UiO : Department of Informatics
University of Oslo

IN3160 IN4160
Finite State Machines

Yngve Hafting

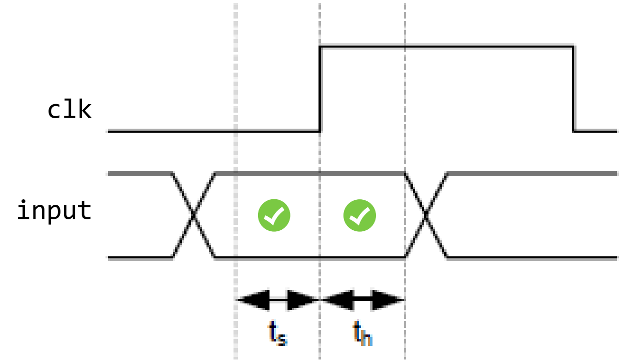
Messages

- Lab supervisor feedback
  - Assignment 1b):
    - about 40-50% has errors according to specification
      - Task as a whole may be approved
      - => Read comments
        - » Later assignments you will need to meet specification
        - » When in doubt ask the supervisors…
  - QnA- on zoom Tuesday: 2 participants..
    - Did everyone see the video-lecture..?
  - Guest Lecture by Espen Tallaksen from emLogic 7.4
    - Will be announced further later.

More on attributes (from last lecture)

- There are attributes for
  - Signals
    - (previous slides)
  - Types
    - Notable:
      - 'image(v) returns a string ex : `report("current value is: ", integer'image(my_int));`
      - 'value(s) returns a value (opposite of 'image) `integer'value(my_str);`
  - Array types/objects (vectors)
    - 'left, 'right, 'low, 'high, 'range, 'reverse_range, 'length, 'ascending (= false when «downto»), 'element (== subtype of the vector)
  - Entities
    - attributes to get compiled name hierarchy- as seen in questa when selecting signals

Testcase:
Set-up/hold time in flipflops



- To avoid metastability (neither 0 nor 1), inputs must be stable some time before (set-up) and after (hold) clock edge

- Output will return to 0 or 1 after being in the metastable state, but it's not given which one.
  - This means; the system is no longer deterministic.

Timing and logic check

```
entity D_FF is

  port (D, Clk, Set, Reset: in std_logic;
        Q : out std_logic);
begin
  assert (not(Clk ='1' and Clk'EVENT and not D'STABLE(Setup)))
  report "Setup time violation" severity WARNING;
  assert (not(Clk ='1' and D'EVENT and not Clk'STABLE(Hold)))
  report "Hold time violation" severity WARNING;
  assert (not(Set ='0' and Reset = '0'))
  report "Set and Reset are both asserted"
  severity ERROR;
end entity D_FF;
```

- The stable attribute can be used to check set-up- and hold times
  - Returns true if a signal has been stable >= time given as input parameter
- Assert in an entity =>
  checking is being done for all architectures that belongs to this entity.

CAUTION! *Care should be taken using asserts. Vivado can only support static asserts that do not create, or are created by, behavior. For example, performing as assert on a value of a constant or a operator/generic works; however, as asset on the value of a signal inside an if statement will not work.*

Clock generator

- Asymmetric low and high time (dutycycle)

```
entity clock_gen is
  generic (Freq : REAL := 10.0;   -- MHz
           Mark : REAL := 0.3);   -- Mark length (0-1.0)
end entity clock_gen;

architecture cg of clock_gen is
  -- Mark time in us
  constant ClockHigh :TIME := (Mark/Freq)*us;
  -- Space time in us
  constant ClockLow :TIME := ((1.0-Mark)/Freq)*us;
  signal clock : std_logic := '0';
begin
    process is
      begin
      wait for ClockLow;
      clock <= '1';
      wait for ClockHigh;
      clock <= '0';
    end process;
end architecture cg;
```

Example: Clock with jitter

- Jitter:
  - (random) variable delay
  - Occurs naturally in all digital electronic
- math_real.uniform:

```
procedure UNIFORM(
  variable SEED1, SEED2 : inout POSITIVE;
  variable X : out REAL);
```

  - pseud-random number generator procedure
  - uniform distribution
  - alters seed values and sets rnd number

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

Entity RAND_CLOCK is
-- generic parameters
  generic (delay : DELAY_LENGTH := 100 ns);
  port(clock : out std_logic);
end entity RAND_CLOCK;

architecture RTL_RAND_CLOCK of RAND_CLOCK is

begin

RAND_CLK:
process
  variable seed1, seed2 : INTEGER := 42;
  variable rnd : REAL;
begin
  loop
    clock <= '0';
    uniform (seed1, seed2, rnd);
    wait for delay + (rnd - 0.5) * (10 NS);
    clock <= '1';
    uniform (seed1, seed2, rnd);
    wait for delay + (rnd - 0.5) * (10 NS);
  end loop;
end process;

