Hardware Modeling [VU] (191.011)
– WS24 –
Finite-State Machine Modeling

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

This lecture shows how finite-state machines can be modelled in general and introduces the model we use in our courses. Furthermore, we will look into two concrete examples showing how FSMs can be modelled given a description of their behavior.

Hardware Modeling [VU] (191.011)
– WS24 –
Finite-State Machine Modeling

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-12, 16:31 (21636bb)

Why not simply draw circuits?

We will start this lecture with discussing how we can model FSMs. But why is this even necessary? After all, since we end up implementing them in circuits, can't we simply draw circuits for modelling them?

Motivation

- Why not simply draw circuits?

Why not simply draw circuits?
Error-prone and scaling poorly for complex FSMs

First, recall the circuits we already saw during this course, especially in the exercise part. While there are structures within them that we can observe, understand, and map to code, circuits quickly become quite big for even simple designs. Since we want to use state machines as a more abstract design method for modelling complex circuits, this is obviously of no good. Furthermore, modelling an FSM by drawing a respective circuit is an error-prone task for humans.

Motivation

- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs

Why not simply draw circuits?
Error-prone and scaling poorly for complex FSMs
Hard to understand and maintain

Another issue with specifying FSMs as circuits is sharing designs and working on them in a team. Spotting all the state register, output logic and state transition function in an FSM circuit you did not design yourself takes quite some time and effort, as does modifying such a circuit.

Motivation

- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs
  - Hard to understand and maintain

Why not simply draw circuits?
Error-prone and scaling poorly for complex FSMs
Hard to understand and maintain
⇒ Increase level of abstraction of description

In conclusion, to really harness the power of FSMs as a more abstract circuit modelling tool, we require a description method for FSMs that operates on a higher level of abstraction as well.

Motivation

- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs
  - Hard to understand and maintain

⇒ Increase level of abstraction of description

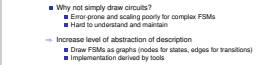
- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs
  - Hard to understand and maintain
- ⇒ Increase level of abstraction of description
  - Draw FSMs as graphs (nodes for states, edges for transitions)

Usually this is done by using graphs to specify FSM behavior. Each state of the FSM corresponds to a dedicated graph node and edges between nodes model state transitions. How outputs are handled will be shown on an upcoming slide. For now, we just want to note that we will also make use of this abstraction during this course. The specific notation we use during this course is also introduced in this lecture.

Motivation

- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs
  - Hard to understand and maintain

⇒ Increase level of abstraction of description
  - Draw FSMs as graphs (nodes for states, edges for transitions)



As we will also see later, such FSM graphs can be converted to VHDL code in an almost automatic manner, leading to common and simple-to-understand code structures. The design tools recognize and understand these structures, which allows them to efficiently implement the described FSMs as circuits.

Motivation

- Why not simply draw circuits?
  - Error-prone and scaling poorly for complex FSMs
  - Hard to understand and maintain

⇒ Increase level of abstraction of description
  - Draw FSMs as graphs (nodes for states, edges for transitions)
  - Implementation derived by tools

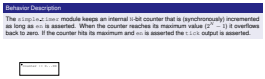
Behavior Description
The simple_timer module keeps an internal N-bit counter that is (synchronously) incremented as long as en is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and en is asserted the tick output is asserted.

Before we show you how we model FSMs, let us together recall the typical model you already know from past courses on theoretical computer science and formal description methods. As an example we use the `simple_timer` module from the lecture about FSM basics. As specified by the description of its behavior, the module shall contain an `N`-bit wide counter register that is incremented when the `en` signal is asserted. Upon an overflow it wraps around to zero and sets its `tick` output when `en` is high. Let us now model this FSM as a graph. With the internal counter being the only register specified by the description, this will be the state register. Thus, for each possible value of this register our graph will contain a node.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.

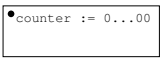


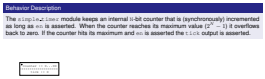
The slide shows a node for the initial state of the FSM, which we mark via a dot in the top-left corner. Inside the node the value of the counter for this state is shown. For the shown state, all its bits are zero. However, as we heard in the lecture about FSM basics, the output function of an FSM is a combinational function. Therefore, in addition to the current state register value, each node must also specify values for the FSM outputs.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



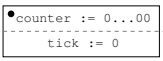


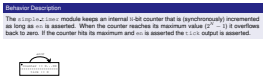
In this case, the FSM only has the single output `tick` which is low in this state of the FSM. We draw this as a separate section in the node, separated by a dashed line. Now that we got the content of our node complete, let us consider the transitions to other states, which we draw as directed edges between state nodes. In this example there are two cases between which we must distinguish.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value $(2^N - 1)$ it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



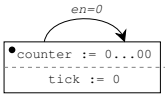


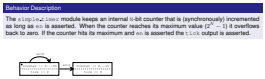
If the `en` signal is low, the counter value is not incremented and the FSM thus stays in its current state. We can draw this via an edge from the node to itself, labelled by the condition under which the respective transition is taken. The second type of transition happens when `en` is high. In this case the counter is incremented, hence leading to a node that represents this new counter value.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



