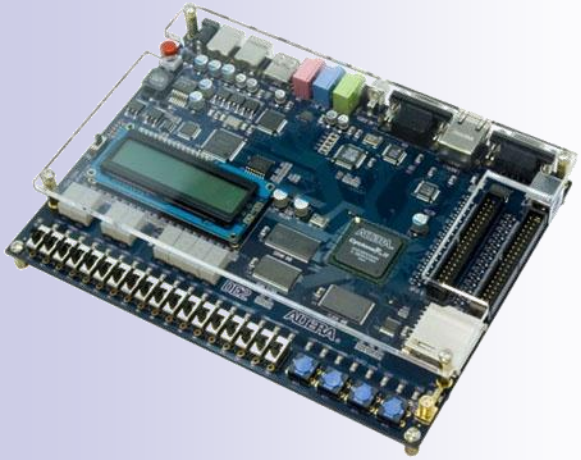
Logic Systems and Processors
cz:Logické systémy a procesory



Lecturer: Richard Šusta

richard@susta.cz, susta@fel.cvut.cz,
+420 2 2435 **7359**

Version: 1.0

Homework

from the previous lecture

What does the code do?

```
function GoS0(stnow:state_t; x, cmp:integer) return state_t is
  begin if x=cmp then return stnow; else return s0; end if;
  end function;
begin
 if rising_edge(CLK) then
   x:=to_integer(unsigned(std_logic_vector'(swc&swb&swa)));
   if x=0 then state:= s1;  else
    case state is
      when s1 => if x=1 then state:=s2; else state:=s0; end if;
      when s2 => if x=3 then state:=s3; else state:=GoS0(state, x, 1); end if;
      when s3 => if x=7 then state:=s4; else state:=GoS0(state, x, 3); end if;
      when s4 => state:=GoS0(state, x, 7);
      when others =>
   end case;
   end if;
 end if;
```

It's the FSM! So let's build its transition table.

What does the code do?

```
library ieee; use ieee.std_logic_1164.all;use ieee.numeric_std.all;
entity UnknownFSM is port ( CLK, swa, swb, swc, ACLRN : in std_logic; ixDebug: out unsigned(2 downto 0); q : out std_logic);
end;
architecture rtl of UnknownFSM is begin
  process (CLK, ACLRN) -- state register
  type state_t is (s0, s1, s2, s3, s4); variable state: state_t:=s0;
  variable x : integer range 0 to 7;
    function GoS0(stnow:state_t; x, cmp:integer) return state_t is
    begin if x=cmp then return stnow; else return s0; end if;
    end function;
begin
  if ACLRN='0' then state:=s0; elsif rising_edge(CLK) then x:=to_integer(unsigned(std_logic_vector'(swc&swb&swa)));
    if x=0 then state:= s1;
    else case state is
          when s1 => if x=1 then state:=s2; else state:=s0; end if;
          when s2 => if x=3 then state:=s3; else state:=GoS0(state, x, 1); end if;
          when s3 => if x=7 then state:=s4; else state:=GoS0(state, x, 3); end if;
          when s4 => state:=GoS0(state, x, 7);
          when others =>
        end case;
    end if;
  end if;
  if state=s4 then q<='1'; else q<='0'; end if;
  ixDebug <= to_unsigned(state_t'POS(state), ixDebug'LENGTH); -- state_t'POS(state) returns integer order number of state
end process; end architecture;
```

It's the FSM! So let's build its transition table.

What does the code do?

Transition Table

x=	0	1	2	3	4	5	6	7	Q
s0	s1	s0	s0	s0	s0	s0	s0	s0	0
s1	s1	s2	s0	s0	s0	s0	s0	s0	0
s2	s1	s2	s0	s3	s0	s0	s0	s0	0
s3	s1	s0	s0	s3	s0	s0	s0	s4	0
s4	s1	s0	s0	s0	s0	s0	s0	s4	1

State Transition Table

	s0	s1	s2	s3	s4
s0		0			
s1	x/=1		1		
s2	x/=3 /=0	0		3	
s3	x/=7 /=0	0			7
s4	x/=7 /=0	0			

Detects the turning on of 3 switches in the sequence
SWA → SWB → SWC.

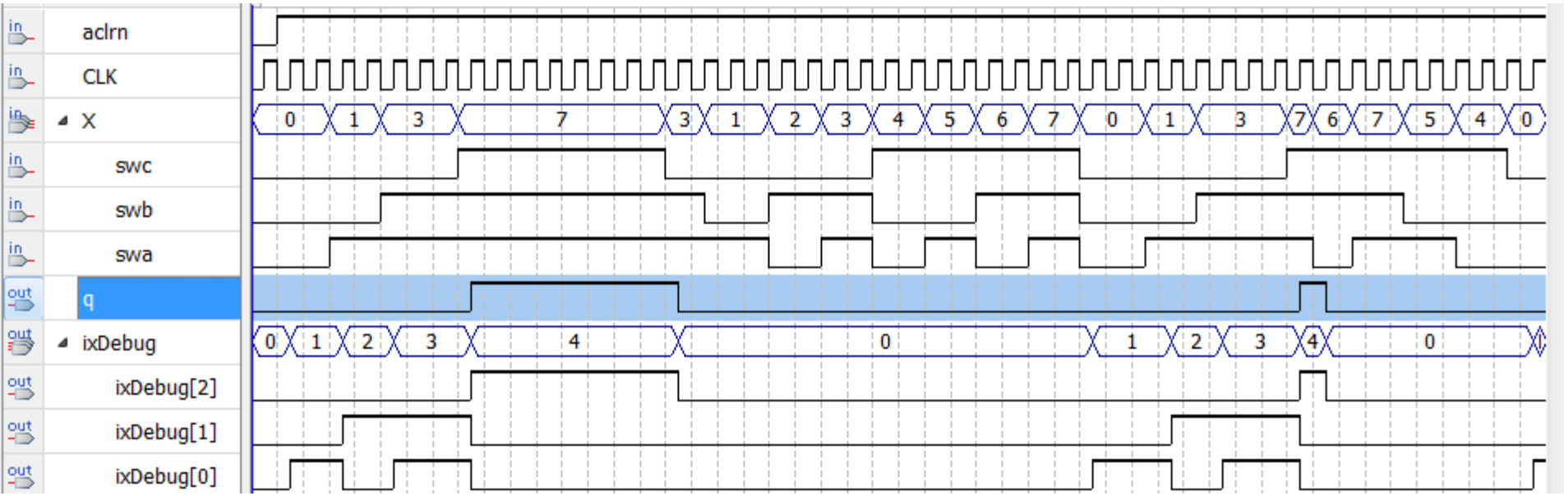
FSM Terminology:

Transition graph + **Transition table** + **State transition table**

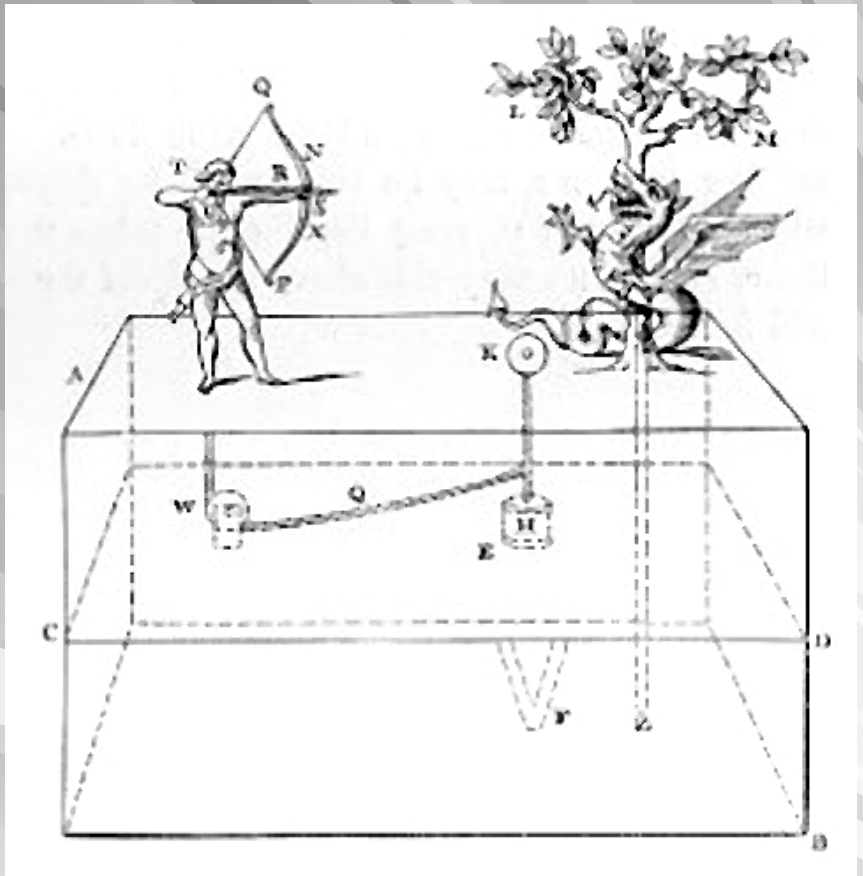
Beware:

Transition matrix belongs to the control theory terminology.

Simulation

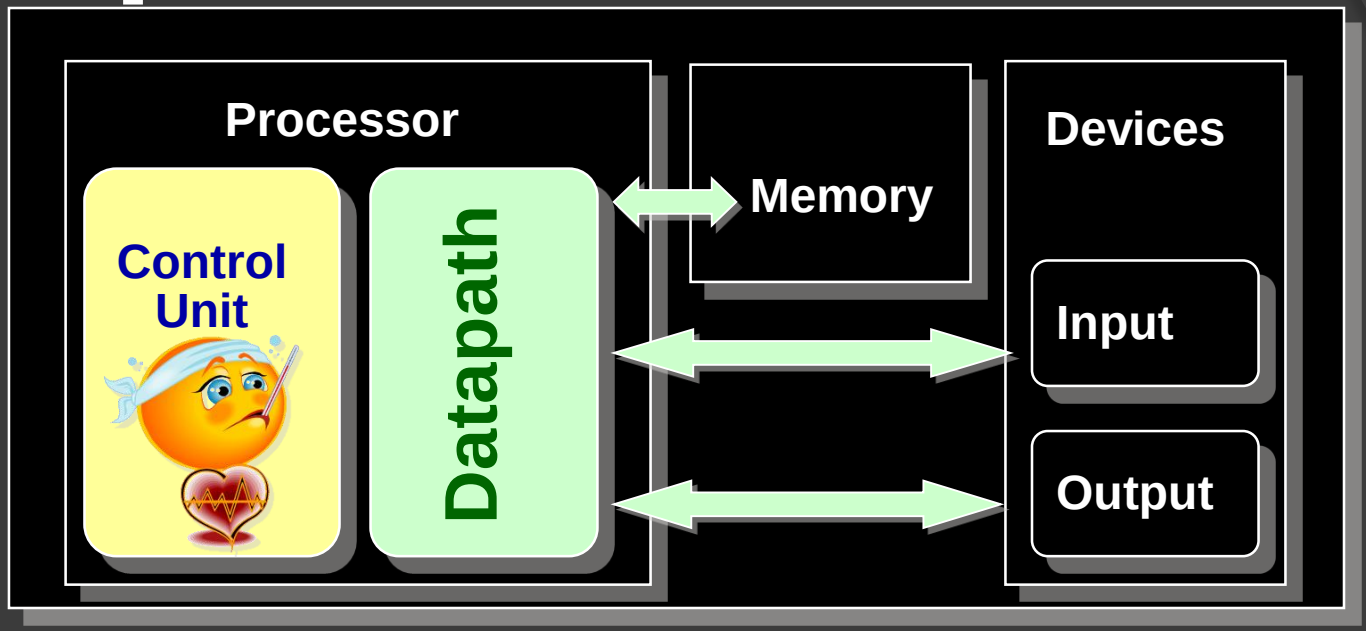

aclrn
CLK
X
swc
swb
swa
q
ixDebug
ixDebug[2]
ixDebug[1]
ixDebug[0]
0 1 3 7 3 1 2 3 4 5 6 7 0 1 3 7 6 7 5 4 0
0 1 2 3 4 0 1 2 3 4 0

Control Units
Cz: Řadiče

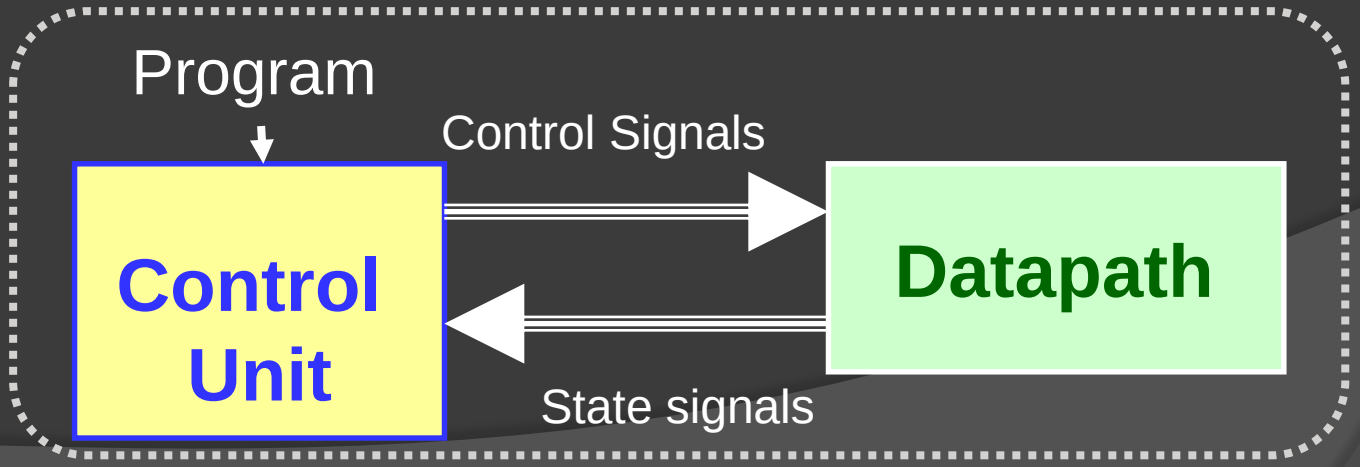


The ancient controller:
On the apple being lifted, Hercules shoots the Dragon who then hisses. Hero from Alexandria, *Pneumatica*.

Computer



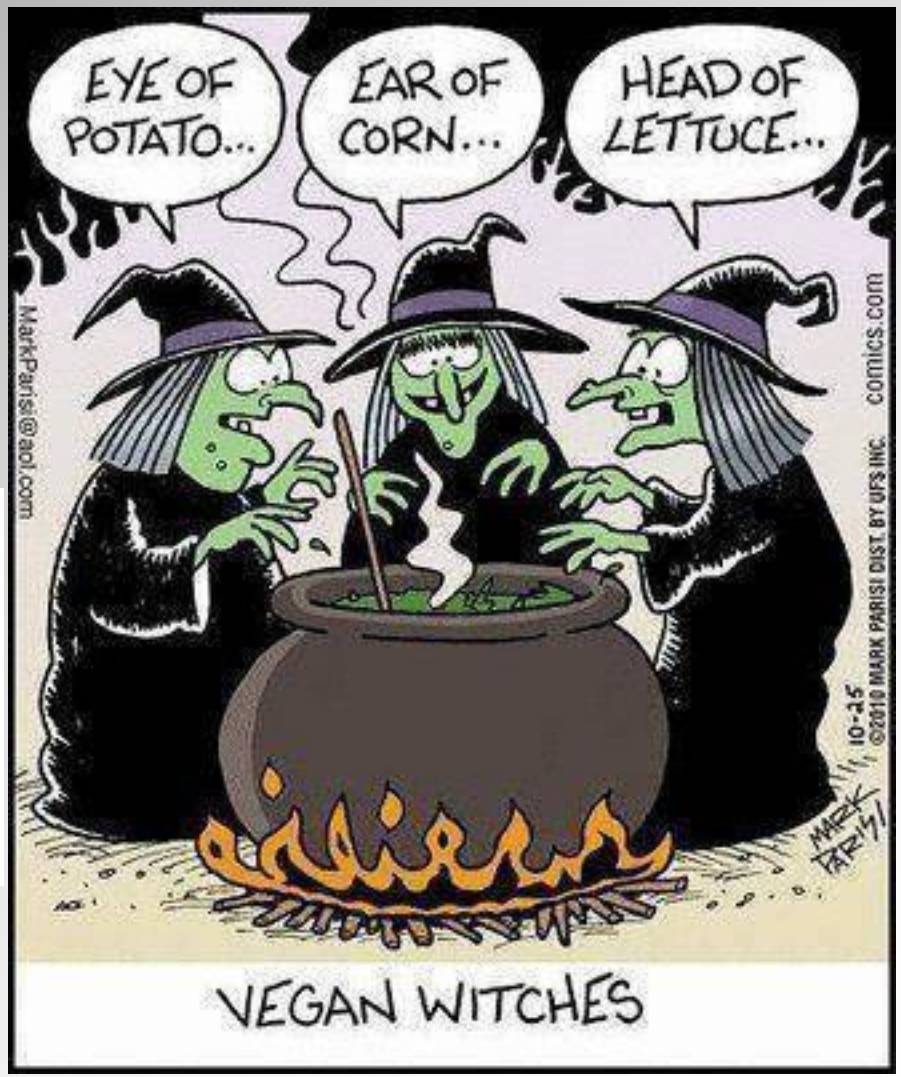
Processor



Design the ecological controller for boilers to reduce their consumption of fossil fuels

This meaning of "ear" is from old English "ere"=edge

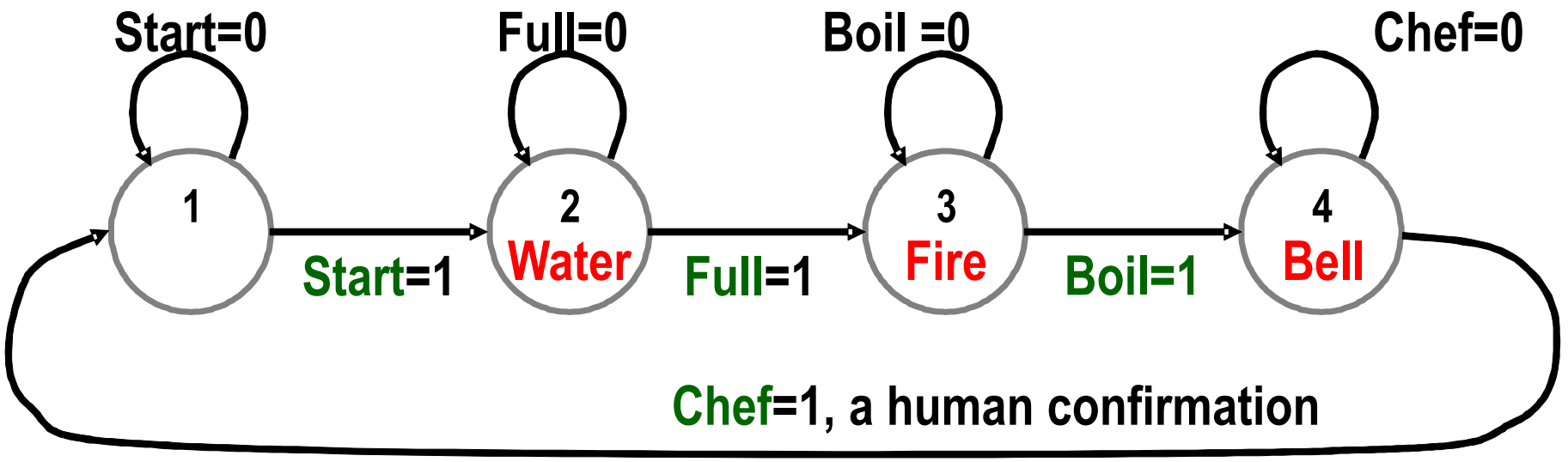
Source: https://chatham.ces.ncsu.edu/



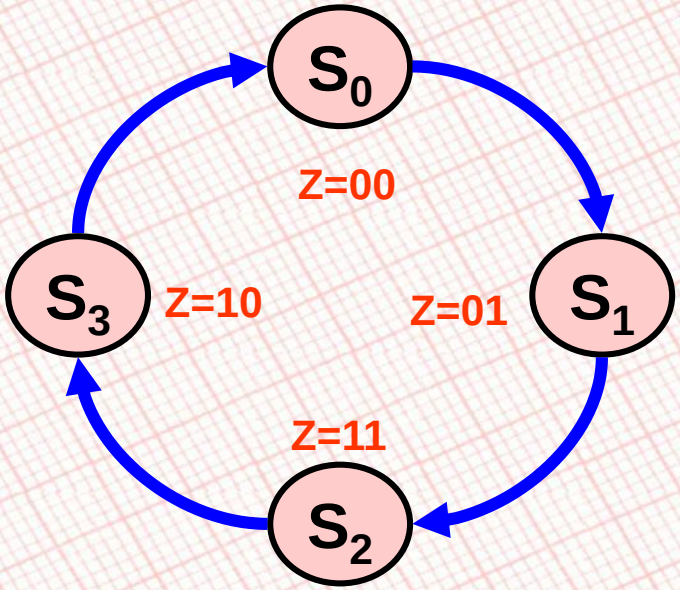
Source: https://grandmalin.wordpress.com/

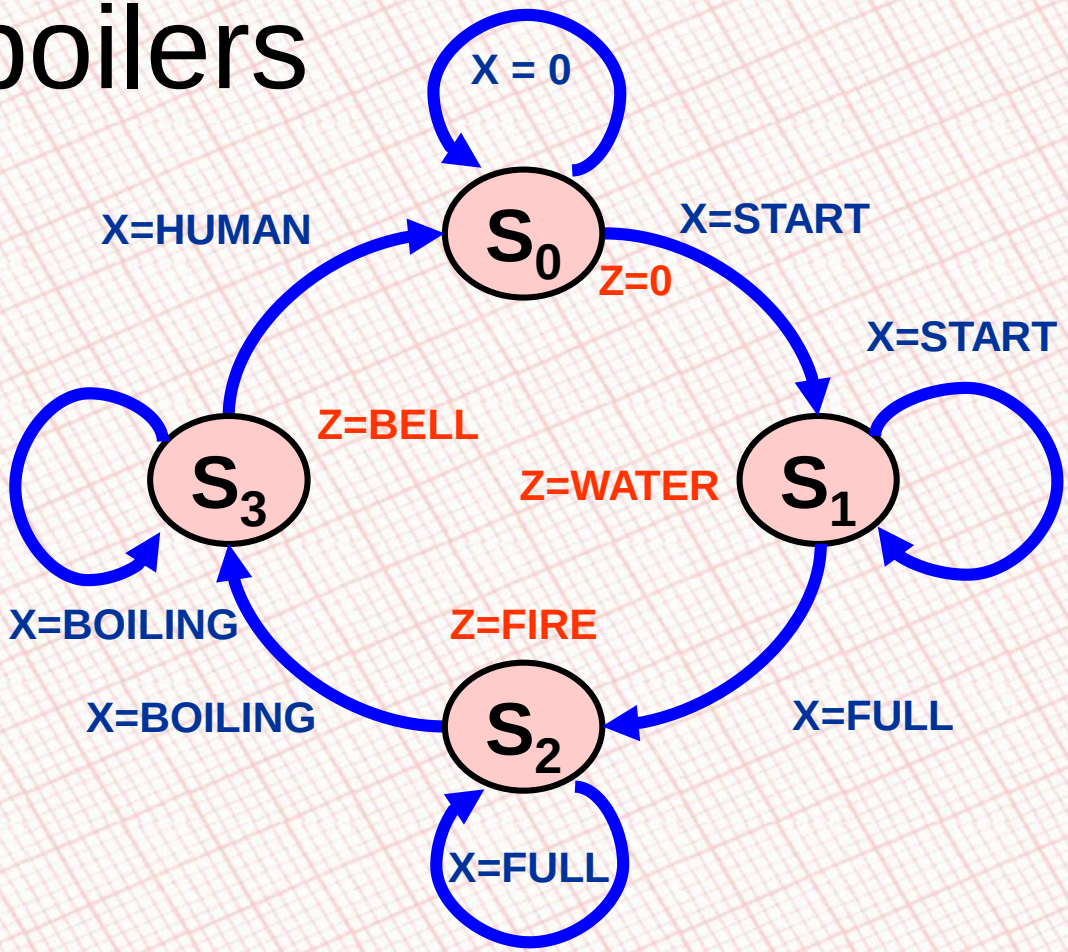
Control Units (in Czech "řadiče")

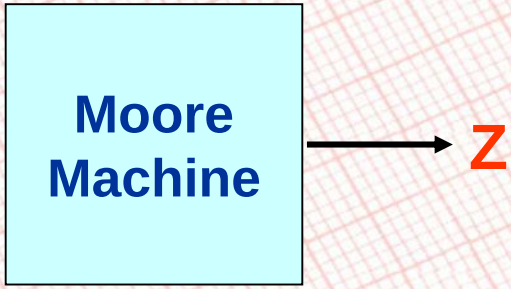
- Their most probable action is movement to the next state.
- They generate control signals according to input sequences.
- The finite state machine (FSM) can be used to represent their behavior.

