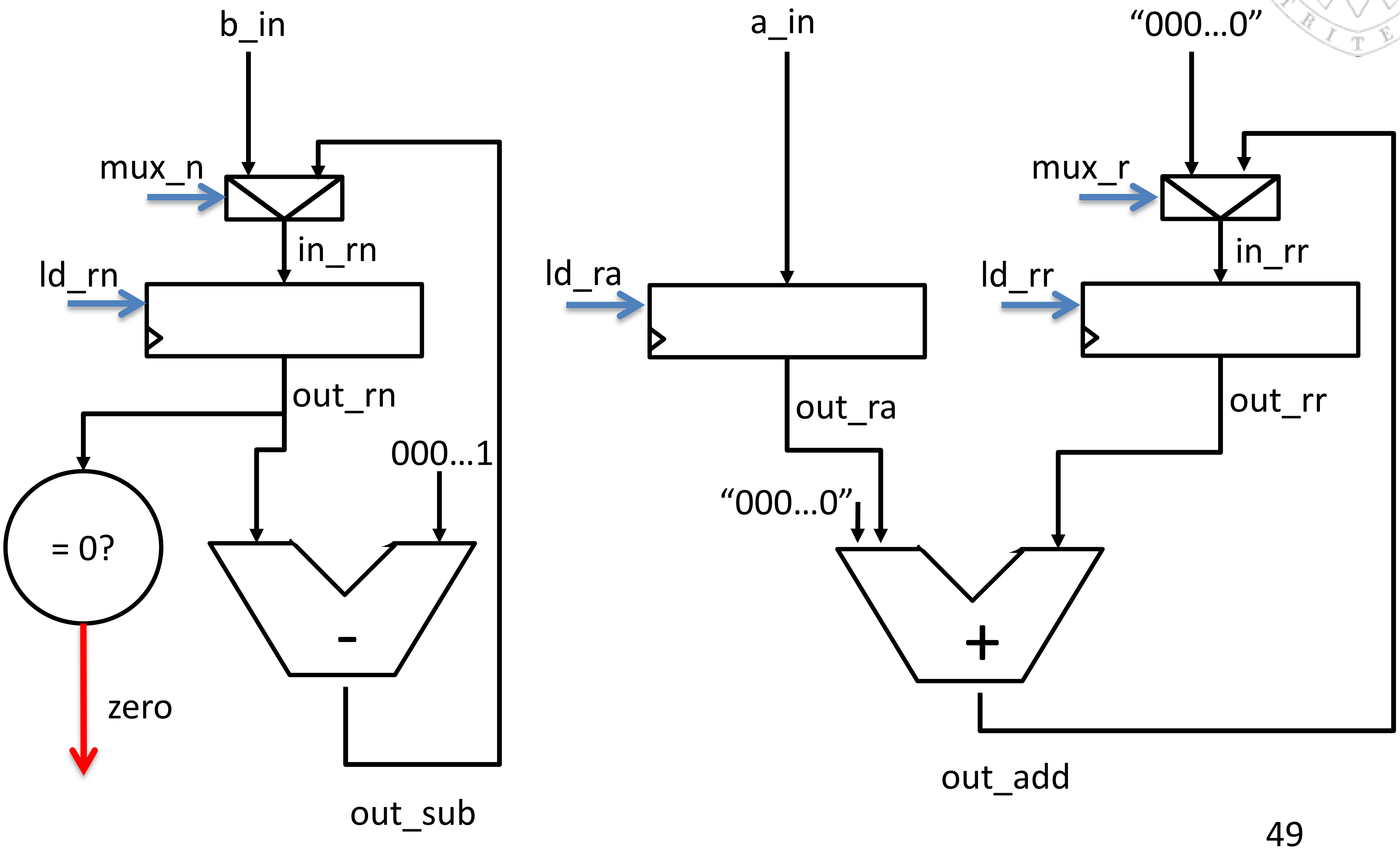
```
package definitions is
    constant W_FACTORS : integer := 4;
    constant W_RESULT  : integer := (W_FACTORS*2);

    -- Control Constants
    constant ld_ra     : integer := 0;
    constant ld_rn     : integer := 1;
    constant ld_rr     : integer := 2;
    constant mux_n     : integer := 3;   -- mux_n = '1' for external input
    constant mux_r     : integer := 4;   -- mux_r = '1' for external input
    constant W_CONTROL : integer := 5;   -- Control vector width

    -- Status Constants
    constant zero      : integer := 0;   -- reg n = 0?
    constant W_STATUS  : integer := 1;   -- Status vector width
end package definitions;
```