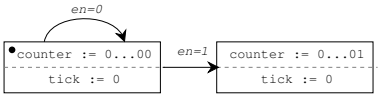


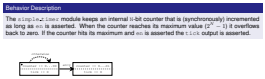
The slide shows this succeeding node and how the previous node is connected to it. At this point we want to point out something paramount: The set edges going out of a state node must cover all possible conditions, meaning all possible combinations of input values. In a majority of cases this results in a self-loop edge that is conditioned on the conjunction of the negation of all other outgoing edges.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



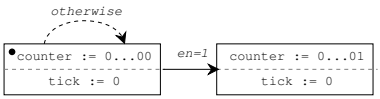


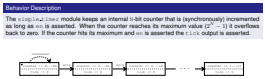
Since writing down this negation can be quite verbose and is also not really necessary, we use the keyword `otherwise` in such cases, as shown on the slide. In the very common case of a self-loop labelled with this particular condition, we omit drawing the edge at all and thus implicitly assume that an FSM remains in its current state if no condition of a transition to a different node is satisfied. From now on, we will no longer draw such `otherwise` self-loops.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



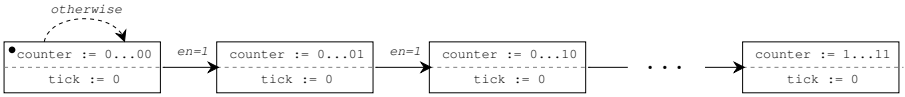


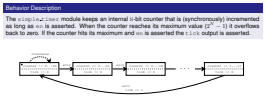
We continue adding such state nodes and transition edges until we enumerated all 2^N possible states. Our FSM specification is now almost complete, barring for the final outgoing edge of the last state node, which is slightly different from the other edges. The specification states that when the counter hits its maximum value that `tick` is only set if `en` is high.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



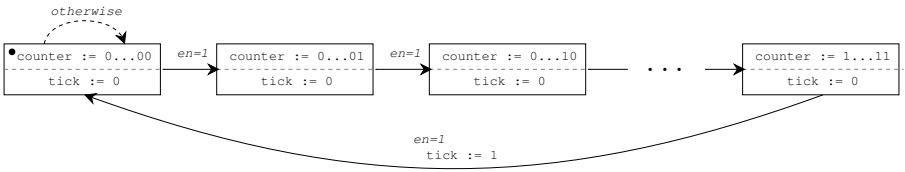


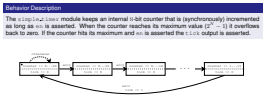
This is modelled as follows: In addition to labelling edges with the respective condition, we can also label them with assignments to FSM outputs. Such edge assignments always take precedence over the assignments contained in the state node, allowing nodes to express default behavior and edges more specialized one.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



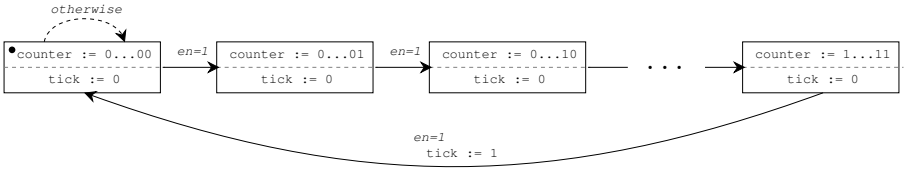


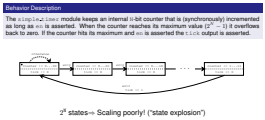
Finally, with our graph for this simple FSM being complete we want to remark that this description method is indeed easier to use and comprehend.

Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



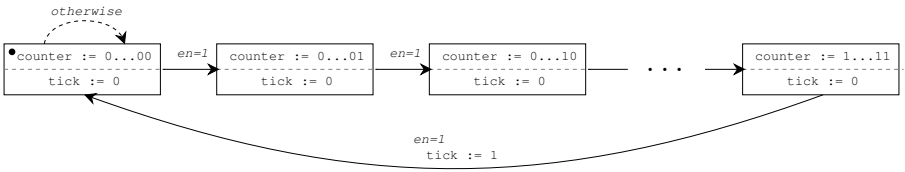


However, by enumerating all possible state values the size of the graph has a size exponential in the with of the state register, which makes it infeasible to draw such graphs for non-toy FSMs. We will now present our FSM model used in this course, which scales significantly better in a majority of cases.

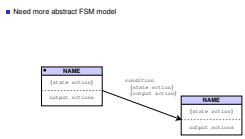
Common FSM Model

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.



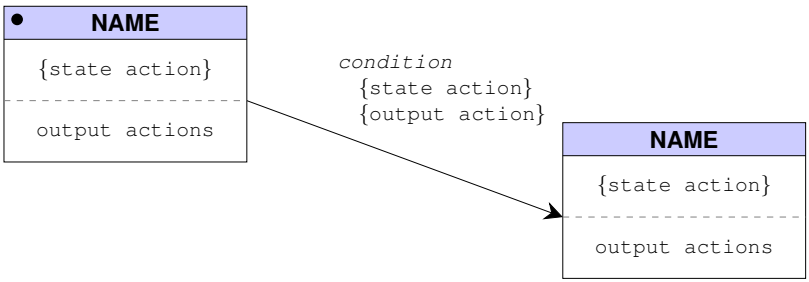
2^N states⇒ Scaling poorly! ("state explosion")

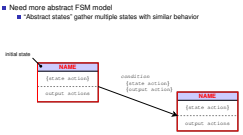


On the previous slide we could observe that the often-encountered way to model FSMs via graphs is infeasible for our purpose. However, we could also observe that, although the values of the state register differed, all states, except for the final one, where on an abstract level identical as they all just set the output to low and, when enabled, incremented the counter by one. We will now present the FSM model used in this course, which harnesses this key observation that multiple states might behave the same on an abstract level.

