UiO : Department of Informatics
University of Oslo

IN3160 IN4160

Datapath state machines

Yngve Hafting

Course Goals and Learning Outcome

https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html

In this course you will learn about the **design of advanced digital systems**. This includes programmable logic circuits, a **hardware design language** and system-on-chip design (processor, memory and logic on a chip). **Lab assignments provide practical experience in how real design can be made**.

After completion of the course you will:

- understand important **principles for design** and testing of digital systems
- understand the relationship between behaviour and different construction criteria
- **be able to describe advanced digital systems at different levels of detail**
- be able to perform simulation and synthesis of digital systems.

Goals for this lesson:

- Know what is
  - Datapath state machines (FSMD)
- Know how to divide larger designs and state machines
  - Principles
  - Design strategies
    - Divide and conquer-

Next lesson:

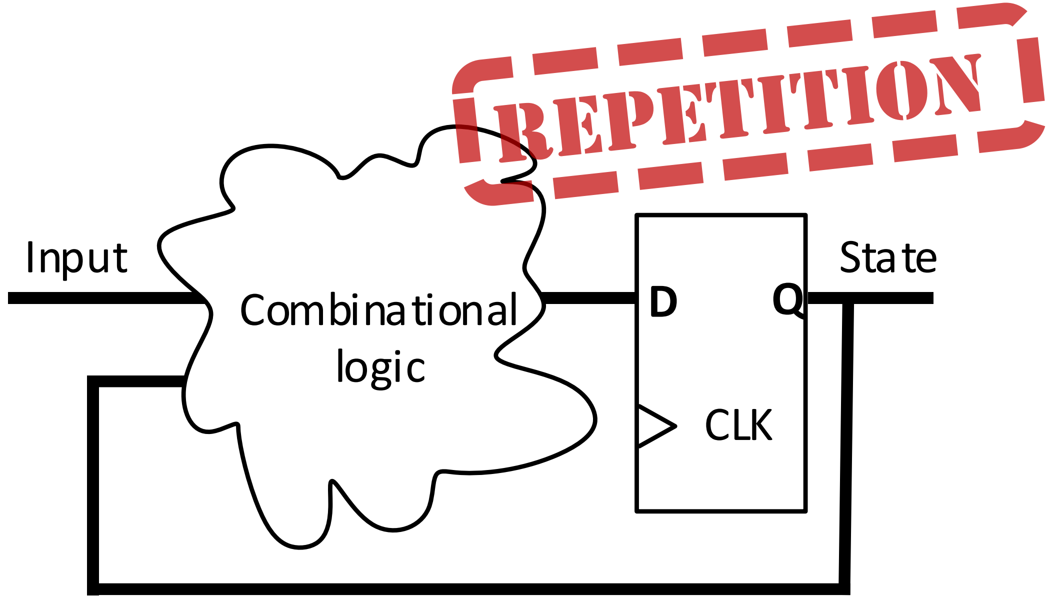
- Diagrams and schematics?
- ~~Microcoded state machines~~
- ~~Microcoded processors~~

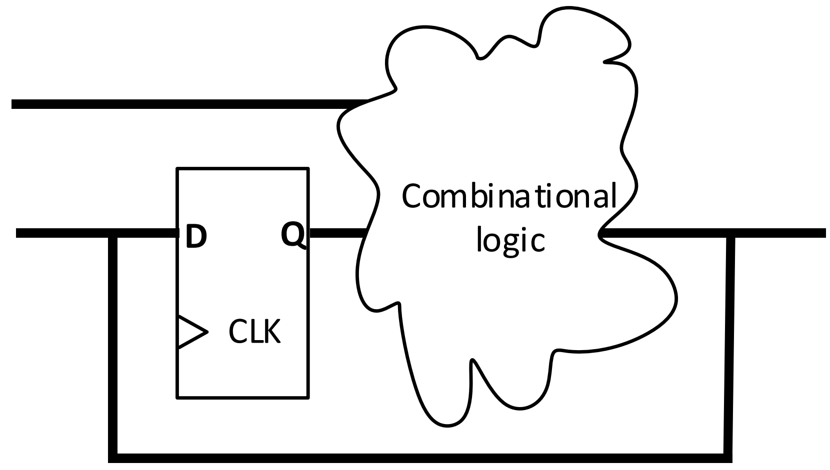
Overview

- What is data path finite state machines (FSMD)?
  - Example with code and diagrams
- Factoring state machines
  - When and how do we split
- Next lesson:
  - Examples with diagrams and code

General FSM

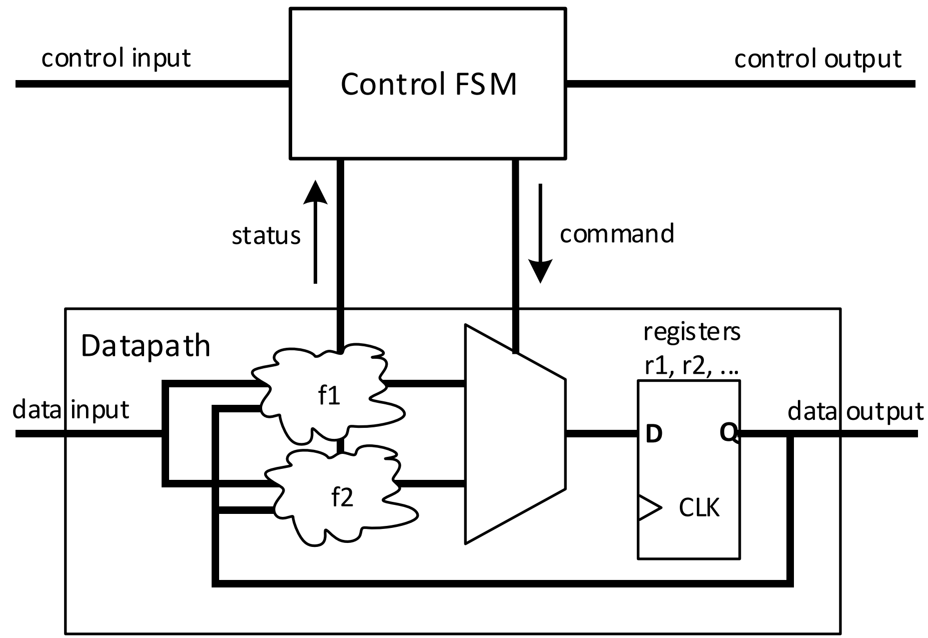
- General FSM
  - Combinational logic connected to registers with feedback