end architecture RTL_RAND_CLOCK;
```

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 different types of state machines
  - What is a state machine
  - Moore type machines
  - Mealy type machines
- To specify state machine functionality using
  - State tables
  - State diagrams
  - Algorithic state machine diagrams
  - VHDL
- To know pro's and con's for
  - Moore and Mealy
  - Different state machine representations

Overview Today

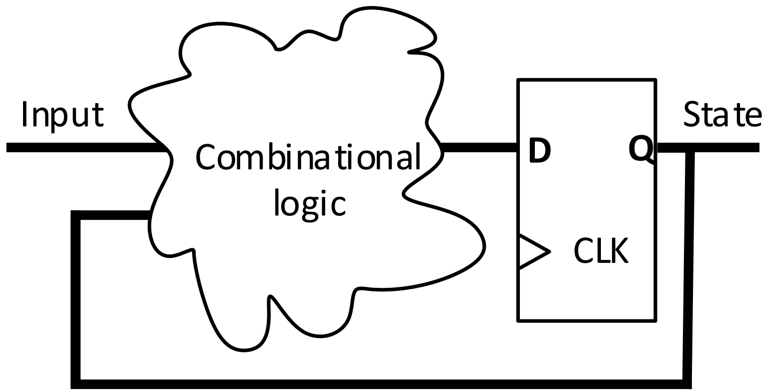
- What is finite state machines (FSM)?
  - General FSM
  - Moore type FSM
  - Mealy
  - Synchronized Mealy
- FSM representations
  - State diagram
  - State output table
  - Algorithmic State Machine (ASM) diagrams
- FSM example with VHDL and testbench

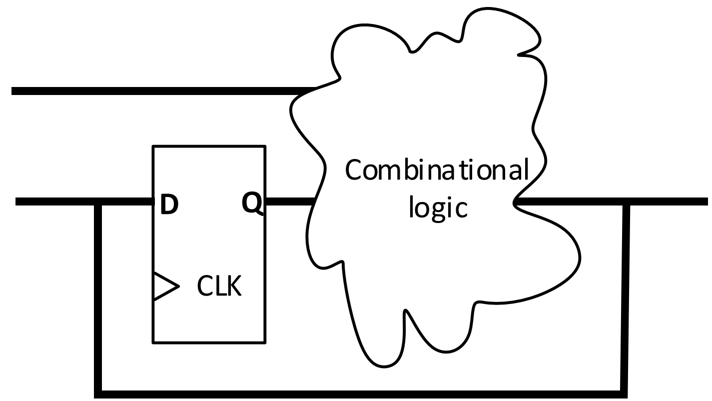
Next:

- Divide & Conquer
- Datapath state machines

General FSM

- General FSM
  - Combinational logic connected to registers with feedback
  - *Can be nearly any clocked logic*
    - Counter
    - LFSR
    - Shiftregister
    - Timer
    - Microprocessor
    - Vending machines…
    - Etc.

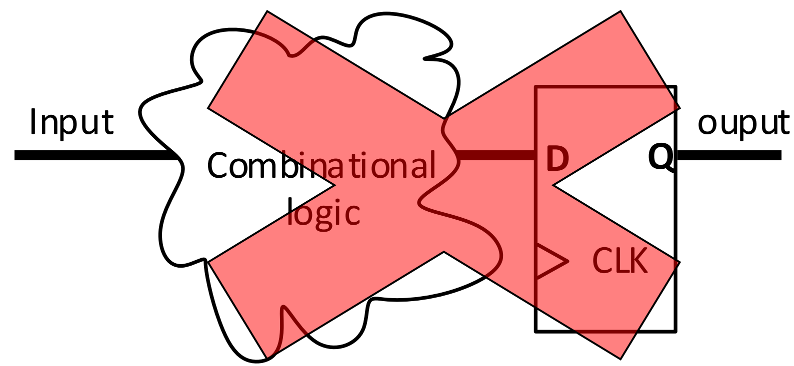




Non FSM sequental logic?

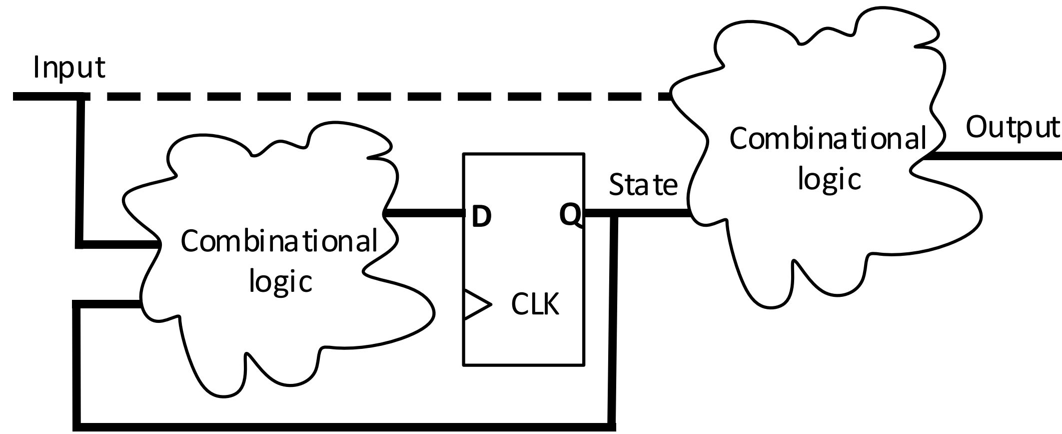
- *Strictly speaking* any logic using registers (FFs) are FSMs, but…

- We usually don't refer to things as FSMs when they
  - Have near-infinite states
    - Counters, Timers
    - Microprocessors
    - LFSR (linear feedback shift registers)
  - have other well known names:
    - Shift registers, …
  - are pure datapath representations
    - No feedback after registers

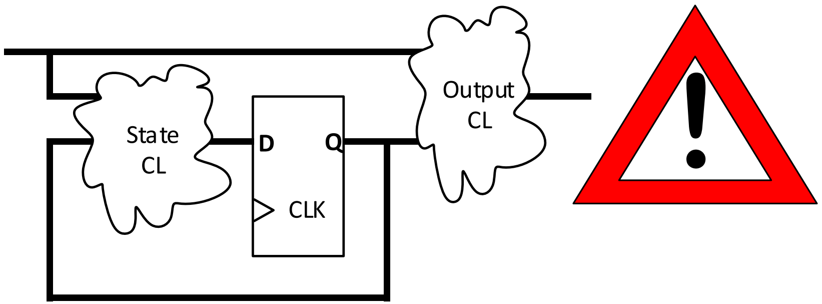


Output decoding in FSM

- Two types:
  - Moore
    - Output is entirely decoded from state registers
  - Mealy
    - Output is decoded from input and state registers



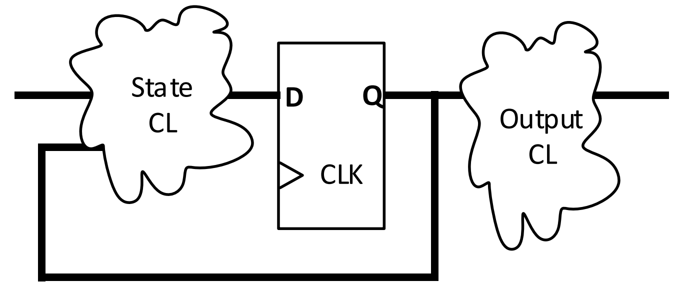
Mealy FSM



- Fast output
- Asynchronous output!

Hazards!

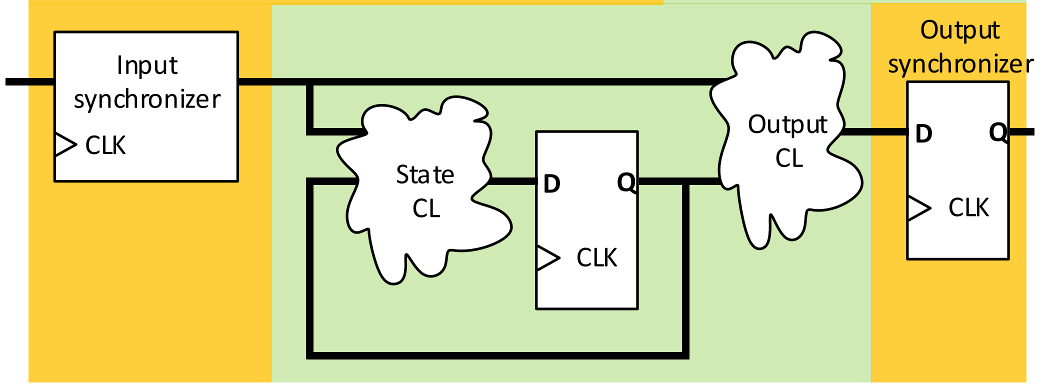
Moore FSM



- Output will always be delayed by at least one clock cycle
  - *Requires more states*

Output hazards still present, although synchronized