HWMod FSM Model

- Need more abstract FSM model

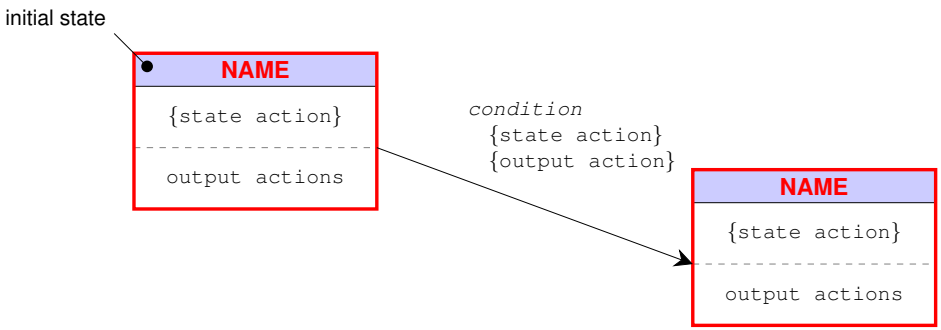


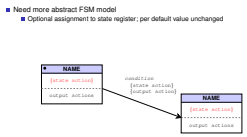


Let us start with the key difference to the model from the previous slide, the abstract states. Instead of each node corresponding to exactly one state register value, in our model each node corresponds to a set of state values. These sets are not chosen arbitrary, but rather such that the FSM next-state and output logic are similar for states that are grouped together. To highlight this conceptual similarity between the FSM states grouped in an abstract state, we give these states descriptive names. Note that an FSM must have one initial state. As before, we mark this state via a dot in the top-left corner in the respective node.

HWMod FSM Model

- Need more abstract FSM model
  - “Abstract states” gather multiple states with similar behavior

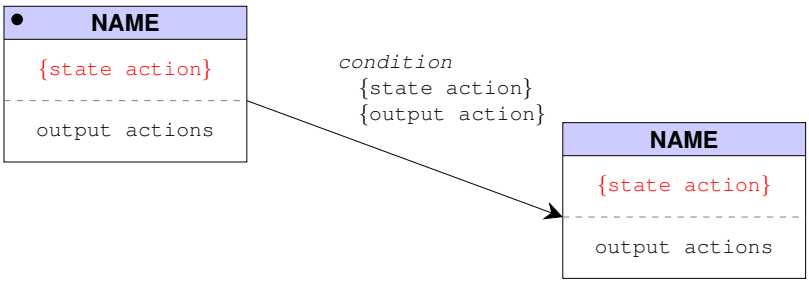


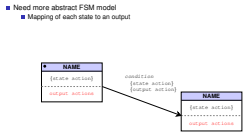


Next, similar to before, each node may contain assignments to the state register. We refer to this as state actions, where a single state action is an assignment to one part of the state register. This will become clear at later examples. In case a node does not contain a state action, the state register is simply left unchanged. Note that this means that a state can contain arbitrary many state actions, including none at all. We emphasize this via the enclosing braces.

HWMod FSM Model

- Need more abstract FSM model
  - Optional assignment to state register; per default value unchanged

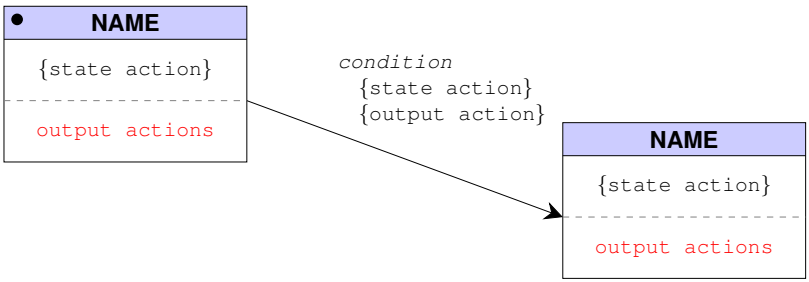


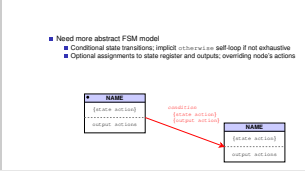


As before, each state must be mapped to values for the FSM outputs. This happens in the separated output actions part of a node. Again, each output action corresponds to a single assignment to an output. Note that the output actions of a node are mandatory and that it must contain an assignment for **each** FSM output in order to mitigate latches being modelled. A state node without output action is therefore not allowed, which is also the reason why the outputs actions are not enclosed via braces.