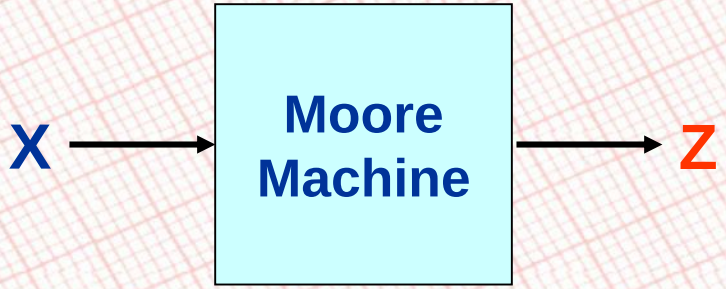


4 states counter versus control unit of boilers





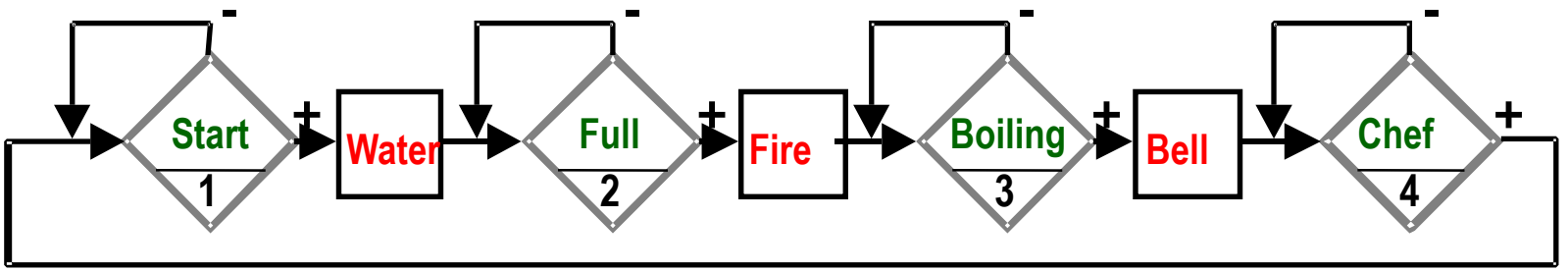


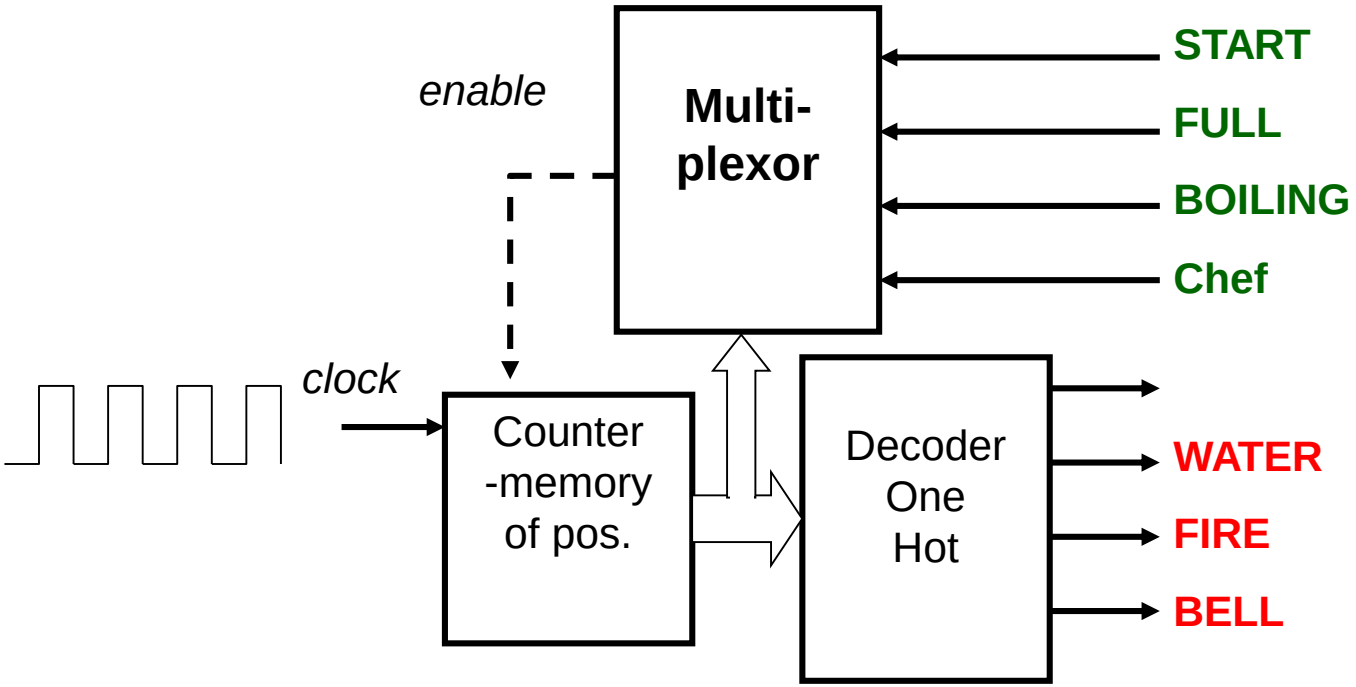


Control Units and their Analogies

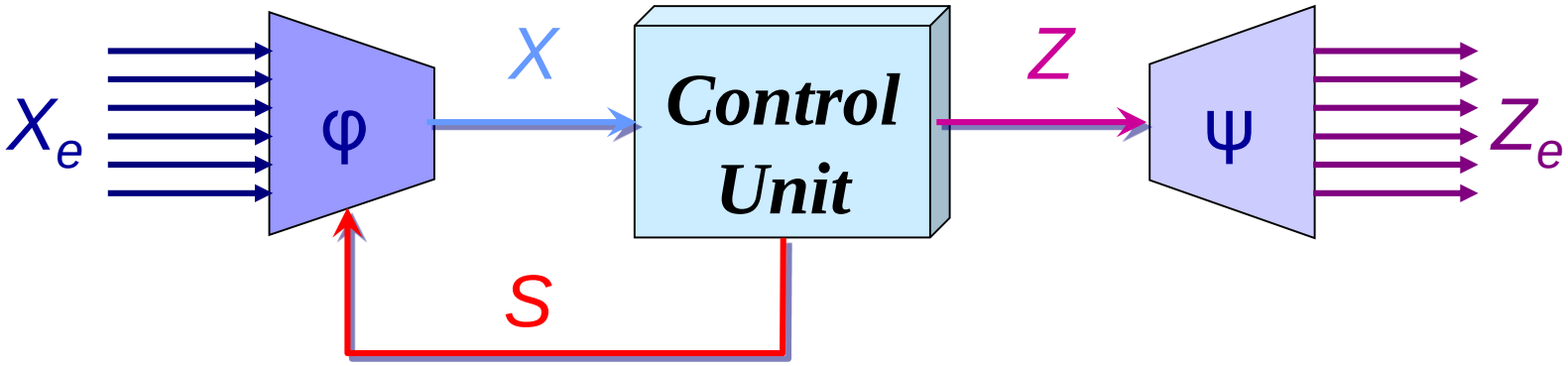
Music box, Leopold Aucac Aine, Paris

Control unit drawing style



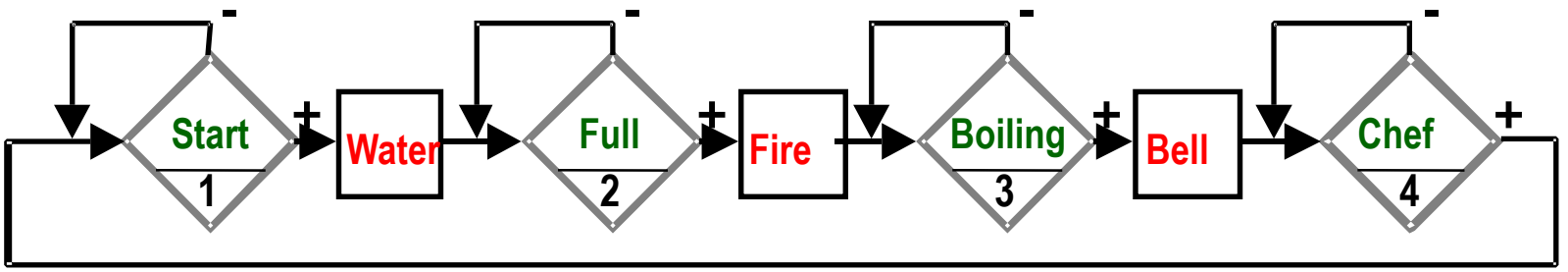


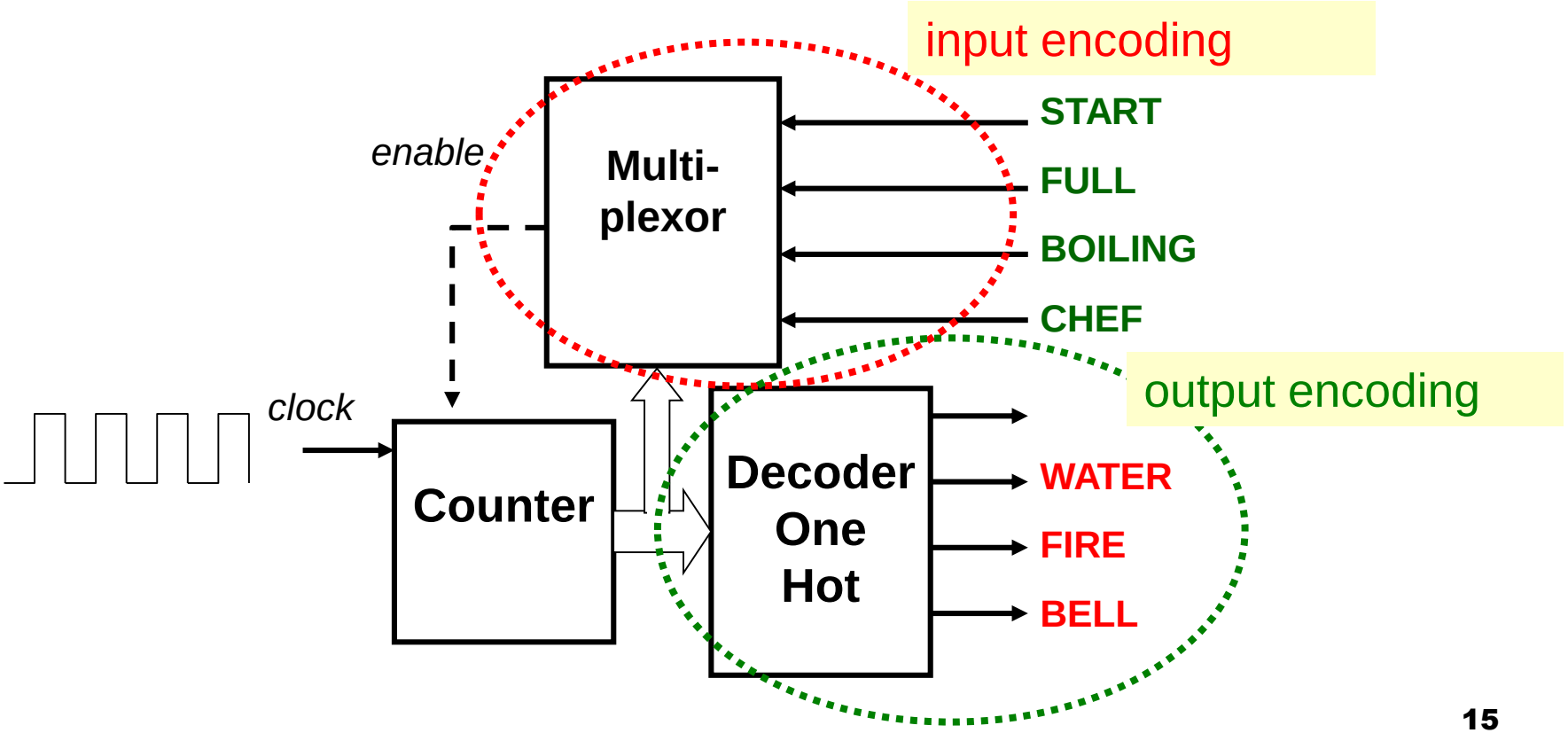
Encoding of Inputs and Outputs



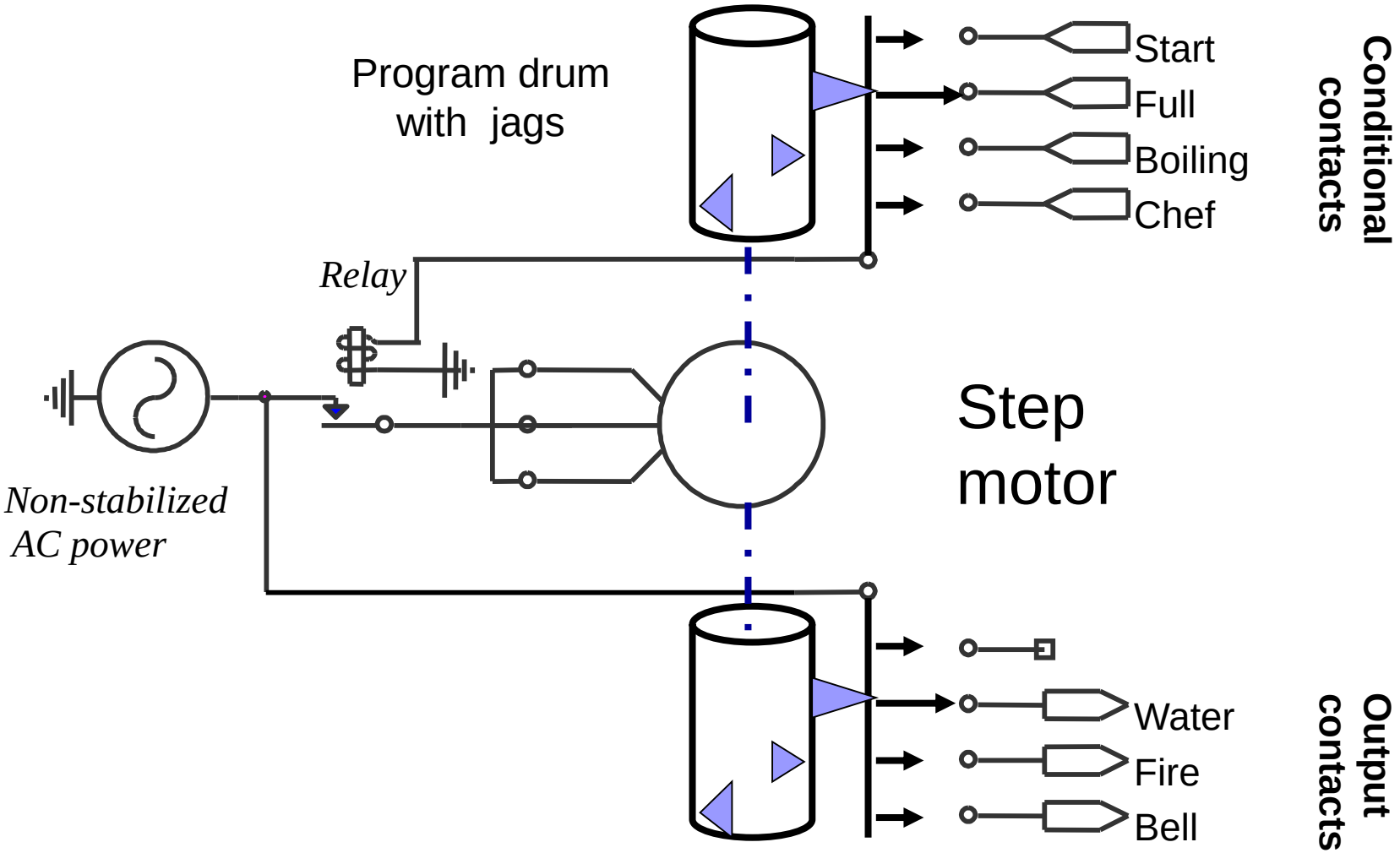
- *External inputs X_e and outputs Z_e can be encoded by*
  - φ: $X_e \times S \rightarrow X$, *where usually* $|X_e|>|X|$
  - ψ: $Z \rightarrow Z_e$, ,
    where usually $z^m \in Z$, $z_e^n \in Z_e$; $m < n$
- *Encoding simplifies design of control units*

Control Unit

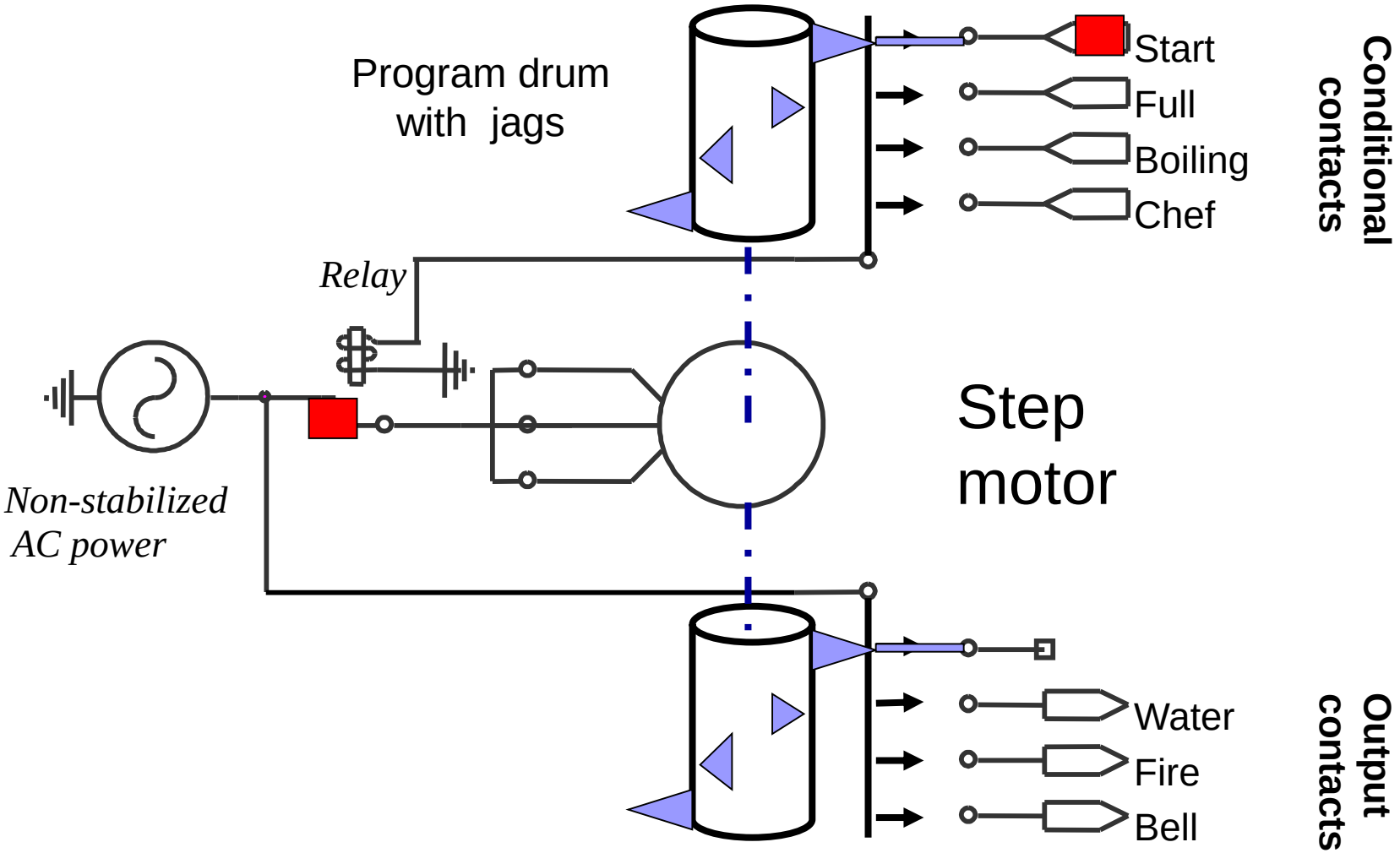




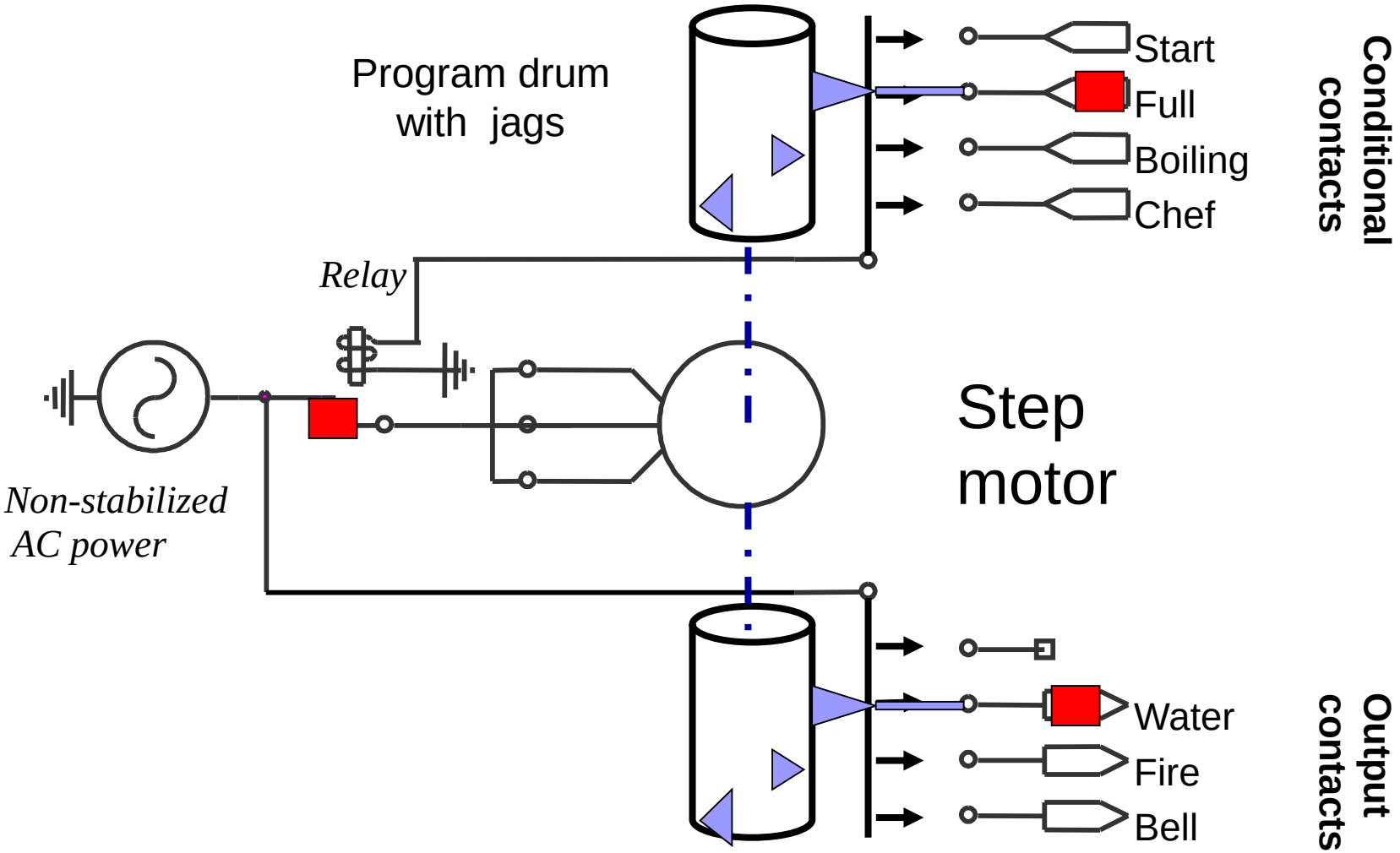
Electromechanical Control Unit



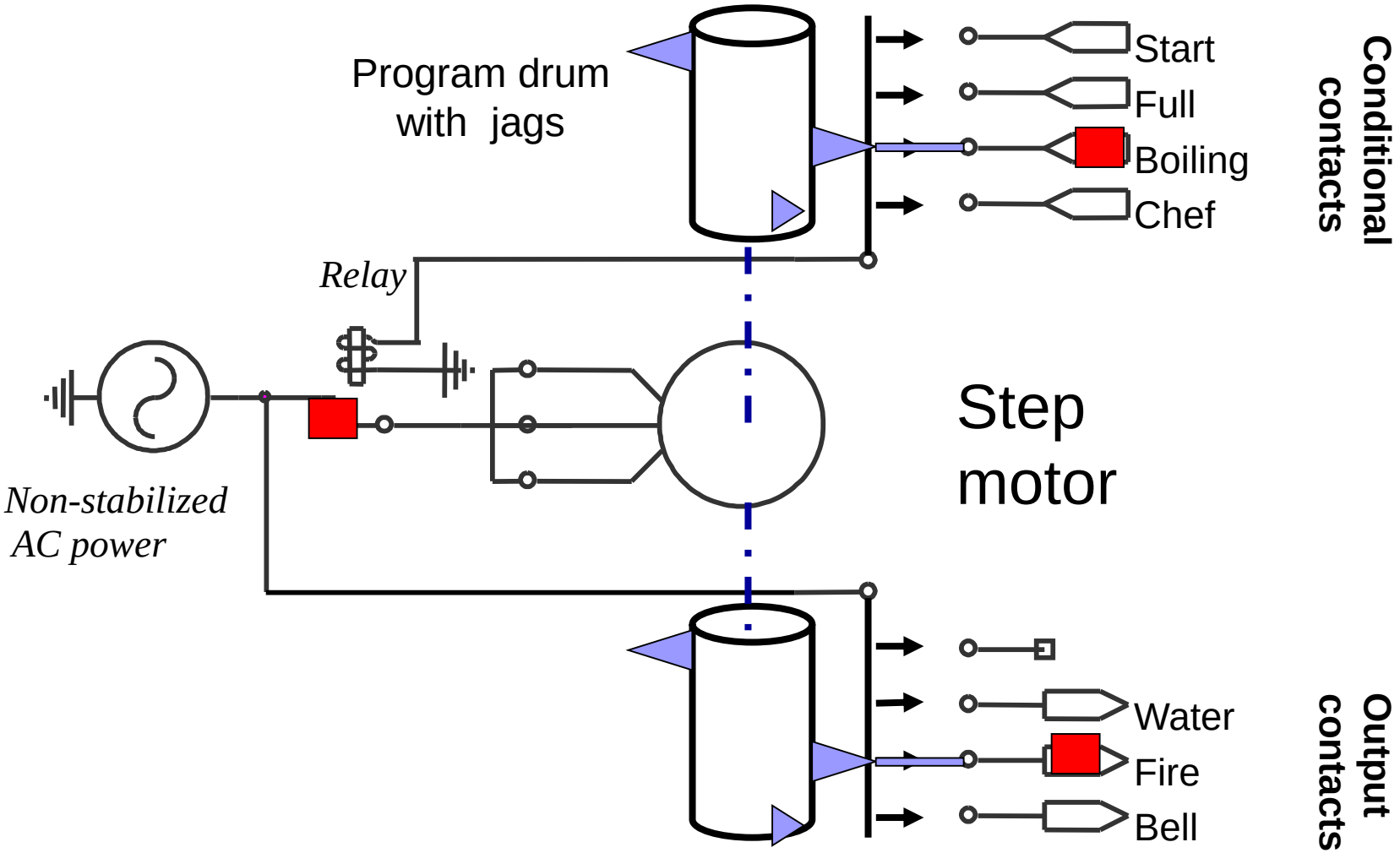
Animation: Electromechanical Control Unit 1/5



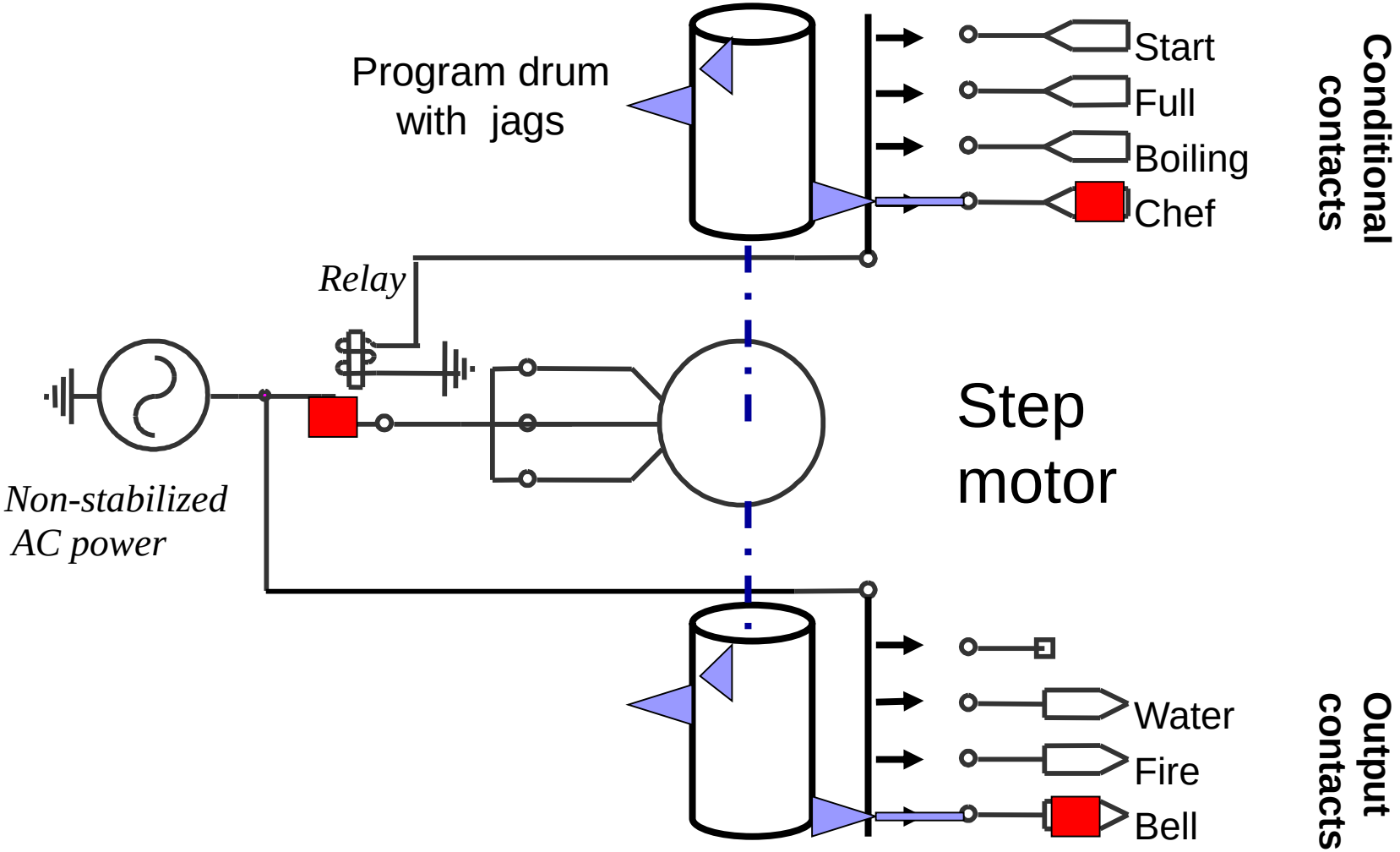
Animation: Electromechanical Control Unit 2/5



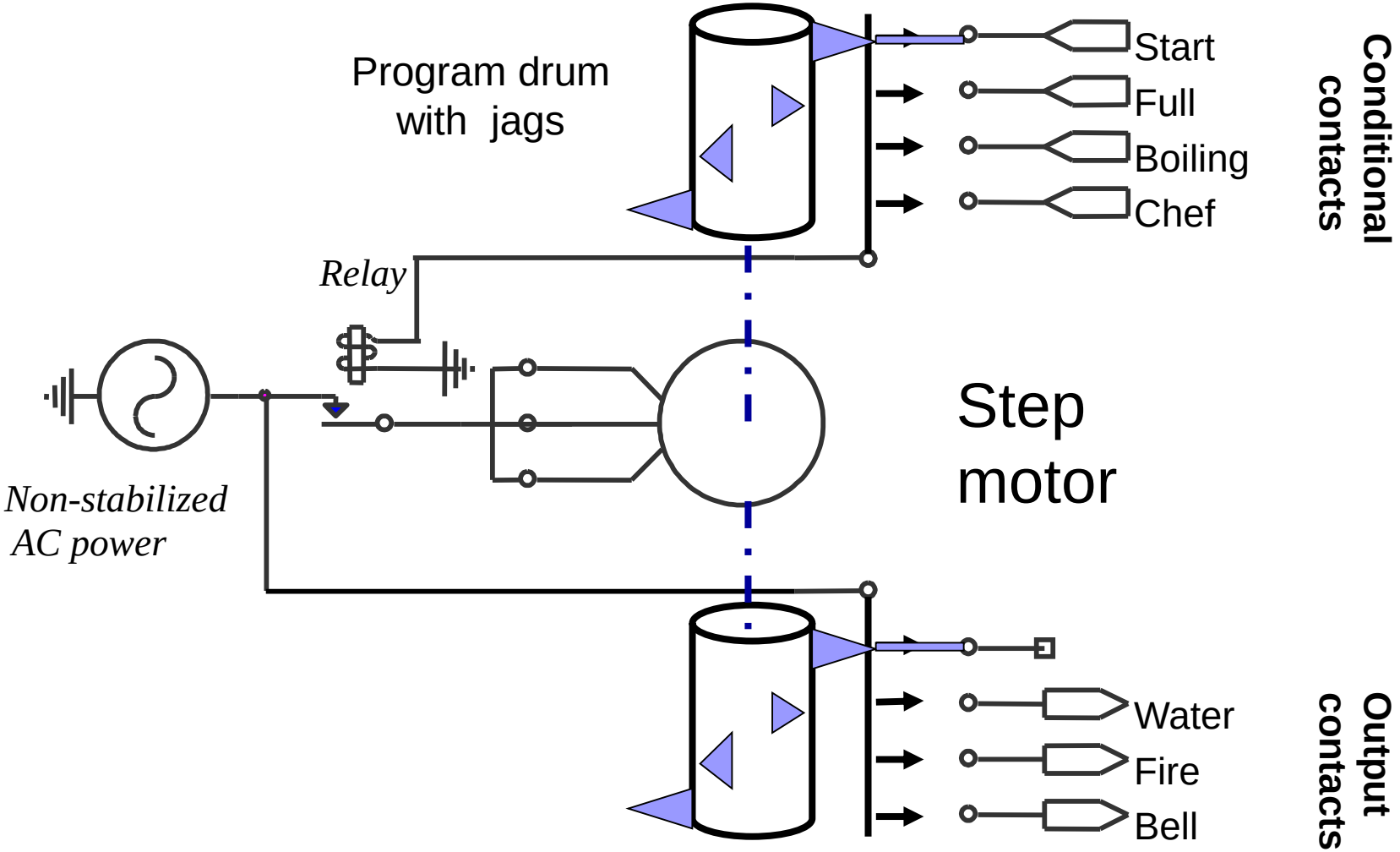
Animation: Electromechanical Control Unit 3/5



Animation: Electromechanical Control Unit 4/5

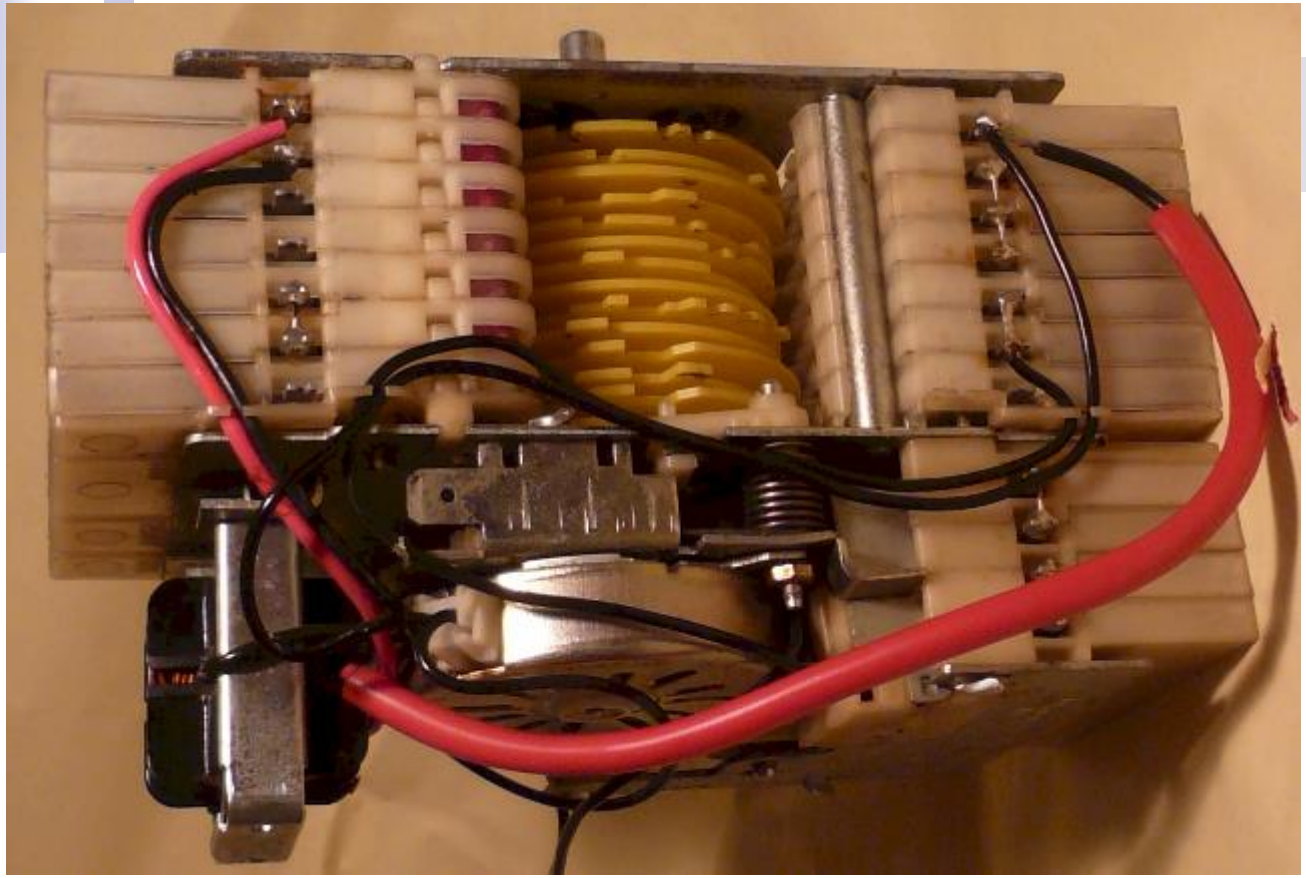


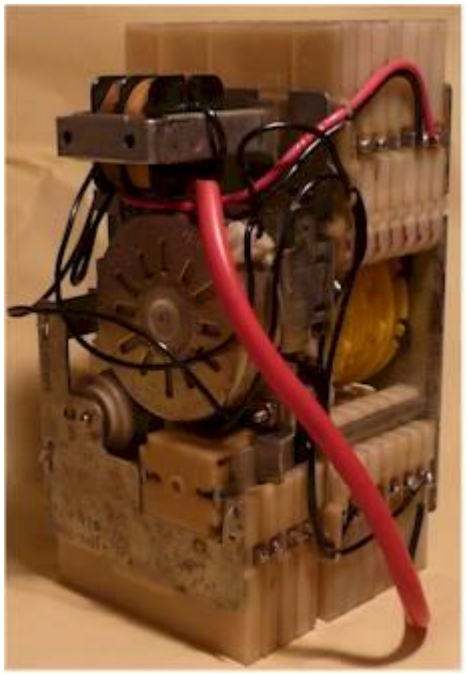
Animation: Electromechanical Control Unit 5/5