Solution: Fully synchronized Mealy FSM



- Input synchronizer: Synchronizes signals from other clock domains
- Output synchronizer: Removes hazards from output

- Technically this is a «Moore» type machine altogether
  - But we operate with **minimum delay within the state machine** design
  - **Synchronizers can be** added in **separate** modules, processes or statements
  - Thus it makes sense to refer to the state machine inside as Mealy type

Moore vs mealy conclusion:

- Can we afford having synchronization:

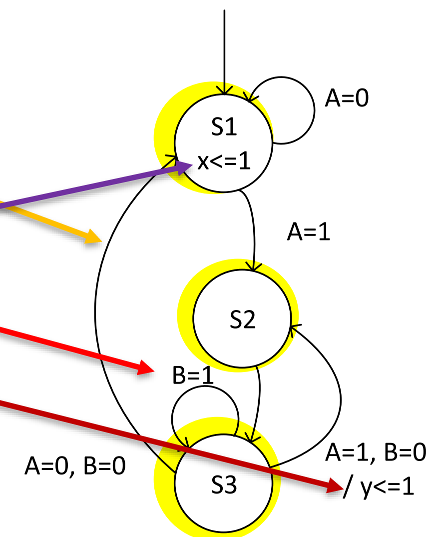
- If we cannot have other synchronization:

Three ways of representing state machines

- State diagram
- State (output) table
- Algorithmic state machine (ASM) diagram

State diagram

- States
- Transitions between states
- Beside transition arc:
  - Descision parameter
  - / Mealy output
- Inside bubble:
  - Moore output

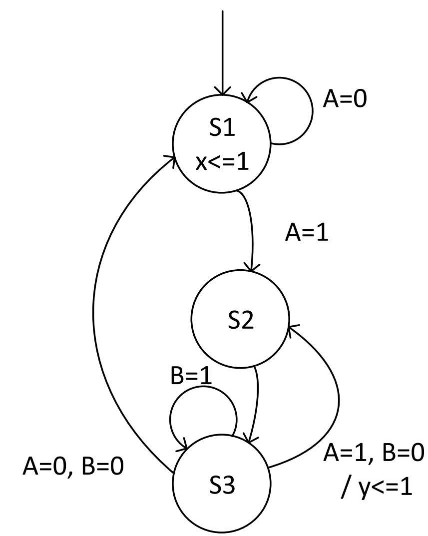


- Frequently used, but not always with all parameters.
- Note: *Default values often omitted*
  - Here: default: x, y = 0 (boolean false)

State output table, input & state table

Input(BA) /State	11	10	01	00	x	y
S1	S2	S1	S2	S1	1	0
S2	S3	S3	S3	S3	0	0
S3	S3	S3	S2	S1	0	$A \text{ and } \bar{B}$

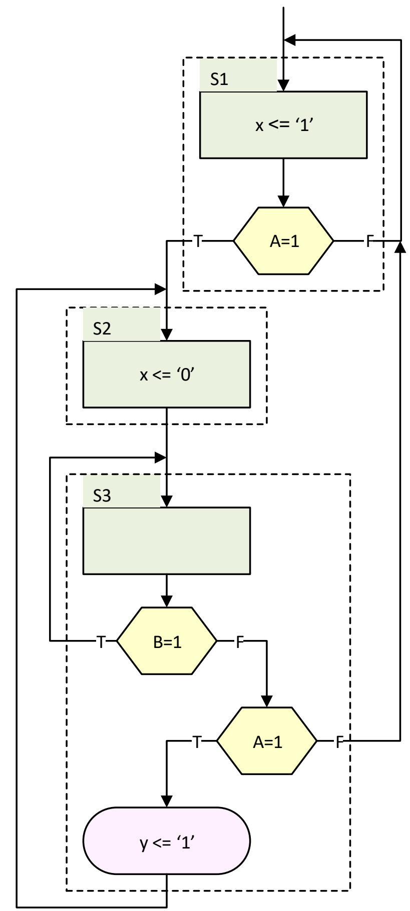
- Moore output is simple
- Mealy outputs becomes functions or table must be extended

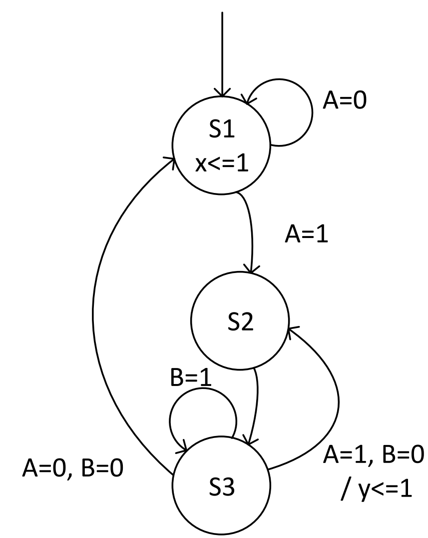


- *Next state and output* as a function of *input vs current state*
  - Can become large tables when having multiple outputs and inputs
  - Good representation when there are many exit paths from each state.

ASM chart (Algorithmic state machine)

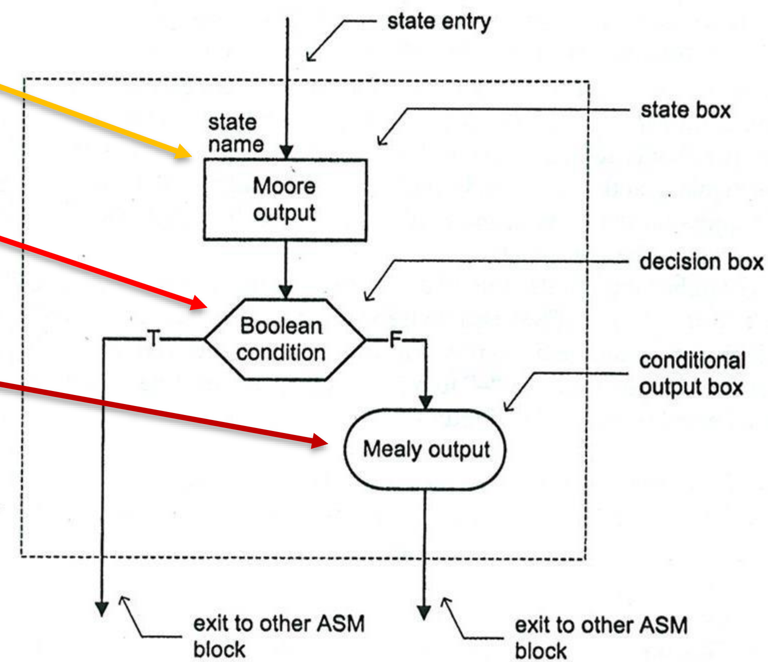
- Standardized way of displaying FSMs
- More descriptive than state diagram?
  - Always prioritized conditions
    - = Synthesizable
- Works well with boolean conditions for transitions and assignment
- Can become very large when having multiple exit paths for each state.



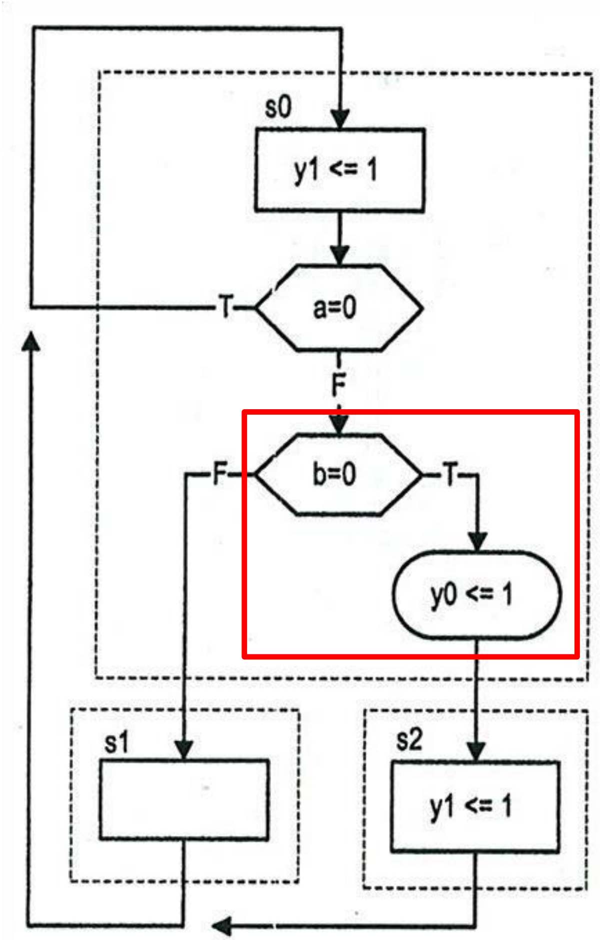


ASM (Algorithmic State Machine) block