HWMod FSM Model

- Need more abstract FSM model
  - Mapping of each state to an output

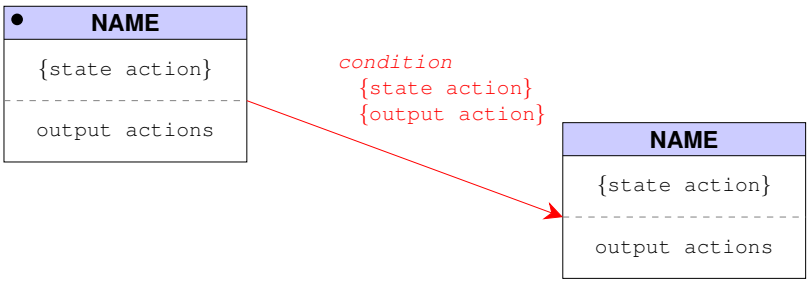


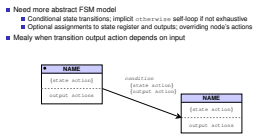


Finally, our model also contains edges for transitions between states. Again, these edges are labelled with a mandatory condition. However, since in our model the state nodes can correspond to multiple state register values, the conditions do not only depend on the inputs but also on the state register. Again, the conditions of all transition edges together must exhaustively cover all possibilities. As before, if this is not the case, we implicitly assume a self-loop on the current state labelled with `otherwise`. Furthermore, our model also allows transitions to optionally assign values to the state register or the outputs. Such assignments always override the ones contained in the node.

HWMod FSM Model

- Need more abstract FSM model
  - Conditional state transitions; implicit `otherwise` self-loop if not exhaustive
  - Optional assignments to state register and outputs; overriding node's actions

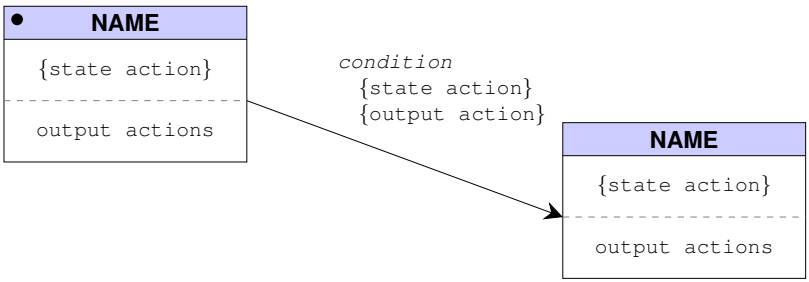




Finally, before we look at an example, we want to point out a minor drawback of our model. In particular, the information whether an FSM is of Moore or Mealy type is not immediately visible from looking at the graph. To determine this, one needs to determine if there exists a transition edge that contains an output action that either directly contains an input in its assignment, or depends on one via the condition. If that is the case, the FSM is of Mealy type.

HWMod FSM Model

- Need more abstract FSM model
  - Conditional state transitions; implicit `otherwise` self-loop if not exhaustive
  - Optional assignments to state register and outputs; overriding node's actions
- Mealy when transition output action depends on input

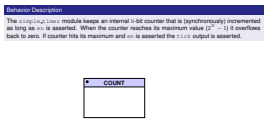


As an example, let us now create a description for the `simple_timer` module using our FSM model.

Example I: `simple_timer`

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value $(2^N - 1)$ it overflows back to zero. If counter hits its maximum and `en` is asserted the `tick` output is asserted.

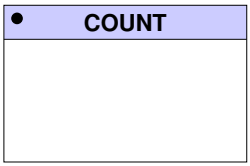


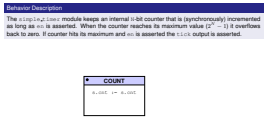
From our previous graph of this FSM we already know that the FSM essentially does the same state and output actions independent of the particular state register value. We capture this by creating a single abstract state node called COUNT.

Example I: simple_timer

Behavior Description

The simple_timer module keeps an internal N-bit counter that is (synchronously) incremented as long as en is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If counter hits its maximum and en is asserted the tick output is asserted.



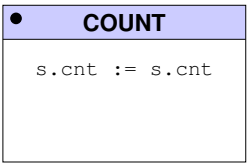


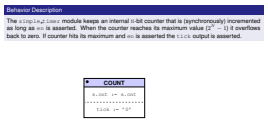
Next, let us specify the state actions. Since our FSM only contains a single state register, we only have a single assignment in here. As stated before, the state action inside a node should ideally correspond to a default assignment that can then be overridden by the state actions of conditioned transition edges. In this example, the count state register shall not change in the default case. At this point there are two remarks to be made: First, note how we refer to values of the state register via the prefix "`s.`". While this might appear strange right now, it will make sense later when we convert our FSM models to VHDL code. Therefore, we decided to introduce this notation early. Furthermore, note how this assignment is not strictly necessary in our model, as we assume the state register to hold its value per default. However, we wanted to make this explicit during the examples to keep them more comprehensible.

Example I: `simple_timer`

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If counter hits its maximum and `en` is asserted the `tick` output is asserted.



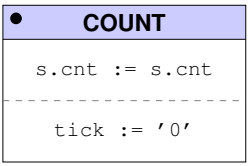


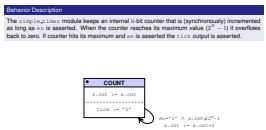
Next, we create a mapping for the state to the FSM output. In this case this mapping simply always assigns low to `tick`, letting the edges handle the special case where the output is asserted.

Example I: simple_timer

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If counter hits its maximum and `en` is asserted the `tick` output is asserted.