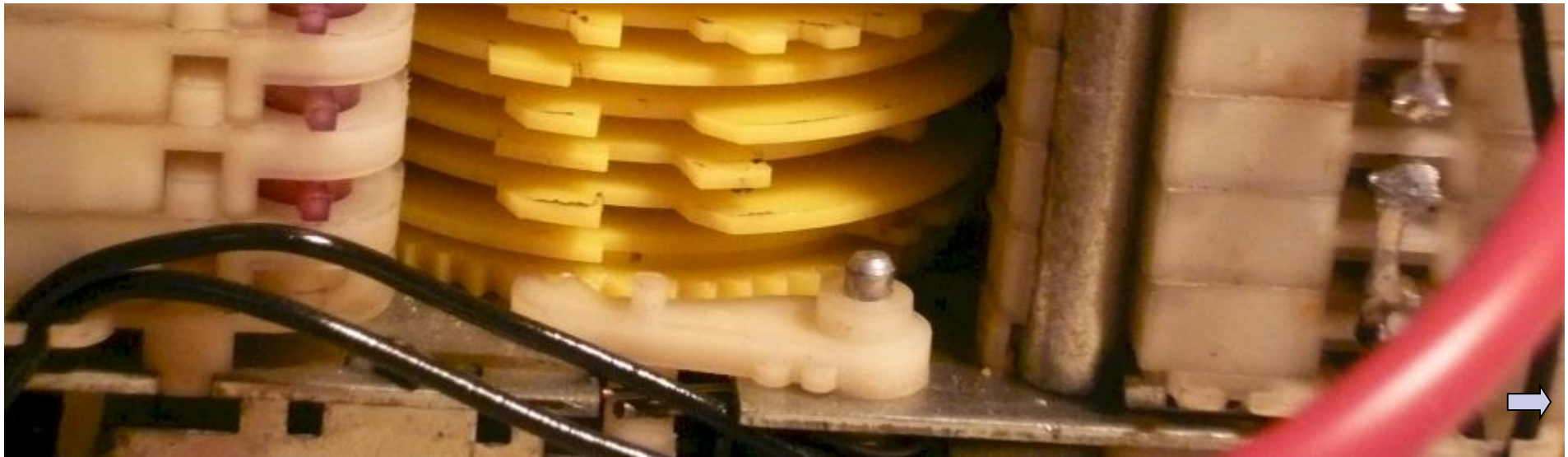


Control Unit

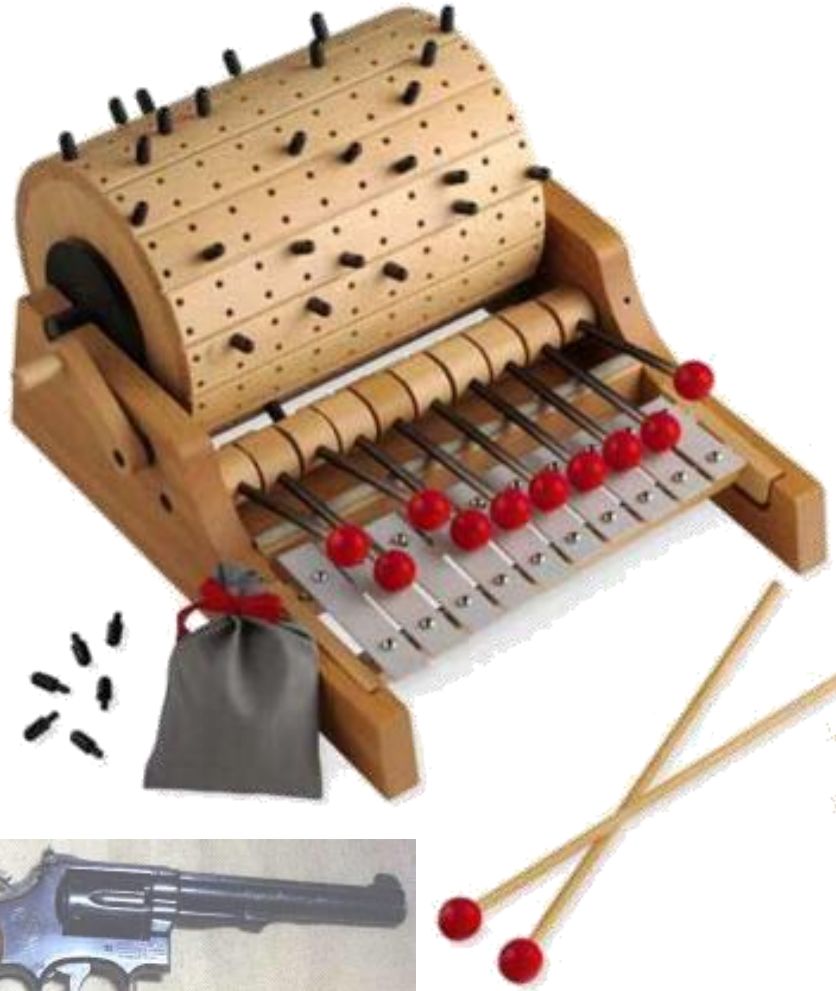
You know it...

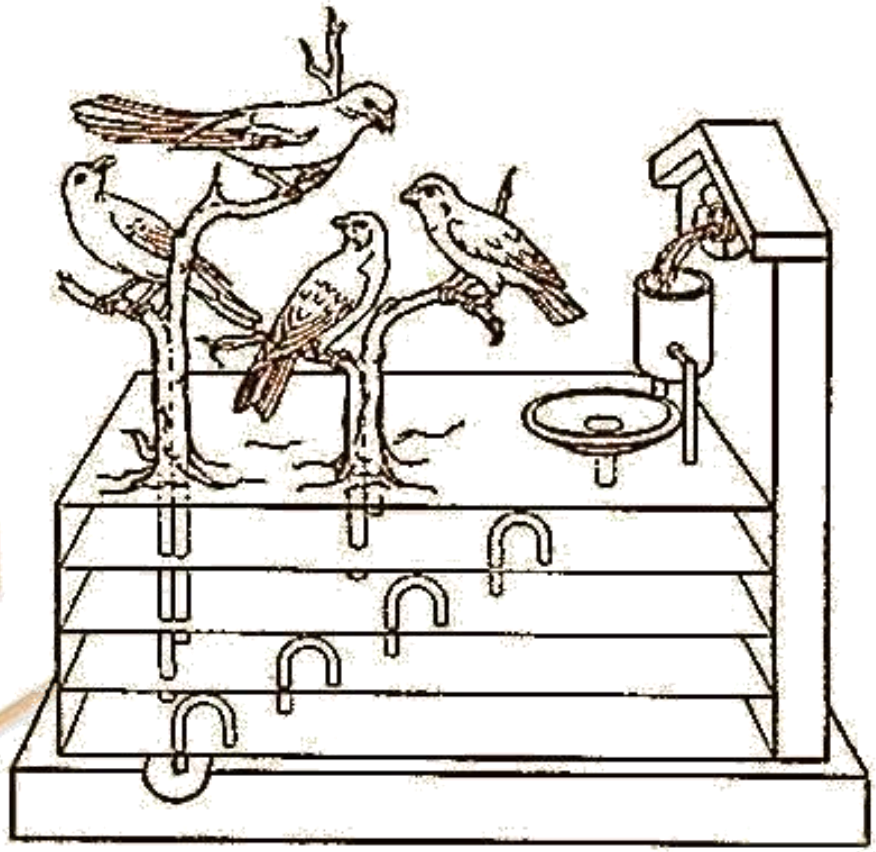






Control Units w/o Input Conditions





Heron of Alexandria, 1st century

Famous Control Unit from 19th century

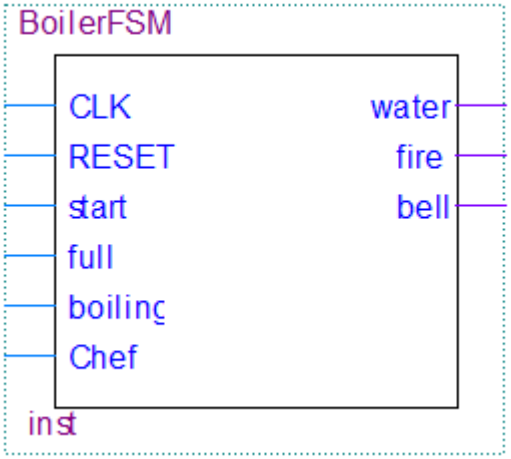
Boiler in VHDL

Key Code is Simple

```
case state is
   when S0 =>   if start = '1' then  state := S1;   end if;
   when S1 =>   if full = '1' then state := S2; end if;
   when S2 =>   if boiling = '1' then  state := S3; end if;
   when S3 =>   if Chef = '1' then state := S0; end if;
end case;
```

Boiler FSM - definitions (1/5)

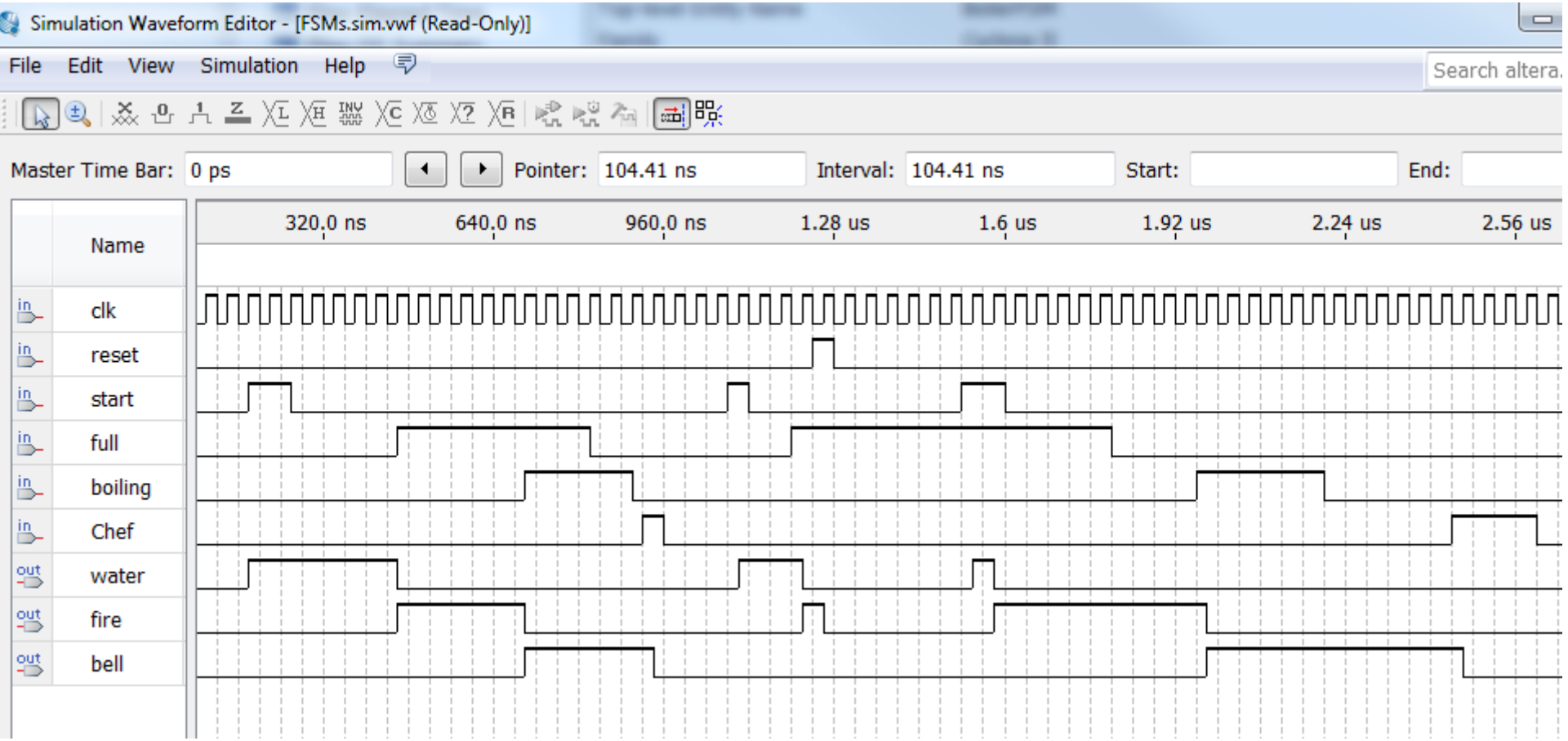
```
library ieee; use ieee.std_logic_1164.all;
entity BoilerFSM is
    port( CLK, RESET: in std_logic;
          start, full, boiling, Chef : in std_logic;
          water, fire, bell : out std_logic);
end;
architecture rtl of BoilerFSM is
begin
 iboiler : process (CLK)
          type state_t is (S0, S1, S2, S3);
          variable state : state_t:=S0;
          begin


          end process;
end architecture;
```



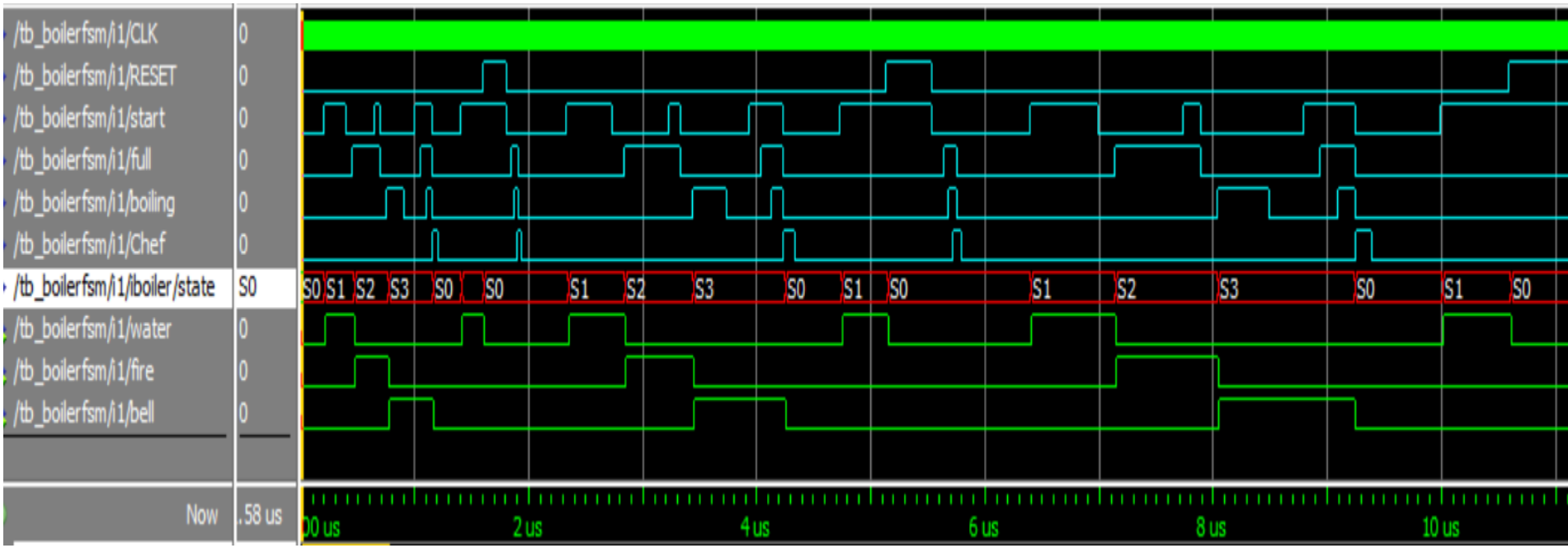
Boiler FSM - Process (2/5)

```
iboiler : process (CLK)
        type state_t is (S0, S1, S2, S3);
       variable state : state_t:=S0;
       begin
          if rising_edge(clk) then
            if RESET='1' then  state := S0;
            else
              case state is
                when S0 =>   if start = '1' then  state := S1;   end if;
                when S1 =>   if full = '1' then state := S2; end if;
                when S2 =>   if boiling = '1' then  state := S3; end if;
                when S3 =>   if Chef = '1' then state := S0; end if;
                when others => report "Reach undefined state" severity error; -- safety check
              end case;
          end if;
      end if; -- rising_edge(clk)
    case state is
      when S0=> water <= '0'; fire<='0'; bell <= '0';  when S1=> water <= '1'; fire<='0'; bell <= '0';
      when S2=> water <= '0'; fire<='1'; bell <= '0';  when S3=> water <= '0'; fire<='0'; bell <= '1';
    end case;
end process; end architecture;
```

Boiler FSM - Simulation (3/5)

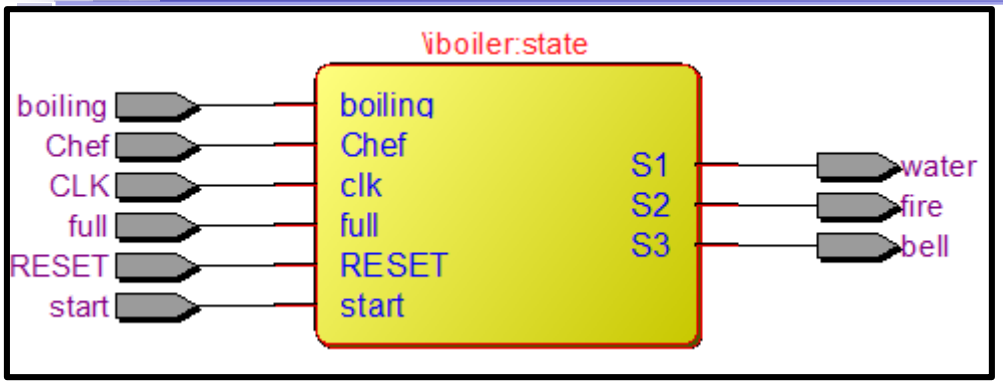


Boiler FSM (4/5)

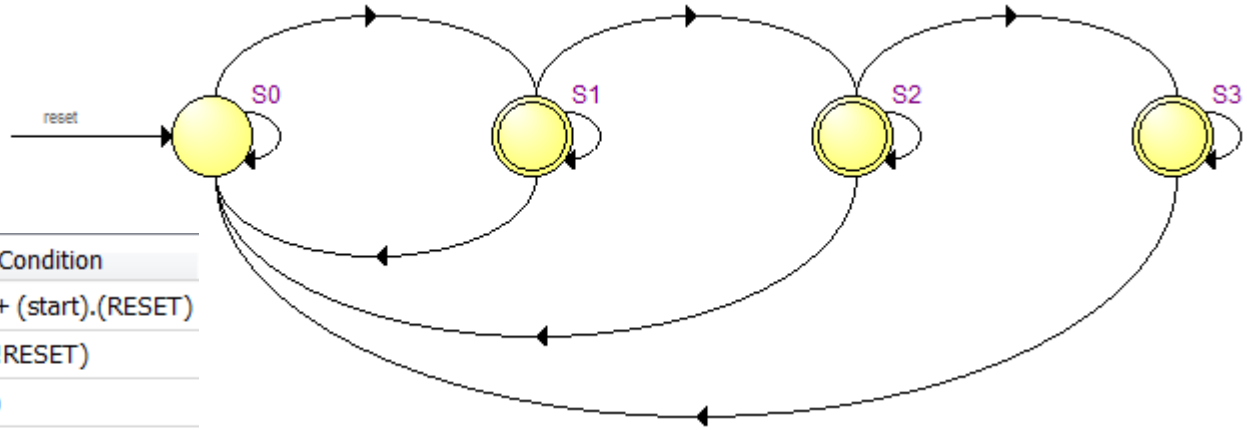


ModelSim shows also internal variables

RTL Viewer: Boiler as FSM (5/5)



We see properly designed FSM RTL Viewer as yellow rectangles. After double-clicking, they are opened in "State Machine Viewer".



	Source State	Destination State	Condition
1	S0	S0	(!start) + (start).(RESET)
2	S0	S1	(start).(!RESET)
3	S1	S0	(RESET)
4	S1	S1	(!full).(!RESET)
5	S1	S2	(full).(!RESET)
6	S2	S0	(RESET)
7	S2	S2	(!boiling).(!RESET)
8	S2	S3	(boiling).(!RESET)
9	S3	S0	(!Chef).(RESET) + (Chef)
10	S3	S3	(!Chef).(!RESET)

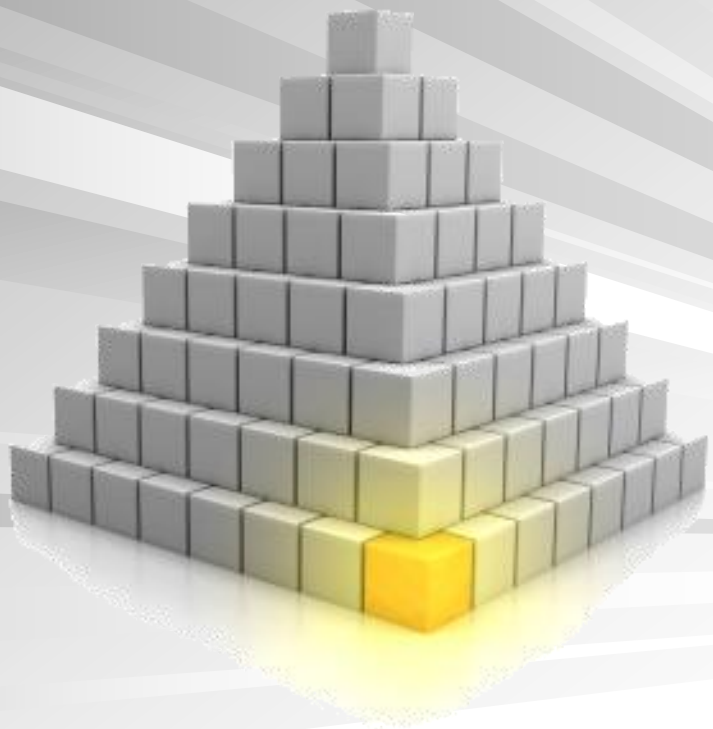
Transitions | Encoding

	Name	S3	S2	S1	S0
1	S0	0	0	0	0
2	S1	0	0	1	1
3	S2	0	1	0	1
4	S3	1	0	0	1

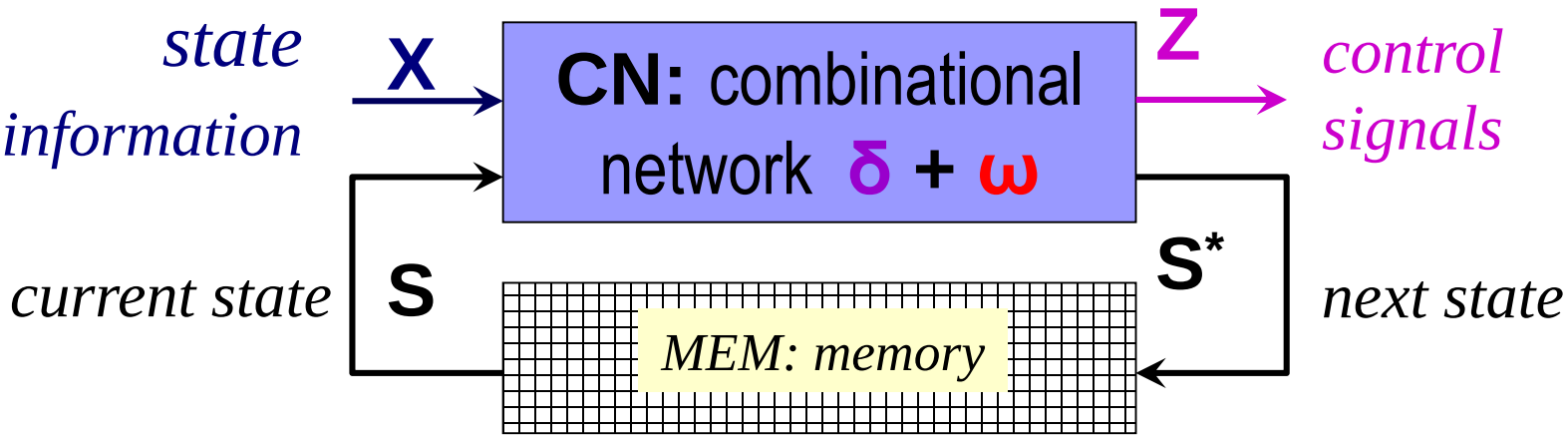
Transitions | Encoding

Control Units with Microcodes aka Micro-programmed Control Units

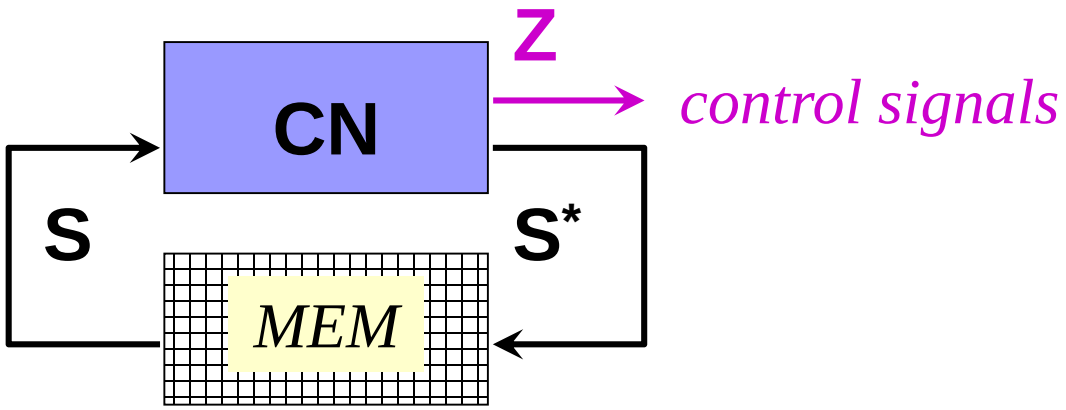
- Foundation (Corner) Stones of Computers