- The **state box** represents a state in the FSM,
  - State based output is shown inside (i.e. the **Moore outputs**).
- The **decision box** tests an input condition to determine the exit path of the current ASM block.
- A **conditional output box ("Mealy box")**
  - lists conditionally asserted signals.
  - Can only be placed after an exit path of a decision box
  - (i.e. the **Mealy outputs** that depends on the state and input values).

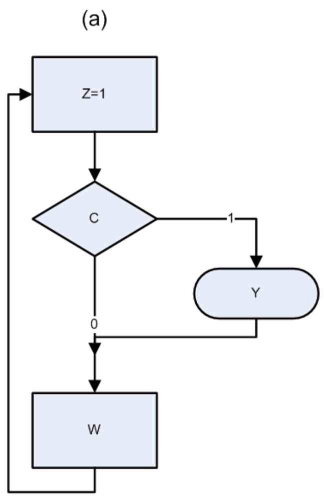


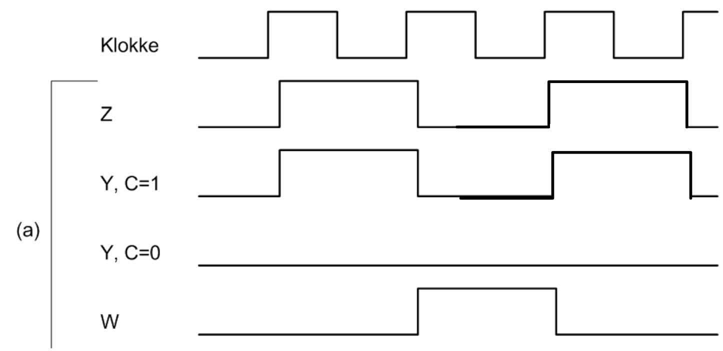
ASM Chart Example 1



- Conditional output (Mealy) box can only be placed after an exit path of a decision box.
- **<=** is used for assigning signal values
  - *Don't expect full consistency… some will use "=“*
- Unless specified (assigned) values are assumed to take their default values
  - Except when we introduce register operations which is noted with '←'
    - Registers will be updated on the next clock cycle
    - This can cause great confusion (be careful)
- Signals that are boolean are assumed set or found active ('1') when mentioned alone.
  - Here: we could have seen
    - "y1" in place of "y1<='1'" and
    - "not b" in place of "b=0"

ASM chart example 2, Mealy vs Moore output

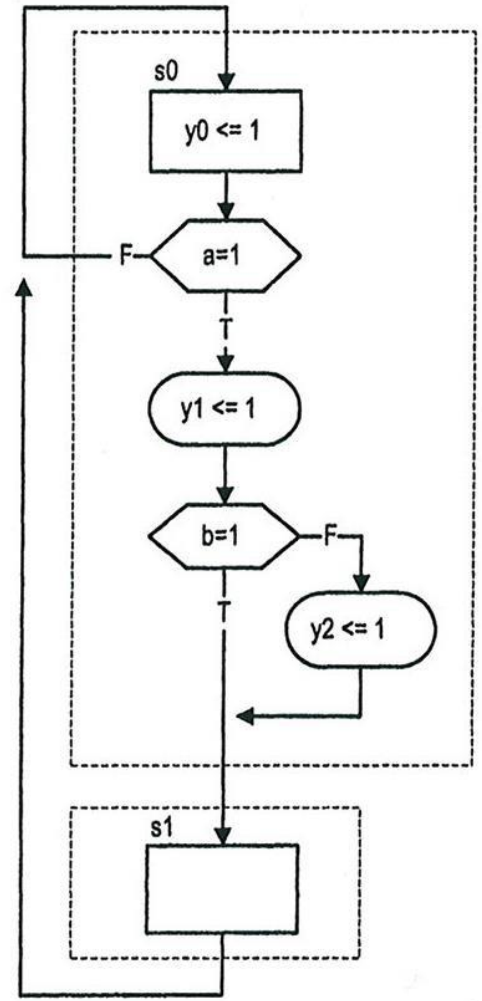




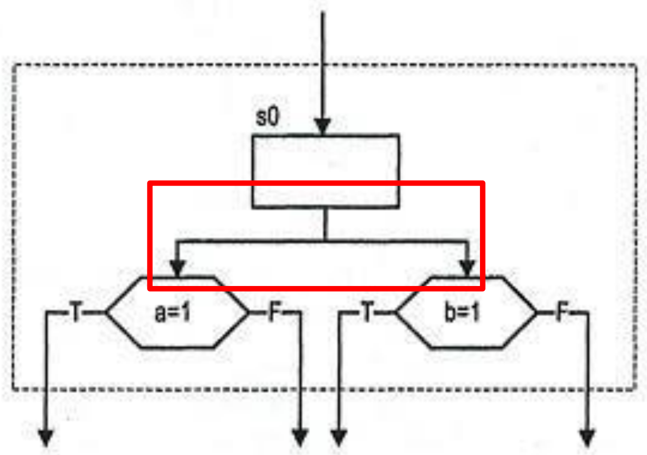
Two ASM FSM rules apply

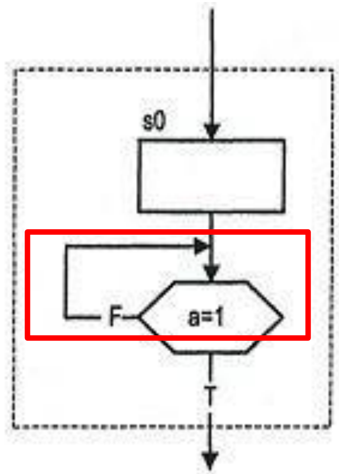
1. For any given input combination, there is one unique exit path from the current ASM block.

2. The exit path of an ASM block must always lead to a state box.
   - Can be the state box of the current or any other ASM block.



Common Errors in ASM Charts





This case violates rule one since it is two exit paths that are not governed by an input

You cannot enter two states at the same time in one state machine…

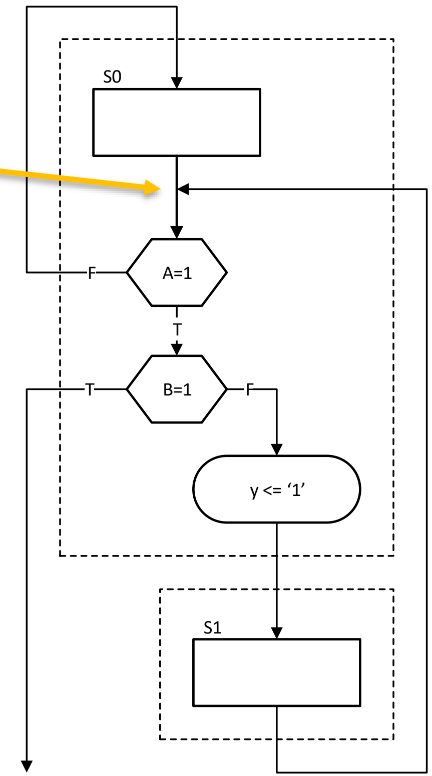
The case above violates the first rule since there is no exit path when the condition in the decision box is false.

A state shall be entered each clock cycle...

Common errors in ASM charts (2/2):

- exit path of the S1 block does not go into a state box