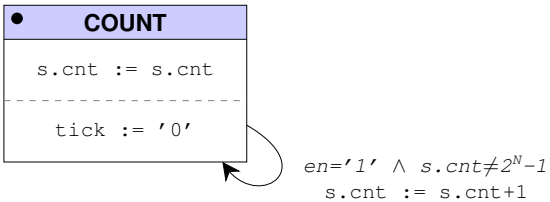


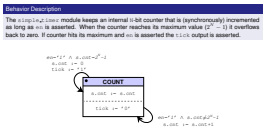
Speaking of which, there are two edges which we need to introduce in order to update the state register and set the output correctly. First, we have the case where the `en` signal is high, and the counter register has not yet reached its maximum value. In this case, the counter register is updated by one.

Example I: simple_timer

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If counter hits its maximum and `en` is asserted the `tick` output is asserted.



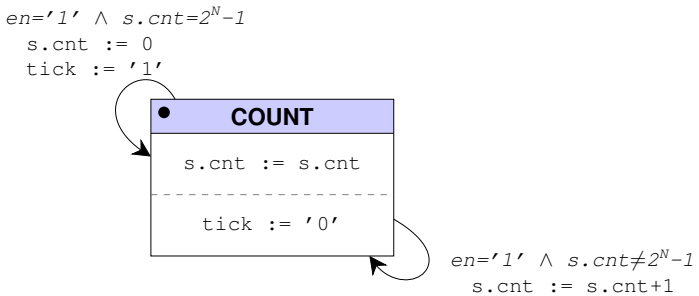


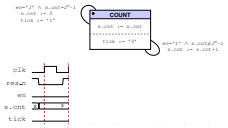
Second, there is the case where en is high and the counter has reached its maximum value. In this case, the counter is set to zero and the output to high. And with that, our model for the simple_timer FSM is complete. Note how we just require a single abstract state using our model, independent of the counter width. We will now continue by looking at how our model conceptually relates to synchronous circuits.

Example I: simple_timer

Behavior Description

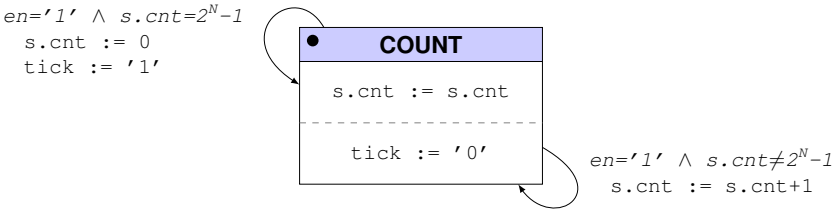
The simple_timer module keeps an internal N-bit counter that is (synchronously) incremented as long as en is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If counter hits its maximum and en is asserted the tick output is asserted.

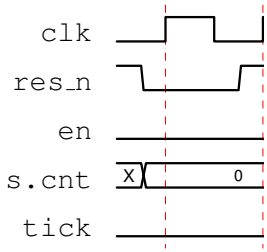


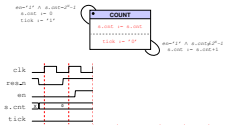


In general this relation is quite simple. During the reset of the synchronous circuit implementing an FSM the state register is set to an initial value. In our example this initial value is 0, as shown by the wave diagram on the slide. After the reset, the FSM operates synchronous to a clock, meaning that it only changes its state and output at active clock edges. We will now consider the first active clock edge.

Synchronous FSM Behavior

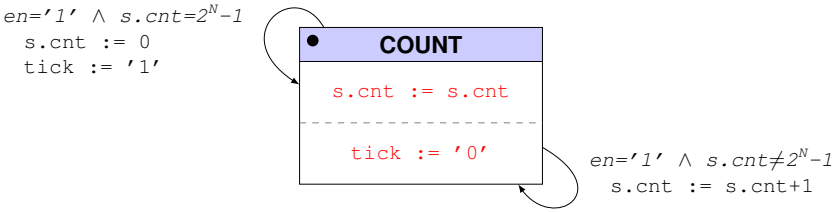


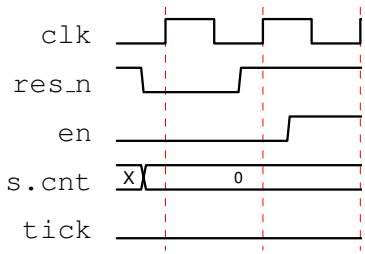


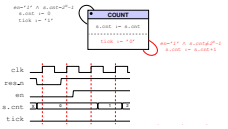


As `en` is low and the counter value 0, none of the edge conditions is true and the FSM will thus simply perform the default actions contained in the node, highlighted in red.

Synchronous FSM Behavior

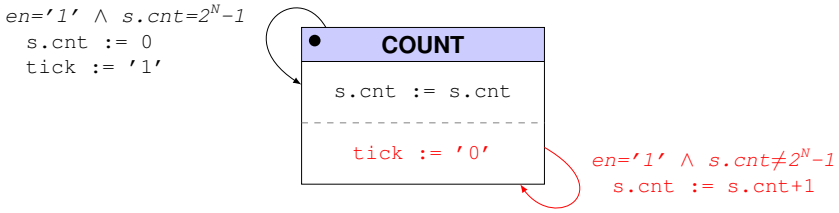


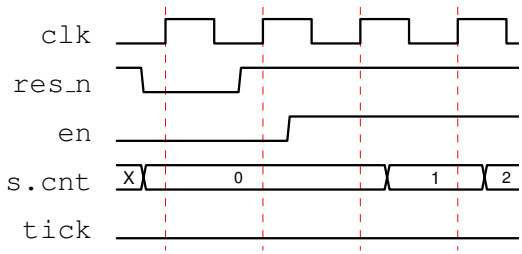


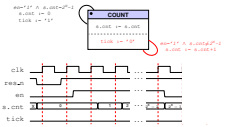


At the next active clock edge, the `en` signal is high. Since the counter is still zero, and thus has not yet reached its maximum value, the condition of the bottom right edge is satisfied. Therefore, the counter register will be incremented by one. Since this edge does not override the default action for the output, `tick` is set to zero again.