«Datapath» FSM

- Datapath is described by a function rather than a table
  - Counters
  - Mathematical operations
  - Shift registers
  - Etc.
- We usually divide into control FSM and Datapath



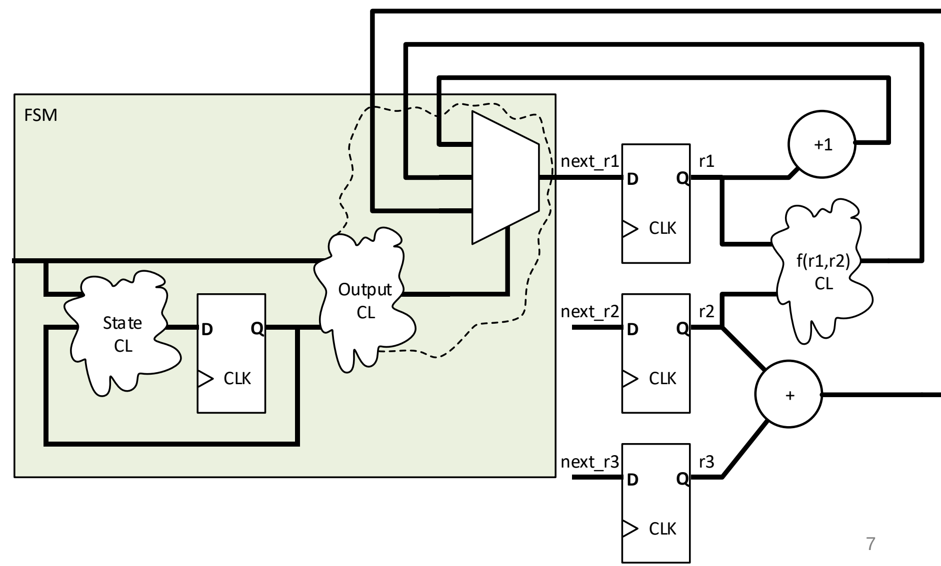
«Register operations» in data-path FSM (FSMD) -and how to deal with it

- Common notations for register operations:
  - on clock edge we increment r1 → $r1 \leftarrow r1 + 1$
  - on clock edge we update r1 based on a function of register outputs → $r1 \leftarrow f(r1,r2)$
  - on clock edge, set r1 to r2+r3 → $r1 \leftarrow r2 + r3$

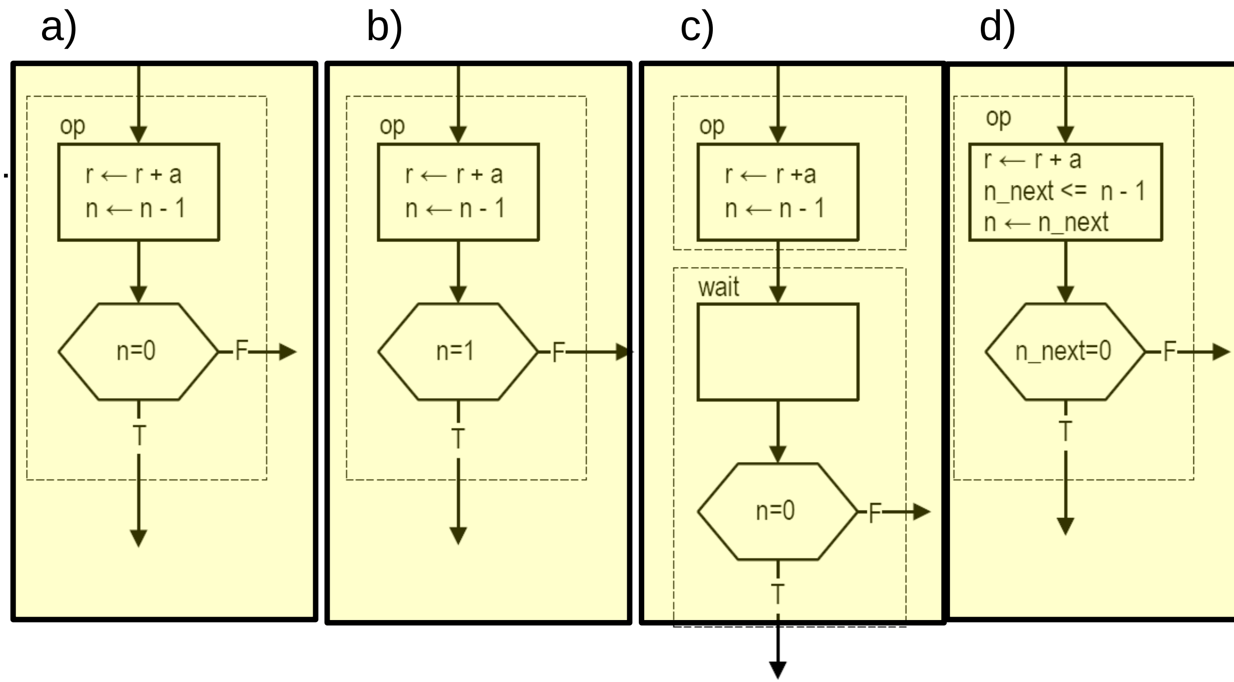
This notation can be confusing, as it implies one clock delay if it is put into an ASM chart.

Solution:

Use '←' for datapath only (not for FSM)
Know that '←' implies the use of registers that are not a part of the FSM states



Use of register in decision box



a) n will be updated after n=0 check

b) n will be updated after n=1 check…

- Even if we want this behavior, it is poor design…
  - it seems we do not know what we are doing, as with a).