toc

Implementation of the top module

```
architecture arch_ASM_mult of ASM_multiplier is
  component controller
    port( clk, reset, init : in std_logic;
          status           : in  std_logic_vector(W_STATUS-1  downto 0);
          control          : out std_logic_vector(W_CONTROL-1 downto 0);
          done             : out std_logic );
  end component controller;

  component data_path
    port( clk, reset : in  std_logic;
          a_in, b_in : in  std_logic_vector(W_FACTORS-1 downto 0);
          control    : in  std_logic_vector(W_CONTROL-1 downto 0);
          status     : out std_logic_vector(W_STATUS-1  downto 0);
          r          : out std_logic_vector(W_RESULT-1  downto 0) );
  end component data_path;

  signal status:  std_logic_vector(W_STATUS-1  downto 0);
  signal control: std_logic_vector(W_CONTROL-1 downto 0);

begin

  U_CNTRL: controller port map(clk, reset, init, status, control, done);

  U_DP:    data_path  port map(clk, reset, a_in, b_in, control, status, r);

end arch_ASM_mult;
```

RTL with control & status signals



Data Path

```
architecture arch_dp of data_path is
  component asynch_reg
    generic (n: natural := 8);
    port( clk, rst, load : in  std_logic;
          din            : in  std_logic_vector (n-1 downto 0);
          dout           : out std_logic_vector (n-1 downto 0) );
  end component asynch_reg;
  component adder_sub
    generic( n: natural := 8 );
    port( a   : in std_logic_vector(n-1 downto 0);
          b   : in std_logic_vector(n-1 downto 0);
          op  : in std_logic;
          res : out std_logic_vector(n-1 downto 0) );
  end component adder_sub;

  signal in_ra, in_rn, out_ra, out_rn   : std_logic_vector(a_in'RANGE);
  signal zeroes, out_sub, inb_sub       : std_logic_vector(a_in'RANGE);
  signal in_rr, ina_add, out_rr, out_add: std_logic_vector(r'RANGE);
```

```
begin
    zeroes <= (others => '0');
    in_ra  <= a_in;
    r      <= out_rr;

    U_REG_A: asynch_reg
        generic map(W_FACTORS)
        port map(clk, reset, control(ld_ra), in_ra, out_ra);
    U_REG_N: asynch_reg
        generic map(W_FACTORS)
        port map(clk, reset, control(ld_rn), in_rn, out_rn);
    U_REG_R: asynch_reg
        generic map(W_RESULT)
        port map(clk, reset, control(ld_rr), in_rr, out_rr);

    inb_sub <= (0=>'1', others=>'0');
    U_SUB: adder_sub
        generic map(W_FACTORS)
        port map(out_rn, inb_sub, '1', out_sub);

    ina_add <= (W_RESULT-1 downto W_FACTORS => '0') & out_ra;
    U_ADD: adder_sub
        generic map(W_RESULT)
        port map( ina_add, out_rr, '0', out_add);

    in_rn <= out_sub when control(mux_n)='0' else b_in;
    in_rr <= out_add when control(mux_r)='0' else (others =>'0');

    status(zero) <= '1' when out_rn = zeroes else '0';

end arch_dp;
```