- If we need the same output logic, it must be copied for S1.
  - *(unless S1 is redundant and can be removed entirely)*



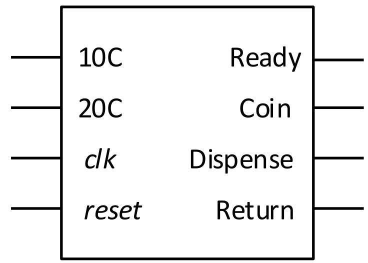
Example State Machine: Vending Machine

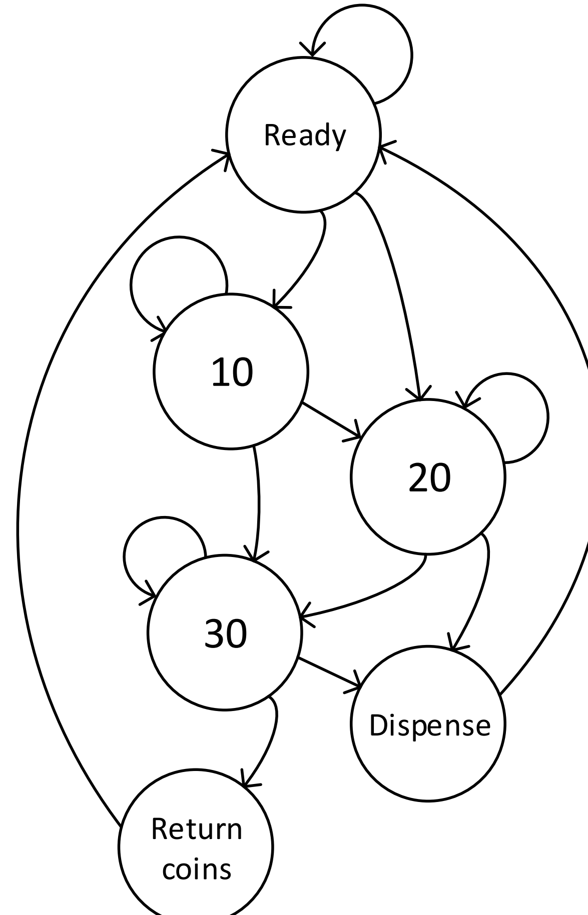
- Specification:
  - We want to design a vending machine that sells drinks for 40c.
  - The machine accepts 20c and 10c coins (all others will be rejected mechanically).
  - If 40c are inserted a drink shall be dispensed
  - If more than 40c is inserted all coins are returned
  - The machine has two lights
    - One to show that it is ready
    - One to show that further coins are needed

- Work order:
  - Define the entity
  - Find/Define the states
    - State diagram, ASM chart or both?
      - How to find redundant states?
    - Create an ASM chart
      - Be aware of Moore and mealy output
  - Once you have the ASM chart, with as few possible states: start coding
  - Decide before synthesizing:
    - One hot?
      - (FF's are cheap in an FPGA)
    - Binary counter?
    - Gray code?
      - (minimum noise / switching current)
    - *can the synthesizer decide for me?*

Example: Vending machine

- Sketch state diagram and entity
- May give you enough overview that you can simplify

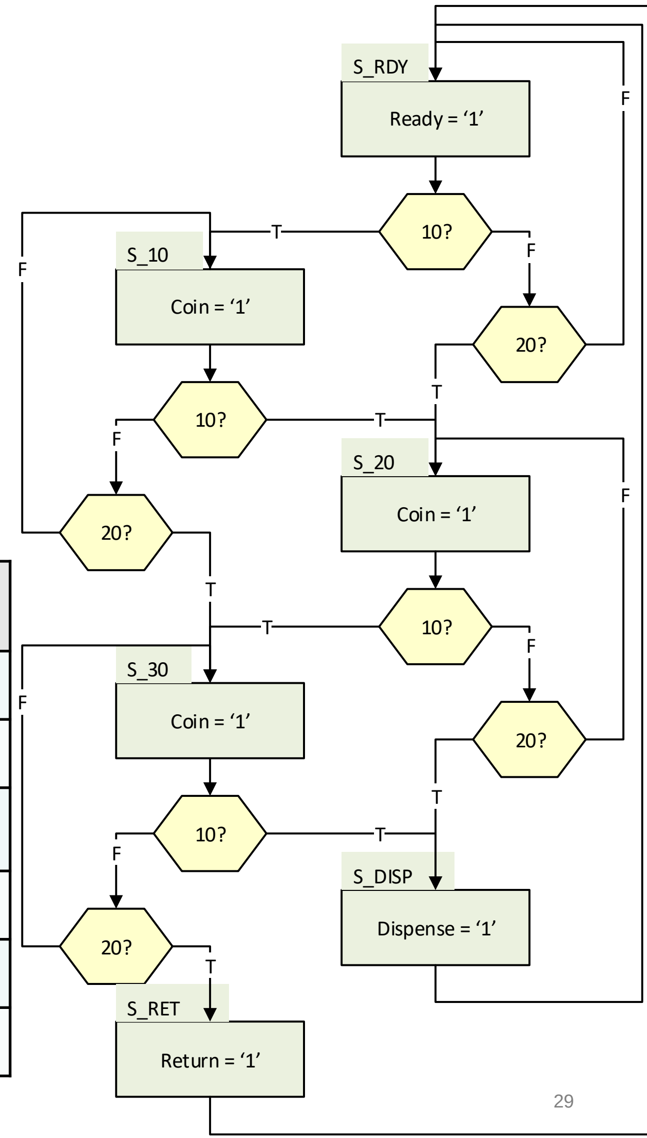




ASM diagram & State and ouput table

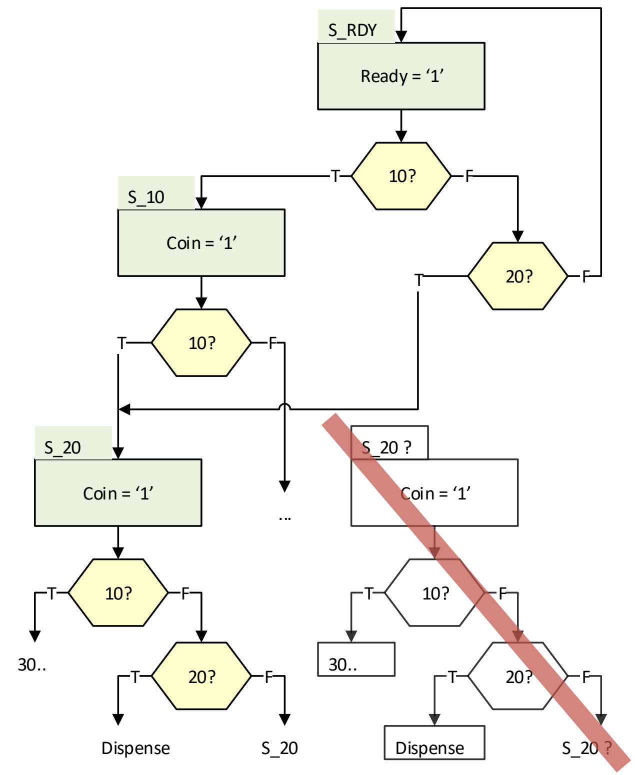
- If possible- simplify early.
  - Both state and output tables and ASM charts can be used to find redundancy

State	10c	20c	No coin	Ready	Coin	Dispense	Return
S_RDY	S_10	S_20	Self	1	0	0	0
S_10	S_20	S_30	Self	0	1	0	0
S_20	S_30	S_DISP	Self	0	1	0	0
S_30	S_DISP	S_RET	Self	0	1	0	0
S_DISP	S_RDY	S_RDY	S_RDY	0	0	1	0
S_RET	S_RDY	S_RDY	S_RDY	0	0	0	1



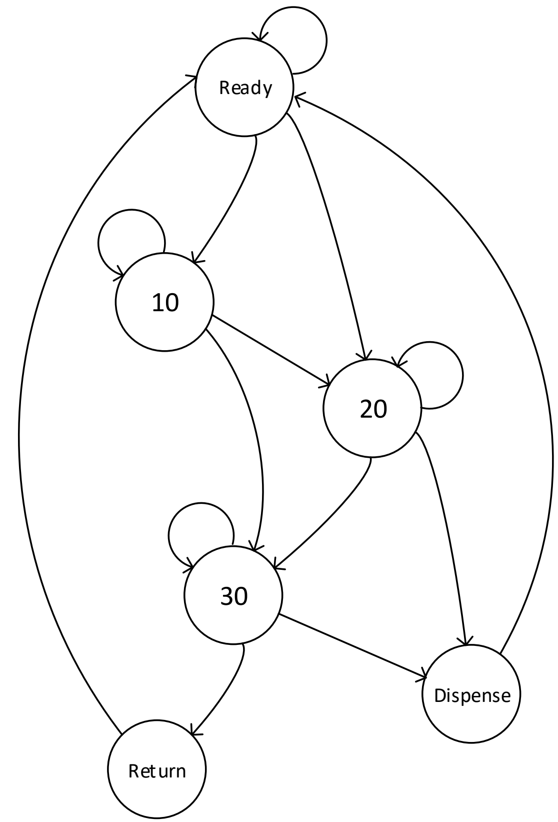
Redundant states in ASM

- If we start out as a descision tree *-always branching to new states-* we will get redundant states.
  - When both next state and output is equal to another state, the state can be removed if we move the inputs to the other state



Redundant states in state diagram

