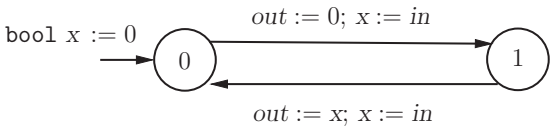
2 Synchronous Model

Solution 2.1: In every odd round, the output of the component `OddDelay` is 0. In every even round, the output equals the value of the input from the previous round. That is, for every $j \geq 1$, if j is an odd number, then the output o_j equals 0, else it equals the input i_{j-1}. Thus the component `OddDelay` alternates between producing the fixed output value 0 and behaving like the component `Delay`. For the given sequence of inputs for the first six rounds, the component has a unique execution shown below, where a state is specified by listing the value of x followed by the value of y:

$$00 \xrightarrow{0/0} 01 \xrightarrow{1/0} 10 \xrightarrow{1/0} 11 \xrightarrow{0/1} 00 \xrightarrow{1/0} 11 \xrightarrow{1/1} 10.$$

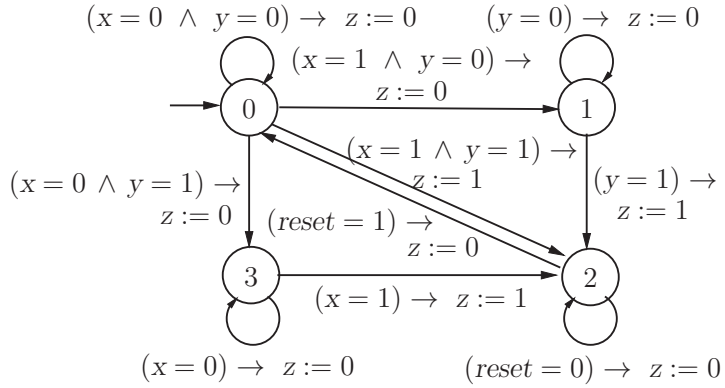
■

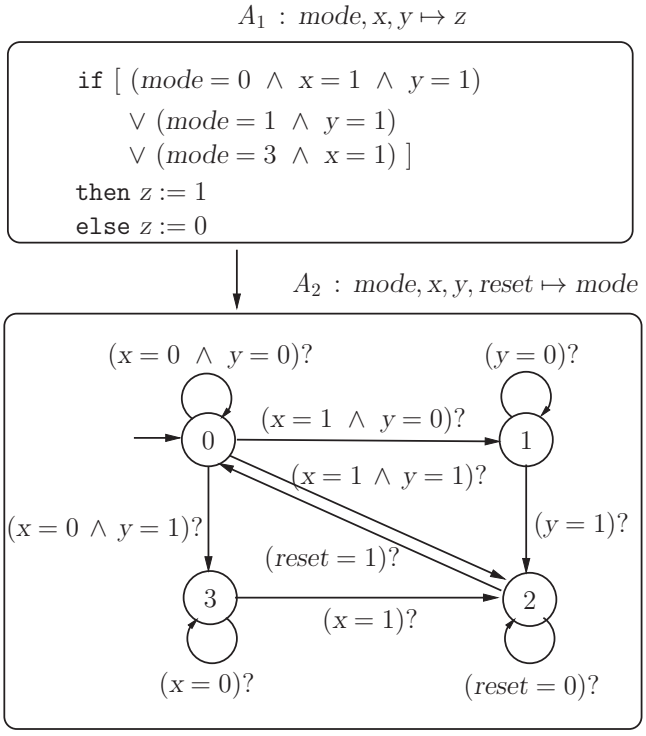
Solution 2.2: The extended-state-machine corresponding to the component `OddDelay` is shown below. The modes correspond to the values of the state variable y.