toc

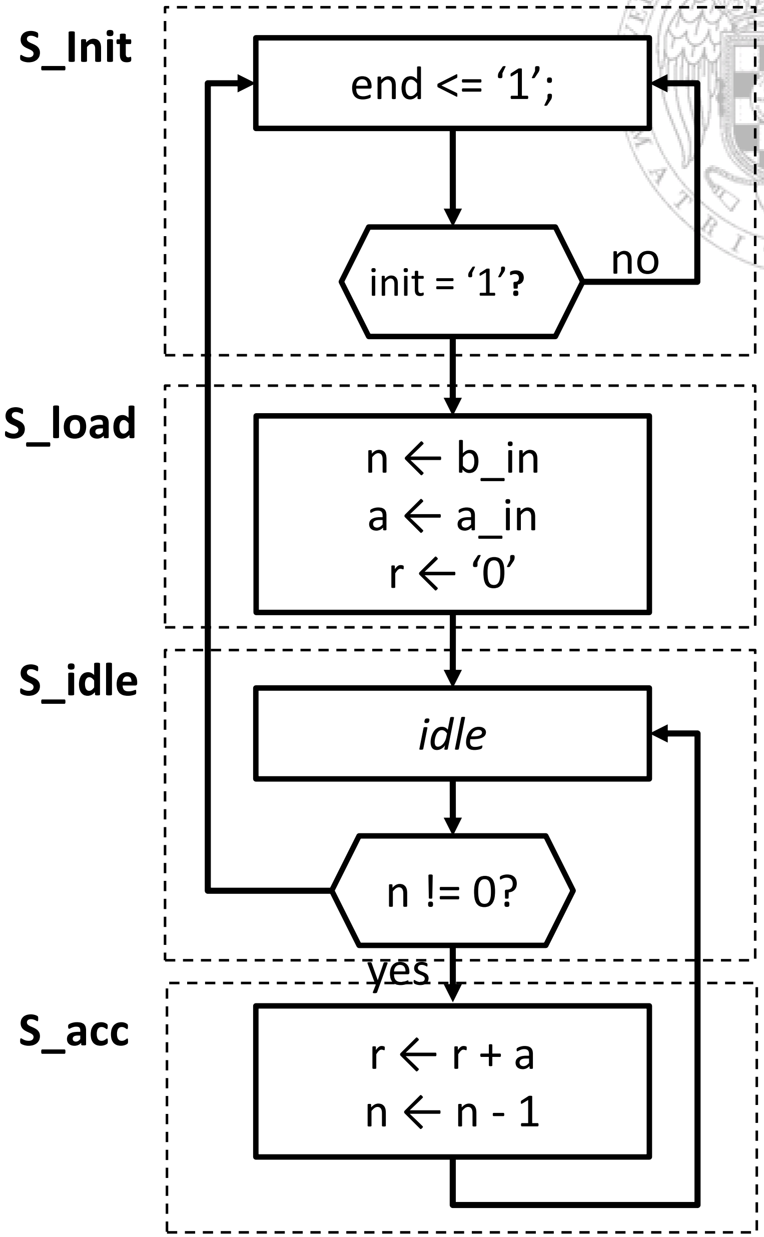
Controller

- The controller features 4 states

```
architecture arch_controller of controller is
    type T_STATE is (S_init, S_load, S_idle, S_acc);
    signal STATE, NEXT_STATE: T_STATE;
begin

    SYNC_STATE: process (clk, reset)
    begin
        if clk'event and clk = '1' then
            if reset = '1' then
                STATE <= S_init;
            else
                STATE <= NEXT_STATE;
            end if;
        end if;
    end process SYNC_STATE;
```

toc

```
COMB: process (STATE, init, status)
begin
    control <= (others => '0');
    done    <= '0';
    case STATE is
        when S_init =>
            done <= '1';
            if (init='1') then
                NEXT_STATE <= S_load;
            else
                NEXT_STATE <= S_init;
            end if;
        when S_load =>
            control(ld_ra) <= '1';
            control(ld_rn) <= '1';
            control(ld_rr) <= '1';
            control(mux_n) <= '1';
            control(mux_r) <= '1';
            NEXT_STATE <= S_idle;
        when S_idle =>
            if (status(zero) ='1') then
                NEXT_STATE <= S_init;
            else
                NEXT_STATE <= S_acc;
            end if;
        when S_acc =>
            control(ld_ra) <= '1';
            control(ld_rn) <= '1';
            control(ld_rr) <= '1';
            control(mux_n) <= '0';
            control(mux_r) <= '0';
            NEXT_STATE      <= S_idle;
    end case;
end process;
```