- *Can* be easier to spot
  - But we need to know all output based on state to be sure



Redundant states in output table

- Can be difficult to spot…
  - Names may confound
  - Both state and output must be checked
    - Here: otherwise DISP = RET ..?
  - It may be useful to use «self» rather than state name when going to the same state.

State	10c	20c	No coin	Transition to self	Ready	Coin	Dispense	Return
S_RDY	S_10	S_20	S_RDY	Yes	1	0	0	0
S_10	S_20_2	S_30	S_10	Yes	0	1	0	0
S_20	S_30	S_DISP	S_20	Yes	0	1	0	0
S_20_2	S_30_2	S_DISP	S_20_2	Yes	0	1	0	0
S_30	S_DISP	S_RET	S_30	Yes	0	1	0	0
S_30_2	*S_DISP*	*S_RET*	*S_30*	*Yes*	*0*	*1*	*0*	*0*
S_DISP	-	-	S_RDY	No	0	0	1	0
S_RET	-	-	S_RDY	No	0	0	0	1

- States that we only sweep through are candidates for creating mealy outputs…

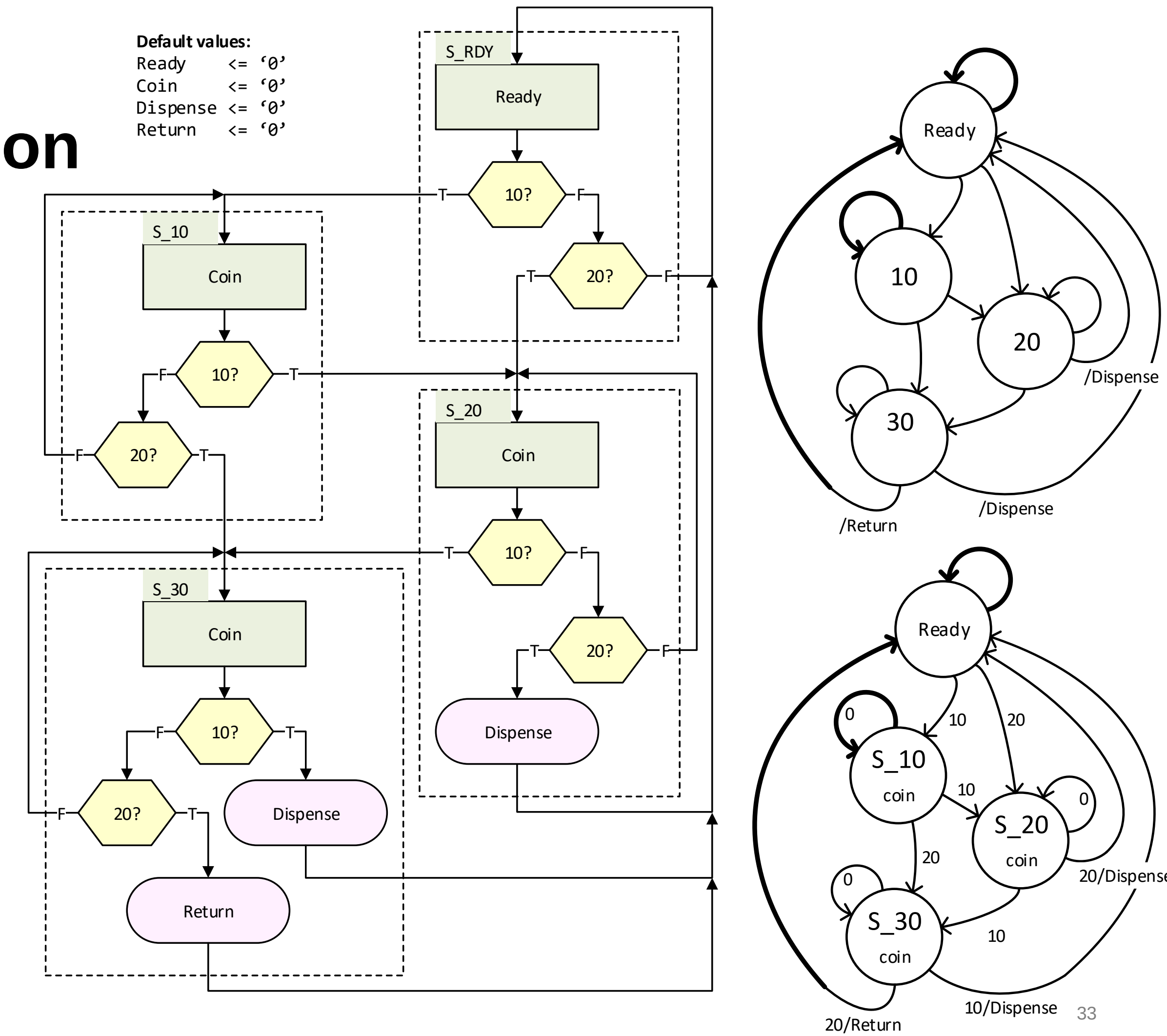
Mealy optimization

Identify states that are run through in one clock cycle without descision boxes

1. Is the output depending on being decoded in a different state than the previous?

2. Does timing requirements that dictates a separate state?

- If no on both:
  create a Mealy-ouput box, in place of the old state



Coding state machine using VHDL

- Make your own states as «enumerated» type.
  – This simplifies reading a lot (example next slide)

- Use three processes / statements
  1. One for assigning the state
     - based clock (and reset when asynchronous reset)
  2. One for deciding the next state (next_state CL).
     - based on previous state and inputs
  3. One for setting outputs (ouput CL)
     - based present state (and inputs if Mealy type)

  – Sometimes 2. and 3. can be combined
     - In simple cases where the output has very little decoding

FSM in VHDL 1/2

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

entity vending is
  port(
    clk, reset, twenty, ten : in std_logic;
    ready, coin, dispense, ret : out std_logic);
end entity vending;

architecture asm of vending is
  type state_type is (S_RDY, S_10, S_20, S_30);
  signal present_state, next_state : state_type;
begin
```