Control unit with state information

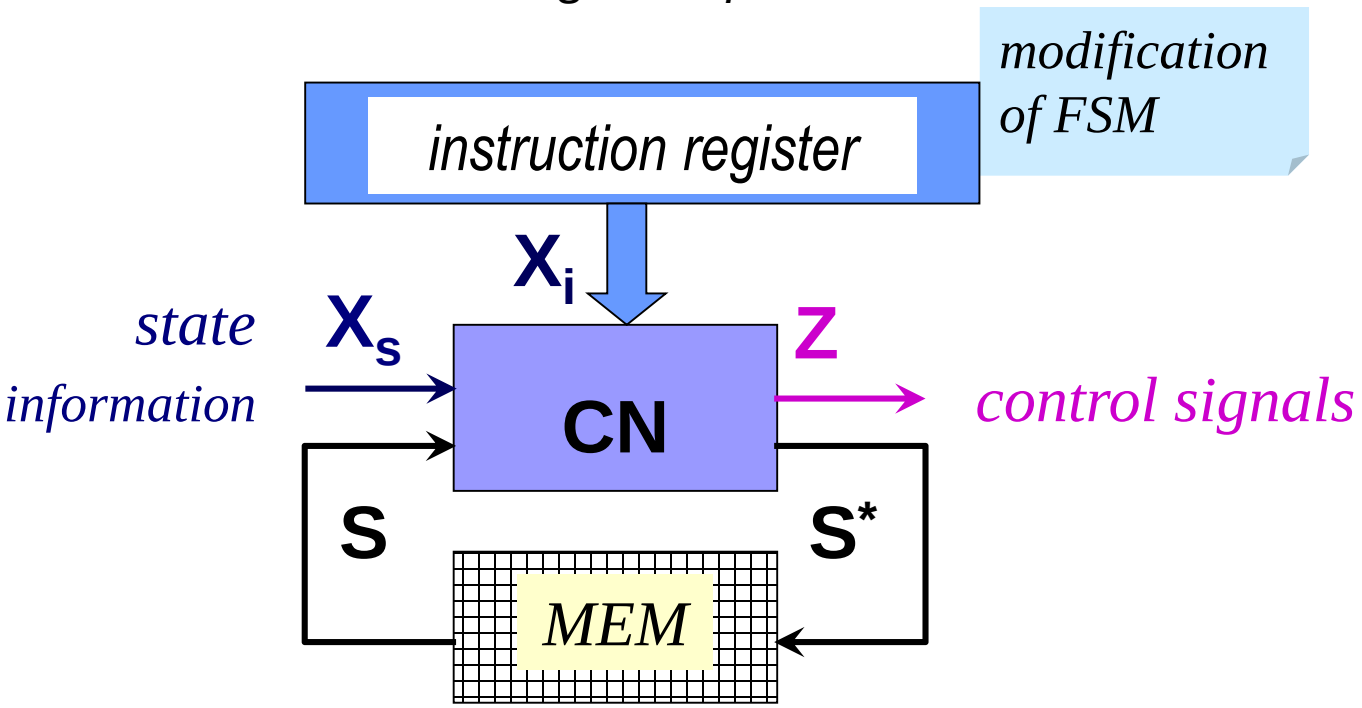


- *modification:* **autonomous control unit** – state information ***X***={ ∅ }

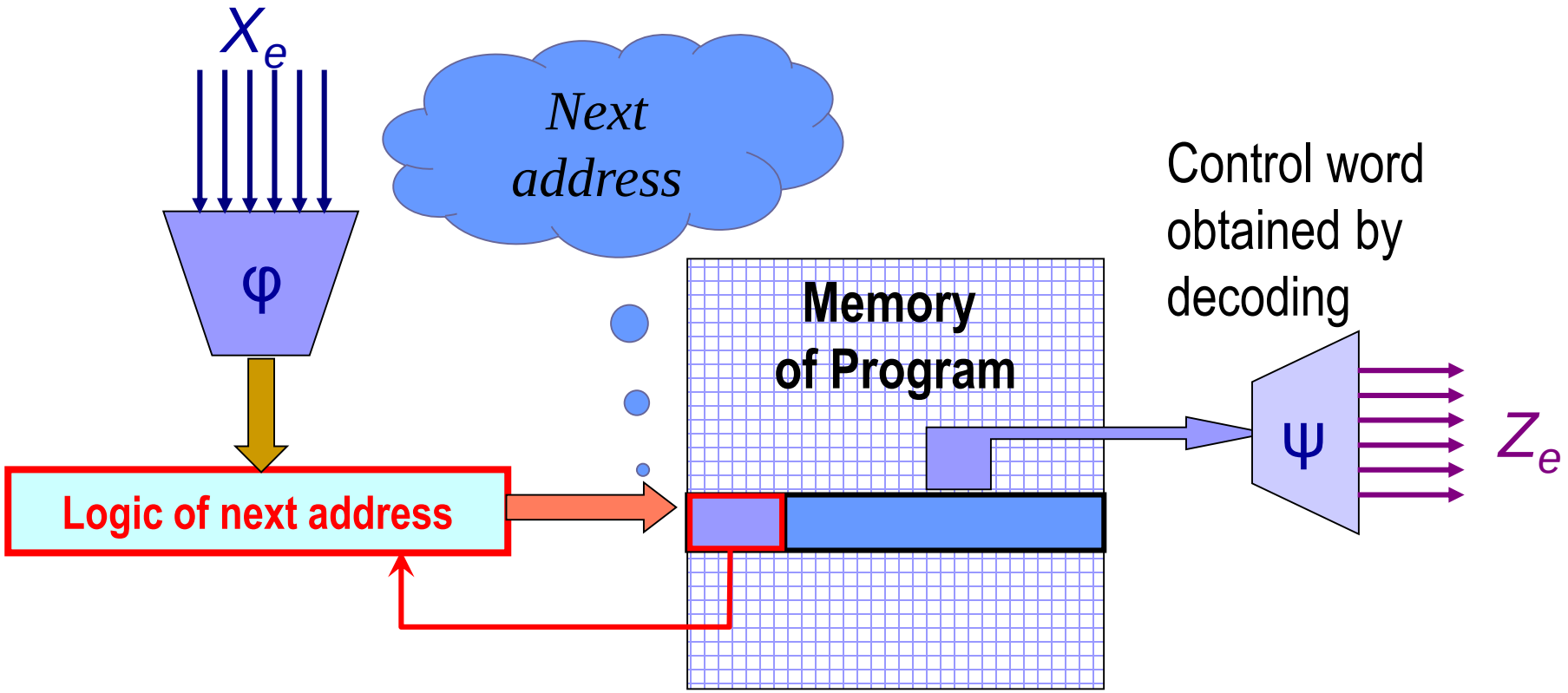


Control unit with instruction register

State information $X = X_s \cup X_i$

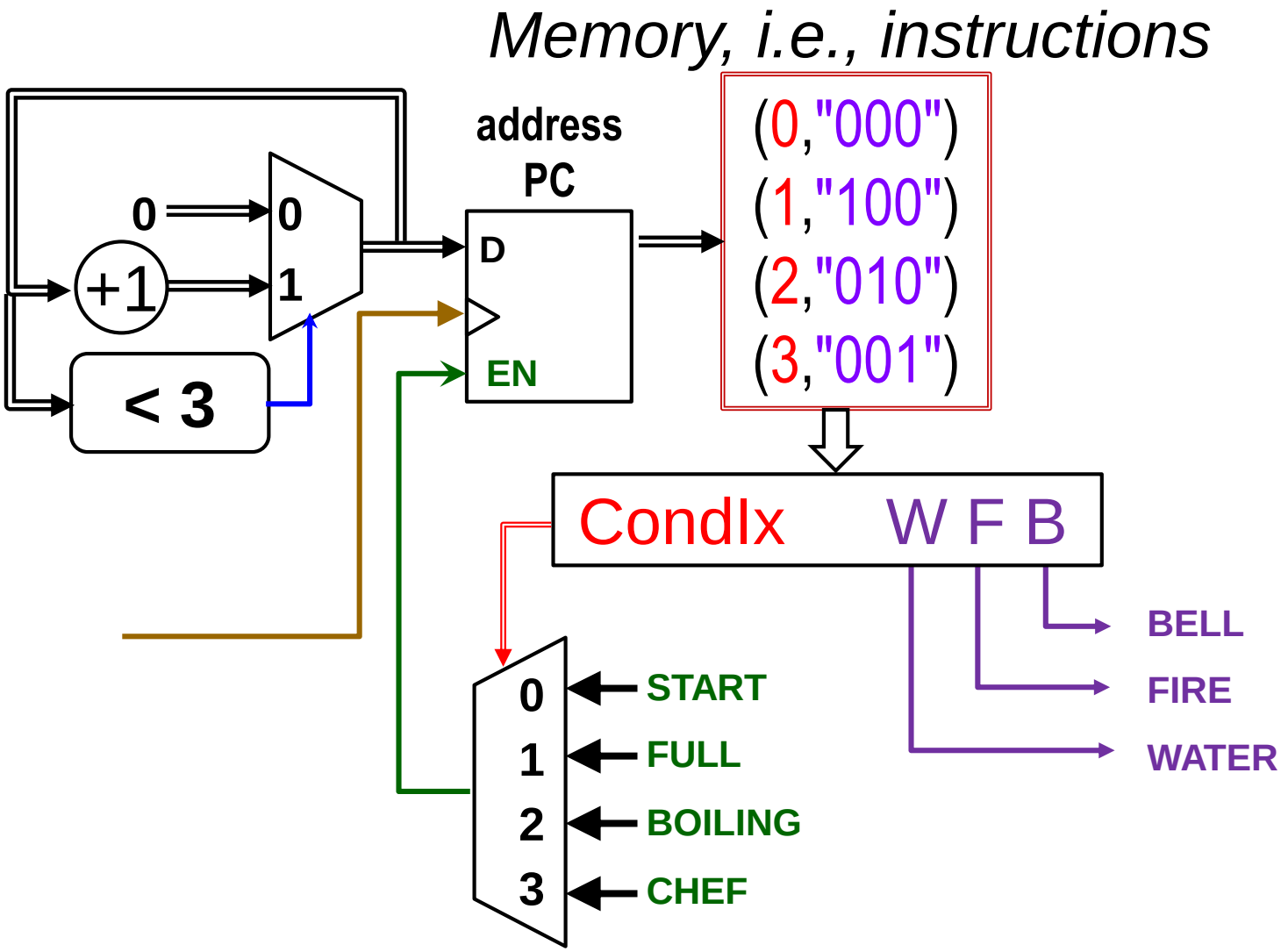


Control Unit with Micro-program



X_e are flags of ALU, status signals

Control Unit of Boiler



Key Code

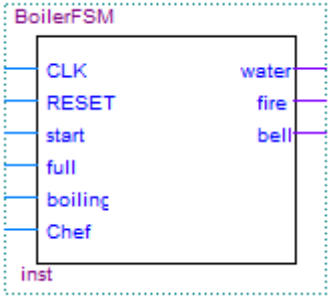
```
type cmd_t is record
    CondIx: integer range 0 to 3;  -- index of input
    WFB: std_logic_vector(0 to 2); -- Water, Fire, Bell values
end record;
type mem_t is array(natural range <>) of cmd_t;
constant mem : mem_t :=
          ((0,"000"),(1,"100"),(2,"010"),(3,"001"));
```

Key Code

```
instr:=mem(address);
BInputs:=start & full & boiling & Chef;
if BInputs(instr.CondIx)='1' then
     if address<mem'HIGH then address:=address+1;
                          else address:=0;
    end if;
end if;
-------------------------------
water<=instr.WFB(0); fire<=instr.WFB(1);
bell<=instr.WFB(2);
```

Control Unit Architecture of Boiler (1/4)

```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity BoilerFSM is
    port( CLK,RESET: in std_logic;
          start, full, boiling, Chef : in std_logic;
          water, fire, bell : out std_logic);
end entity;
architecture controlUnit of BoilerFSM is
begin
iboiler : process (CLK)
          -- next page
          end process;  end architecture;
```



Control Unit Architecture of Boiler (2/4)

```
iboiler : process (CLK)
  type cmd_t is record
    CondIx: integer range 0 to 3;  -- index of input
    WFB: std_logic_vector(0 to 2); -- Water, Fire, Bell values
  end record;
  type mem_t is array(natural range <>) of cmd_t;
  constant mem : mem_t := ((0,"000"),(1,"100"),(2,"010"),(3,"001"));
  variable address : integer range 0 to mem'HIGH:=0;
  variable Binputs : std_logic_vector(0 to 3):=(others=>'0');
  variable instr:cmd_t:=(0,"000");
  begin
    -- next page
  end process;
```

Note: Unconstrained array

The specification "natural range <>" in the definition

```
type mem_t is array(natural range <>) of cmd_t;
```

creates an unconstrained array, i.e. of unspecified length.
An object (signal, variable or constant) of an unconstrained array type must have it's index type range defined when it is declared.

```
constant mem : mem_t := ((0,"000"),(1,"100"),(2,"010"),(3,"001"));
```

In the previous definition, the range is derived from the initialization.
The std_logic_vector type is also defined as the unconstrained array in library ieee.std_logic_1164:

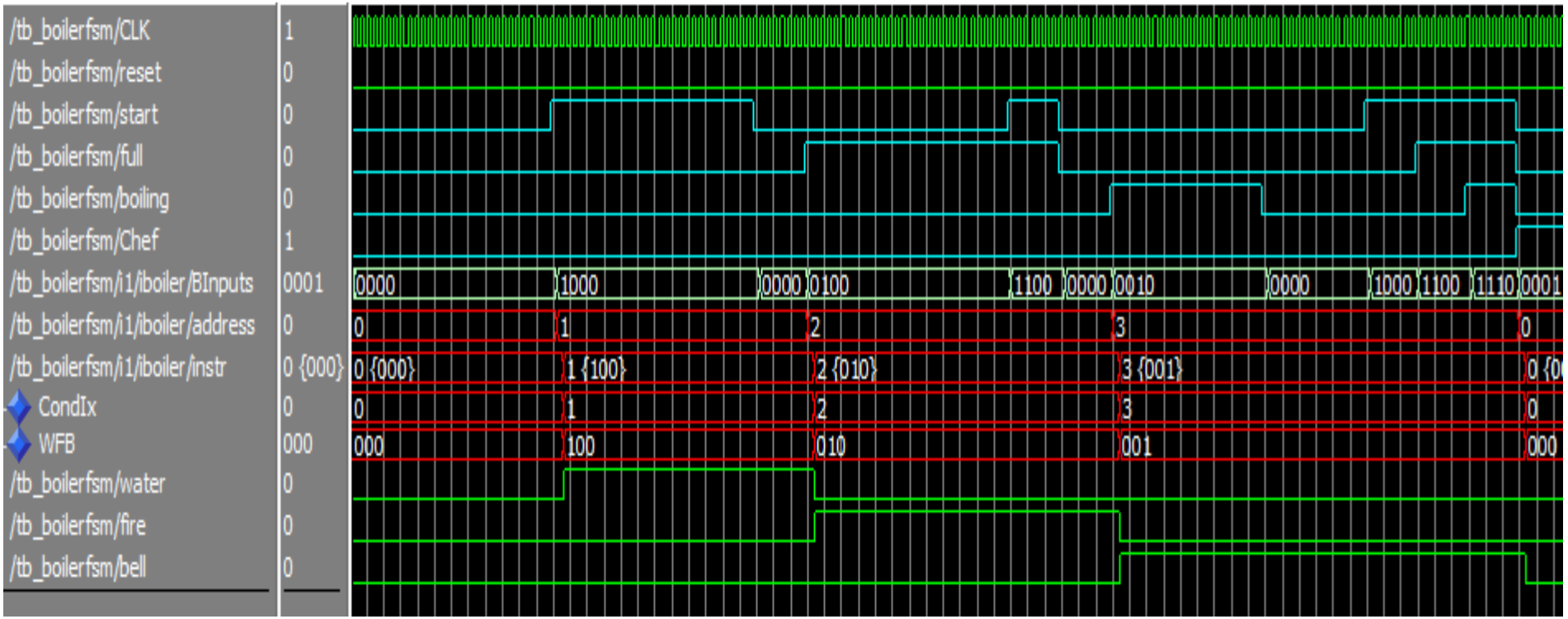
```
type std_logic_vector is array (natural range <>) of std_logic;
```

Starting with VHDL 2008, records may also contain unconstrained data types. In VHDL 1993, not yet.

Control Unit Architecture of Boiler (3/4)

```
begin -- process
  if rising_edge(CLK) then
        instr:=mem(address);
        BInputs:=start & full & boiling & Chef;
        if BInputs(instr.CondIx)='1' then
              if address<mem'HIGH then address:=address+1;
                                       else address:=0;
              end if;
        end if;
   end if;
// control signal - the output of control unit
 water<=instr.WFB(0); fire<=instr.WFB(1); bell<=instr.WFB(2);
 end process;
end architecture;
```

Control Unit Architecture of Boiler (4/4)



Unconstrained Arrays in Testbenches 1/3

```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all; library work;
entity tb_BoilerFSM is end entity;
architecture rtl OF tb_BoilerFSM IS
  signal reset, start, full, boiling, Chef, water, fire, bell:
std_logic:='0';
  signal CLK:std_logic:='0';
  component BoilerFSM is
   port( CLK, reset, start, full, boiling, Chef : in std_logic;
         water, fire, bell : out std_logic);
   end component;
```

Unconstrained Arrays in Testbenches 2/3

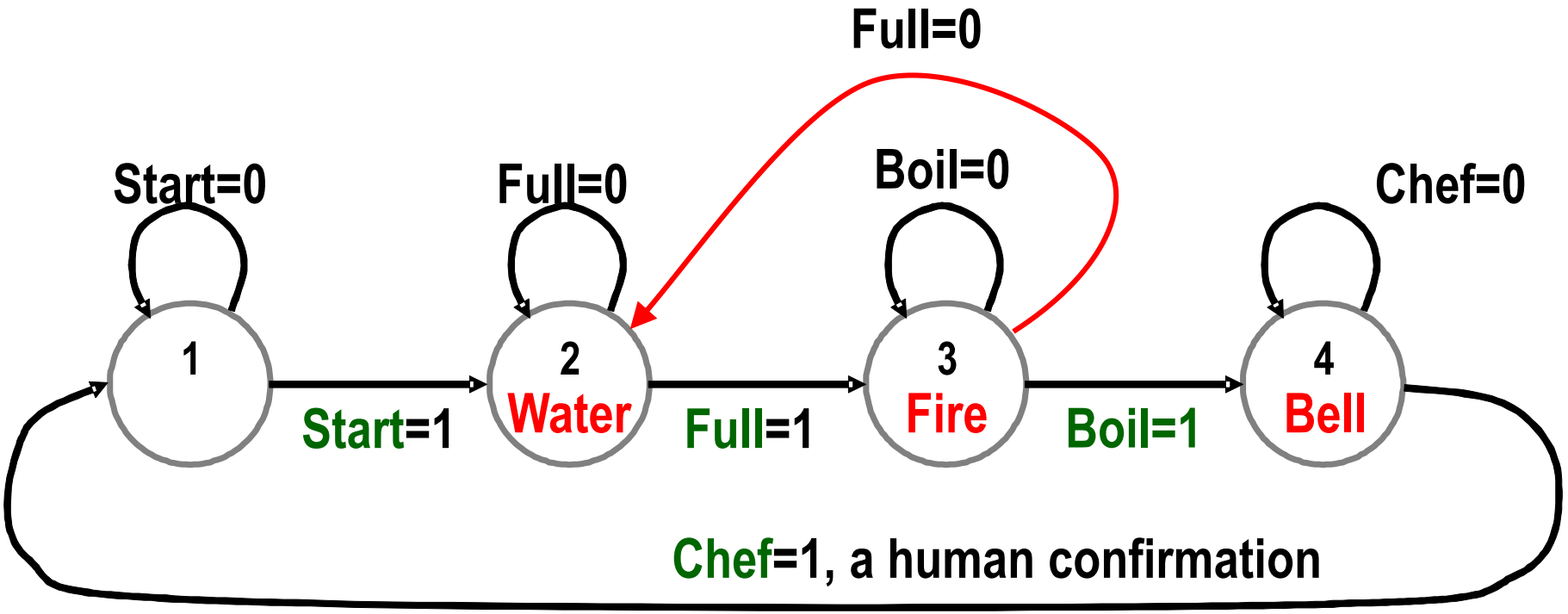
```
type R_t is record
      D:std_logic_vector(0 to 4);
      T:TIME;
end record;
type Stimul_t is array(natural range <>) of R_t;
constant STIMULS:Stimul_t:=
  (("00000",200 ns),("01000",200 ns), ("00000",50 ns),  ("00100",200 ns),
  ("01100",50 ns),   ("00000",50 ns),   ("00010",150 ns), ("00000",100 ns),
  ("01000",50 ns),   ("01100",50 ns),   ("01110",50 ns),    ("00001",50 ns),
  ("00000",200 ns), ("01000",200 ns), ("11000",200 ns), ("00000",50 ns),
  ("00100",20 ns),   ("00110",20  ns),    ("00111",20 ns),   ("00001",20 ns));
```

Unconstrained Arrays in Testbenches 3/3

```
begin - architecture
  CLK <= not CLK after 10 ns;
  i1 : entity work.BoilerFSM (controlUnit)
        port map(CLK, reset, start, full, boiling, Chef, water, fire, bell);
 always : process
   variable x:std_logic_vector( STIMULS(0).D'RANGE );
   begin
   irep: for j in 1 to 3 loop
     iloop: for ix in 0 to STIMULS'LENGTH-1 loop
          x:=STIMULS(ix).D;
          reset<=x(0); start<=x(1);full<=x(2);boiling <=x(3);Chef<=x(4);
         wait for j*STIMULS(ix).T;
     end loop iloop;
   end loop irep;
   assert false  report "End of simulation" severity failure;
end process; end architecture;
```

Task: New Requirements

We now require always full water during heating.



We can design this boiler also as a control unit by adding some other conditions into our microcode, but FSMs are more suitable here. Control units best solve tasks in which they always move to the next state.

New Boiler FSM

```
iboiler : process (CLK)
        type state_t is (S0, S1, S2, S3);   variable state : state_t:=S0;
       begin
          if rising_edge(clk) then
           if RESET='1' then  state := S0;
           else
             case state is
               when S0 =>   if start = '1' then  state := S1;   end if;
               when S1 =>   if full = '1' then state := S2; end if;
               when S2 =>   if boiling = '1' then state := S3;
                            elsif full = '0' then  state := S1; end if;
               when S3 =>   if Chef = '1' then state := S0; end if;
               when others => report "Reach undefined state" severity error; -- safety check
             end case;
          end if;
       end if; -- rising_edge(clk)
    case state is
       when S0=> water <= '0'; fire<='0'; bell <= '0';  when S1=> water <= '1'; fire<='0'; bell <= '0';
       when S2=> water <= '0'; fire<='1'; bell <= '0';  when S3=> water <= '0'; fire<='0'; bell <= '1';
    end case;
end process; end architecture;
```

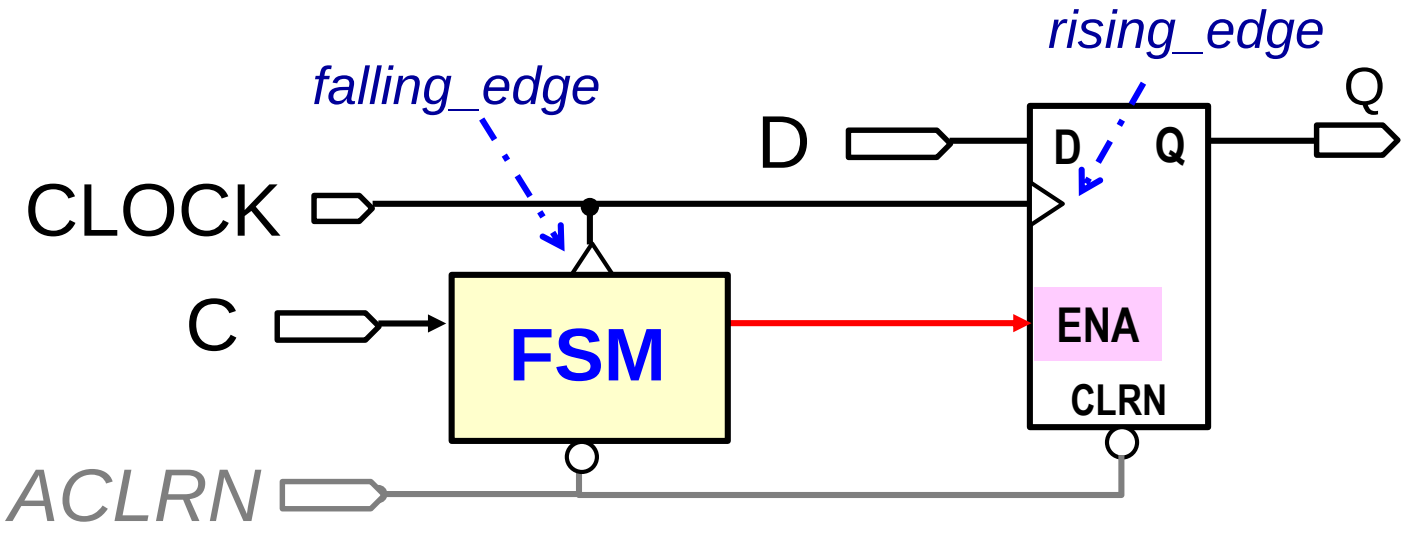
Task

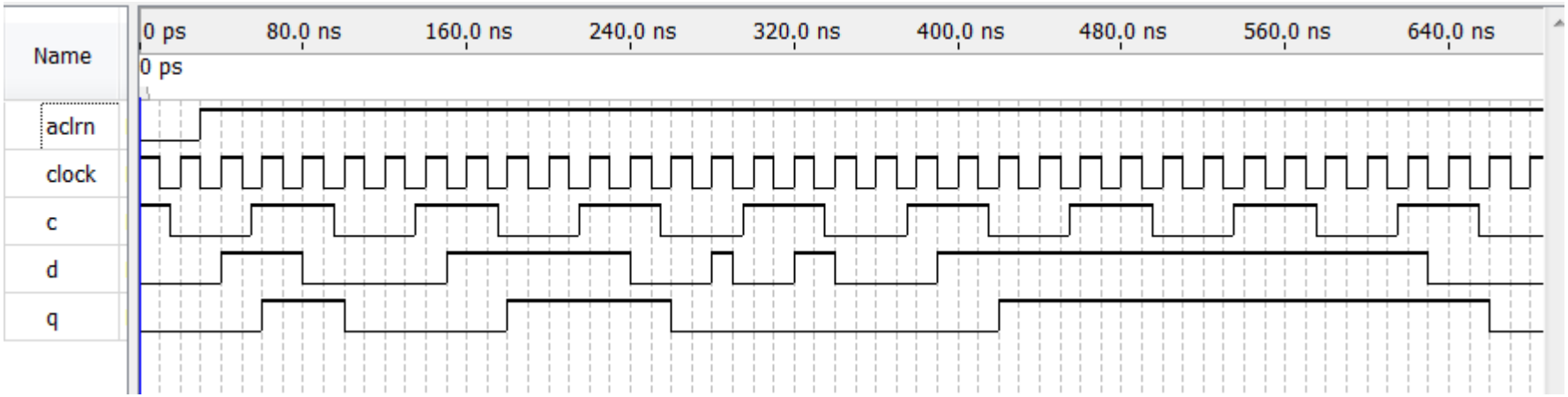
Sample D input on the rising and falling edges of C clock.

A direct solution would be complicated

- The number of 2 possible values of the C input can be multiplied by the number of D values up to 4 possible states.
- In Moore's automaton we also need to distinguish the values of the output, which is generated from the state, thus we have 8 possible combinations.
- To these we must add one more for initialization, so our solution can consider up to 9 states.
- Their number would be reduced during the proposal, but not in a transparent way.
- However, the task can be decomposed into a clearer FSM and DFF flapping circuit:
  - The first of these sets the ENA (enable) of the DFF flip-flop circuit at both the rising and falling edges of input C. But these are detected by comparing with their past values.
  - The FSM must operate on CLOCK clocks having a much higher frequency than C changes and also on the opposite edge.

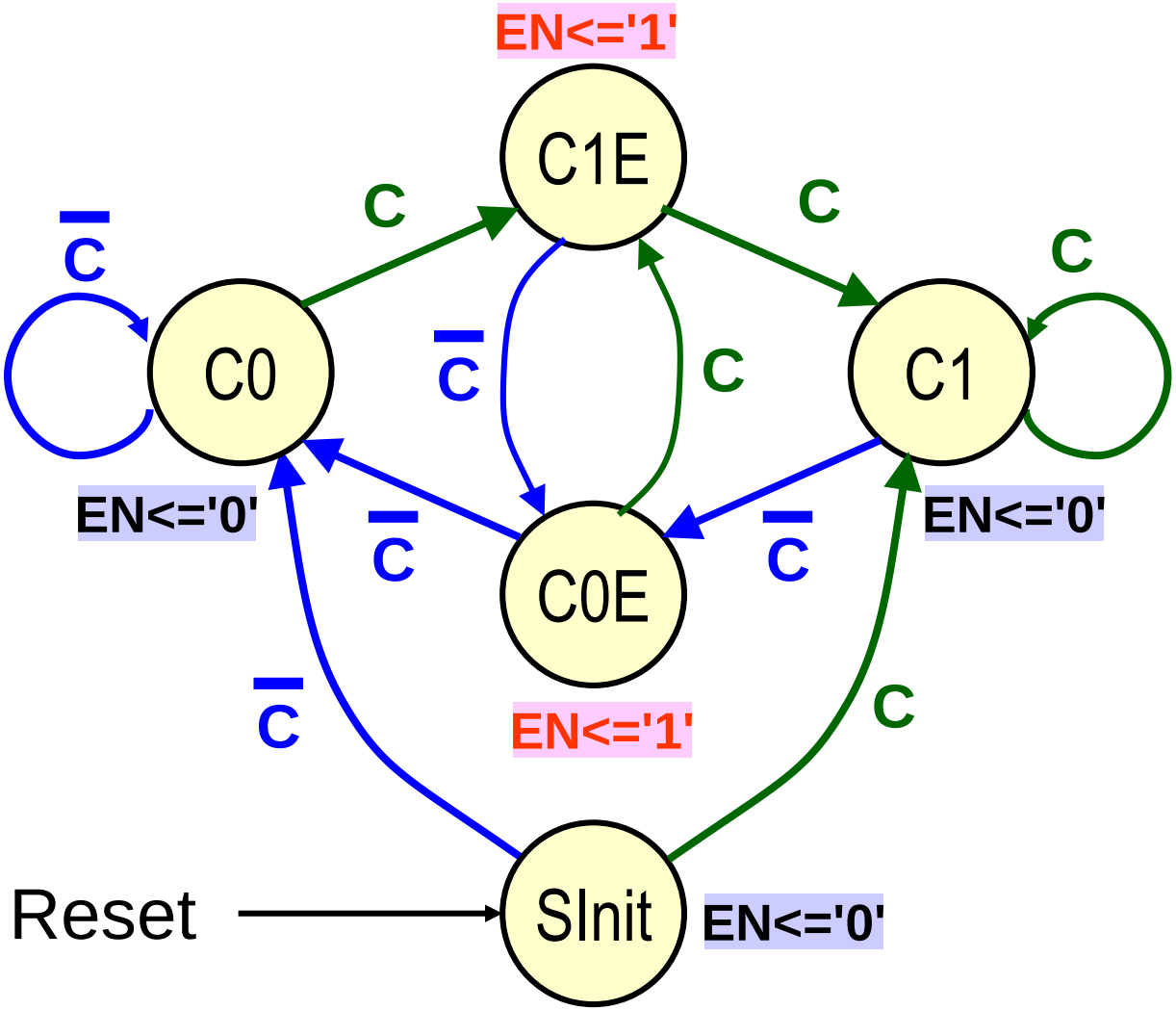
Task decomposition



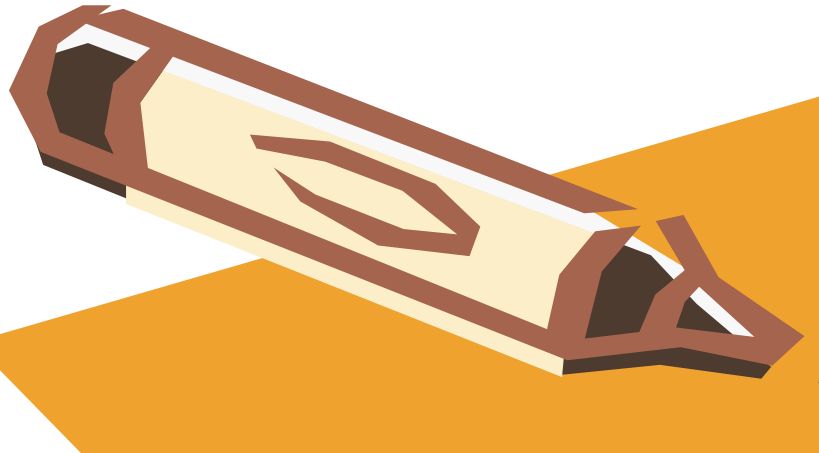


Moore's automaton

	C '0'	C '1'	EN
SInit	C0	C1	0
C0	C0	C1E	0
C1En	C0E	C1	**1**
C1	C0E	C1	0
C0En	C0	C1E	**1**



It can be simplified by converting it to a smaller Mealy automaton

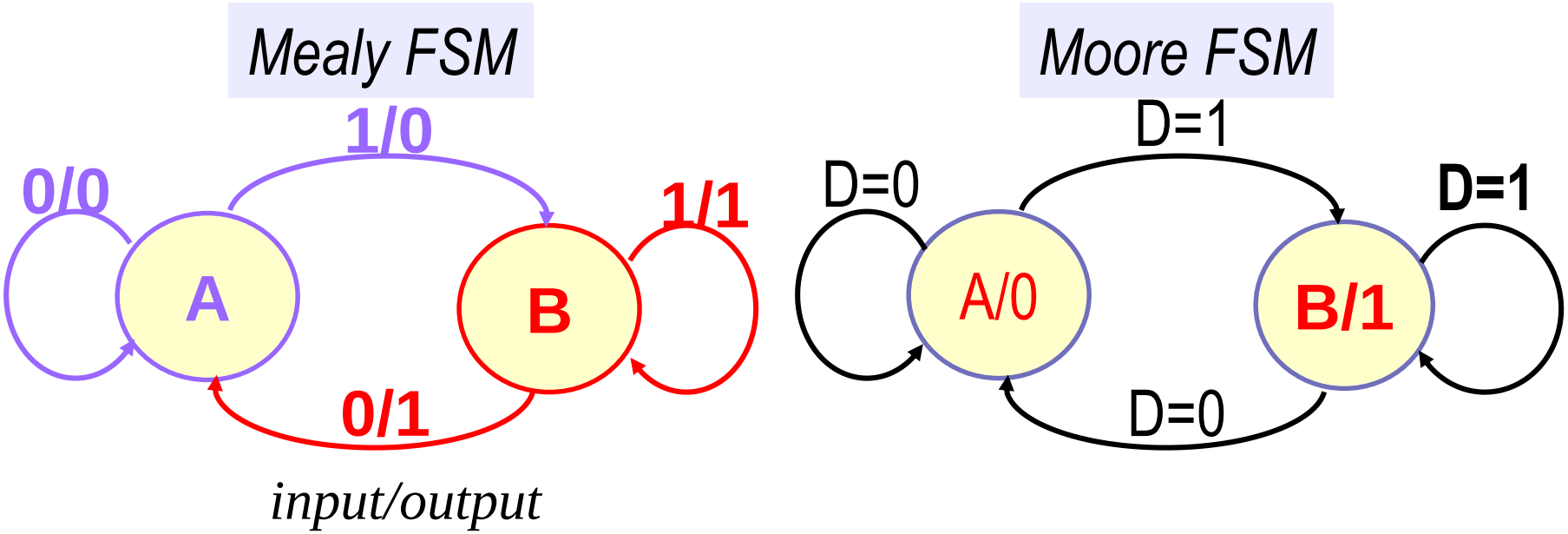


Moore vs Mealy

comparison

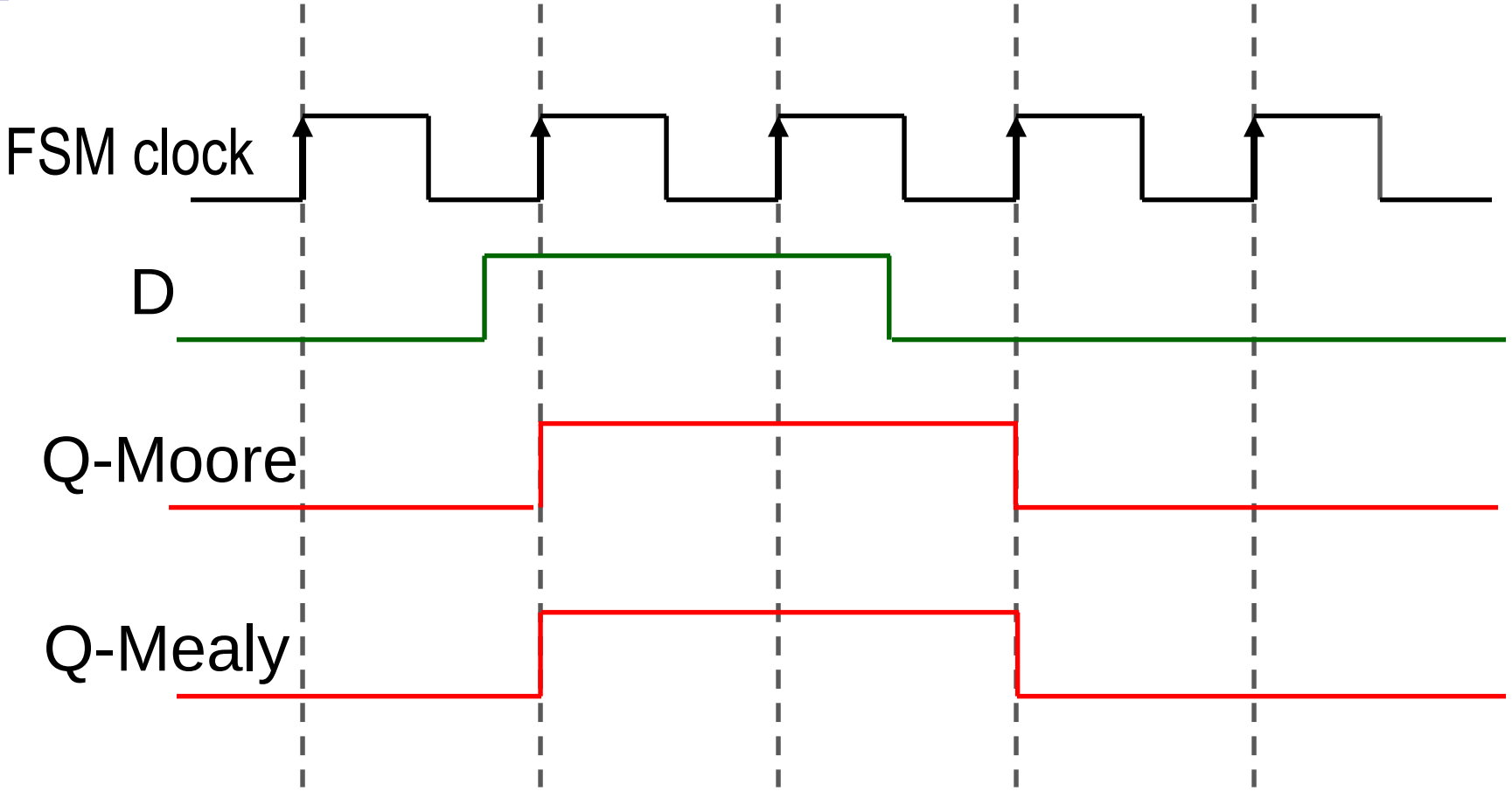
D-register - Mealy and Moore automaton

- Mealy: Output is a function of input and state.
- Moore: The output is only determined by the state.