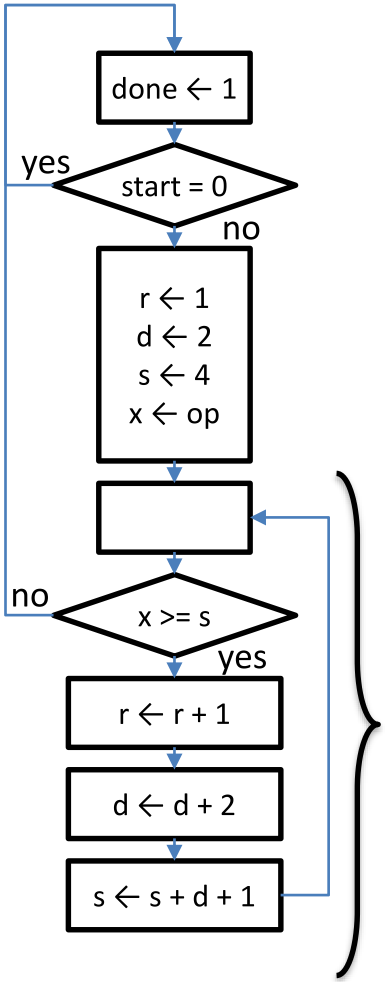


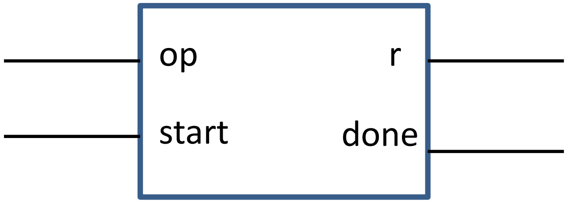
Integer square root. Example II

$$isqrt(r) = floor(\sqrt{op})$$

```
-- In:  op
-- Out: r

r <= 1;
d <= 2;
s <= 4;
x <= op;
while x >= s
      r <= r + 1;
      d <= d + 2;
      s <= s + d + 1;
end
```

Option I



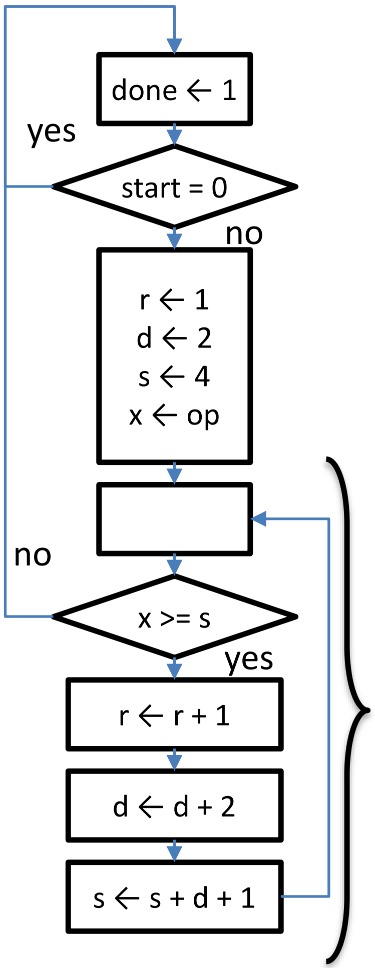


Loop:

- 2 adders
- 4 cycles/iteration

Can it be done in a different way?