- *Continues next slide*

```
-- 1: sequential state assignment:
present_state <=
  S_RDY when reset else
  next_state when rising_edge(clk);

-- 2: combinatorial next_state logic
next_state_CL: process(twenty, ten, present_state) is
begin
  case present_state is
    when S_RDY =>
      next_state <=
        S_10 when ten else
        S_20 when twenty else
        S_RDY;
    when S_10 =>
      next_state <=
        S_20 when ten else
        S_30 when twenty else
        S_10;
    when S_20 =>
      next_state <=
        S_30 when ten else
        S_RDY when twenty else
        S_20;
    when S_30 =>
      next_state <= S_30 when not(ten or twenty) else S_RDY;
  end case;
end process next_state_CL;
```

FSM in VHDL 2/2

```
  -- 3: combinatorial output logic
  output_CL: process(all) is
  begin
    --default output values
    ready    <= '0';
    dispense <= '0';
    ret      <= '0';
    coin     <= '0';
    -- state based assignment
    case present_state is
      when S_RDY =>
        ready    <= '1';
      when S_10 =>
        coin     <= '1';
      when S_20 =>
        coin     <= '1';
        dispense <= '1' when twenty;
      when S_30 =>
        coin     <= '1';
        dispense <= '1' when ten;
        ret      <= '1' when twenty;
    end case;
  end process output_CL;
end architecture asm;
```

```
-- ALTERNATIVE ouput_CL:
 ready <= '1' when present_state = S_RDY else '0';
 coin  <= not ready;
 dispense <= '1' when
   (present_state = S_20 and twenty = '1') or
   (present_state = S_30 and    ten = '1') else '0';
 ret <= '1' when (present_state = S_30 and twenty = '1') else '0';
```