They both behave identically because Mealy's automaton generates **the same outputs** in each state on all paths out of it.

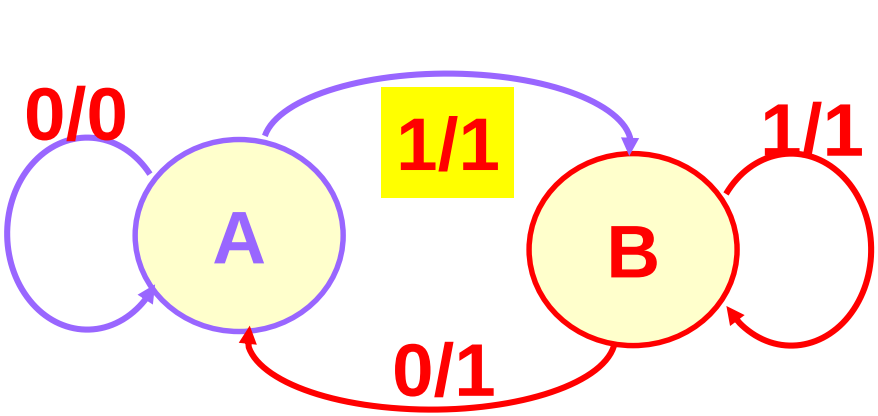
Moore & Mealy FSMs : D-register

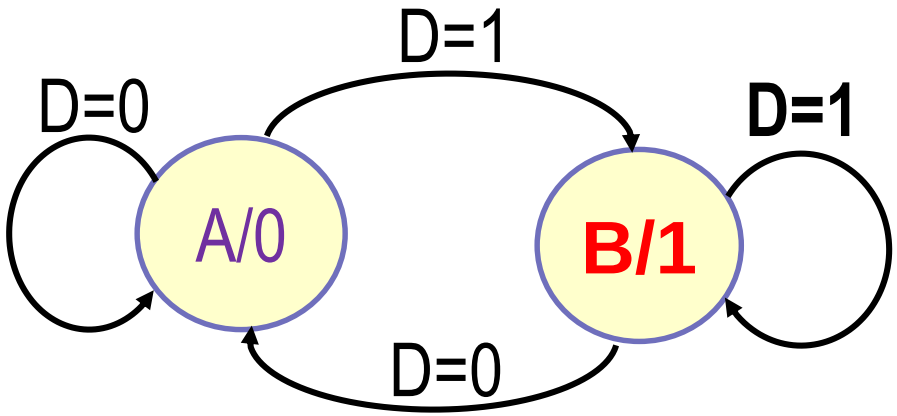


D-register - Mealy and Moore automaton

Mealy's machine can also predict

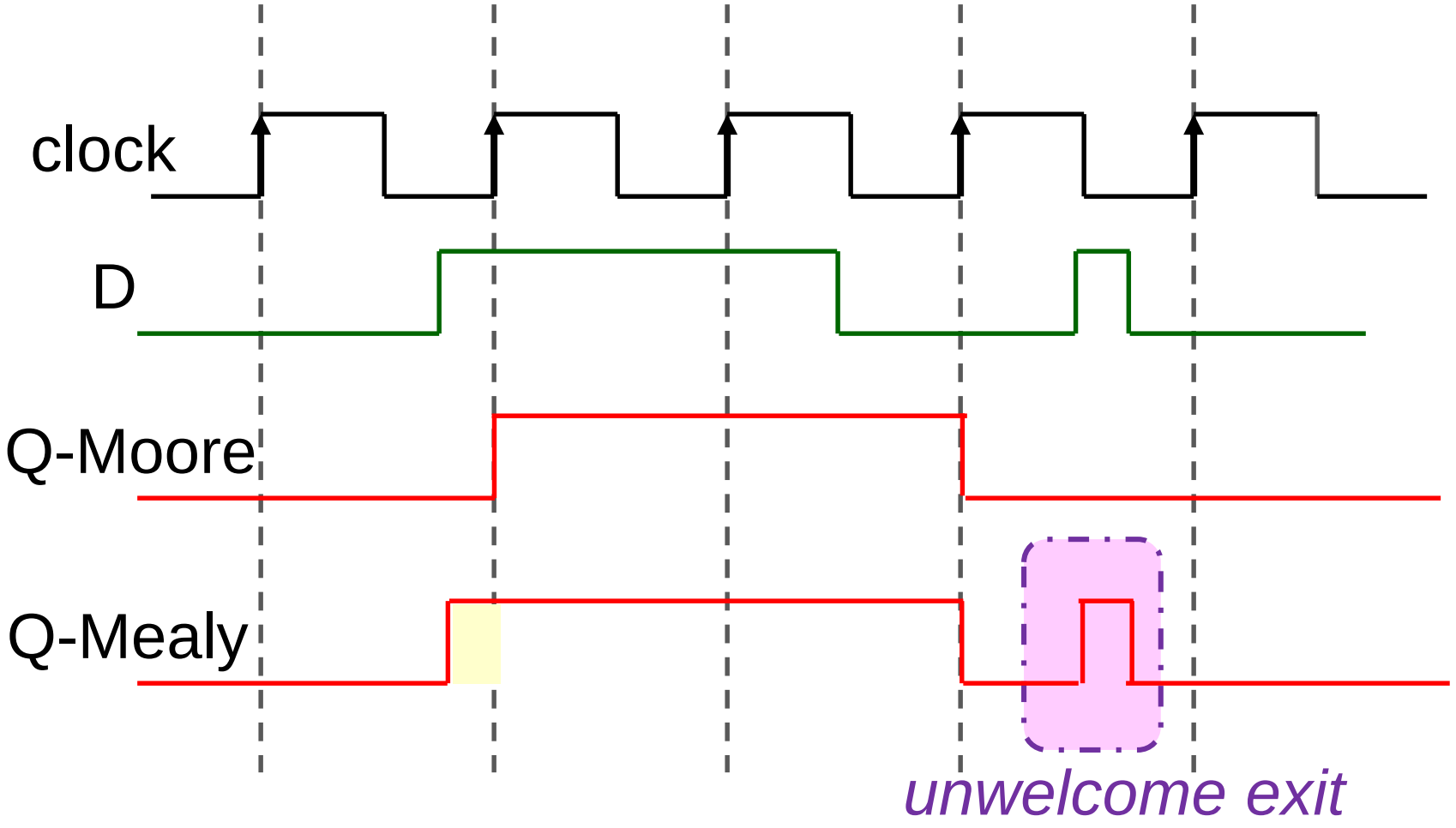
Moore's automaton is always without prediction





D-register Moore's automaton & Mealy's with prediction

The output of the Mealy automaton better always be synchronized in the circuit !



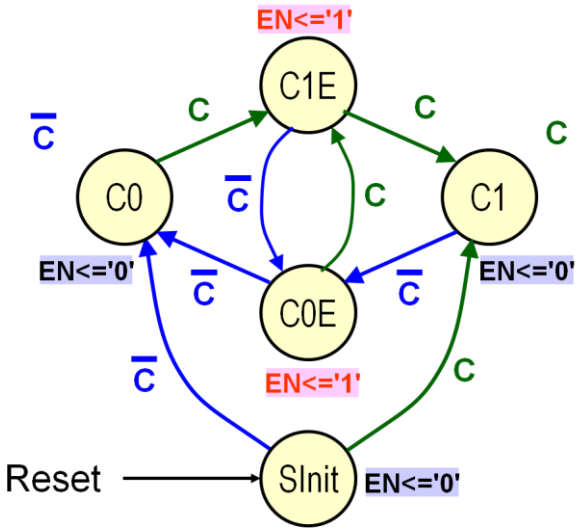
Moore's automaton

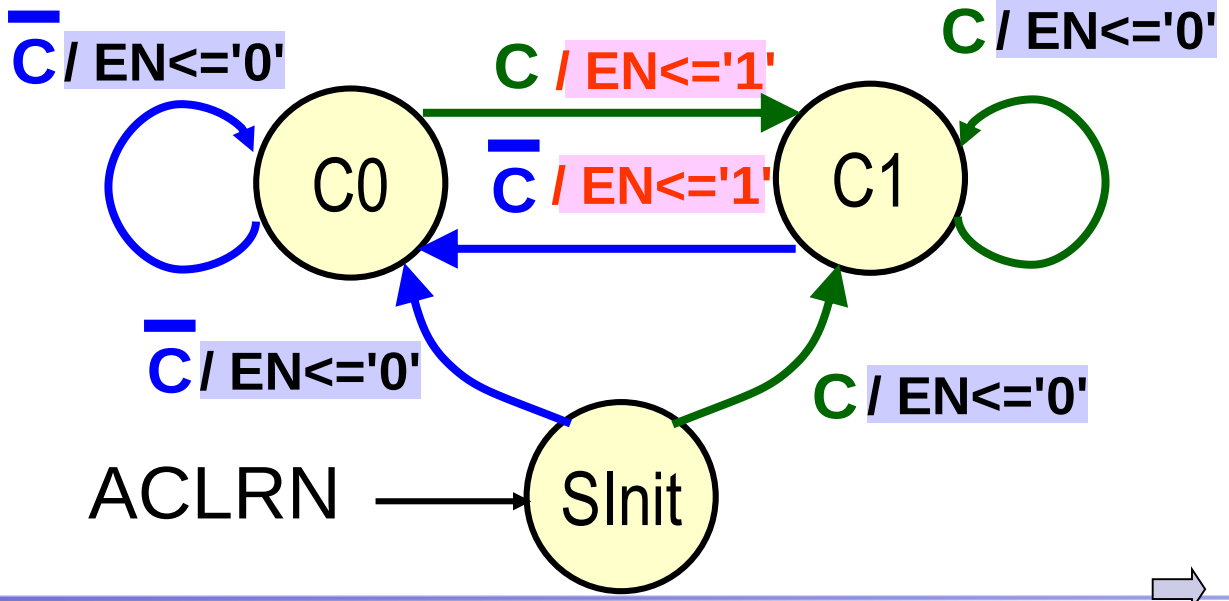
	C '0'	C '1'	EN
SInit	C0	C1	0
C0	C0	C1E	0
C1E	C0E	C1	**1**
C1	C0E	C1	0
C0E	C0	C1E	**1**

merge of states

Smaller Mealy's machine

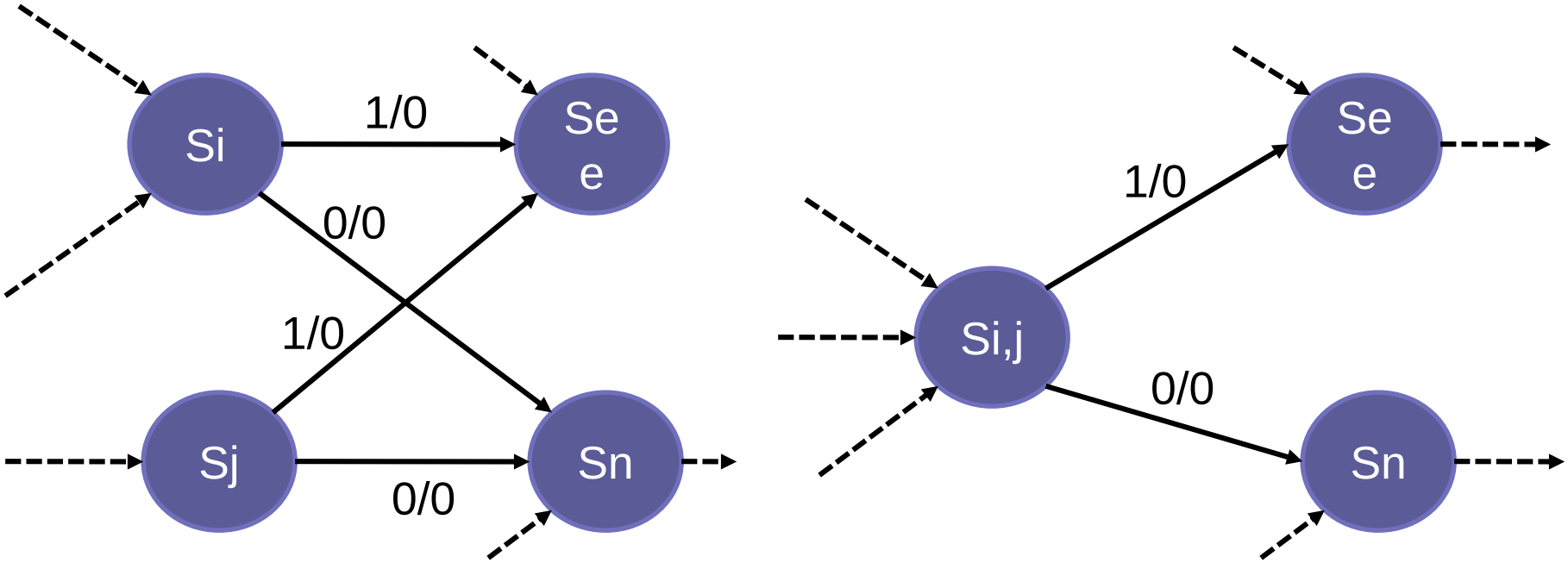
			EN	
	C '0'	C '1'	C '0	C '1'
SInit	C0	C1	0	
C0x	C0x	C1x	0	**1**
C1x	C0x	C1x	**1**	**0**





Equivalent States

- Two states of an FSM are *equivalent* (or *indistinguishable*) if for each input they produce the same output and their next states are identical.



Si *and* Sj *are equivalent and merged into a single state*.

```
case state is
   when SInit =>
      if C='1' then state:=C1x; else state:=C0x; end if;
   when C0x =>
      if C='1' then state:=C1x; m:=D; end if;
   when C1x =>
      if C='0' then state:=C0x; m:=D; end if;
end case;

-------------------------------------

Q<=m;
```

VHDL Mealy FSM

```
library ieee; use ieee.std_logic_1164.all;
entity RFE is
    port ( CLK, C, D, ACLRN : in std_logic;
           Q : out std_logic);
end entity;

architecture rtl of RFE is
begin
   process (CLK, ACLRN)
    -- next
  end process;
end architecture;
```

Code

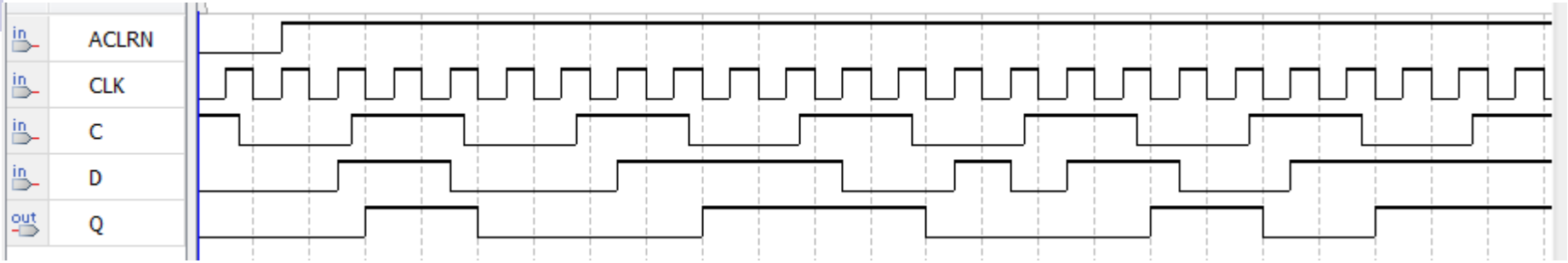
```
process (CLK, ACLRN)
type state_t is (SInit, C0x, C1x);
variable state : state_t;
variable m:std_logic;
begin
  if ACLRN='0' then state:=SInit; m:='0';
  elsif falling_edge(CLK) then
     case state is
      when SInit => if C='1' then state:=C1x; else state:=C0x; end if;
      when C0x => if C='1' then state:=C1x; m:=D; end if;
       when C1x => if C='0' then state:=C0x; m:=D; end if;
      end case;
  end if;
  Q<=m;
end process;
```

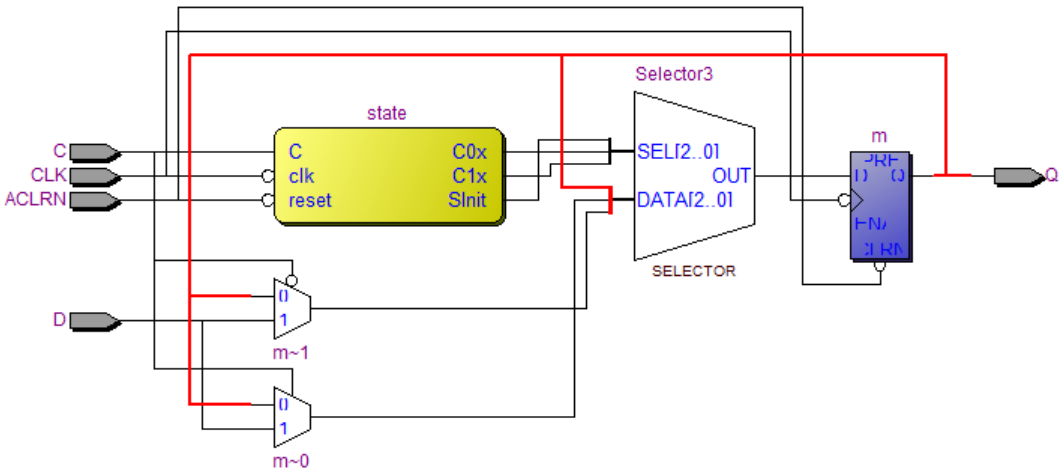
The variable m is not implemented by a register because it is always assigned a new value.

Function outline of the previous code

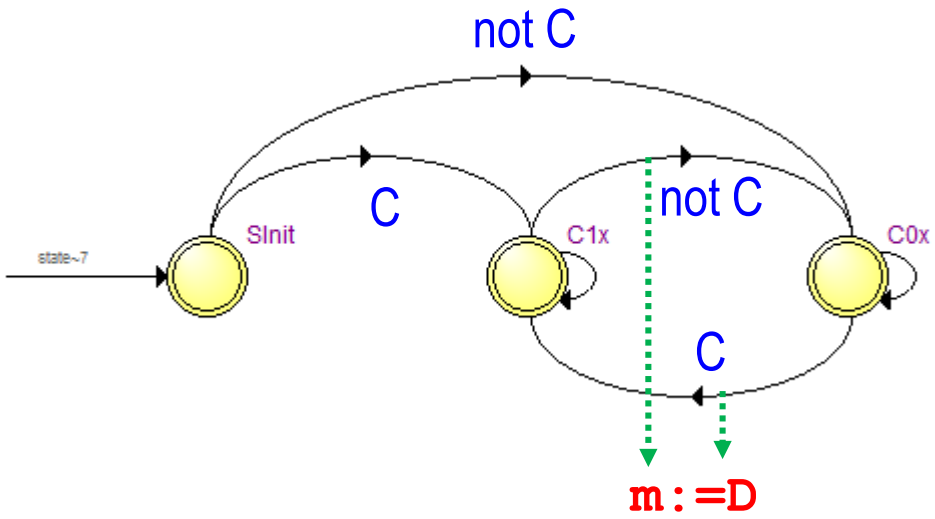
```
variable var_en:boolean;

-- ....
if ACLRN='0' then state:=SInit; m:='0';
elsif falling_edge(CLOCK) then
        var_en:=FALSE;
        case state is
           when SInit => if C='1' then state:=C1x; else state:=C0x; end if;
           when C0x => if C='1' then state:= C1x; var_en:=TRUE; end if;
           when C1x => if C='0' then state:= C0x; var_en:=TRUE; end if;
         end case;
         if var_en then d:=m; end if;
 end if;
```

Variable var_en is not implemented by the registry because it always gets a new value.





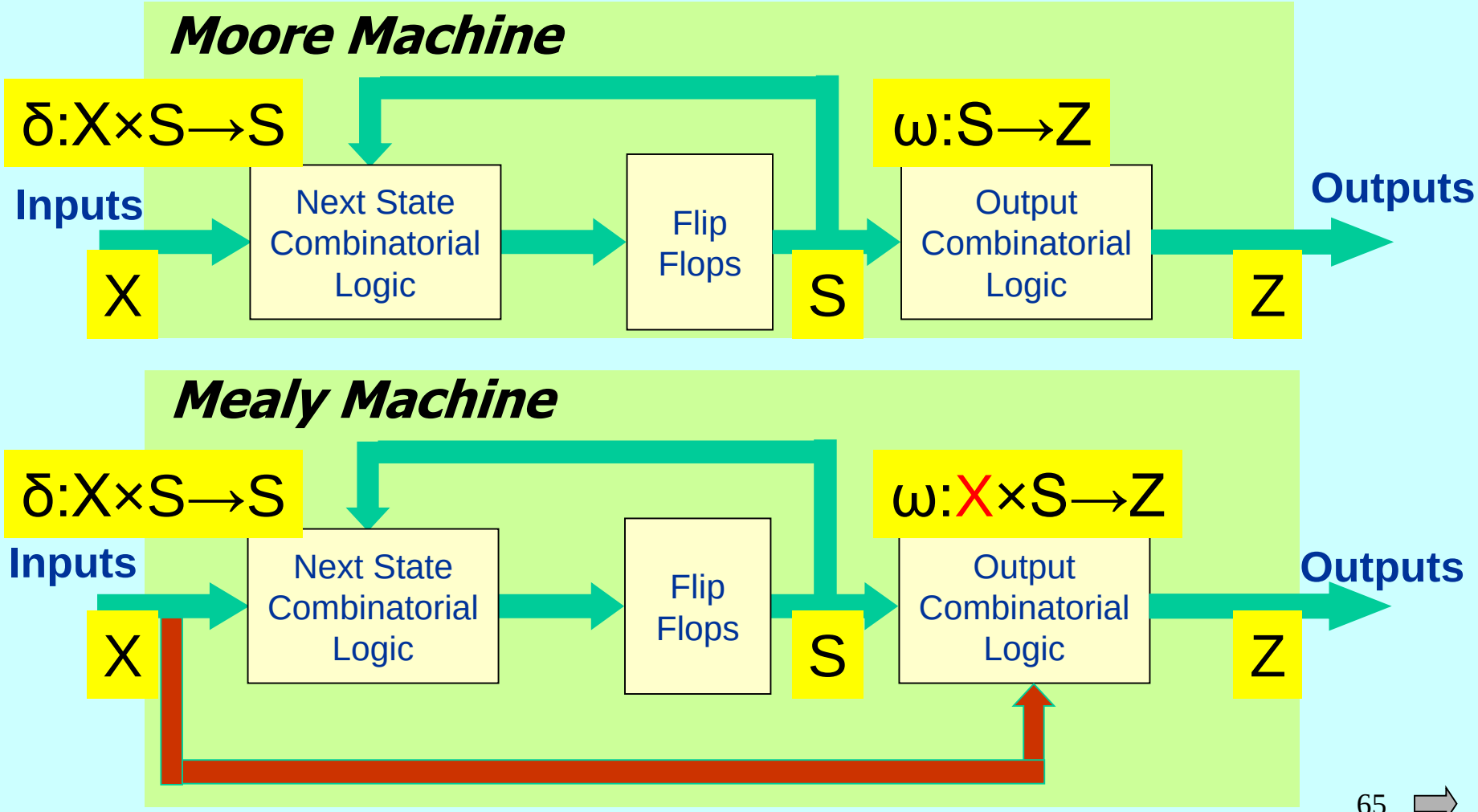
	Source State	Destination State	Condition
1	C0x	C1x	(C)
2	C0x	C0x	(!C)
3	C1x	C1x	(C)
4	C1x	C0x	(!C)
5	SInit	C1x	(C)
6	SInit	C0x	(!C)



Nondeterministic FSMs

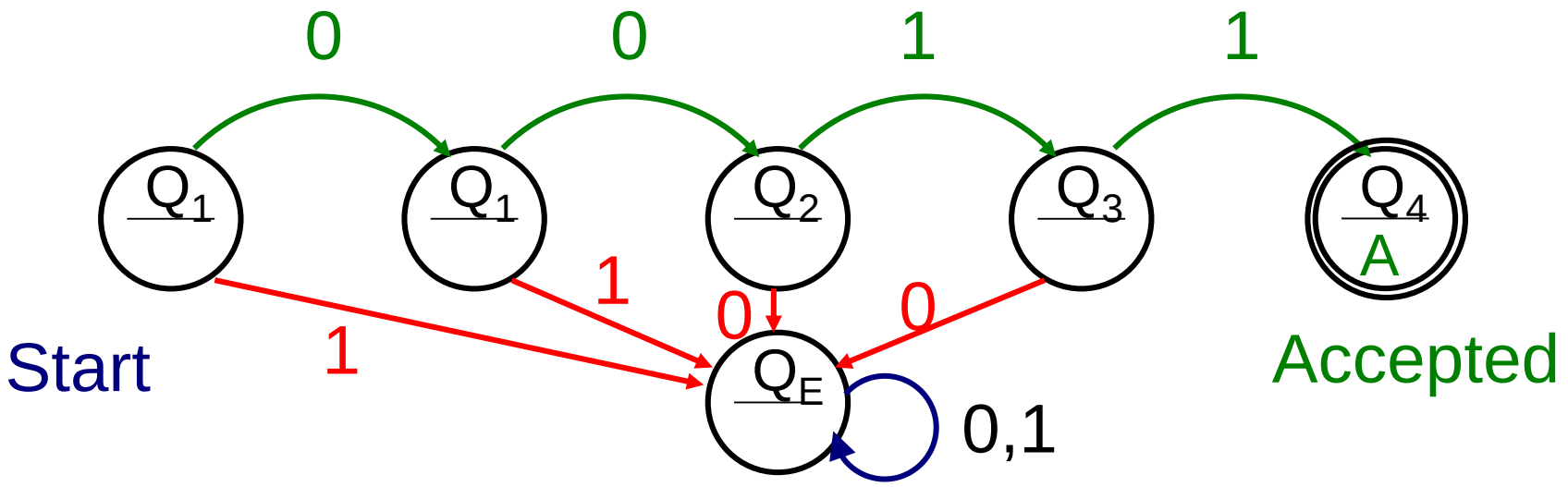
by an example

Mealy and Moore Finite State Machines are Deterministic!