Option I


done ← 1

yes

start = 0

no

r ← 1
d ← 2
s ← 4
x ← op

no

x >= s

yes

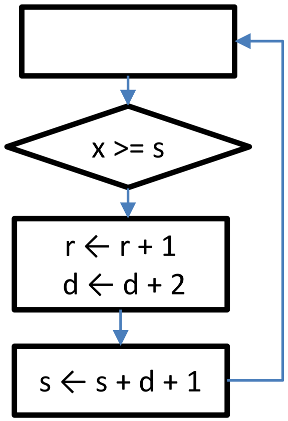
r ← r + 1

d ← d + 2

s ← s + d + 1

2 adders
4 cycles/iteration

Option II

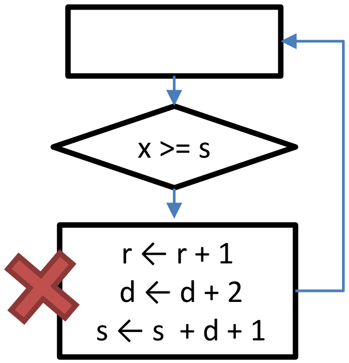

x >= s

r ← r + 1
d ← d + 2

s ← s + d + 1

2 adders
3 cycles/iteration

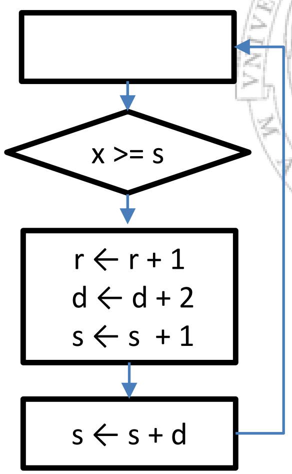
Option III


x >= s

r ← r + 1
d ← d + 2
s ← s + d + 1

4 adders
2 cycles/iteration

Option IV

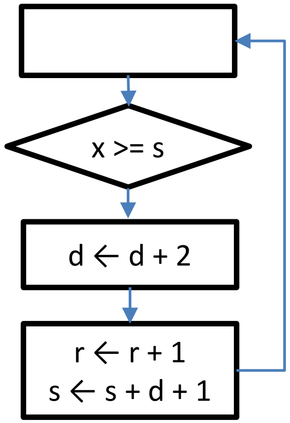

x >= s

r ← r + 1
d ← d + 2
s ← s + 1

s ← s + d

3 adders