■

Solution 2.3: The extended-state-machine below implements the desired component. The initial mode is 0. When the input x is 1, the component switches to the mode 1, and subsequently when the input y is 1, it switches to the mode 2. Symmetrically, in the initial mode, when the input y is 1, the component switches to the mode 3, and subsequently when the input x is 1, it switches to the mode 2. Note that in the initial mode, if both input variables x and y equal 1, the component directly switches to the mode 2. The transitions to the mode 2 set the output z to 1, and all other transitions set the output to 0. In mode 2, when the condition $(reset = 1)$ holds, the component returns to the initial mode.





■

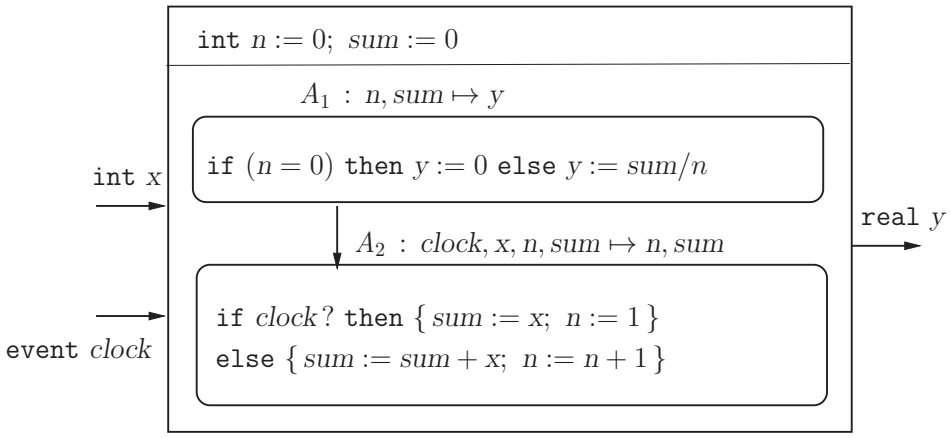
Solution 2.12: Since the task A_2 writes the output z, and z does not await the input x, we can conclude that the task A_2 does not read x and nor does a task that must precede A_2. Since the output y produced by the task A_1 awaits x, it must be the case that A_1 reads x. It follows that there cannot be a precedence edge from the task A_1 to A_2, that is, $A_1 \not\prec A_2$. This means that either there are no precedence constraints (that is, the relation $\prec$ is empty), or the task A_2 precedes A_1 (that is, $A_2 \prec A_1$). ■

Solution 2.13: The reactions of the component are listed below (the output lists the values of y and z in that order):

$$0 \xrightarrow{0/00} 0; \quad 0 \xrightarrow{1/00} 1; \quad 0 \xrightarrow{1/01} 1; \quad 1 \xrightarrow{0/10} 0; \quad 1 \xrightarrow{0/11} 0; \quad 1 \xrightarrow{1/11} 1.$$

The output y *does not* await the input x. The output z awaits the input x. ■

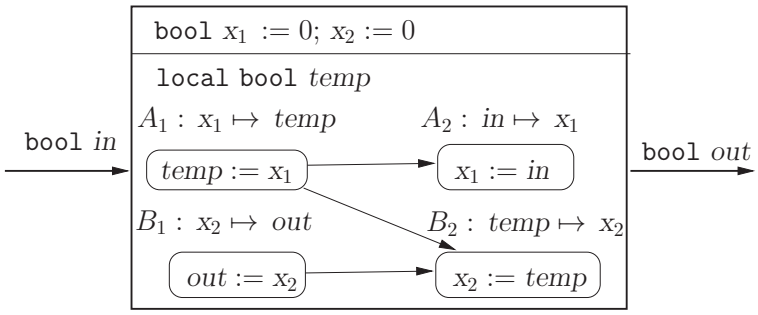
Solution 2.14: The component `ComputeAverage` is shown below. It maintains an integer state variable n that tracks the number of rounds elapsed since the presence of the input event *clock*, and an integer state variable *sum* that maintains the sum of the values of the input variable x since the presence of the input event *clock*. The task A_1 computes the value of the output y based on the current state, and the task A_2 then updates the state variables based on the inputs.