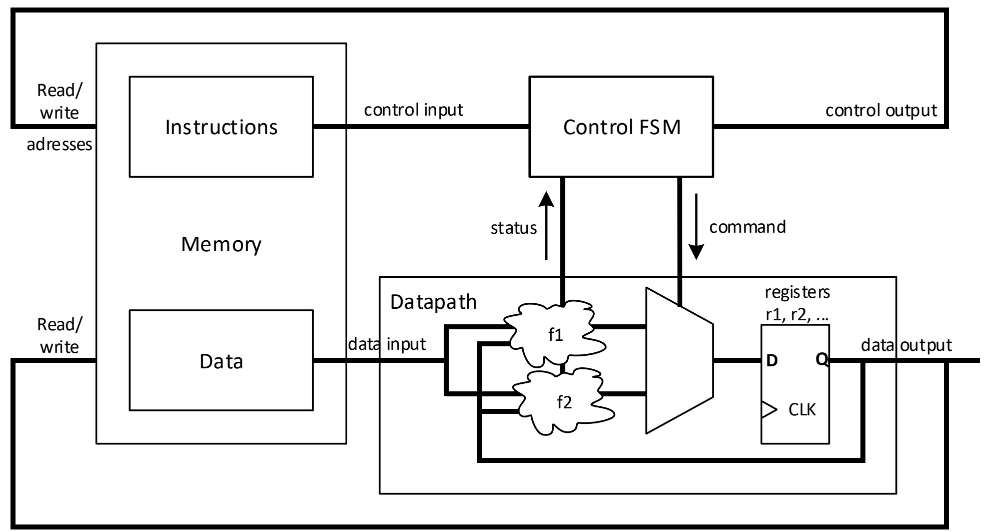
c) Do we need to introduce single cycle wait states?..

d) is clear about
  - ***what we want*** and
  - ***how we will do** it*
  - => no doubt on our intention

- Register is updated when the FSM exits current state
  - NOTE*: We "exit" current state each cycle- even* if we *re-enter*…
- => Use solution d)!

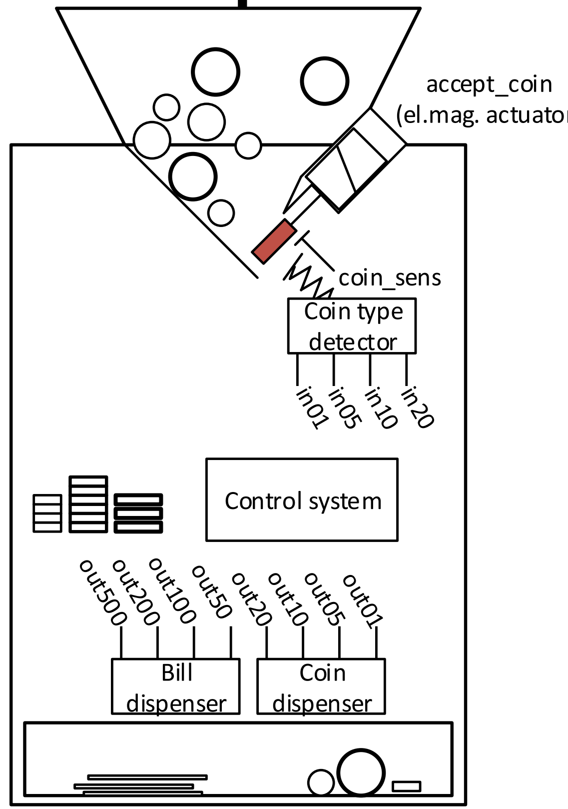
Processor system is a datapath FSM

- Control output is memory instructions
- FSM decodes instructions and decides which part of the datapath is used
  - Pipeline flushes, stalls etc.
- Datapath contains ALU, pipeline registers etc.



Example Factoring state machine with Datapath

- Exhange machine
  - Green LED ‘ready’/ can accept coins
  - Can take a number of up to 100 coins
    - Count each coin type
    - 1, 5, 10, 20 NOK
    - Close intake at maximum (! Green)
    - Close intake when counting
    - Close intake when no more coins (assume new coin each clock edge)
  - Give out the highest possible bills (assuming infinite supply)
    - 50, 100, 200 NOK
  - Return the least amount of coins
    - Use only coin from machine



When state count is nuts…

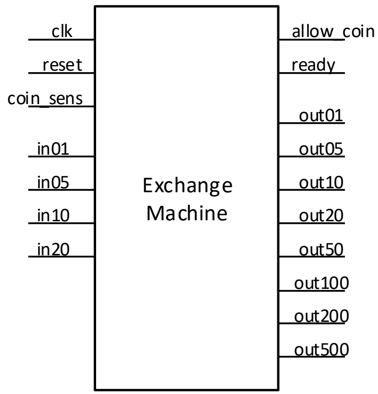
- Millions of states possible => Cannot make «one» FSM

=> several smaller state_machines or
state machine + data path with registers

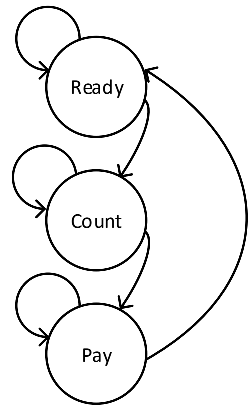
Divide into models that can be conquered

- Partition by..?
  - State (FSMs vs datapath),
  - Task (counters, FSMs,…)
  - Interface (entities)

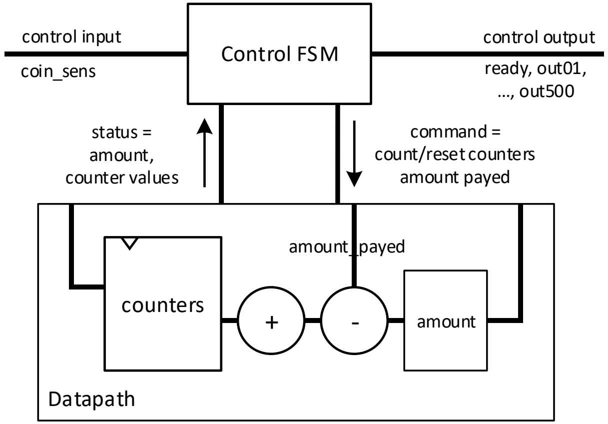
- Entity:



