C. Clauß , U. Donath, A. Schneider, E. Weber
Fraunhofer Institute for Integrated Circuits, University of Applied Sciences Mittweida, Germany

Standard Package Modelica.Electrical. Digital

pp. 539-547

Standard Package Modelica.Electrical.Digital

Christoph Clauß[1], **Ulrich Donath**[1], **André Schneider**[1], **Enrico Weber**[2]
1) Fraunhofer Institute for Integrated Circuits, Branch Lab Design Automation
Zeunerstraße 38, D-01069 Dresden, Germany
2) University of Applied Sciences, Technikumplatz 17, D-09648 Mittweida
{clauss, donath, schneider}@eas.iis.fraunhofer.de

Abstract

According to the IEEE 1164 standard the Modelica.Electrical.Digital library was developed which uses nine-valued logical signals. The first stage of extension contains basic gate devices, sources, delay devices, and convertes. The main principles of implementation are demonstrated as well as some examples which show some possibilities of usage. Using converters, the electrical digital components are capable of interacting with the components of other Modelica libraries.

1 Introduction

The Modelica language [1], [2] already allows the formulation of logic behaviour using both the predefined Boolean variable type (true, false) and Boolean operators (or, and, not). For many applications these possibilities are sufficient. However, the description of complex digital electronic behaviour requires a very extension of the simple Boolean logic. The reason is that some of the properties of electronic circuits have to be transmitted to the logic approach, e.g. the existence of an *unknown* signal state, of different signal strengths etc..

Considering the VHDL language, the IEEE 1164 standard [3], [4], [5], [6], [7] is generally accepted and widely used for the description of digital electronic devices. It is based on nine-valued logical signals and defines the behaviour of simple and more general digital devices including time-dependencies. Due to the importance of this standard the digital electronic library should be developed in accordance with it.

In this paper an overview is given on the devices available. Details of the implementation are presented as well as some questions of the usage in combination with other libraries. Many examples give an impression of the actual state of the library.

2 Overview

The nine digital signal values are 'U' (uninitialized), 'X' (forcing unknown), '0' (forcing 0), '1' (forcing 1), 'Z' (high impedance), 'W' (weak unknown), 'L' (weak 0), 'H' (weak 1), '-' (don't care).

The library is devided into:
- delay models (transport, inertial, sensitive inertial)
- basic gates without delay (Not, And, Nand, Or, Nor, Xor, Xnor)
- basic gates including intertial delay (InvGate, AndGate, NandGate, OrGate, NorGate, XorGate, Xnorgate, BufGate)
- sources (Set, Step, Table, Pulse, Clock)
- converters (for connections with Boolean, and with Real, and for the restriction of the digital logic values to 'X01' or to 'X01Z' or to 'UX01')
- auxiliary subpackages of interface definitions and tables
- examples

The model definition can be seen in the library. Some of the models are explained in detail within the next paragraph. The icons of some models can be seen in **Fig. 1**. Most of the icons correspond to the European standard [8].

The digital library will be developed in at least two steps. The first step contains the devices mentioned above. Components like flip-flops, transfer gates, memories (RAM, ROM), and multiplexers are still missing. The behavioural models of these components will be added within the second step of library development. At the present stage such devices must be composed using the available gates. Examples of such compositions can be found in the example subpackage.

3 Details of Implementation

The basic idea was to offer a library of digital logic devices which can be placed and connected by the user to model a digital logic scheme. Otherwise, Modelica also allows to create models in a netlist like way by instan-