Synchronous FSM Behavior

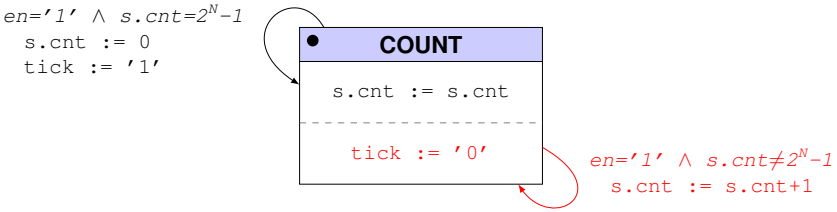


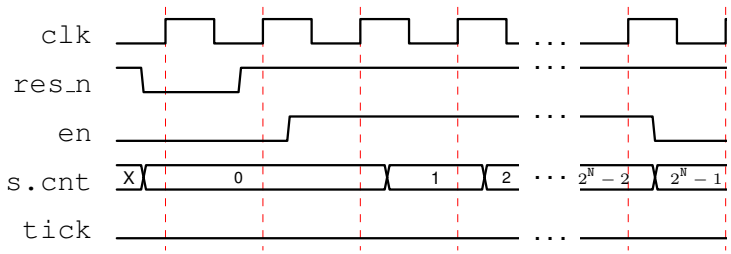


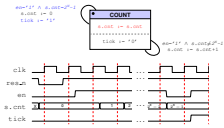


We continue in this manner until the counter register finally reaches its maximum value.

Synchronous FSM Behavior

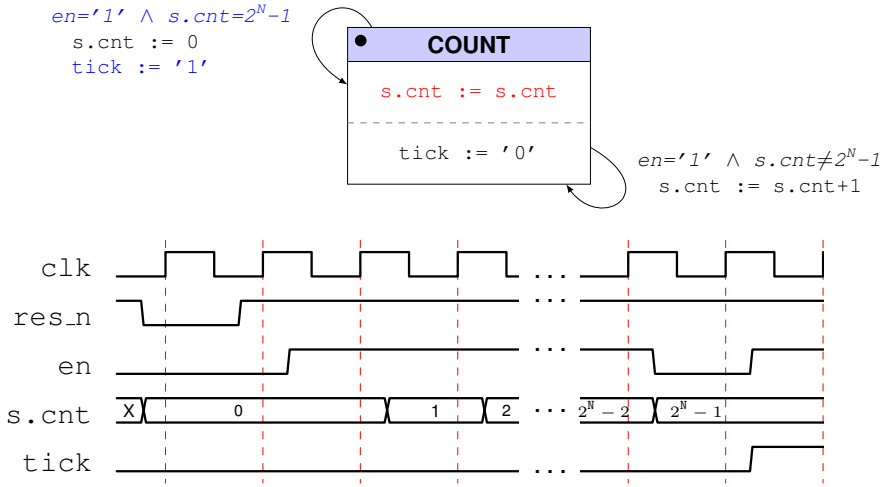


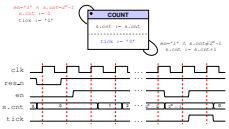




In the current clock cycle, an interesting thing happens: The `en` signal was low at the active clock edge, but becomes high later during the clock cycle. Since the state register is a sequential element, the state actions will always happen at the active clock edge. Hence, the node's default state action will be performed which leads to the counter keeping its value. However, recall that the specification states that the `tick` output is set whenever the counter holds its maximum value and `en` is high. As we saw in the lecture about FSM basics, the `simple_timer` is a Mealy type FSM and the output combinationally depends on the `en` signal. Therefore, as shown on the slide, with the input becoming high during this clock cycle, the output will be asserted. We highlighted this in blue in the model.