- FSM(s)

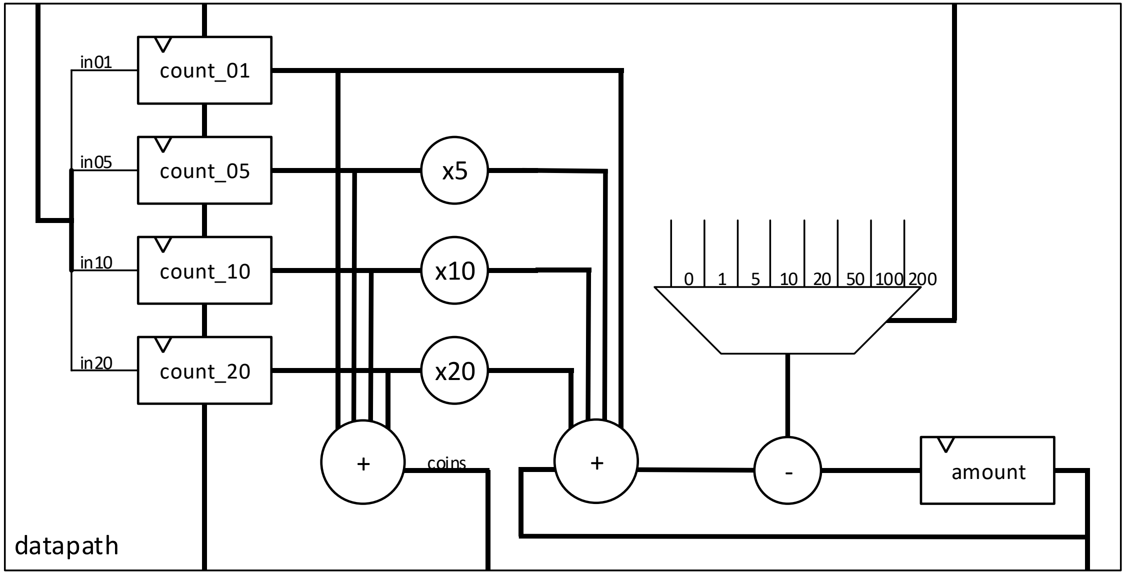


- Datapath



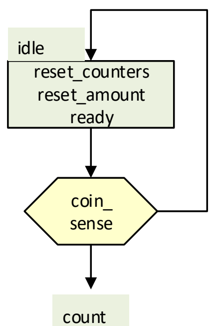
Detailed datapath

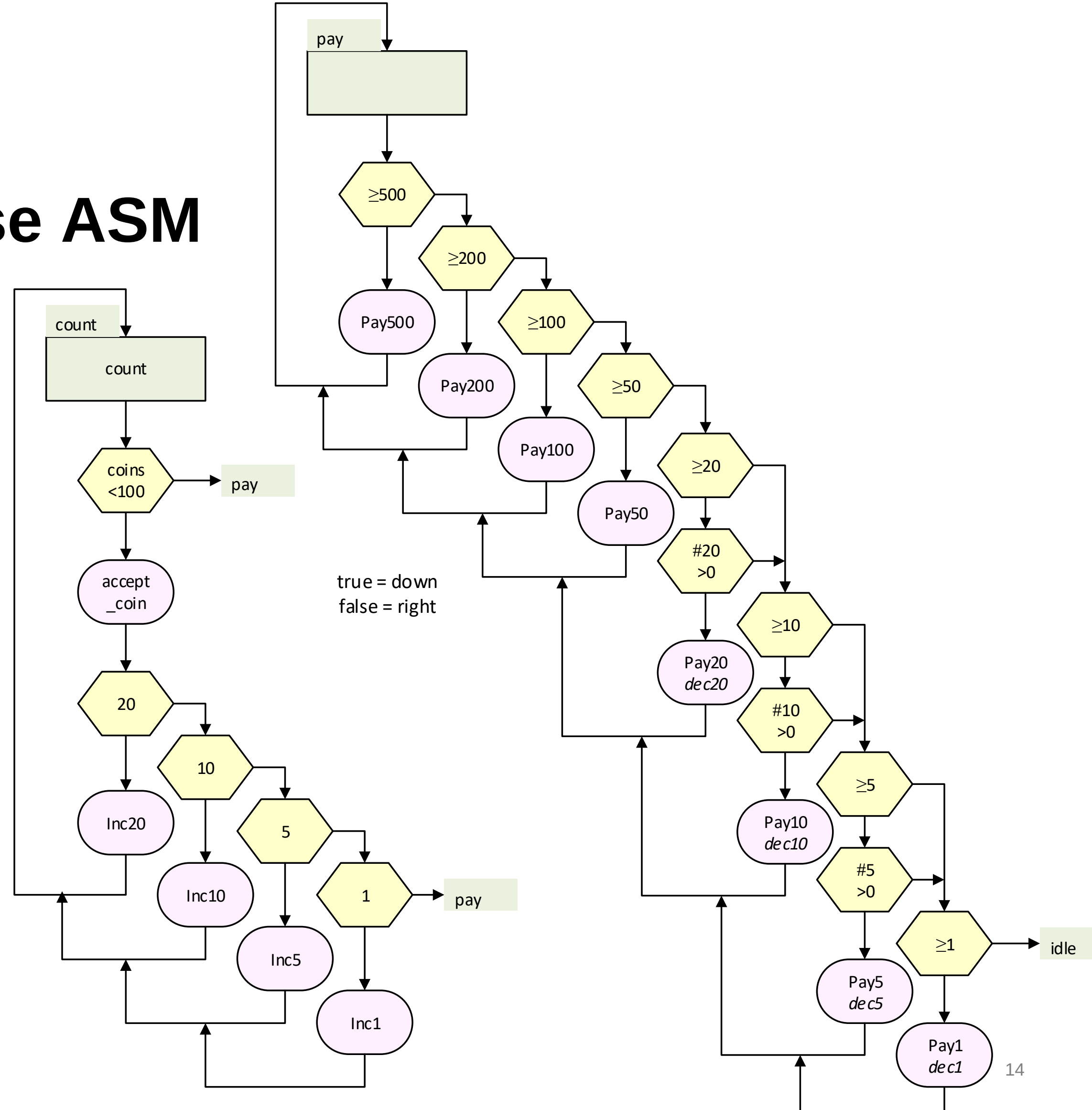
- 4 counters
  - Can they be of the same type?
  - Up/ down / reset
- «Coins» and «amount»
  - Why/ why not registers?