- *Optional alternative replaces 3*
  - *Consider how compactness affects readability*

Test bench for FSM

- Uses file I/O template from previous lecture-
- Input procedural
- Output in a separate process

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

entity tb_vending is
end entity;

architecture behavioral of tb_vending is
  component vending is
    port(
      clk, reset, twenty, ten : in std_logic;
      ready, coin, dispense, ret : out std_logic);
  end component;
  signal clk, reset, twenty, ten: std_logic := '0';
  signal ready, coin, dispense, ret: std_logic;
  constant CLK_PERIOD : time := 10 ns;

begin
  DUT: vending
  port map(
    clk => clk,
    reset => reset,
    twenty => twenty,
    ten => ten,
    ready => ready,
    coin => coin,
    dispense => dispense,
    ret => ret);

  clk <= not clk after CLK_PERIOD/2;
```

```vhdl
check_output: process(clk) is
  variable in_machine: integer := 0;
  constant COIN_DIGITS : integer := 3;
  constant SPACER : integer := 1;
  -- log output file
  file log_file: text open write_mode is "vending_log.txt";
  variable log_line: line;

begin
  if rising_edge(clk) then
    --keep track of coins
    if ret = '1' or dispense = '1' then
      in_machine := 0;
    elsif ten then
      in_machine := in_machine + 10;
    elsif twenty then
      in_machine := in_machine + 10;
    end if;
    --report errors to console
    assert (in_machine < 40)
      report ("coin overflow: ", integer'image(in_machine))
      severity error;
    -- report to file
    write(log_line, in_machine, field => COIN_DIGITS);
    write(log_line, ready, field => + 2*SPACER);
    write(log_line, coin, field => + 2*SPACER);
    write(log_line, dispense, field => + 2*SPACER);
    write(log_line, ret, field => + 2*SPACER);
    writeline(log_file, log_line);
  end if;
end process;
```

Stimuli next slide

TB stimuli:

- Usually one main process for stimuli
  - Except for clock generation
- Use procedures for file IO
- Testing can be done for each new input data or in a separate process...

```
process is
  type t_coin is (te, tw); -- ten, twenty abbreviated
  file stimuli_file: text open read_mode is "vending_stimuli.txt";
  variable stimuli_line: line;
  variable stimuli_coin: t_coin;
  variable stimuli_periods: integer := 0;
  variable str : string(2 downto 1);

  procedure set_stimuli is
  begin
    readline(stimuli_file, stimuli_line);
    read(stimuli_line, str);
    stimuli_coin := t_coin'value(str);
    read(stimuli_line, stimuli_periods);
    ten <= '1' when stimuli_coin = te else '0';
    twenty <= '1' when stimuli_coin = tw else '0';
  end procedure;

begin
  -- initial reset:
  wait for CLK_PERIOD/2;
  reset <= '1';
  wait for CLK_PERIOD;
  reset <= '0';
  wait for CLK_PERIOD;

  while not endfile(stimuli_file) loop
    set_stimuli;
    wait for CLK_PERIOD;
    ten <= '0';
    twenty <= '0';
    wait for CLK_PERIOD*stimuli_periods;
  end loop;
  file_close(stimuli_file);
  -- file_close(log_file);
  report ("Testing finished!");
  std.env.stop;
end process;

end architecture;
```

Additional video resources
(IN3160/V21/forelesningsvideoer/Vigander-17)

- FSM intro (37s)
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/0-fsm-intro.mp4
- FSM Basics (2:05)
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/1-fsm.mp4
- ASM State Diagrams (7:44)
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/2-asm.mp4
  - *Note: 3:50 different interpretation of '←'*
- ASM Examples (5:30)
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/2b-asm-examples.mp4
- **FSM Synthesis to VHDL (4:43)**
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/3-fsm-synthesis.mp4
- FSM Example (6:56)
  - https://www.uio.no/studier/emner/matnat/ifi/INF3430/h17/jn/videos/4-fsm-example.mp4
    - Note: optimizations are possible

Suggested reading

- D&H:
  - 14 p305-324
  - 16 p344-372

Some (free) tools for making charts

- https://app.diagrams.net/ (browser based)
- www.lucidchart.com (browser based, signup)
- Dia (Small, requires installation, all platform GNU)
- LibreOffice (large, GNU)
- http://diagramo.com/ (browser based, signup)