Safety Lock with Binary Key

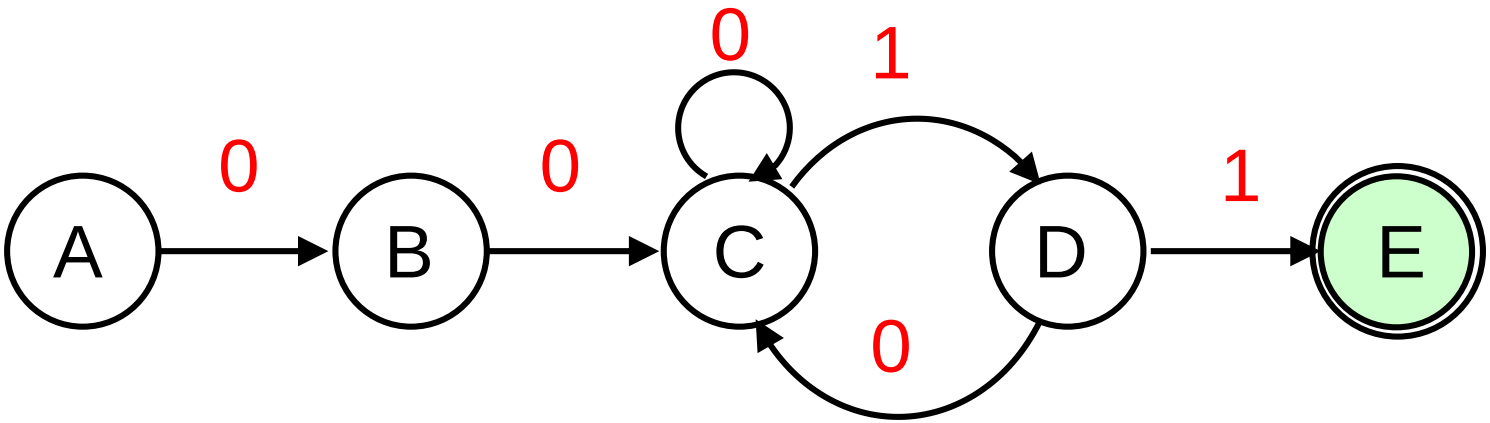
- Opened when 0011, otherwise not.
- Generate A (Accepted) in Q_5 state



Such automaton is called *Finite State Acceptor or Acceptor Finite State Machine.*

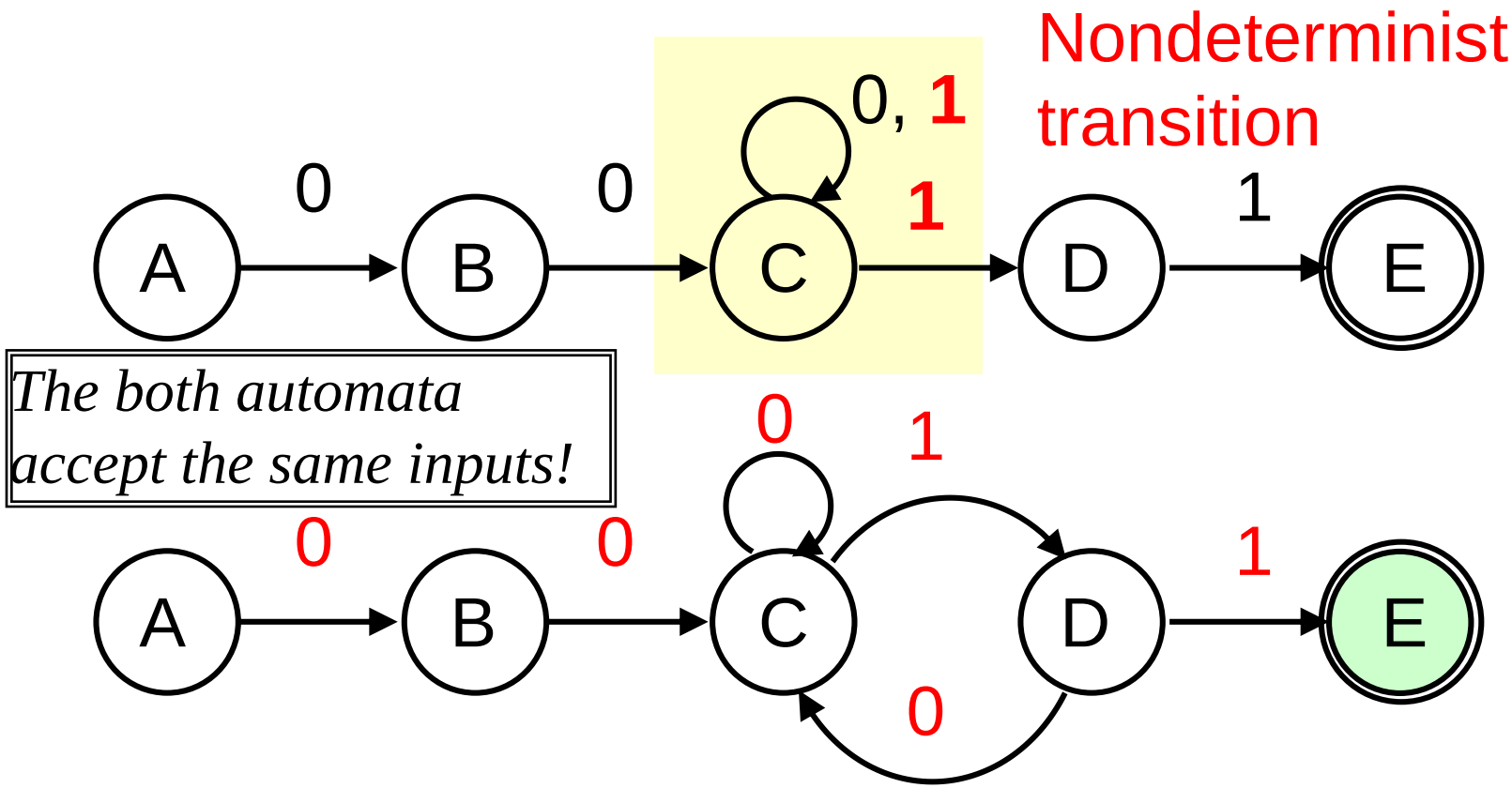
Code Safety Lock 2

Input must begin by 00 and terminate by 11



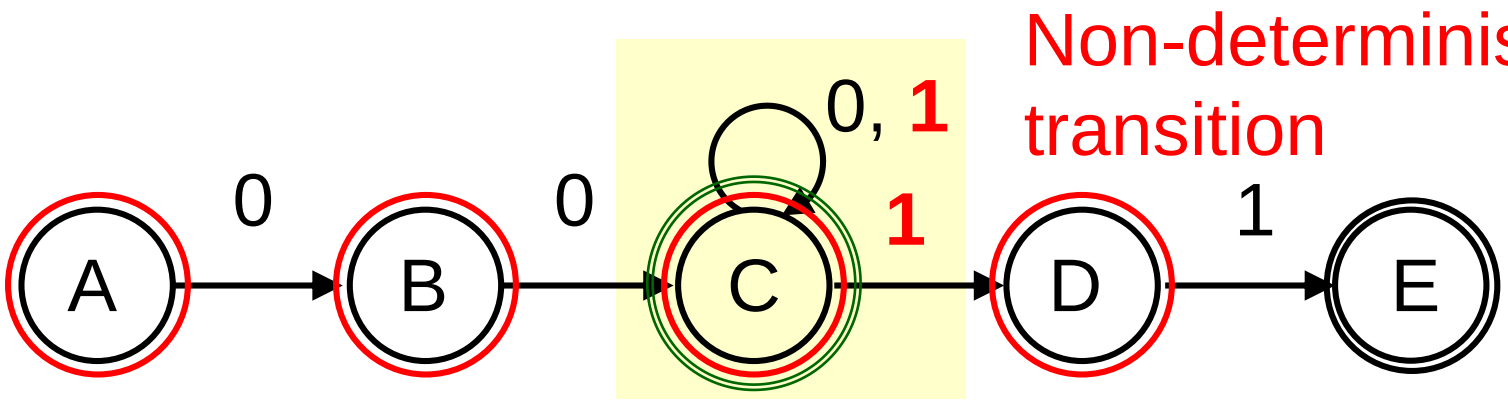
Code Safety Lock 2 cont'd

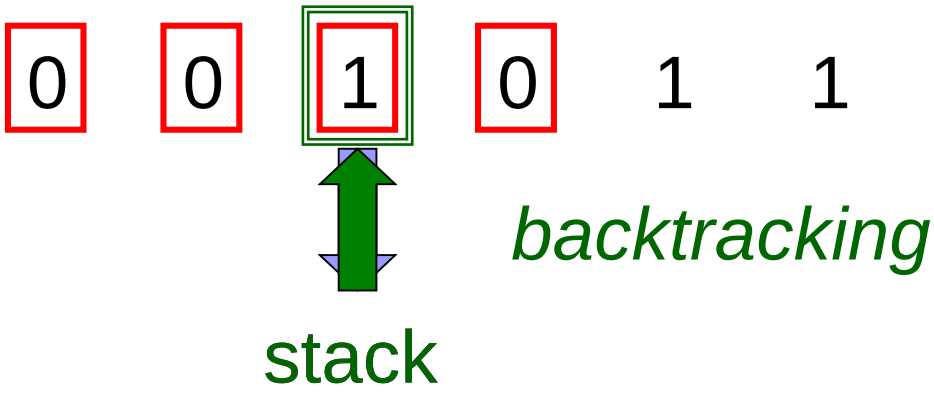
NFA – Nondeterministic Finite Automat/Acceptor



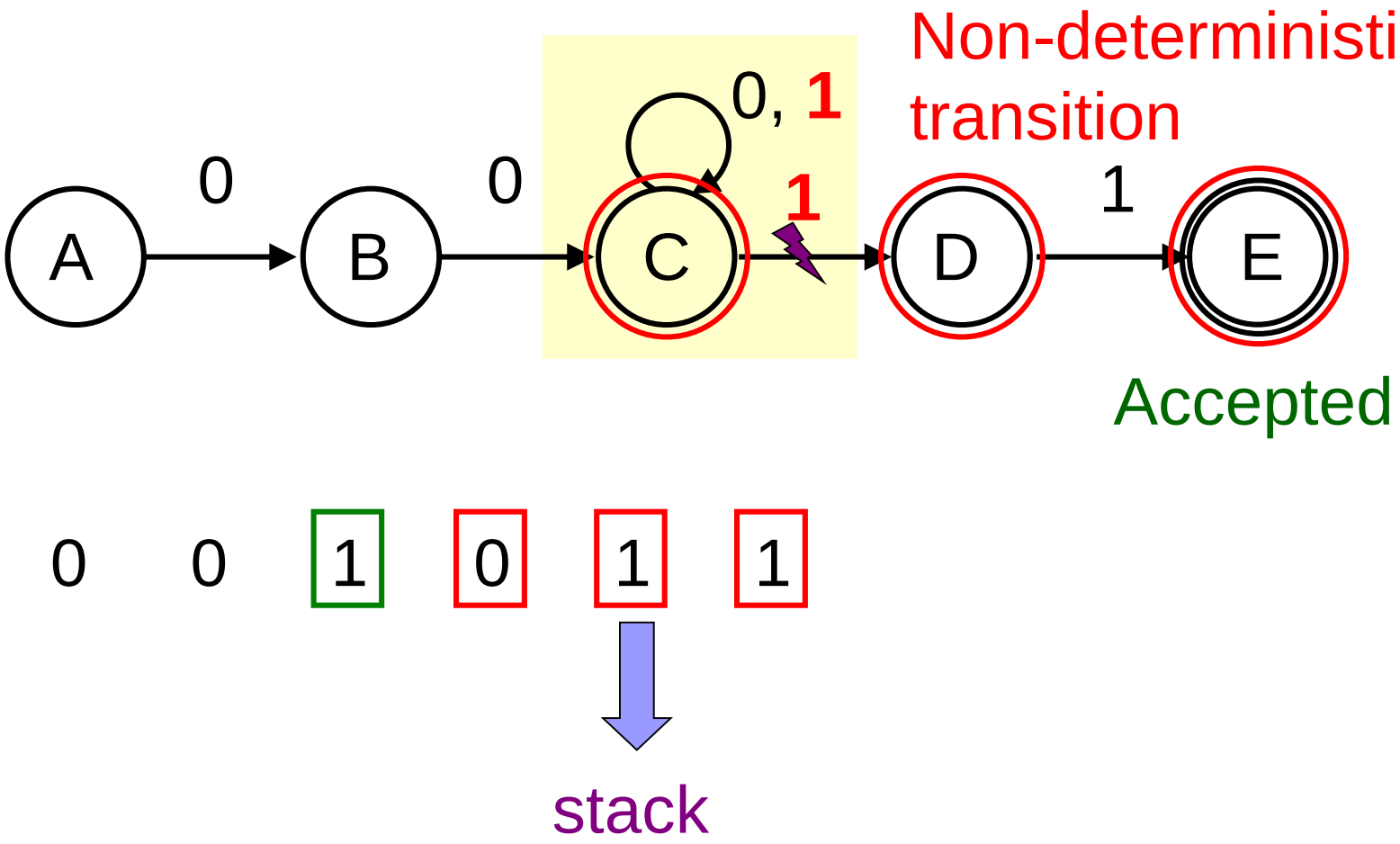
DFA – Deterministic Finite Automat/Acceptor

NFA – Nondeterministic Finite Automaton/Acceptor





NFA – Nondeterministic Finite Automat/Acceptor



Note

- *There are other more complex methods for processing NFA that do not use backtracking and can operate in linear time,*
  e.g. Thompson NFA, see
  http://swtch.com/~rsc/regexp/regexp1.html

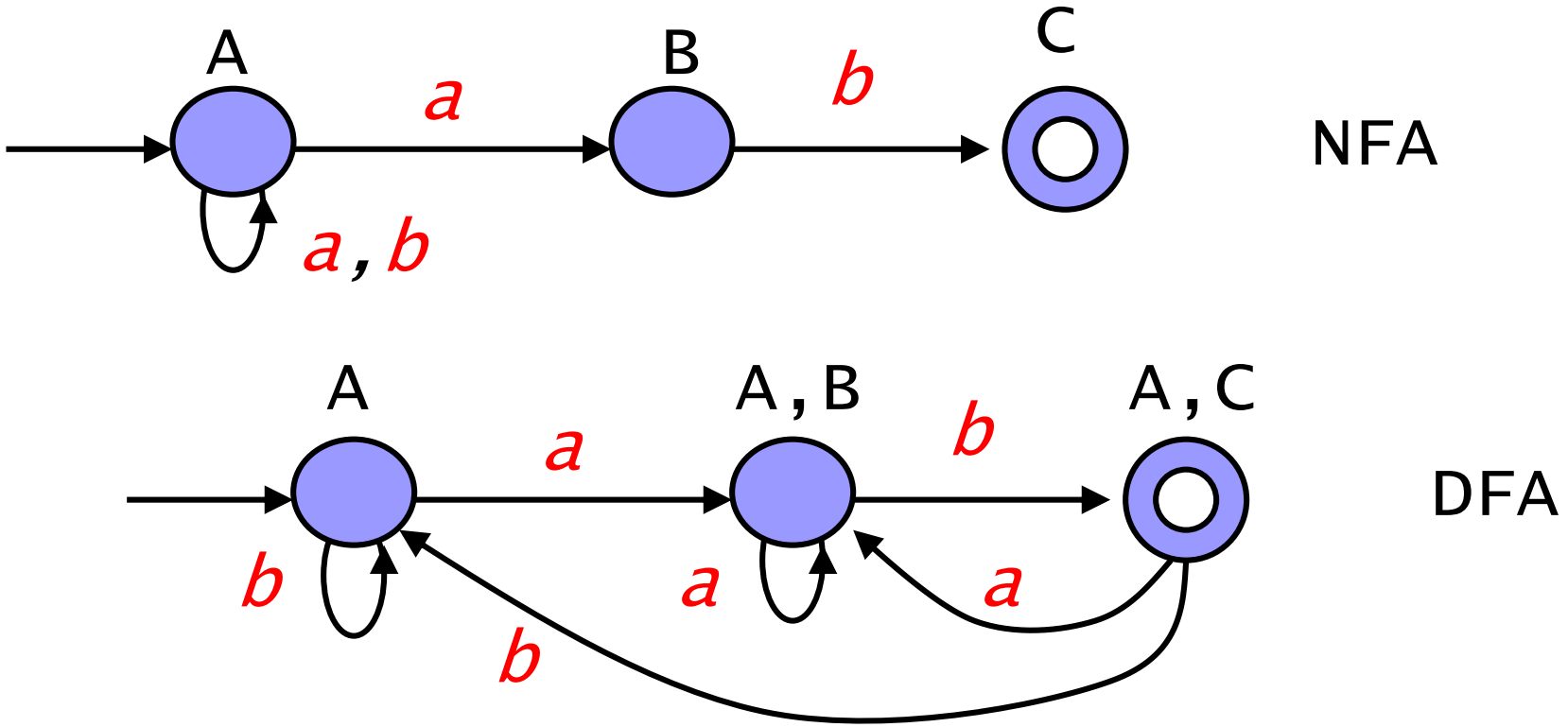
Non-deterministic behavior is not random

- Deterministic:
  f(1) → 1 *always*
- Random:
  f(1) → 0 *in 50% cases,*
  f(1) → 1 *otherwise*
- Non-deterministic behavior
  f(1) → 0 or f(1) → 1,
  we do not tell when it happens.

- Non-deterministic behavior can look as deterministic, random or much worse...

NFA versus DFA

Any sequence of a b that terminates by a b



Finite acceptor

A tuple

$$\mathbf{M = < X, S, \delta, s_0, F >}$$

- **X** – *finite set of input vectors*
- **S** – *finite set of states*
- δ - mapping
  - for DFA -mapping **δ: X** x **S -> S**
  - for NFA – mapping **δ: {X + ε} x S -> S**, where ε is random input
- $\mathbf{s_0}$ – initial state $s_0 \in S$
- **F** – finite set of accepted states $F \subset S$

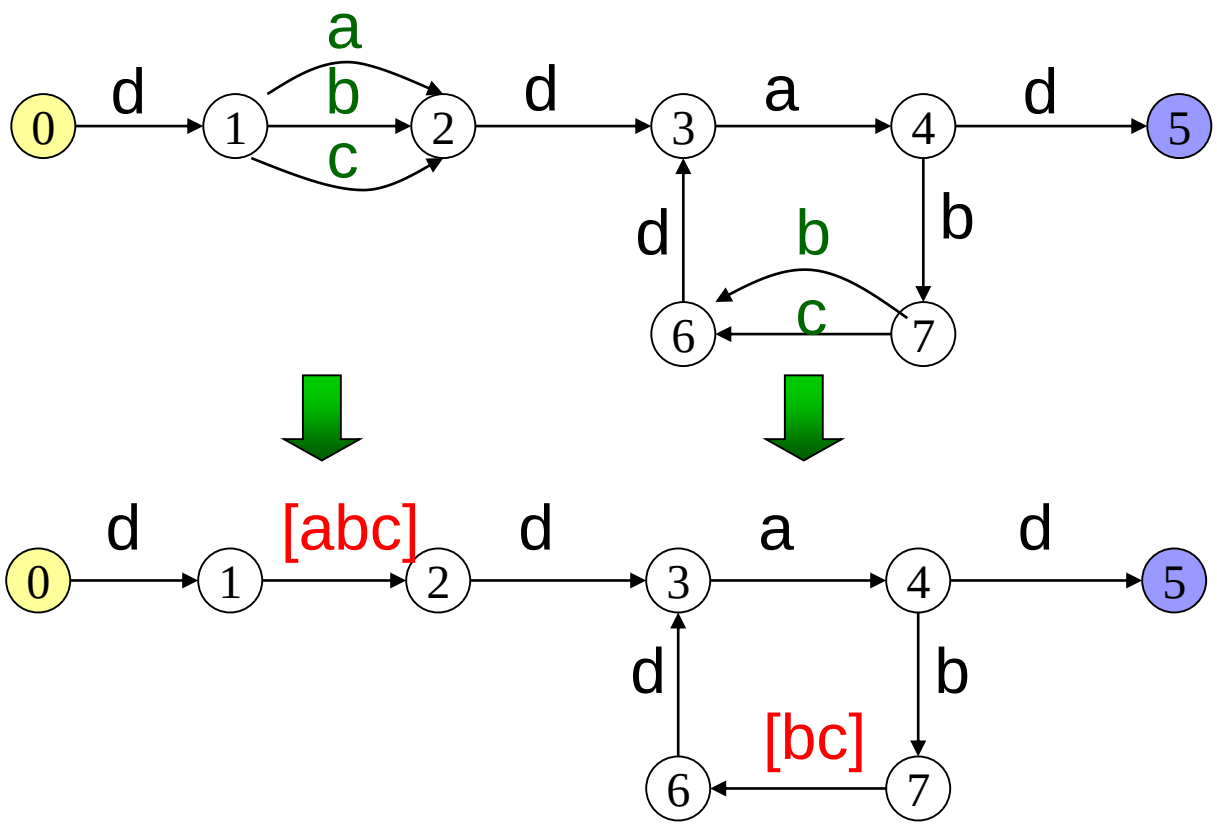
Regular Expresions

http://www.dotnetcoders.com/web/Learning/Regex/default.aspx

Regular Expressions

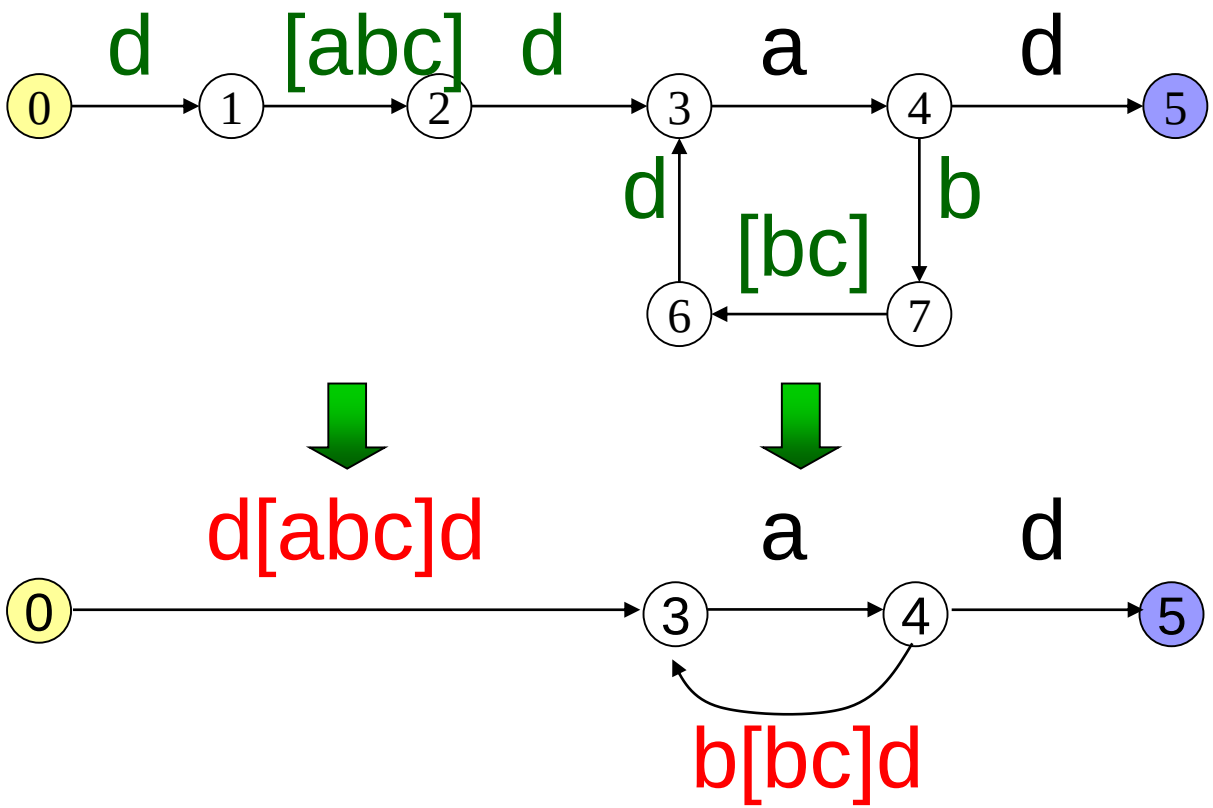
- Formally describe tokens in the language
  - Regular Expressions
  - NFA
  - DFA
- Regular Expressions ➔ finite automata
- Java, PHP, Python, C# and other implements regular expressions by NFA.

Example: Building regular expression 1/5

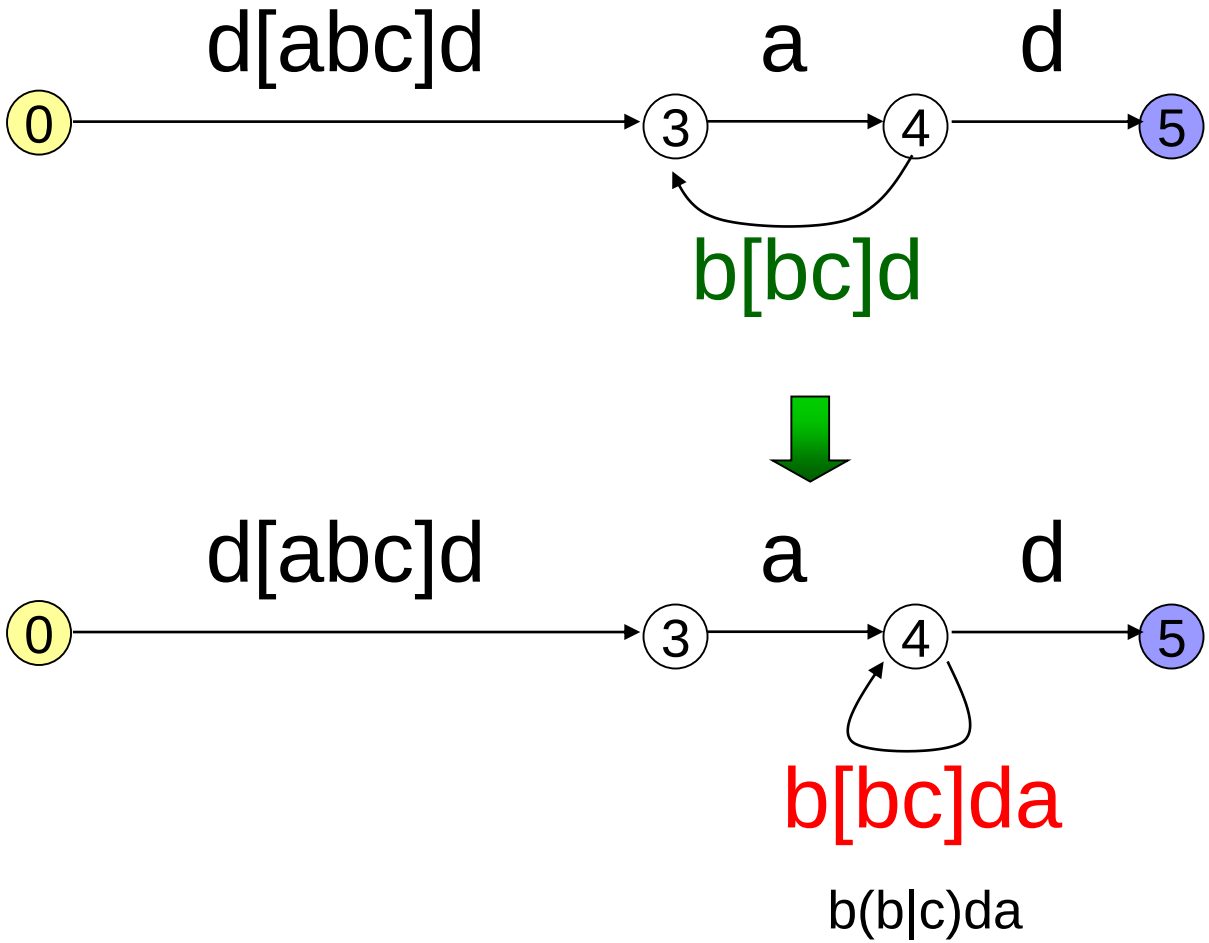


[abc] – on character from [ ] list.
a or b or c.

Example: Building regular expression 2/5

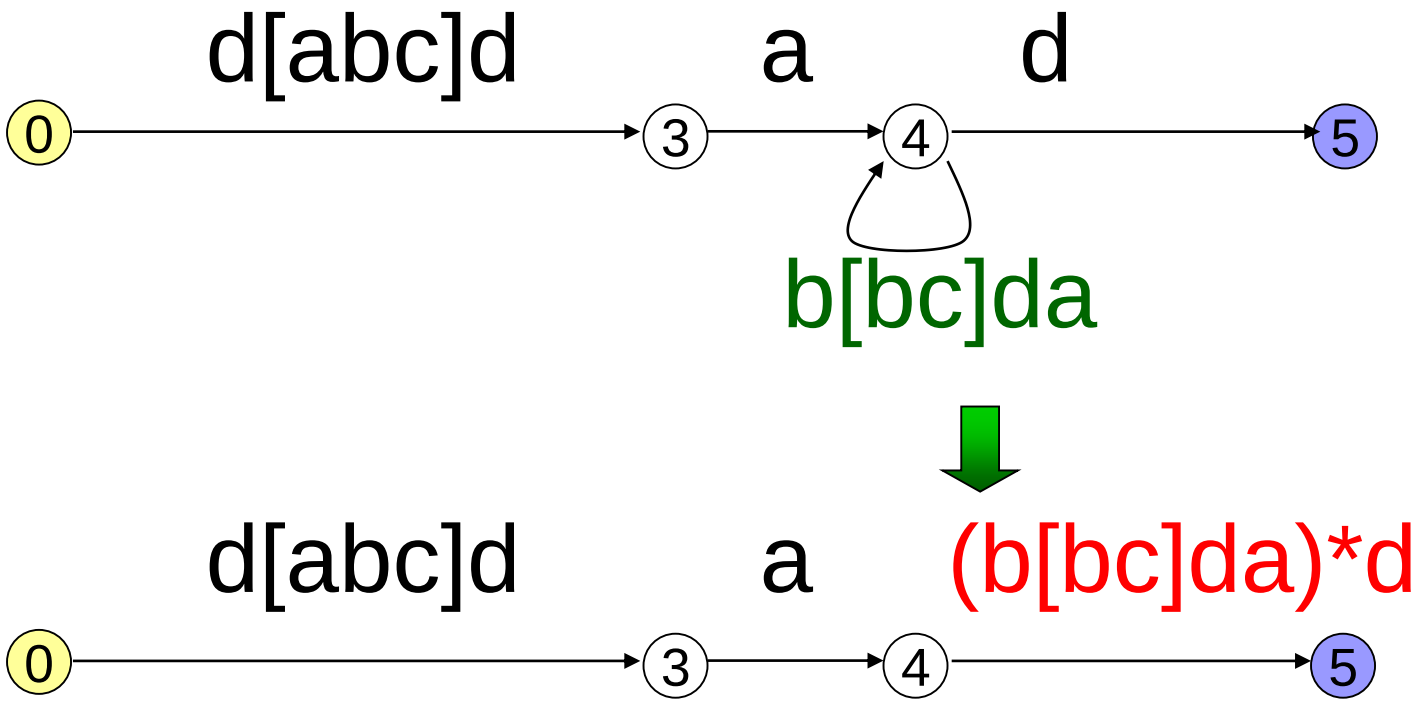


Example: Building regular expression 3/5



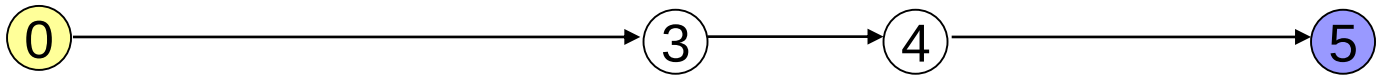
b(b|c)da

Example: Building regular expression 4/5

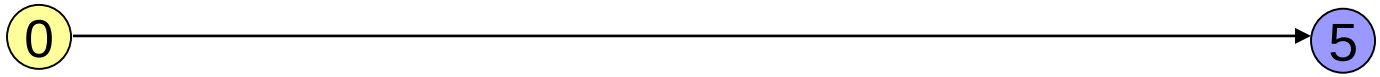


Example: Building regular expression 5/5

d[abc]d a (b[bc]da)*d



d[abc]da(b[bc]da)*d



* - *0 or more repetition of ( ).*
(b[bc]da)

Character classes

.	Matches any character except **\n**. E.g.. a.a matches aea, aia, aca, and a a
[XY]	Matches any single character included in the specified set of characters
[A-Z]	Use of a hyphen (–) allows specification of contiguous character ranges.
[A-Za-z]	form A to Z, from a to z
[^AB]	Matches any single character not in the specified set of characters.

Metacharacters

- Some "character-class" metacharacters
  - **\w** is equivalent to **[a-zA-Z_0-9]**
  - **\W** is equivalent to **[^a-zA-Z_0-9]**.
  - **\s** Matches any white-space character. **[\f\n\r\t\v]**.
  - **\S** Matches any non-white-space character. **[^ \f\n\r\t\v]**.
  - **\d** Matches any decimal digit. **[0-9]**.
  - **\D** Matches any nondigit.

- Example **\d\d\d\s matches: "123 456"**

Example Metacharacters

```
Regex rx1 = new Regex(@"\d\d\d\s");
bool b1 = rx1.IsMatch("123 456");          // true
bool b2 = rx1.IsMatch("   123 456");       // true
bool b3 = rx1.IsMatch("   123   456");     // true
bool b4 = rx1.IsMatch("   123456");        // false
bool b5 = rx1.IsMatch("a123 456");         // true

Regex rx2 = new Regex(@"^\d\d\d\s");
bool b6 = rx2.IsMatch("a123 456");         // false
```

Modificators

|	Matches any one of the terms separated by the | (vertical bar) character; for example, cat|dog|tiger. The leftmost successful match wins.
\	next char is literal
^	from beginning
$	at the end of text

"^list" matches "listing", but not "A list"

"data$" matches "subdata", but not "data1"

Quantifiers

*	Specifies zero or more matches
+	Specifies one or more matches
?	Specifies zero or one matches
{n}	Specifies exactly *n* matches
{n,}	Specifies at least *n* matches
{n,m}	Specifies at least *n*, but no more than *m*, matches
()	Grouping

What is the correct answer?