Detailed FSM = use ASM

- Make sure
  - all transition descisions are covered
  - all control output is set



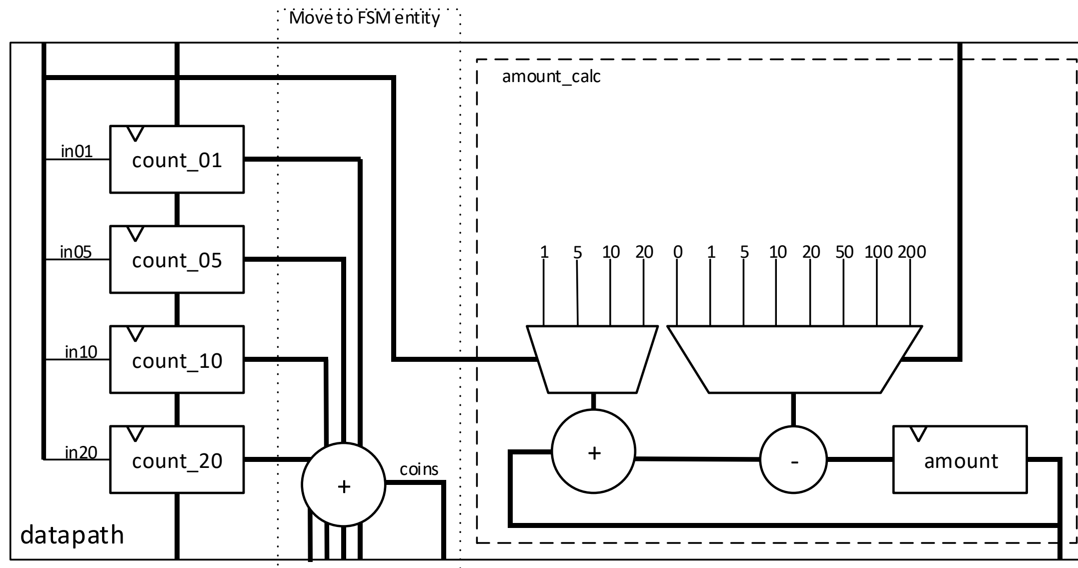


Reiterate and refine

- You will likely need a couple of rounds refining before deciding on VHDL modules
  - Entity
  - FSM(s)
  - Detailed datapath
  - ASM diagrams