■

Solution 2.15: The component `ClockedDelayComparator` has input variables in_1 and in_2 of type `nat`, an input event variable *clock*, and an output variable y of type `event(bool)`. Suppose the input *clock* is present during rounds, say, $n_1 < n_2 < n_3 < \cdots$. Then, in round n_1, the output y is 0; and in round n_{j+1}, for each j, the output equals 1 if the value of the input variable in_1 in the round n_j is greater than or equal to the value of the input variable in_2 in the round n_j, and equals 0 otherwise; and in the remaining rounds (that is, rounds during which the input event *clock* is absent), output is absent. ■

Solution 2.16: The component `DoubleSplitDelay` has input variable *in*, output variable *out*, state variables x_1 and x_2, and local variable *temp*, all of type `bool`. Its reaction description consists of 4 tasks as shown below.



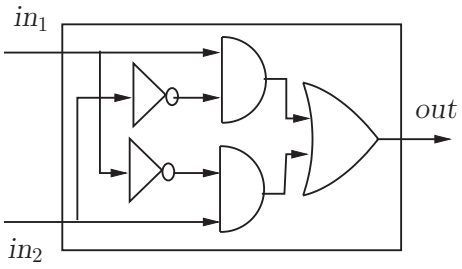
The output variable *out* does not await the input variable *in*. ■

Solution 2.17: The desired component `SecondToHour` is defined as

(`SecondToMinute` ‖ `SecondToMinute`[*minute* ↦ *hour*][*second* ↦ *minute*])*minute*.

■

Solution 2.18: For the component `SyncXor`, its output *out* should be 1 exactly when only one of the inputs in_1 and in_2 is 1. Thus, the output *out* corresponds to the Boolean expression $(in_1 \wedge \neg\, in_2) \vee (\neg\, in_1 \wedge in_2)$. The desired output is computed by the following combinational circuit that uses 2 instances of `SyncAnd`, 2 instances of `SyncNot`, and one instance of `SyncOr`.



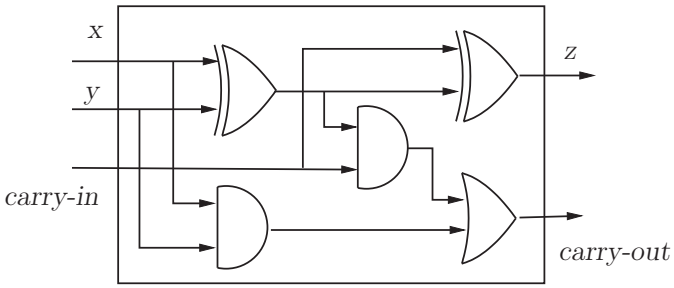
■

Solution 2.19: The parity circuit with n inputs is defined inductively. For $n = 2$, the desired functionality coincides with that of the XOR gate. Thus, the component $\texttt{Parity}_2$ is same as the component `SyncXor` from exercise 2.18. Having defined the circuit $\texttt{Parity}_{n-1}$ that computes the parity of $n-1$ input variables, now we wish to construct the circuit $\texttt{Parity}_n$ with input variables $in_1, \ldots in_n$ and output *out*. Observe that the output should be 1 exactly when either the input in_n is 1 and the parity of the first $n-1$ input variables is even, or the input in_n is 0 and the parity of the first $n-1$ input variables is odd. Thus, the desired circuit is defined as:

$$\texttt{Parity}_n \;=\; (\texttt{Parity}_{n-1}[\,out \mapsto temp\,] \parallel \texttt{SyncXor}[\,in_1 \mapsto temp\,][\,in_2 \mapsto in_n\,])\backslash temp.$$

Note that the circuit $\texttt{Parity}_n$ uses one more instance of `SyncXor` than the component $\texttt{Parity}_{n-1}$, and thus, $n-1$ total instances of `SyncXor`. ■