^([A-Z]\d)$

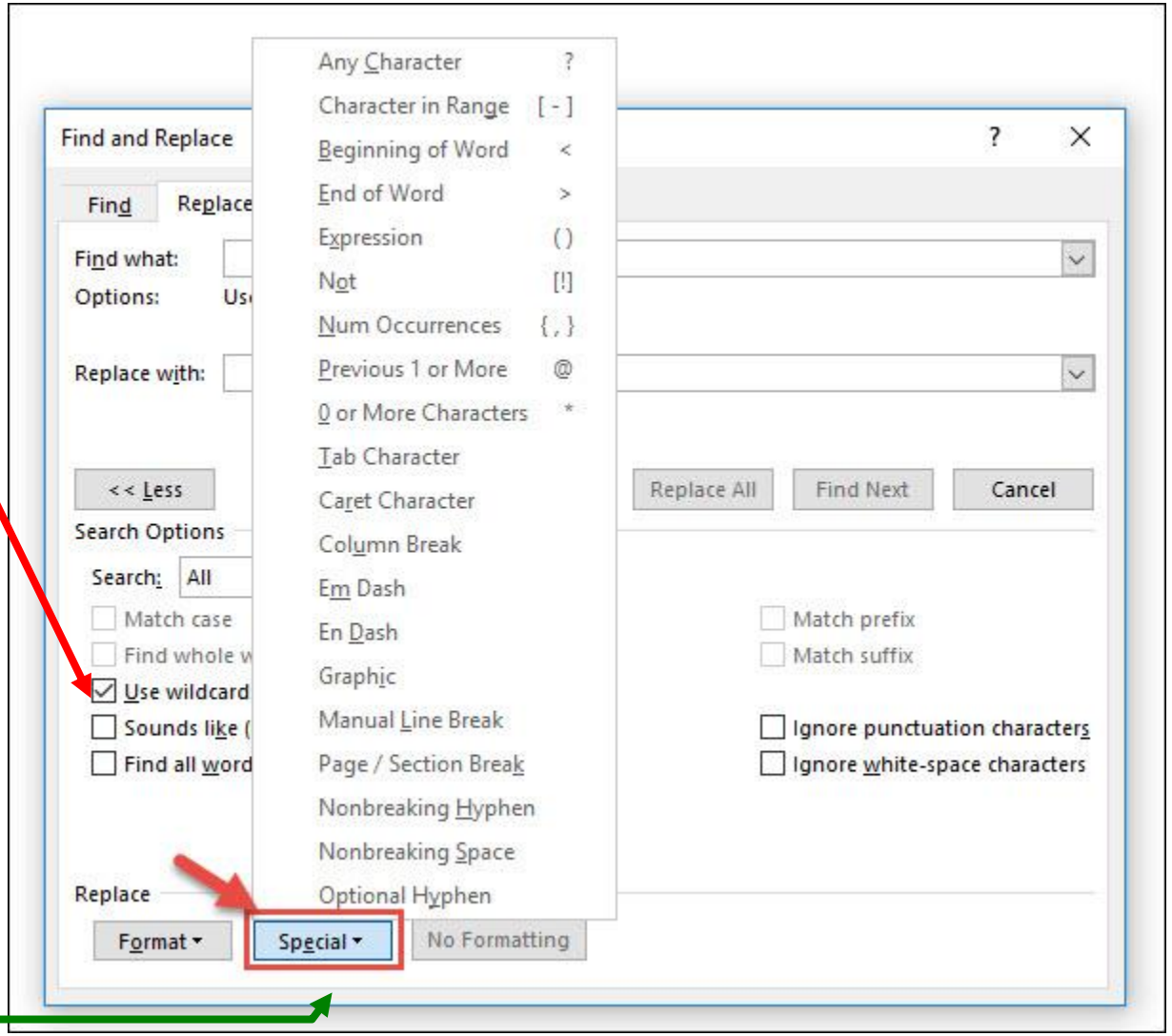
Example of C# Program

```
using System; using System.Text.RegularExpressions;
namespace ConsoleRegex
{  class Program
  {    static void Main(string[] args)
       {  Regex rg = new Regex(@"(0x)?([A-Fa-f\d]+)(h|H)?\w*", RegexOptions.Compiled);
          string [] tokens = new string[] { "1234 ", " 0xABc9 ", "78H", "LSP" };
          for (int i = 0; i < tokens.Length; i++)
           { Match m = rg.Match(tokens[i]);
             if (!m.Success) Console.WriteLine("No hex:{0}", tokens[i]);
             else  {  Console.Write("Decimal={0}\t", UInt64.Parse(m.Groups[2].Value,
                                          System.Globalization.NumberStyles.HexNumber));
                      for (int j = 0; j < m.Groups.Count; j++) Console.Write("\t{0}:{1}", j, m.Groups[j].Value);
                    }
            Console.WriteLine();
       }
    }
  }
}
```

```
Decimal=4660   0:1234    1:     2:1234   3:
Decimal=43977  0:0xABc9  1:0x   2:ABc9   3:
Decimal=120    0:78H     1:     2:78     3:H
No hex:LSP
```

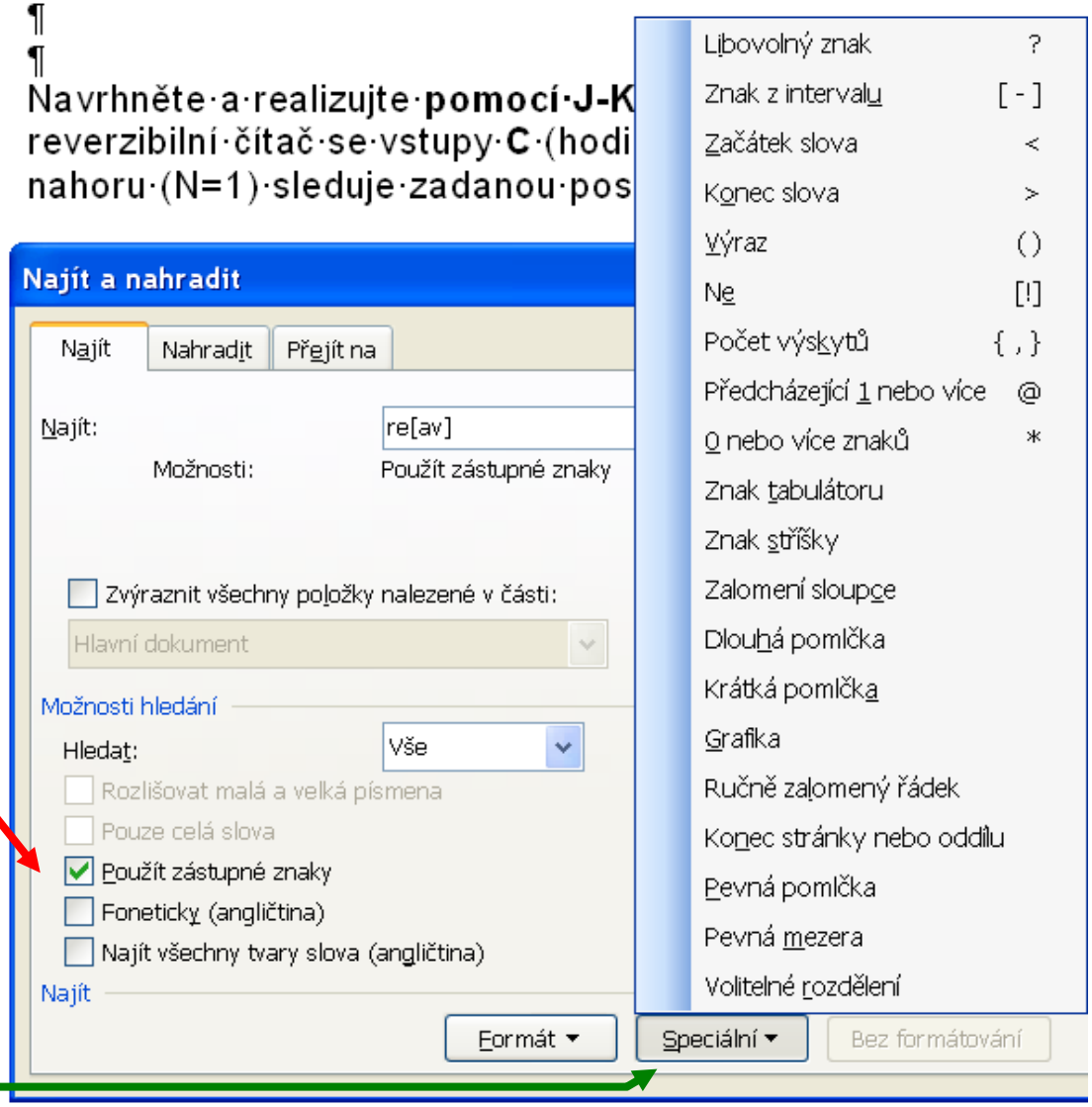

MS-Word knows subset of regular expressions

- Regular-like expressions are wildcards
- Operators are listed in Special



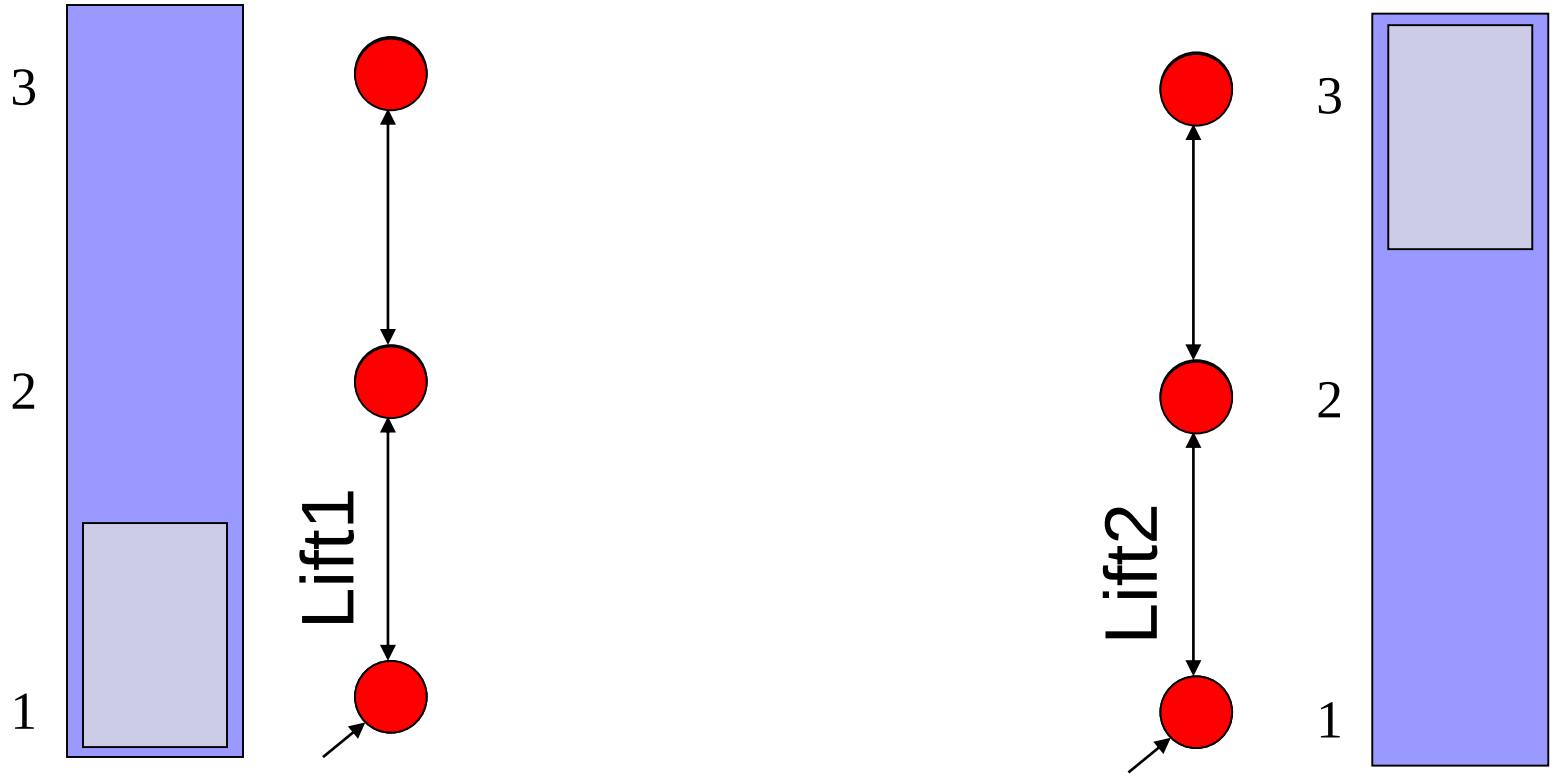
See: https://wordmvp.com/FAQs/General/UsingWildcards.htm

- Regulární výrazy se skrývají pod zástupnými znaky
- Nabídku operátorů najdete pod Speciální



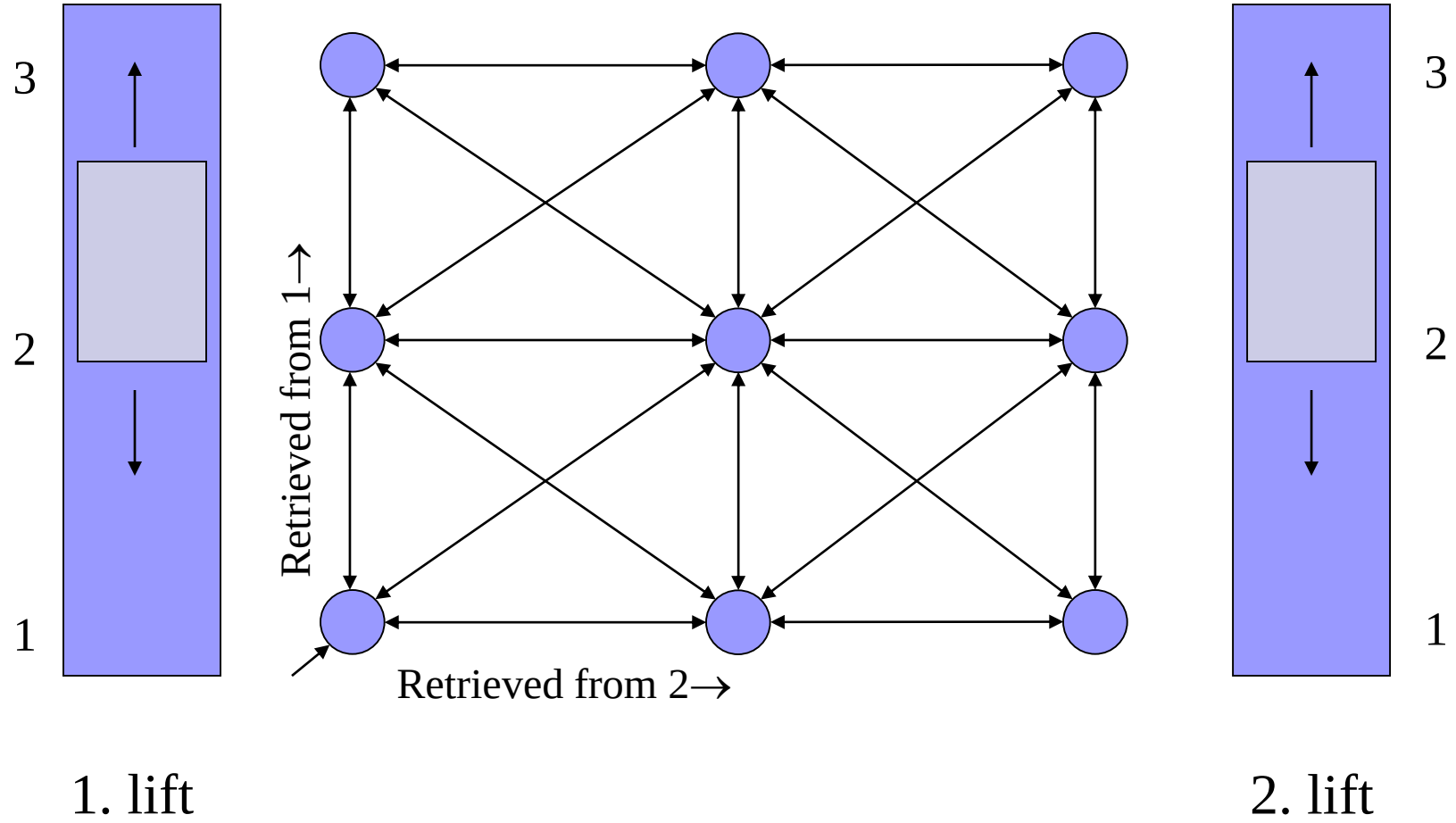
Limits of the automatic model

Description of 2 lifts 2 automats



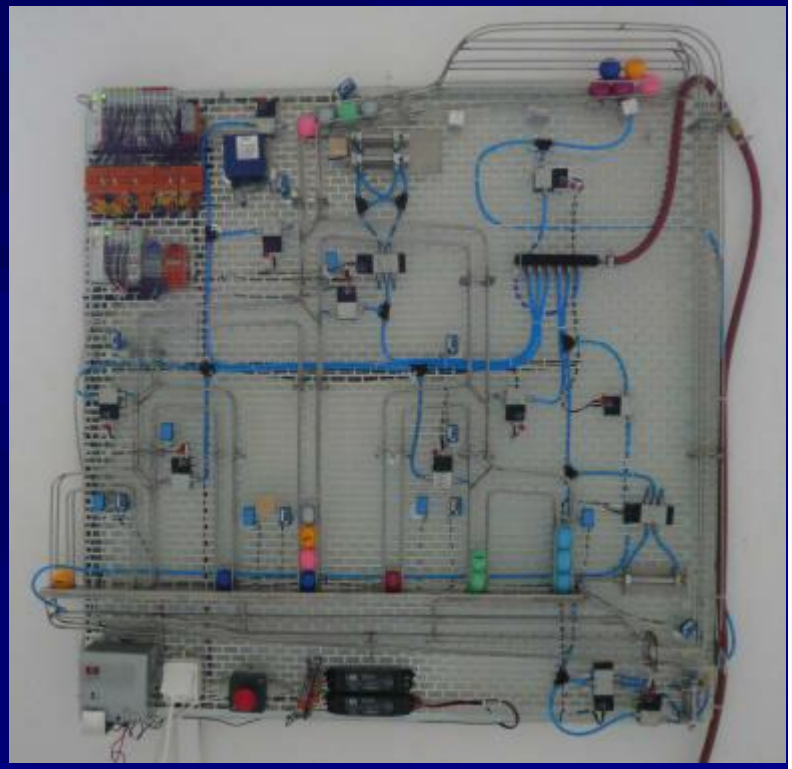
Two lifts with floors can be described as a combination of 2 FSMs.

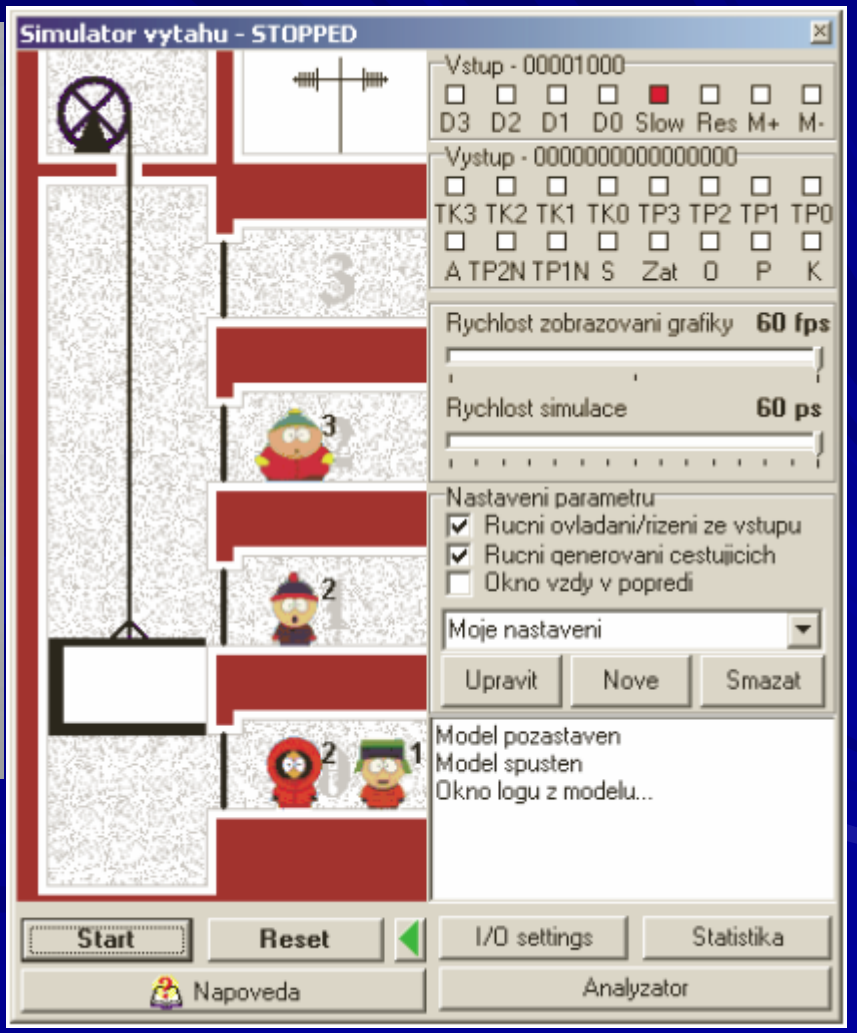
Model 2 lifts 1 automatic



*The FSM is always in **one state***
-> the number of states is a list of all possibilities!

What is solvable by FSMs?





FSM and their Tasks ?

***Jorge Luis Borges, Library of Babel**, 1941*
variant of Jonathan Swift's Word-Machine

The world is bigger than imaginable.

The number of possible situations is just as great.

Every human theory solves only part of the problems.

Gulliver's Travels
Part III.
by
Jonathan Swift,
1726:

Word-Machine of Lagado Academy

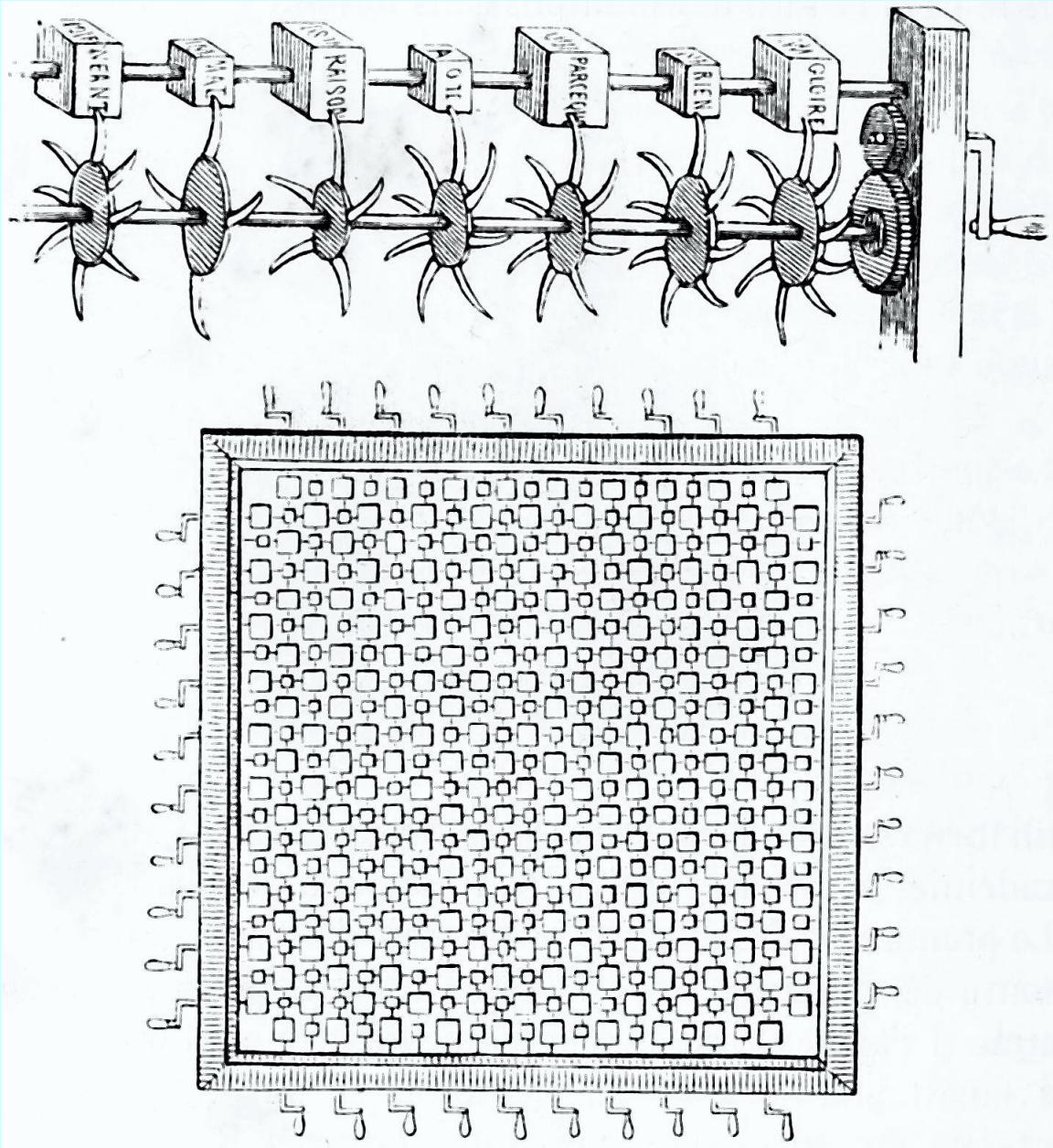
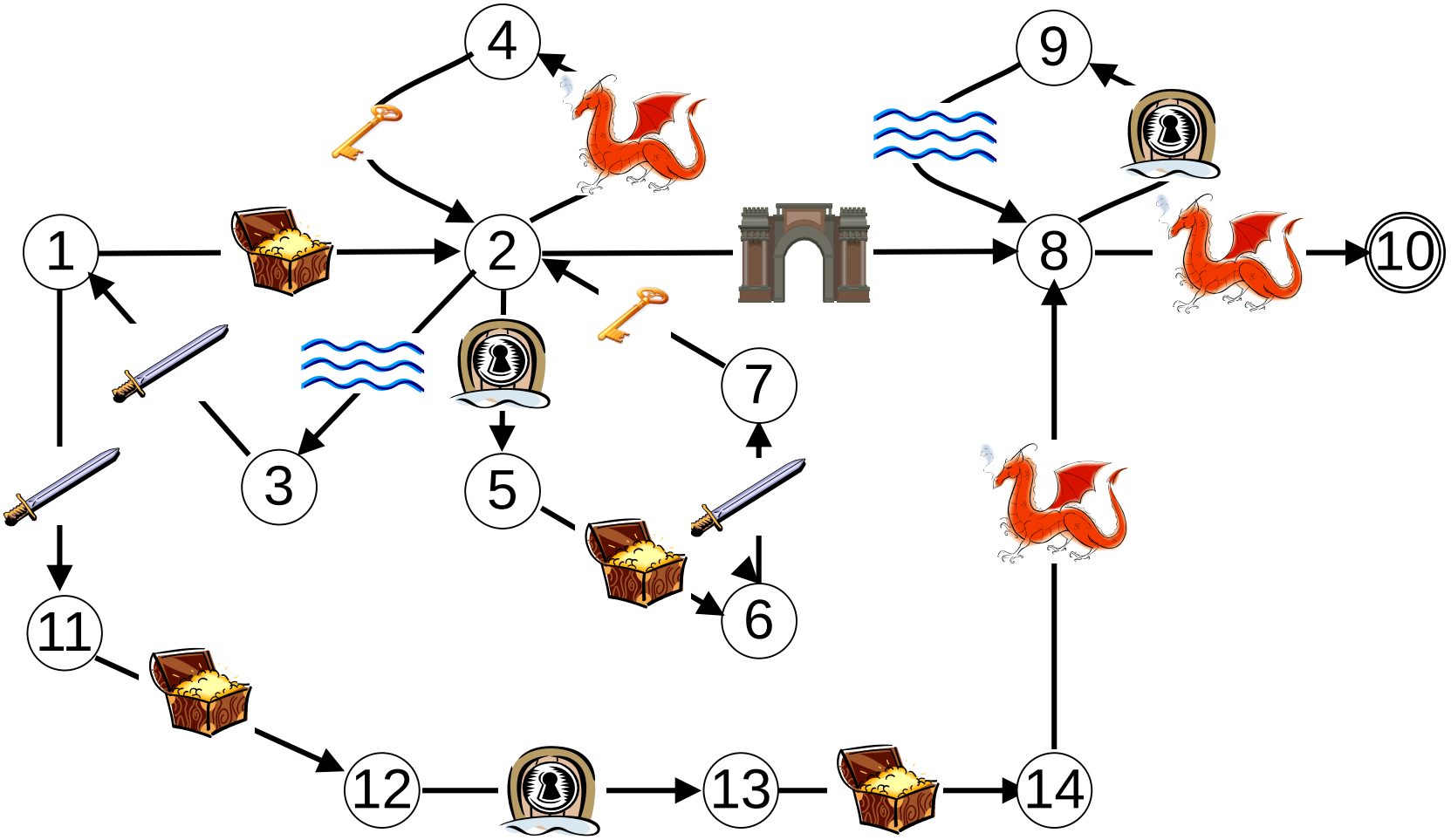


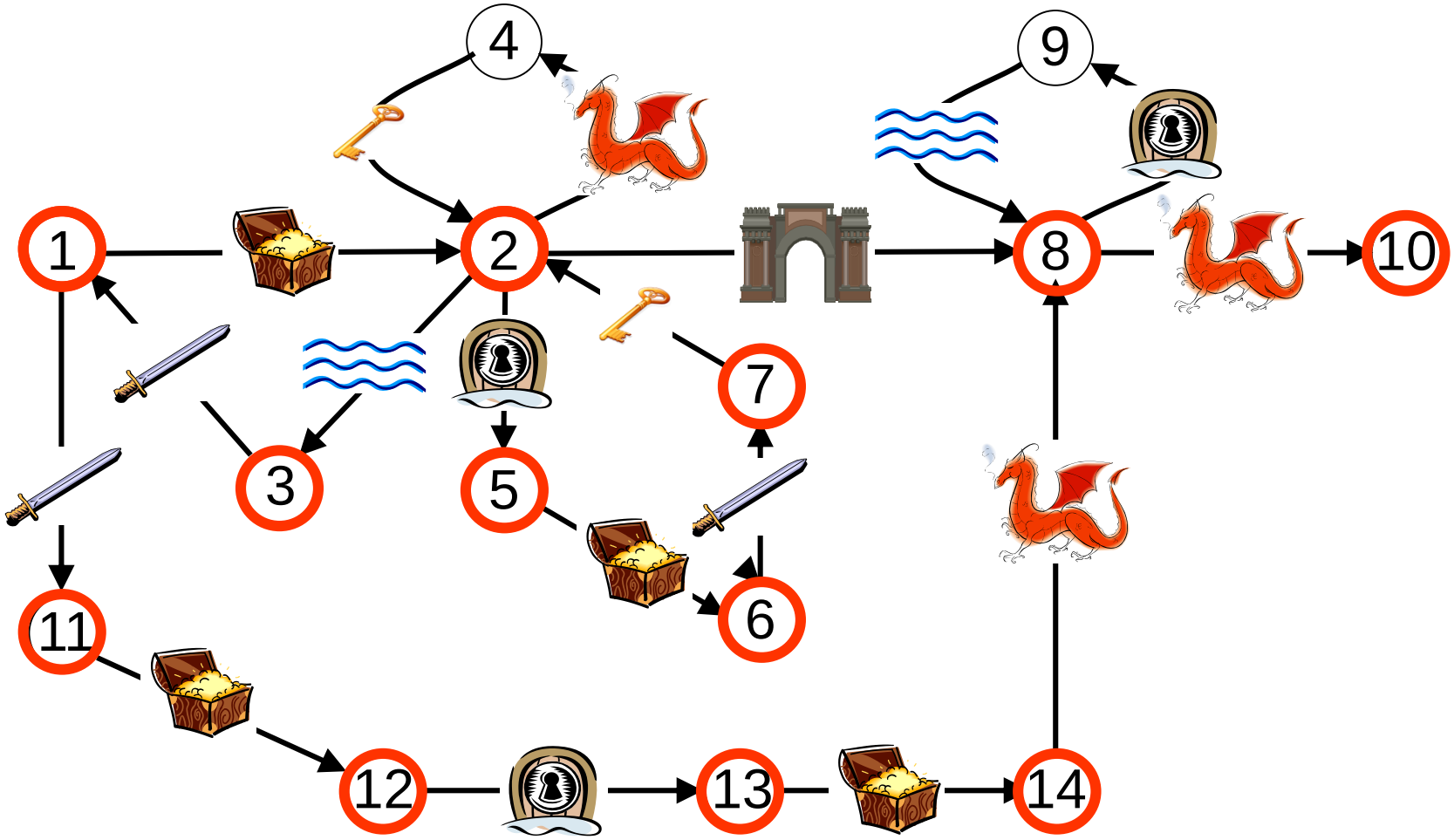
Image: Blog Gulliver's Typewriter

Regular, Contex-free, and Context grammar tasks



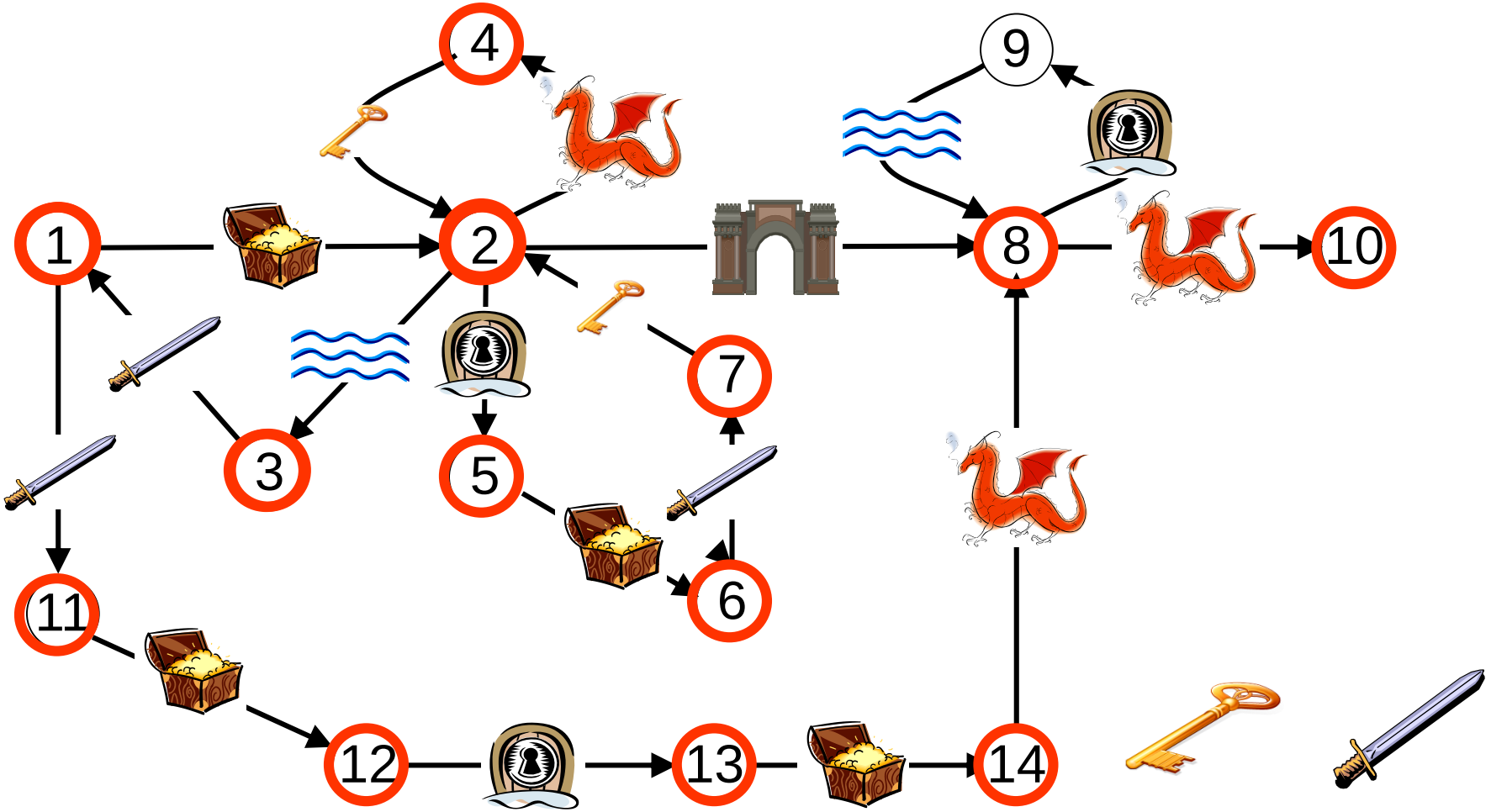
Created according to Brauer, Holzer1, Konig, Schwoon: The Theory of Finite-State Adventures, (*http://www.fmi.uni-stuttgart.de/szs/publications/koenigba/eatcs79.pdf*)