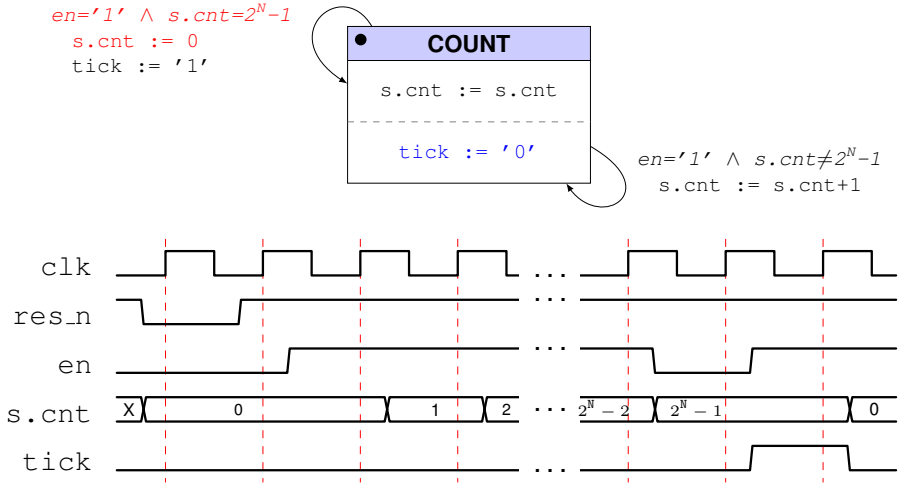
Synchronous FSM Behavior





Finally, the `en` input is high during an active clock cycle, which results in the condition of the top-left edge being satisfied. This leads to the internal counter being reset. However, with counter value being reset, we can observe `tick` transitioning to low.

Synchronous FSM Behavior



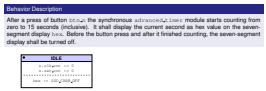
Behavior Description
After a press of button btn_n the synchronous advanced_timer module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display hex. Before the button press and after it finished counting, the seven-segment display shall be turned off.

With our FSM model being introduced, and a simple example being shown to demonstrate its advantages and use, we will now look at a more elaborate example, the `advanced_timer` module. You can find the description of the module's behavior on the slide. In essence, it shall count from zero to fifteen seconds after a button press and display the current second on a seven-segment display. We will now create a model for an FSM implementing this description. Unlike for the `simple_timer` we know nothing about the internals of the desired FSM. Hence, we also start without a clue about its state register and possible values. Let us therefore create the FSM step-by-step by introducing new states for all parts of the desired behavior.

Example II: `advanced_timer`

Behavior Description

After a press of button `btn_n` the synchronous `advanced_timer` module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display `hex`. Before the button press and after it finished counting, the seven-segment display shall be turned off.

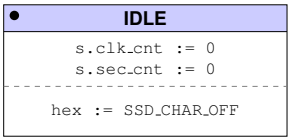


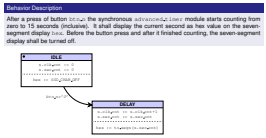
First, we obviously need a state for the FSM to wait for a button press. We call this state IDLE. Furthermore, as the clock period is most likely significantly smaller than one second, we introduce a clock count register which we can use to determine the passing of a second. In addition to that, we introduce a register for counting the seconds themselves. We name these two parts of the state register `clk_cnt` and `sec_cnt`. In the IDLE state they are both set to zero, while the FSM's output is set to the `SSD_CHAR_OFF` constant. Next, the description tells us that a button press leads to a state where seconds are counted. Therefore, we introduce a respective transition to a state for counting seconds.

Example II: advanced_timer

Behavior Description

After a press of button `btn_n` the synchronous `advanced_timer` module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display `hex`. Before the button press and after it finished counting, the seven-segment display shall be turned off.



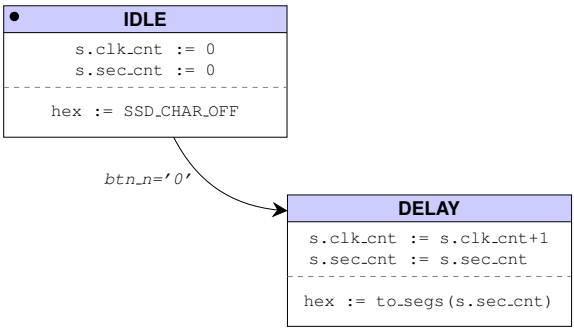


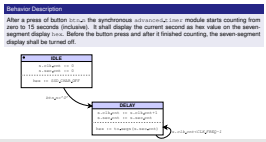
Because we need to wait in this state until a second has passed, we name this state DELAY. In here `clk_cnt` is incremented by one in each clock cycle, whereas `sec_cnt` retains its value, which is mapped to the output as hexadecimal digit via the `to_segs` function. How long do we remain in this state?

Example II: advanced_timer

Behavior Description

After a press of button `btn_n` the synchronous `advanced_timer` module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display `hex`. Before the button press and after it finished counting, the seven-segment display shall be turned off.



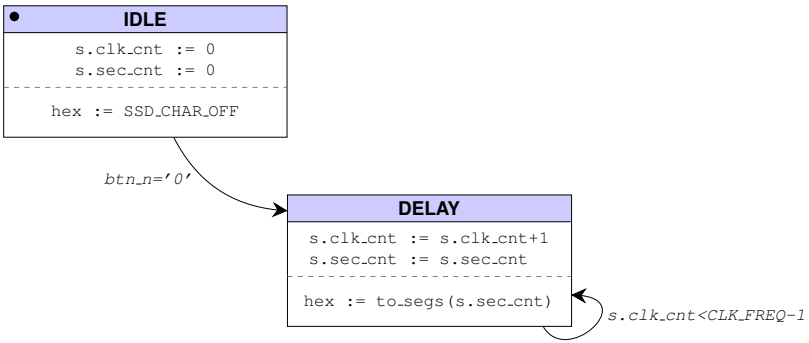


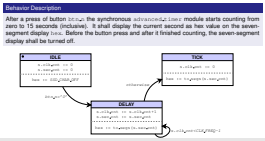
Well, as long as it takes until a second has passed. We model this via a respective self-loop, assuming that we know the constant clock frequency. The minus one has its origin in the fact that a state transition takes one clock cycle. Thus, if we want to reach the next state after exactly one second, we must account for this one cycle.