3 cycles/iteration

Option V

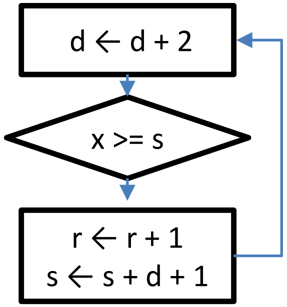

x >= s

d ← d + 2

r ← r + 1
s ← s + d + 1

3 adders
3 cycles/iteration

Option VI

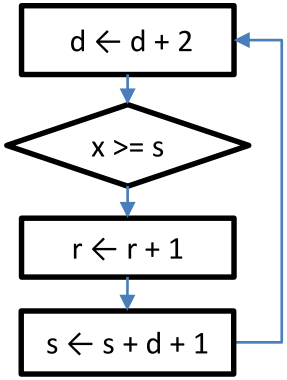

d ← d + 2

x >= s

r ← r + 1
s ← s + d + 1

3 adders
2 cycles/iteration

Option VII


d ← d + 2

x >= s

r ← r + 1

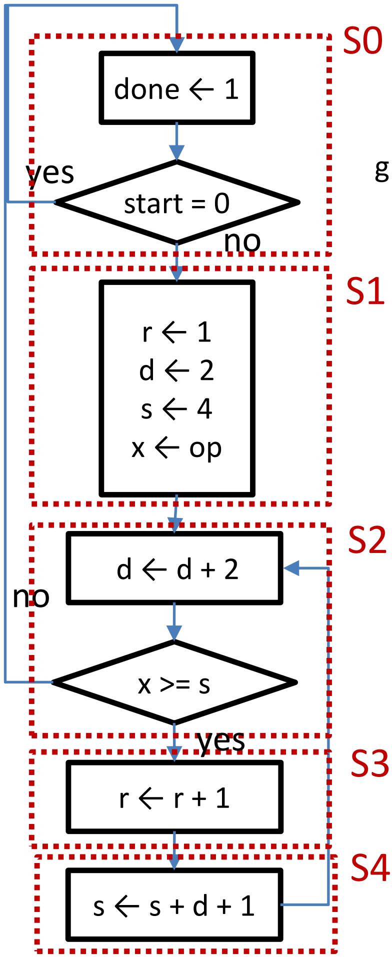
s ← s + d + 1

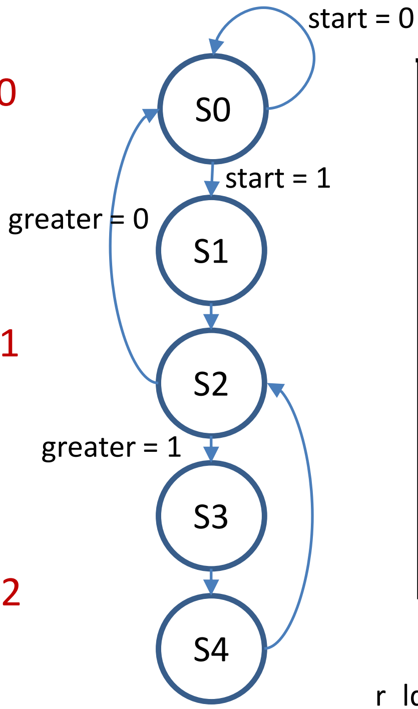
2 adders
3 cycles/iteration

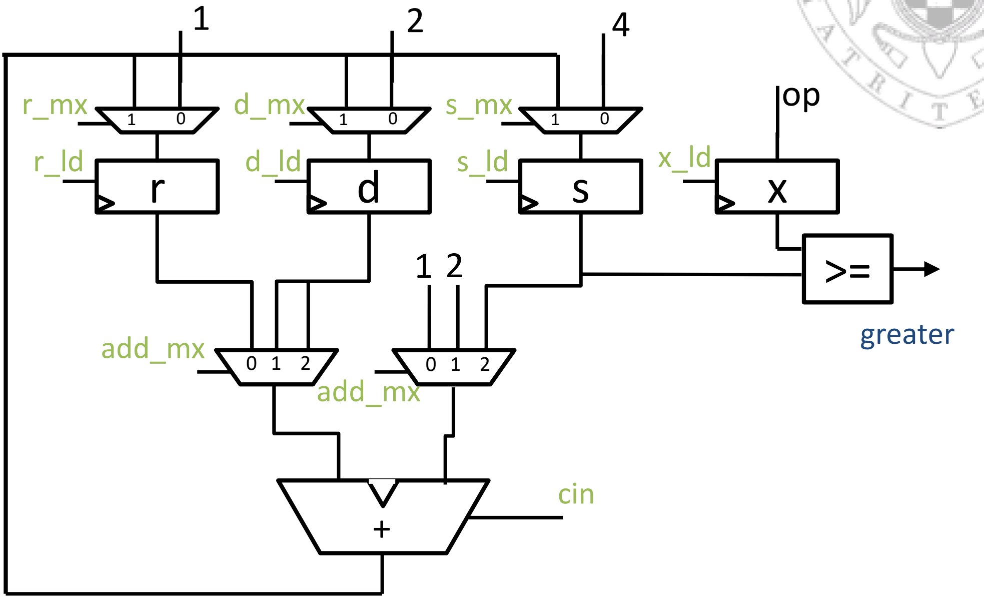
toc

Integer square root. Example II

Option VII







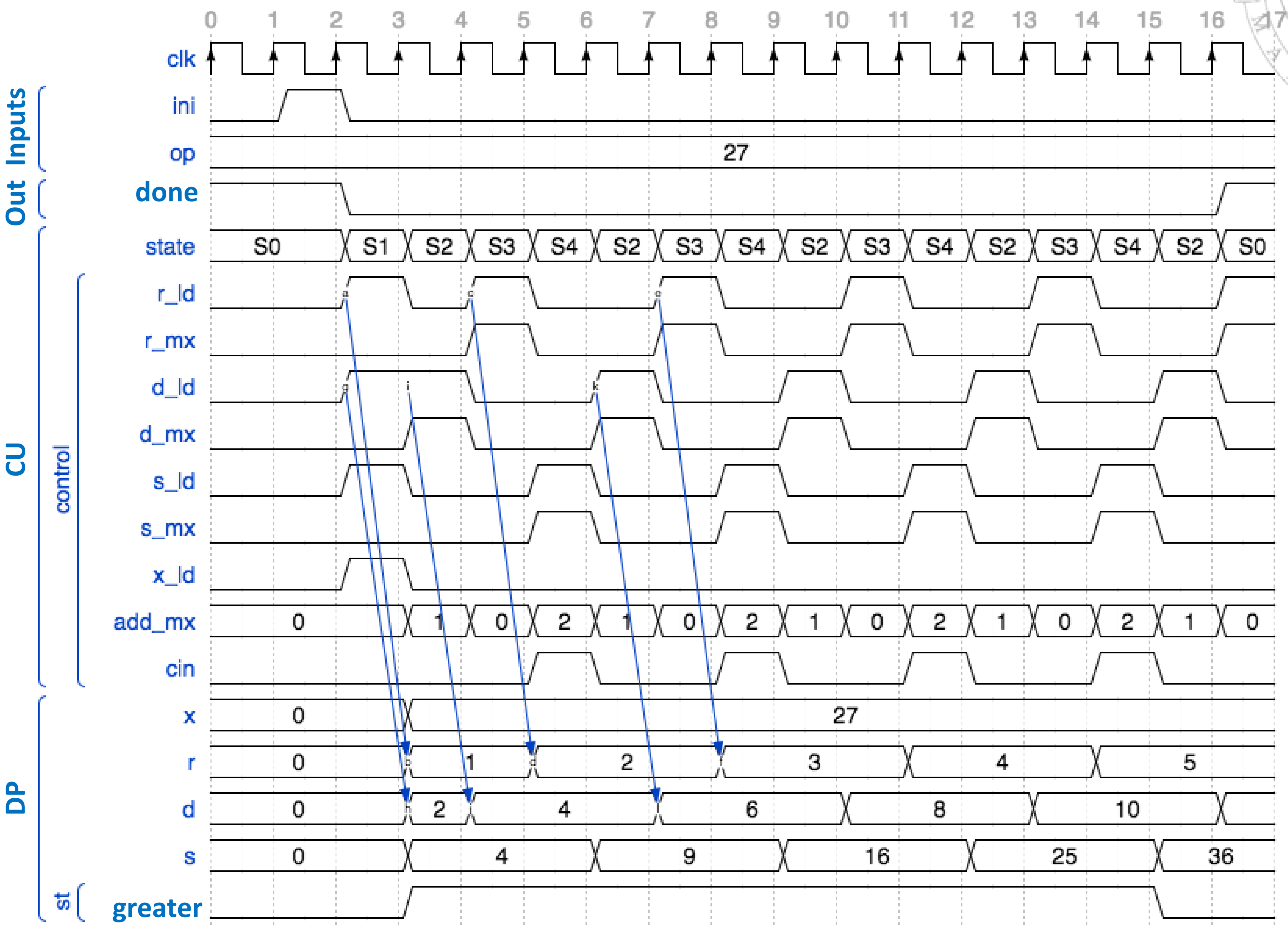
1 adder -> +1 cin

3 cycles/iteration

	r_ld	r_mx	d_ld	d_mx	s_ld	s_mx	x_ld	add_mx	cin	done
S0	0	dc	0	dc	0	dc	0	dc	0	1
S1	1	0	1	0	1	0	1	dc	0	0
S2	0	dc	1	1	0	dc	0	1	0	0
S3	1	1	0	dc	0	dc	0	0	0	0
S4	0	dc	0	dc	1	1	0	2	1	0

toc

Integer square root. Example II


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Inputs
Out
CU
control
DP
st
clk
ini
op
27
done
state
S0
S1
S2
S3
S4
S2
S3
S4
S2
S3
S4
S2
S3
S4
S2
S0
r_ld
r_mx
d_ld
d_mx
s_ld
s_mx
x_ld
add_mx
0
1
0
2
1
0
2
1
0
2
1
0
2
1
0
cin
x
0
27
r
0
1
2
3
4
5
d
0
2
4
6
8
10
s
0
4
9
16
25
36
greater