Animation: Regular grammar - FSM



Collecting treasure without limitations,
no keys and no swords -> Finite state machines

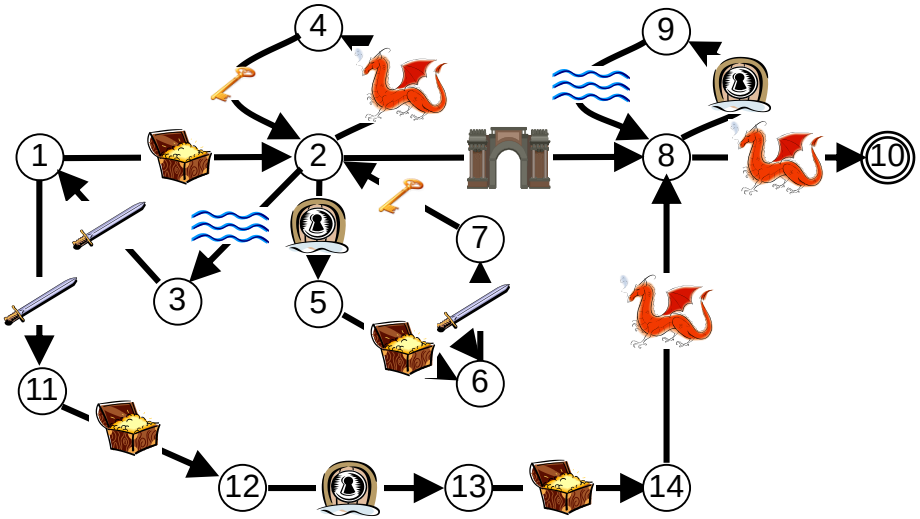
Animation: context-free grammar - push-down automaton



We need a sword to kill dragons and single usage keys

Context-free task *-> push-down automaton*

Context grammar - program



Even more difficult task, this time context

* After slaying the dragon, sword is too sticky from his blood.
  We cannot fight it until it is cleaned in water.
* Possible additional restrictions?
  + with a large amount of gold, we cannot swim
  + we cannot throw away gold if we have picked it

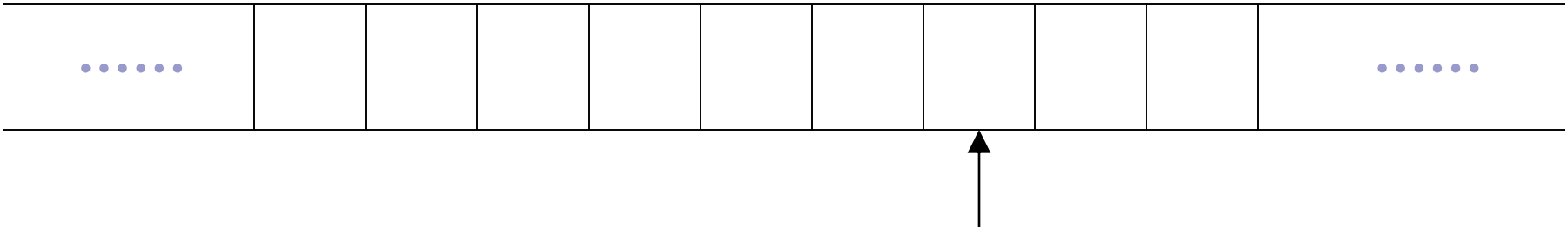
Overview Machines versus Languages

Chomsky's Hierarchy of Languages	Recognized by Machines
Regular (type-3)	**Finite state machines**
Context-free (type-2)	Push-down automata
Context-sensitive (type-1)	Linear-bounded Turing machines *limited tape*
Unrestricted grammar	Turing machines *unlimited tape*

More in Course Languages, Automats and Gramatics
https://intranet.fel.cvut.cz/en/education/bk/predmety/46/81/p4681606.html]

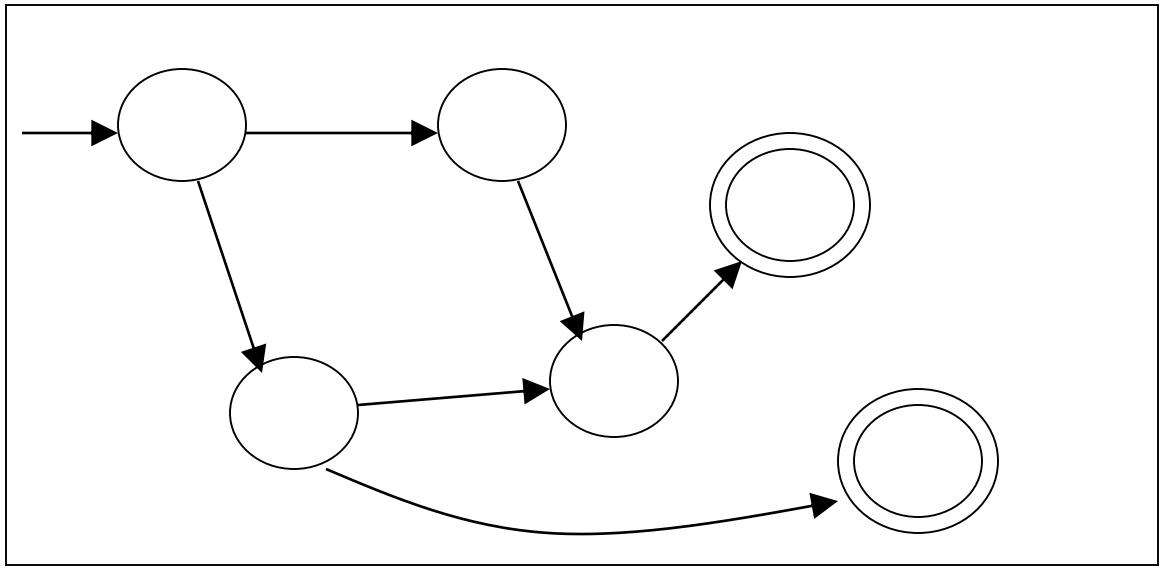
Turing Machine versus Processor

Tape

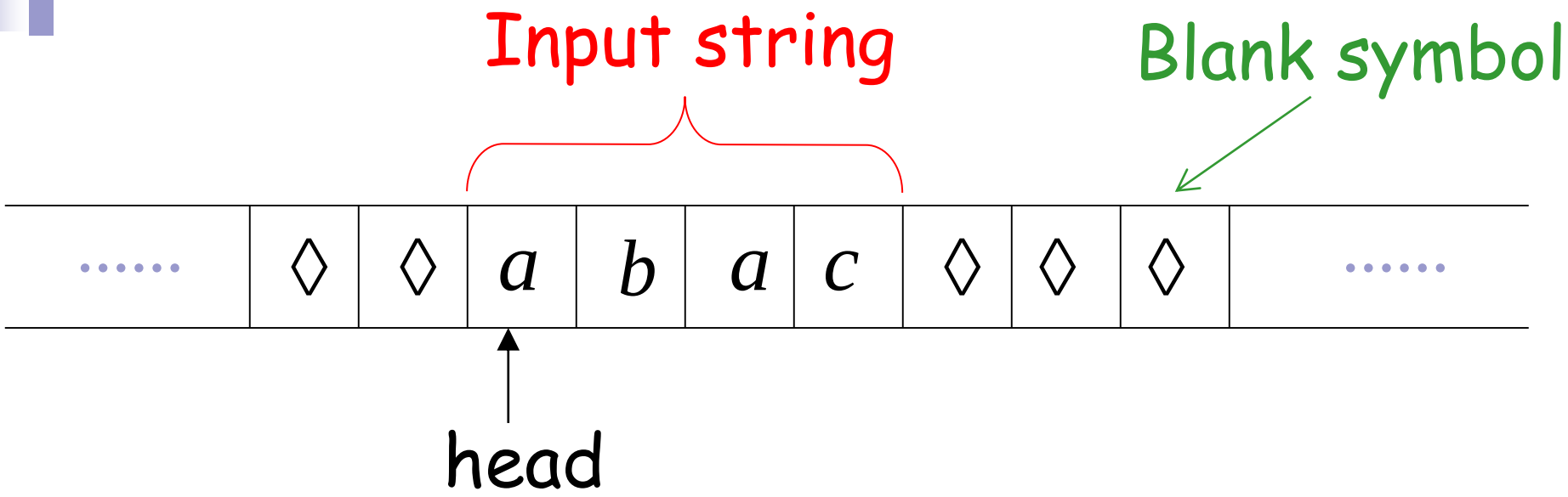


Read-Write head

Control Unit



A Turing Machine Head



The head at each time step:

1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Usage of Turing Machine

- Anything a real computer can compute, a Turing machine can also compute...
- *...but programming of Turing machines is very clumsy and their programs run very slow...*
- *...so Turing machines are not physical objects but mathematical ones suitable only for* ***proving of computability****.*

For more infromation, see AD4M01TAL Theory of Algorithms

Languages and FSMs

The Tower of Babel, Pieter Brueghel, c. 1563, Kunsthistorisches Museum, Vienna

Languages

- **Alphabet** – finite character set (S)
- **String** – finite sequence of characters – can be ε, the empty string
- **Language** – possibly infinite set of strings over some alphabet – can be { }, the empty language.

Examples of Languages

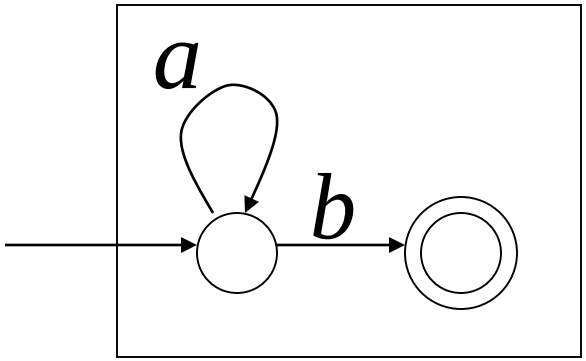
Let be given alphabet $\Sigma = \{a, b, c\}$, than we may create languages:

- {aa,ab,ac,bb,bc,cc}
- {ab,abc,abcc,abccc,. . .}
- { ε } - *it contains empty string*
- { } – *empty language*
- {a, b, c, ε}

Examples DFA

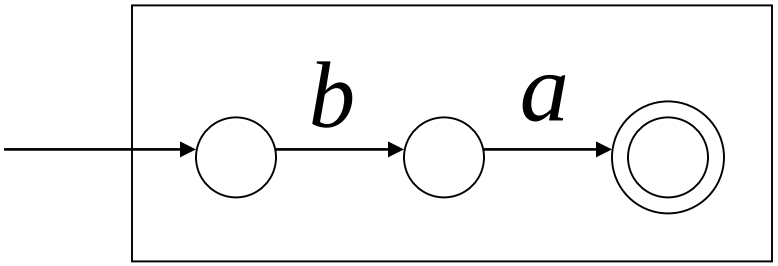
$L_1 = \{a^n b\}$

M_1



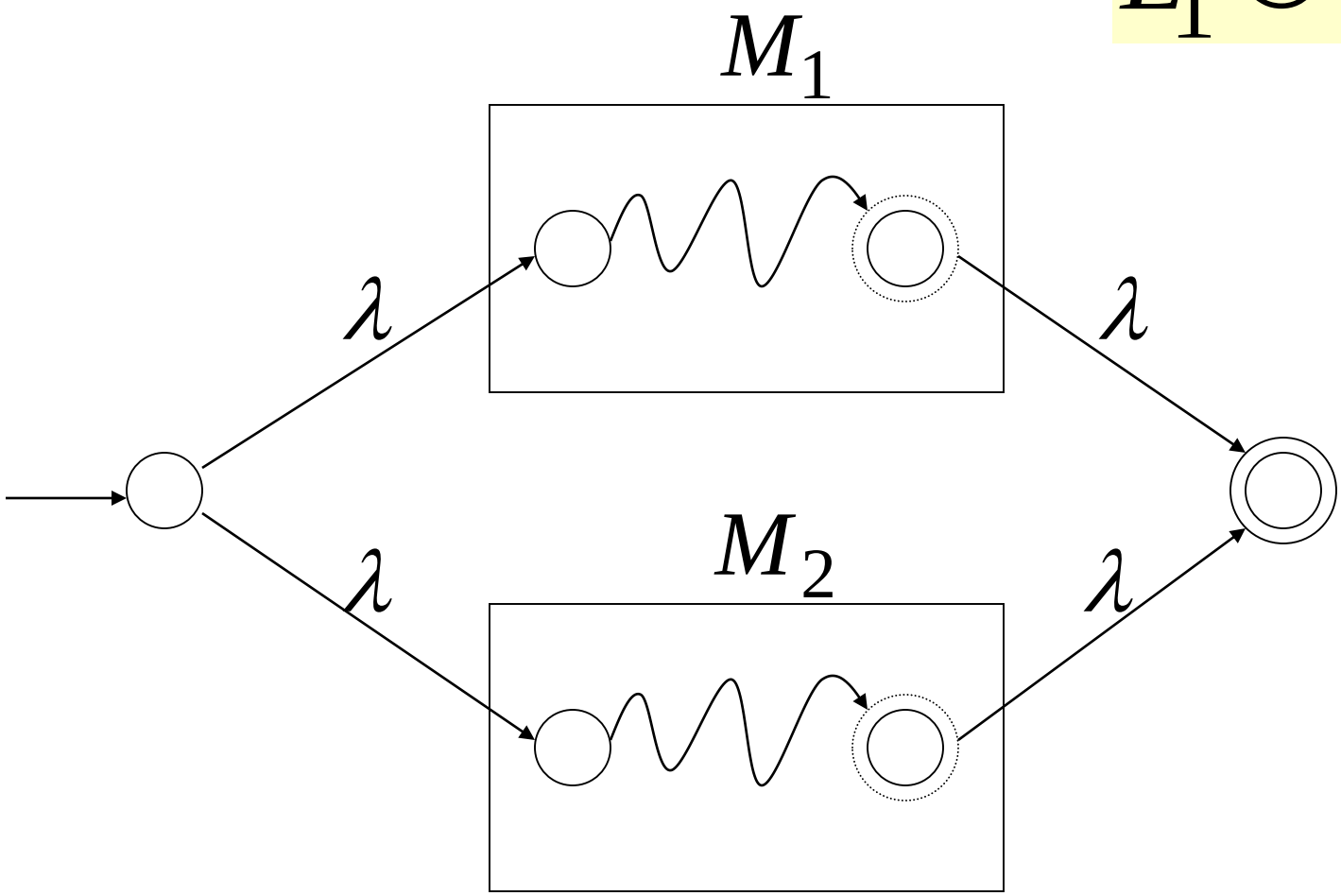
$L_2 = \{ba\}$

M_2



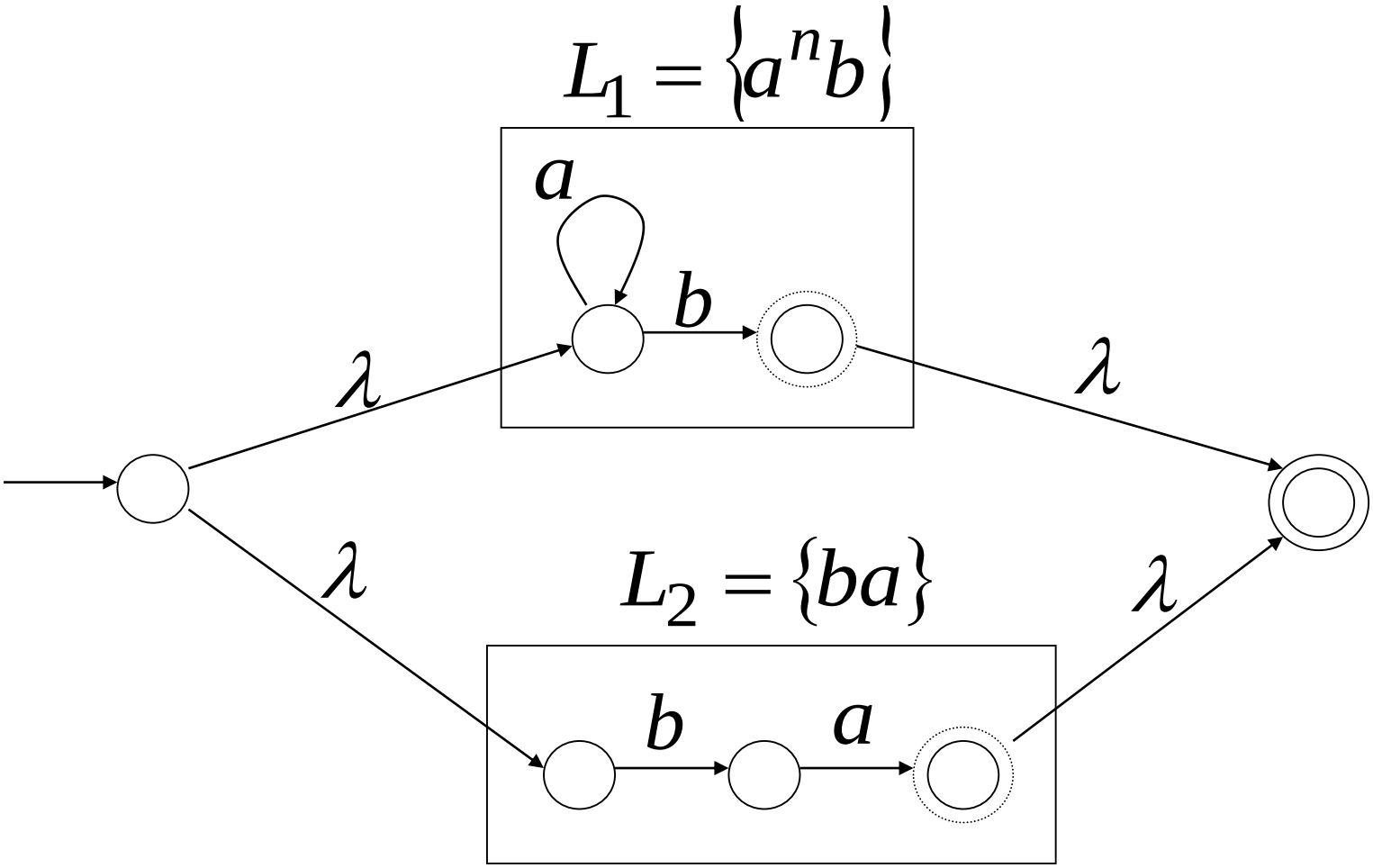
Union

$L_1 \cup L_2$



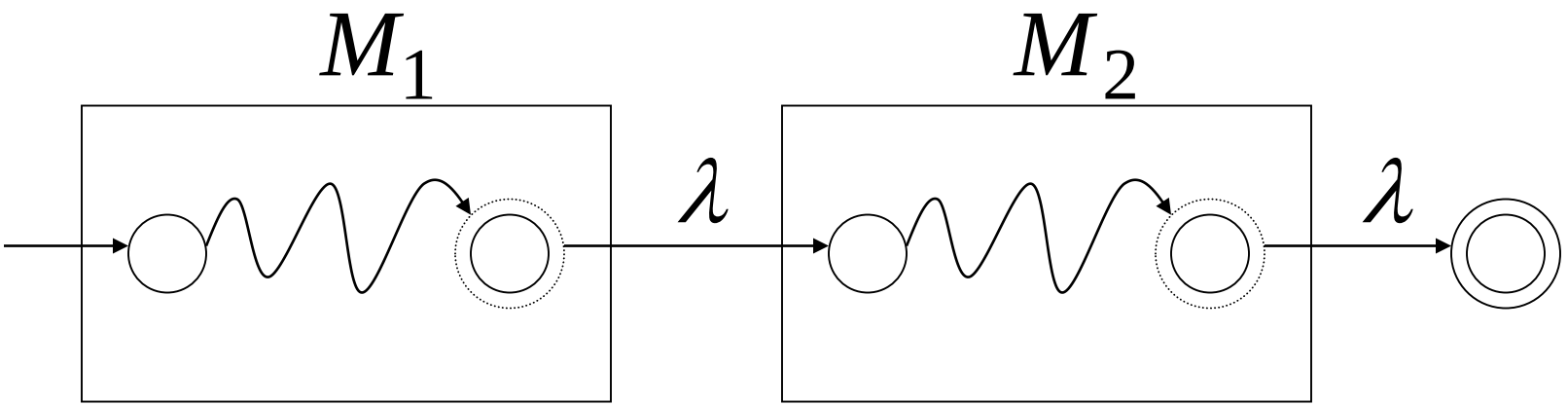
Example

$$L_1 \cup L_2 = \{a^n b\} \cup \{b, a\}$$



Concatenation

$L_1 L_2$

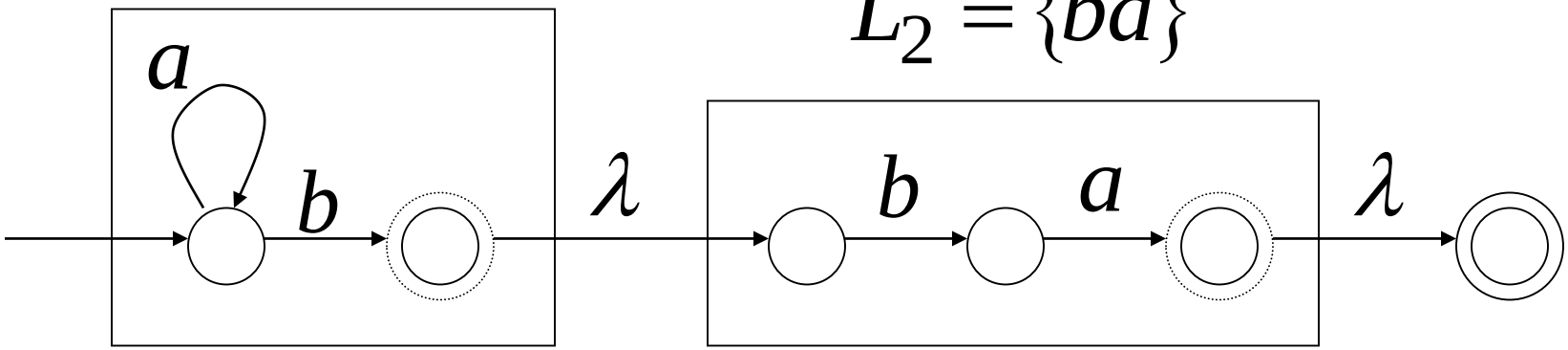


Example

$$L_1 L_2 = \{a^n b\}\{ba\} = \{a^n bba\}$$

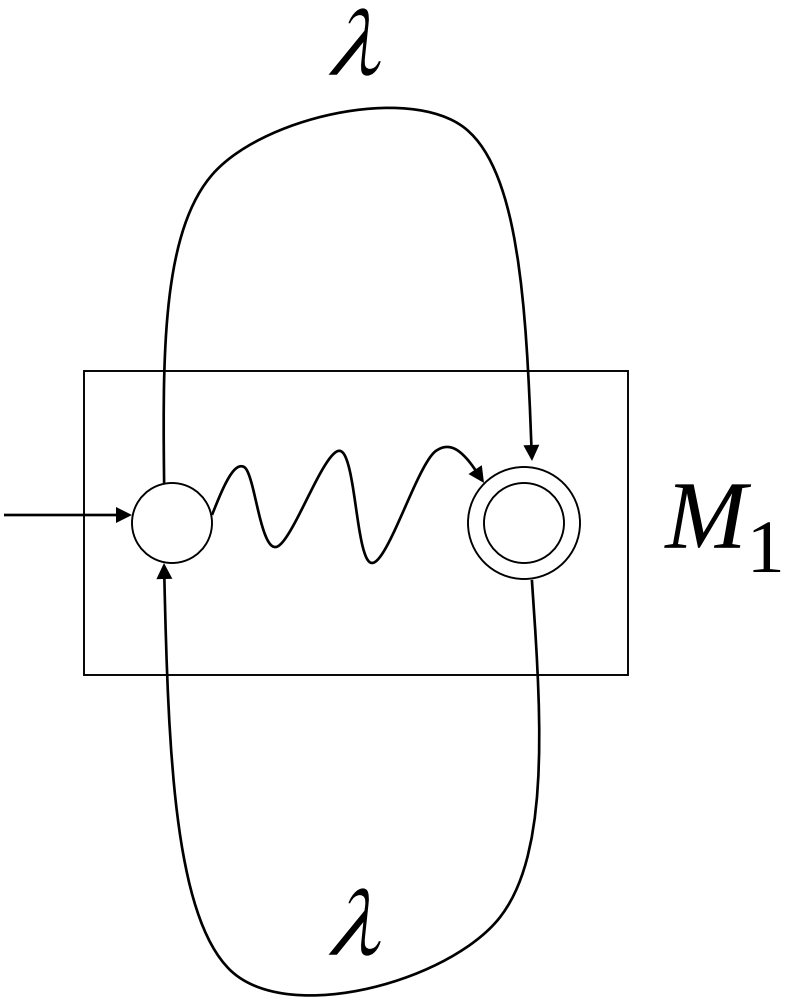
$L_1 = \{a^n b\}$

$L_2 = \{ba\}$



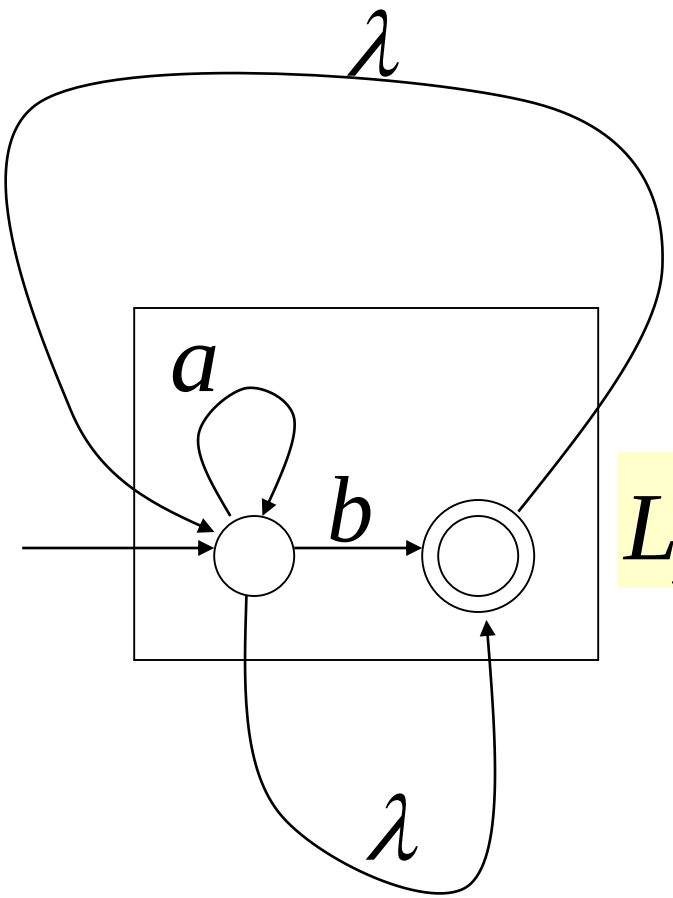
Operation * (Kleene star)

L_1^*



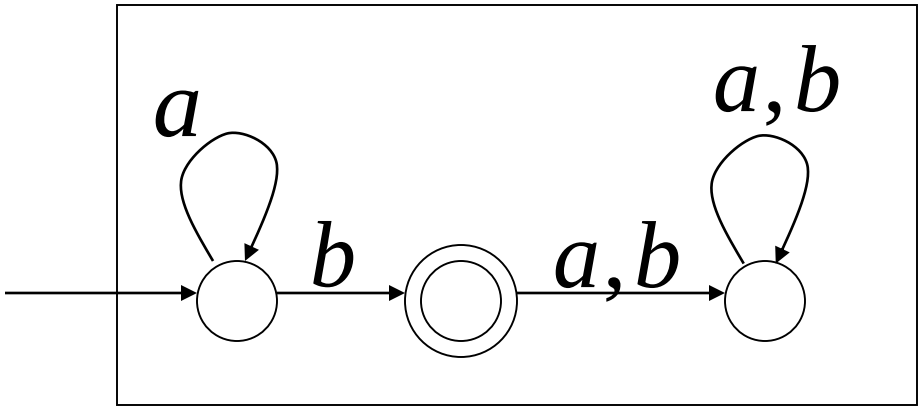
Example *

$$L_1^* = \{a^n b\}^*$$



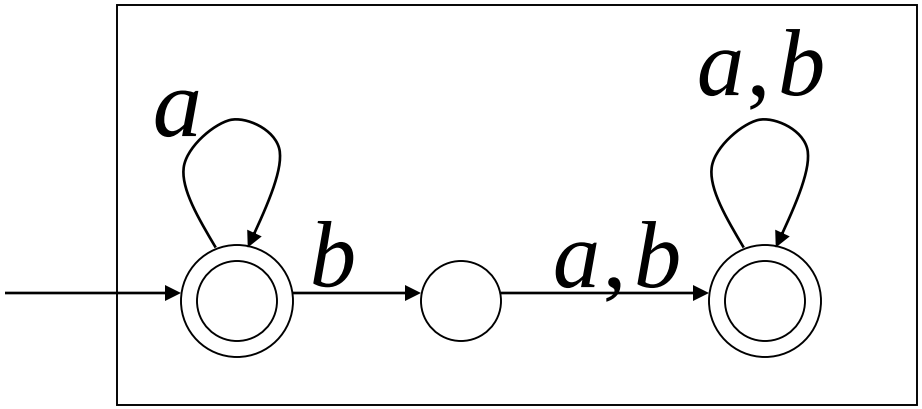
$$L_1 = \{a^n b\}$$

Complement of Language



$$L = \{a^n b\}$$

F_{neg} = S-F , *by words: we invert end states*



$$\overline{L} = \overline{\{a^n b\}}$$