Example II: advanced_timer

Behavior Description

After a press of button btn_n the synchronous advanced_timer module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display hex. Before the button press and after it finished counting, the seven-segment display shall be turned off.



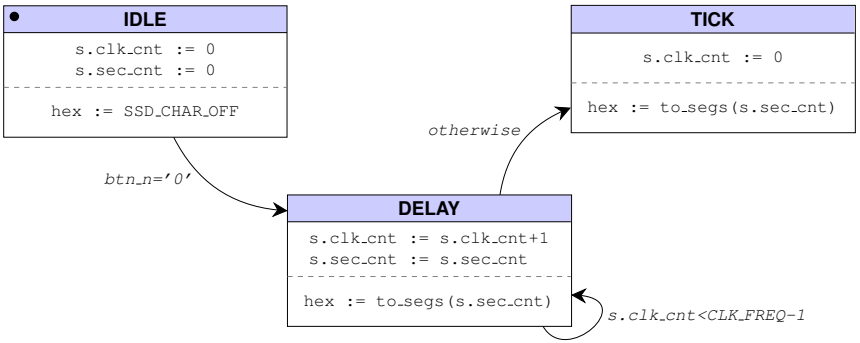


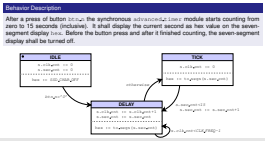
In case the second has passed, we transition to a state called TICK where the clk_cnt register is set to zero for counting the next second if necessary. The output logic is the same as in the previous state. We now have to determine the possible transitions from this to other states.

Example II: advanced_timer

Behavior Description

After a press of button btn_n the synchronous advanced_timer module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display hex. Before the button press and after it finished counting, the seven-segment display shall be turned off.



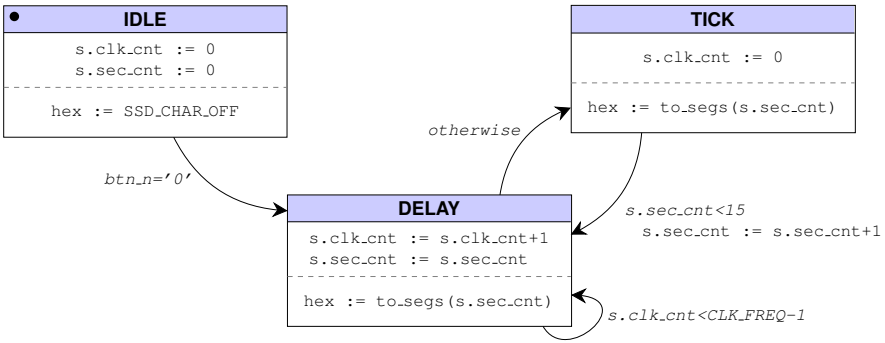


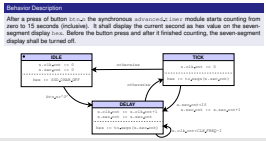
In case the FSM has not yet counted to fifteen, it transitions back to the DELAY state to wait for another second. Furthermore, it increments the value held in `sec_cnt`.

Example II: advanced_timer

Behavior Description

After a press of button `btn_n` the synchronous `advanced_timer` module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display `hex`. Before the button press and after it finished counting, the seven-segment display shall be turned off.



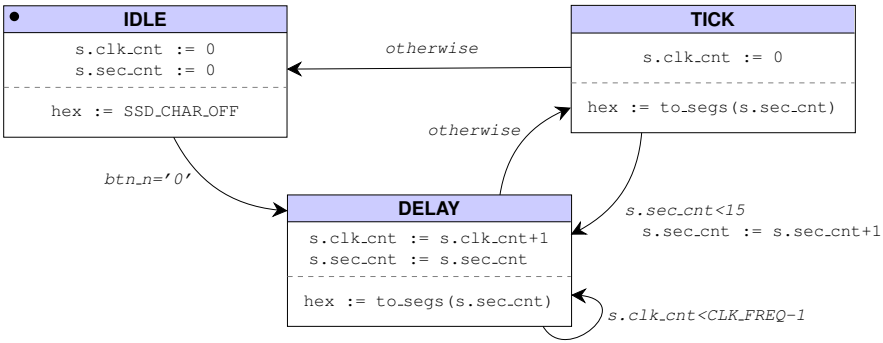


If this is not the case, that is, the FSM has already counted to fifteen, it returns to the IDLE state. Since this state sets the state register to a known value, no specialized state or output action is necessary. This last edge concludes this example. Make sure that you understand all the states, actions and transitions and that you also have an intuition on how the FSM was derived from the description of the behavior alone. Furthermore, try to determine if this is a Moore or a Mealy FSM? Since none of the transition edges features an output action, this models a Moore FSM. This concludes the lecture about modelling FSMs. In an upcoming video we will continue by discussing how such FSM models can be converted to VHDL code.

Example II: advanced_timer

Behavior Description

After a press of button btn_n the synchronous advanced_timer module starts counting from zero to 15 seconds (inclusive). It shall display the current second as hex value on the seven-segment display hex. Before the button press and after it finished counting, the seven-segment display shall be turned off.



Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!