Example reiteration

- Simpler by using
  - Only increments or decrements for amount calculation

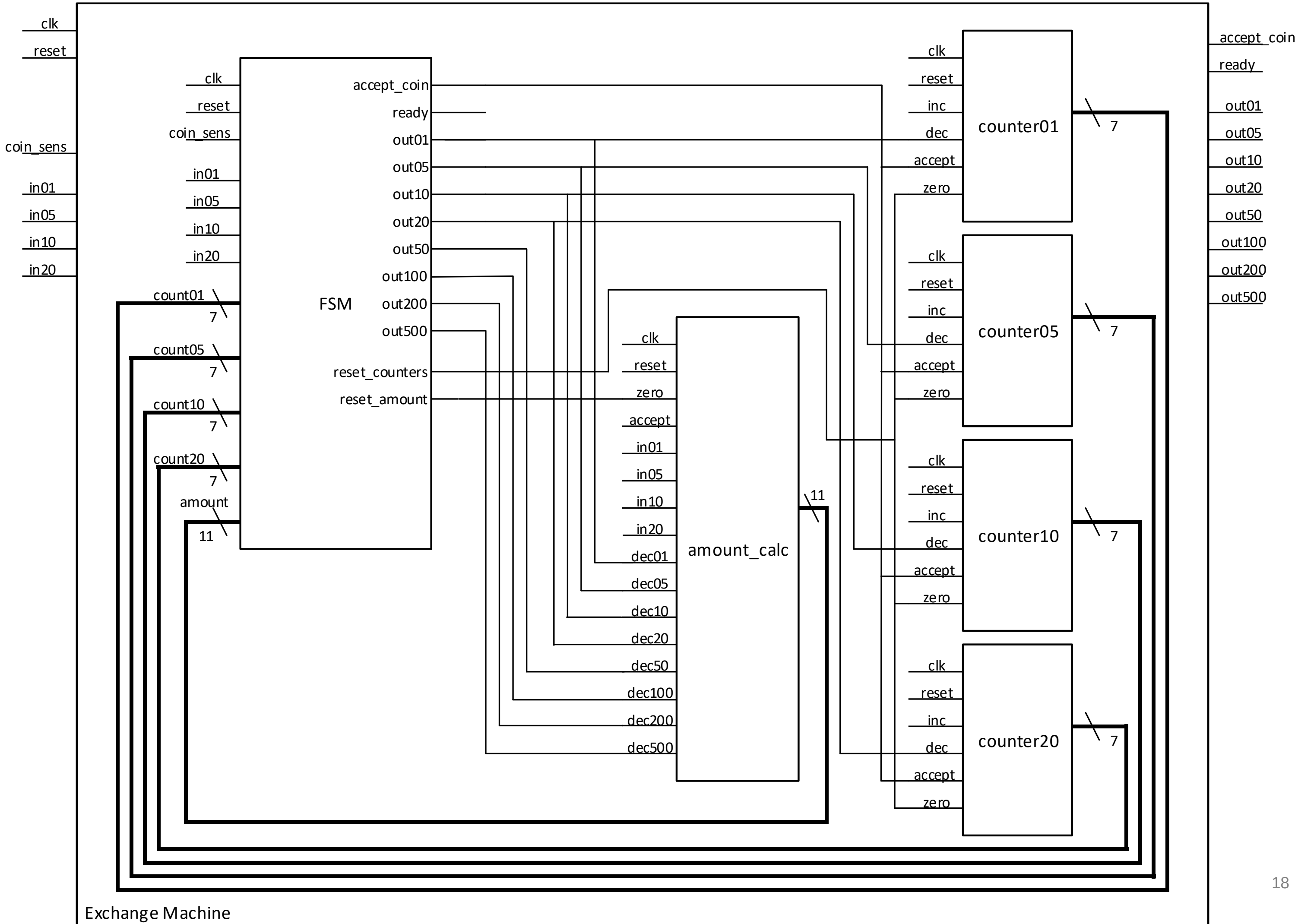


VHDL modules and hierarchy

- What makes a good hierarchy?
  1. Structural top
  2. RTL
  3. data flow modules
  - Complex designs may have several structural layers
    - Do not overdo this
- What makes good modules..?
  - One type of code within module
    - (Structural vs RTL vs Data Flow)
  - One purpose for each module
  - Loosely coupled / few dependencies
    - Minimum communication between modules
    - Changes can be made within one module without changing an other
  - Little or no duplicate code…
    - Use functions, loops, constants etc.
  - Scalable

- Example modules:
  - Toplevel (structural)
  - Control FSM
  - Counter(s)
    - One VHDL module, four instances
  - ~~D[illegible]path..?~~
    - ~~(co[illegible]e within toplevel…)~~
    - ~~Datapath [illegible].?~~
  - *Amount calculation*

Top



Only internal signals are connected in drawing

Toplevel shell

- Filling in the rest should be easy once the modules are ready
- We need names for signals that go between modules.

```vhdl
library IEEE;
  use IEEE.STD_LOGIC_1164.all;

entity exchange_machine is
  port(
    clk, reset                        : in std_logic;
    coin_sens, in01, in05, in10, in20 : in std_logic;
    ready, accept_coin                : out std_logic;
    out01, out05, out10, out20        : out std_logic;
    out50, out100, out200, out500     : out std_logic
    );
end entity exchange_machine;

architecture toplevel of exchange_machine is
  component control_FSM is
  port(
    clk, reset : in std_logic);

  component counter is
  port(
    clk, reset : in std_logic);

  component amount_calc is
  port(
    clk, reset : in std_logic);

-- signal decl. for communication between modules

begin
  FSM: control FSM
  port map(
    clk => clk,
    reset => reset);

  count01: counter
  port map(
    clk => clk,
    reset => reset);

  count05: counter
  port map(
    clk => clk,
    reset => reset);

  count10: counter
  port map(
    clk => clk,
    reset => reset);

  count20: counter
  port map(
    clk => clk,
    reset => reset);

  amount: amount_calc
  port map(
    clk => clk,
    reset => reset);

end architecture toplevel;
```

Counter

- Processes *can be* used to sort priority by order
  - OK when conditions are mutually exclusive?
- When-else can do the same sorting explicitly
  - Less need for process..

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;

entity counter is
generic(
    COUNT_WIDTH : natural := 7);
  port(
    clk, reset  : in std_logic;
    inc, accept : in std_logic;
    dec, zero   : in std_logic;
    count       : out unsigned(COUNT_WIDTH-1 downto 0));
end entity counter;

architecture RTL of counter is
  signal next_count : unsigned(count'range);
begin
  -- registry update
  count <= (others => '0') when reset else next_count when rising_edge(clk);

  --next count CL
  next_count <=
    count + 1 when inc and accept else
    count - 1 when dec else
    (others => '0') when zero else
    count;



end architecture RTL;
```

amount_calc

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;

entity amount_calc is
  generic(
    -- 100*20 = 2000 < 2048 = 2^11.
    AMOUNT_WIDTH : natural := 11);
  port(
    clk, reset                    : in std_logic;
    in01, in05, in10, in20        : in std_logic;
    zero, accept_coin             : in std_logic;
    dec50, dec100, dec200, dec500 : in std_logic;
    dec20, dec10, dec05, dec01    : in std_logic;
    amount: out unsigned(AMOUNT_WIDTH-1 downto 0));
end entity amount_calc;
```

- Process + if because..
  - Use of priority
  - Several levels
    - Single output *can be resolved using when-else* only
      - Readability/Maintainability would suffer (…`and accept_coin x 4`)

```
 architecture RTL of amount_calc is
  signal next_amount : unsigned (amount'range);
begin
  -- registry update
  amount <=
    (others => '0') when reset else
    next_amount when rising_edge(clk);

  -- CL next_amount
  process(all) is
  begin
    -- default statement:
    next_amount <= amount;
    -- conditional statements (priority doesnt matter)
    if zero then
      next_amount <= (others => '0');
    elsif  accept_coin then
      next_amount <= amount +  1 when in01;
      next_amount <= amount +  5 when in05;
      next_amount <= amount + 10 when in10;
      next_amount <= amount + 20 when in20;
    else
      next_amount <= amount - 500 when dec500;
      next_amount <= amount - 200 when dec200;
      next_amount <= amount - 100 when dec100;
      next_amount <= amount -  50 when  dec50;
      next_amount <= amount -  20 when  dec20;
      next_amount <= amount -  10 when  dec10;
      next_amount <= amount -   5 when  dec05;
      next_amount <= amount -   1 when  dec01;
    end if;
  end process;

end architecture RTL;
```