Solution 2.20: The combinational circuit `1BitAdder`, shown below, uses 2 instances of `SyncAnd`, one instance of `SyncOr`, and 2 instances of `SyncXor`. Verify that the output z is 1 when an odd number of the input variables equal 1, and the output *carry-out* is 1 when two or more of the input variables equal 1.



output value only when the variable *new* equals 1, and in each round, if the state variable *id* is modified, the variable *new* is set to 1. The initialization is given by

$$\texttt{nat } id := \texttt{myID};\ r := 1;\ \texttt{bool } new := 1$$

The update code the task A_1 is changed to:

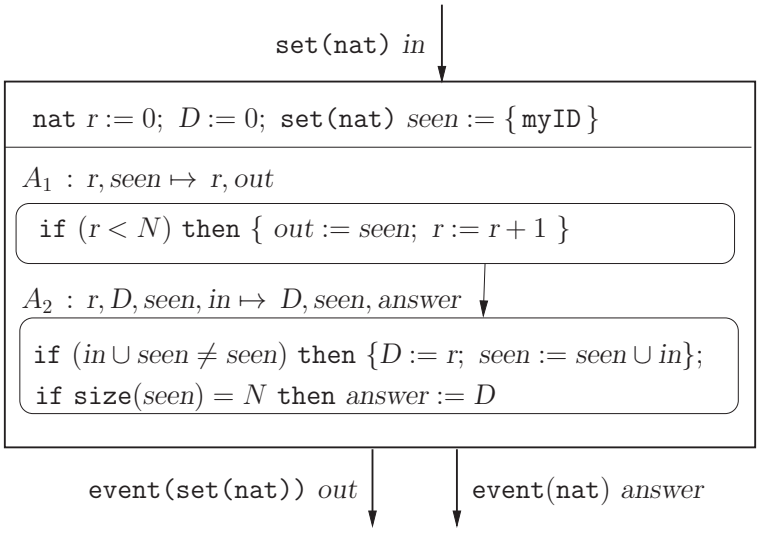
```
if (r < N) then {
  if (new = 1) then out := id;
  r := r + 1 }.
```

The first statement in the update code for the task A_2 is replaced by (the subsequent statement that assigns the value of the output variable *status* stays unchanged):

```
if (in ≠ ∅) then
  if (id < max in) then
    { id := max in; new := 1 }
  else new := 0.
```

■

Solution 2.23: Each node n maintains a state variable r that keeps track of the number of rounds: r is initialized to 0, and is incremented by 1 in each round. To compute the desired value D_n, the node n needs to know, for each node m, the length of the shortest path from the node m to node n. For this purpose, the node n maintains a state variable *seen* which contains the set of identifiers of all the nodes that the node n has encountered so far. Initially, the set *seen* contains just the node n itself. In each round, each node transmits this set to its neighbors, and updates the value of *seen* to include all the identifiers it receives from its neighbors. Thus, if the shortest path from the node m to node n is k, then, during the k-th round, the node n will receive m as one of the identifiers for the first time, and will add it to the set *seen*. To compute the desired value D_n, the node n maintains a state variable D: it is initialized to 0, and during the k-th round, if the node n receives an identifier that it has not seen before (that is, the input contains a value not in the current value of the variable *seen*), the variable D is updated to k. If the network is strongly connected, then within N rounds, the set *seen* will contain all the node identifiers: when the size of this set equals N, the value of the variable D equals the desired diameter D_n, and the node n can output this answer. The synchronous component that implements this protocol is shown below:


set(nat) in
nat r := 0; D := 0; set(nat) seen := { myID }
A1 : r, seen ↦ r, out
if (r < N) then { out := seen; r := r + 1 }
A2 : r, D, seen, in ↦ D, seen, answer
if (in ∪ seen ≠ seen) then {D := r; seen := seen ∪ in};
if size(seen) = N then answer := D
event(set(nat)) out
event(nat) answer

■