FSM

```vhdl
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;

entity control_FSM is
  generic(
    COUNT_WIDTH : natural := 7;
    AMOUNT_WIDTH : natural := COUNT_WIDTH+4;
    COIN_LIMIT : natural := 100);
  port(
    clk, rese : in std_logic;
    coin_sens, in01, in05, in10, in20 : in std_logic;
    count01 : in unsigned(COUNT_WIDTH-1 downto 0);
    count05 : in unsigned(COUNT_WIDTH-1 downto 0);
    count10 : in unsigned(COUNT_WIDTH-1 downto 0);
    count20 : in unsigned(COUNT_WIDTH-1 downto 0);
    amount  : in unsigned(AMOUNT_WIDTH-1 downto 0);
    ready, accept_coin             : out std_logic;
    out01, out05, out10, out20     : out std_logic;
    out50, out100, out200, out500 : out std_logic;
    reset_counters, reset_amount  : out std_logic);
end entity control_FSM;
```

```vhdl
architecture RTL of control_FSM is
  type state_type is (idle, count, pay);
  signal current_state, next_state : state_type;
  signal coins : unsigned(COUNT_WIDTH-1 downto 0);
begin
  -- clocked logic
  current_state <=
    idle when reset else
    next_state when rising_edge(clk);

  -- CL (moved from datapath)
  coins <= count01 + count05 + count10 + count20;

  next_state_cl: process(all) is
  begin
    -- default value prevents latches
    next_state <= current_state;
    case current_state is
      when idle =>
        next_state <= count when coin_sens;
      when count =>
        next_state <= pay when coins > COIN_LIMIT-1;
        next_state <= pay when not (in01 or in05 or in10 or in20);
      when pay =>
        -- this should be equivalent to all tests listed
        next_state <= idle when or(amount) = '0';
    end case;
  end process;

-- more next slide…
```

FSM 2/2

```
output_cl: process(all) is
  begin
    -- default values to prevent latching
    reset_counters <= '0';
    reset_amount   <= '0';
    ready          <= '0';
    accept_coin    <= '0';
    out01          <= '0';
    out05          <= '0';
    out10          <= '0';
    out20          <= '0';
    out50          <= '0';
    out100         <= '0';
    out200         <= '0';
    out500         <= '0';
```

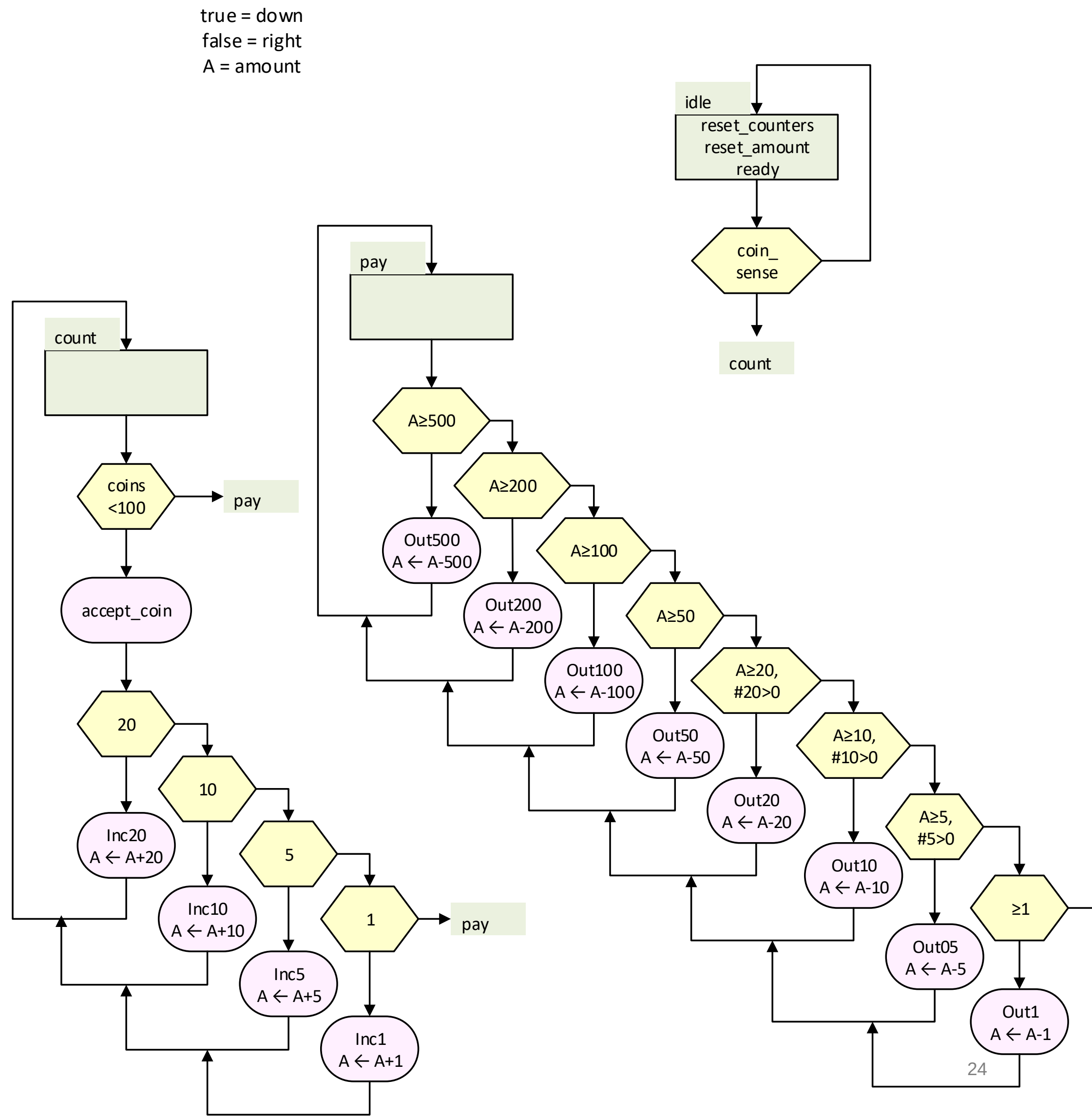
NOTE: with this prioritation order, the sequence becomes more complex than necessary.

```
    case current_state is
      when idle =>
        reset_counters <= '1';
        reset_amount   <= '1';
        ready <= '1';
      when count =>
        accept_coin <= '1' when coins < COIN_LIMIT;
      when pay =>
        if amount >= 500 then out500 <= '1';
        elsif amount >= 200 then out200 <= '1';
        elsif amount >= 100 then out100 <= '1';
        elsif amount >=  50 then  out50 <= '1';
        elsif amount < 50 and amount >= 20 then
          if count20 > 0 then out20 <= '1';
          elsif count10 > 0 then out10 <= '1';
          elsif count05 > 0 then out05 <= '1';
          else out01 <= '1';
          end if;
        elsif amount < 20 and amount >= 10 then
          if count10 > 0 then out10 <= '1';
          elsif count05 > 0 then out05 <= '1';
          else out01 <= '1';
          end if;
        elsif amount < 10 and amount >= 5 then
          if count05 > 0 then out05 <= '1';
          else out01 <= '1';
          end if;
        elsif amount < 5 and amount >= 1 then
          out01 <= '1';
        end if;
    end case;
  end process;

end architecture RTL;
```

- Readability?
  - What would happen if we mixed next_state CL into the output CL?
- Default values for all signals
  - => no latches
    - No need for **else** after **when** or **if** since default clause will apply.
- Use «**if**» to sort priorities, when having multiple conditions and multiple outputs
  - That are depending on each other..

Recap ASMD

- D for datapath ASM
- '←' in a Mealy box?
  - OK because the register is a part of the data path (*and not the FSM itself*)
- Can we go without '←' ?
- Should we?



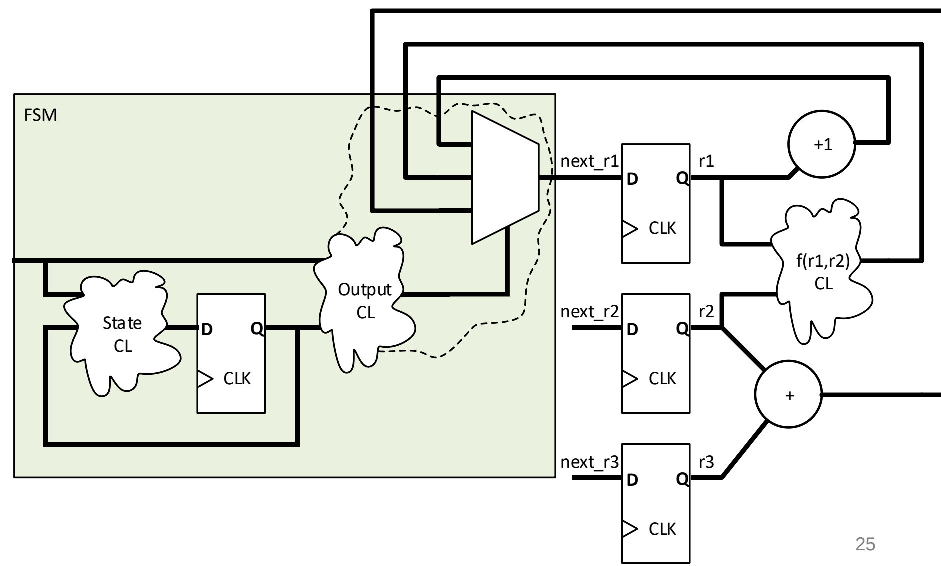
REPETITION

«Register operations» in data-path FSM (FSMD) -and how to deal with it

- Common notations for register operations:
  - on clock edge we update r1 based on a function of register outputs
  - on clock edge we increment r1,
  - on clock edge, set r1 to r1+r2

r1 ← r1 + 1
r1 ← f(r1,r2)
r1 ← r2 + r3

- This notation can be confusing, as it implies one clock delay if it is put into an ASM chart.
- Solution:
  - **Use '←' for datapath only** (not for FSM)
  - Know that '←' implies the use of additional registers



Suggested reading

- D&H:
  - 17 p 375 - 393
  - 21 p 467 – 477

- Hva nå? <=

```
next_page when time_left > 15 min else questions ..?
```

Pushbutton register storage (not in Oblig 6 2021)

- **All storage elements should be clocked!**
- *should reset have top priority?*
  - Use asynchronous reset for the time being (topic for later)

```
process(all) is
  begin
    if enable then
      output <= input;
    elsif reset then
      output <= "00000000";
    end if;
end process;
```

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;

entity my_reader is
  port(
    clk, reset : in std_logic;
    enable : in std_logic;
    input : in std_logic_vector(7 downto 0);
    output : out std_logic_vector(7 downto 0)
    );
end entity my_reader;
```

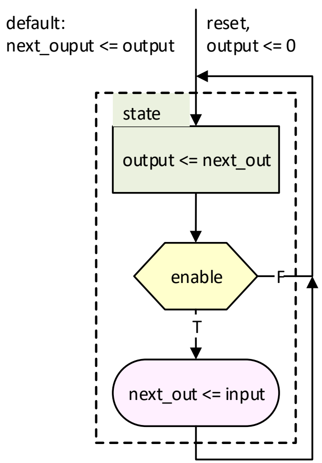
```
architecture single_process of my_reader is
begin
  process(clk, reset) is
  begin
    if reset then
      ouput <= (others => '0');
    elsif rising_edge(clk) then
      ouput <= input when enable;
    end if;
  end process;
end architecture;
```

```
architecture two_statement of my_reader is
  signal next_out: std_logic_vector(7 downto 0);
begin
  process(clk, reset) is
  begin
    if reset then
      ouput <= (others => '0');
    elsif rising_edge(clk) then
      ouput <= next_out;
    end if;
  end process;
  -- latched input : Don't do this
  next_out <= input when enable /*else next_out*/ ;

  -- CL alternative
  with enable select next_out <=
    input when '1',
    output when others;

end architecture FSM_style;
```

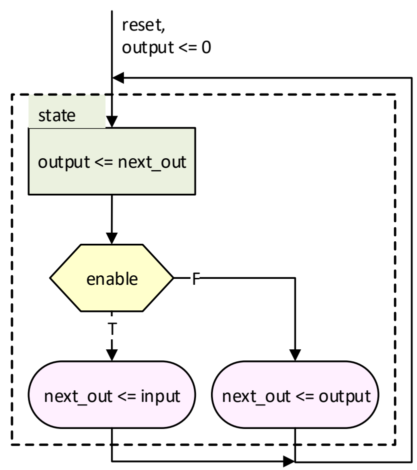
Pushbutton register storage

- Can be seen as a single state storage operation

- With default value



- Without default



- As a register operation