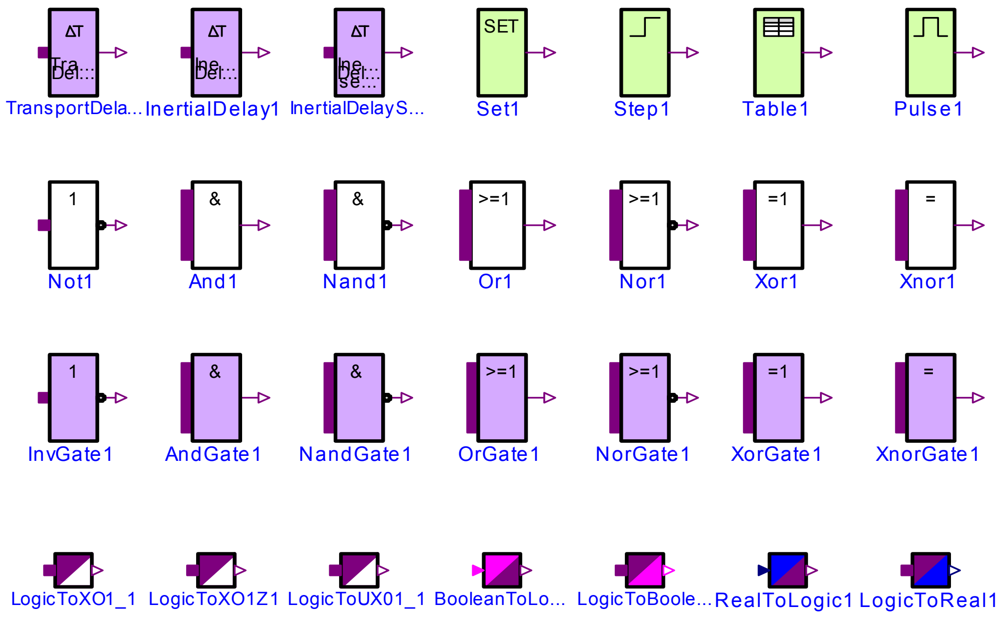


Figure 1: Components of the Modelica package *Modelica.Electrical.Digital*.

tiating and connecting devices on a text level. In both cases, a network of digital devices can be described on its connections where digital logic signals are transmitted. In this paragraph the behavioural modelling of some devices is shown exemplarily.

Since the number of logic values is limited signals do not change continuously but at discrete event times. Furthermore, nothing has to be differentiated. To calculate the output of digital devices an intensive usage of the algorithm section is necessary in the models. The simulator's task is not to solve a DAE but a system of algebraic equations at discrete event time, whose dimension is normally high and which contains lots of conditional clauses.

3.1 Signals and Connectors

The nine logic values are coded using an integer logic type:

```
type Logic = Integer

record LogicValue
  constant Integer min=1;
  constant Integer max=9;
  constant Logic 'U'=1 "Uninitialized";
  constant Logic 'X'=2 "Forcing Unknown";
  constant Logic '0'=3 "Forcing 0";
  constant Logic '1'=4 "Forcing 1";
  constant Logic 'Z'=5 "High Impedance";
  constant Logic 'W'=6 "Weak    Unknown";
  constant Logic 'L'=7 "Weak    0";
  constant Logic 'H'=8 "Weak    1";
  constant Logic '-'=9 "Don't care";
end LogicValue;
```

The sequence coded in this record corresponds to the IEEE 1164 sequence. This way simplifies the adaptation of logic value tables from the standard. Later on this record definition could be replaced by an enumeration type definition.

At the connections (ports) of the devices logic values are transmitted. Therefore, connectors are defined which only need a logic value signal. Since in most cases the signal flow direction is well defined, input and output connectors are specified:

```
connector DigitalSignal=Logic
"Digital port (both input/output
                possible)";
connector DigitalInput=input DigitalSignal;
connector DigitalOutput=
                output DigitalSignal;
```

The signals at the connectors are scalar ones. If vectors of signals are needed vectors of connectors have to be defined. This idea is taken over from the Modelica.Blocks library. The usage of both scalar and vector connectors can be seen at the following partial model for multiple input - single output devices which is used for modeling of Basics and Gates:

```
partial block MISO
  import D = Modelica.Electrical.Digital;
  parameter Integer n(final min=2) = 2
                        "Number of inputs";
  D.Interfaces.DigitalInput x[n];
  D.Interfaces.DigitalOutput y;
end MISO;
```

3.2 Basics

In the Basics subpackage the simple logic operations Not, And, Nand, Or, Nor, Xor, and Xnor are modeled.

The Not model is a single-input-single-output model. The logic input value, which is an integer between 1 and 9, specifies the row in the NotTable in which the output value can be found that negates the input value. The Modelica text of the Not device is:

```
model Not
  import D = Modelica.Electrical.Digital;
  import L = D.Interfaces.LogicValue;
  extends D.Interfaces.SISO;
protected
  D.Interfaces.Logic
                 auxiliary(start=L.'0');
equation
  auxiliary = D.Tables.NotTable[x];
  y = pre(auxiliary);
end Not;
```

The NotTable is defined in the IEEE 1164 standard:

```
input:   U  X  0  1  Z  W  L  H  -
output:  U  X  1  0  X  X  1  0  X
```

Regarding that the input code is used as index in the NotTable array, this is described in Modelica:

```
constant D.Interfaces.Logic
NotTable[L.max]=
 {L.'U',L.'X',L.'1',L.'0',L.'X',
  L.'X',L.'1',L.'0',L.'X'};
```

In the model the result is not put to the output directly but the pre-operator is applied to an intermediate variable. This is necessary to avoid algebraic loops which can appear in some cases. Therefore, the pre-operator is generally used.

As an example with multiple inputs the And model is explained. The source code is without any annotations:

```
model And
  import D = Modelica.Electrical.Digital;
  import L = D.Interfaces.LogicValue;
  extends D.Interfaces.MISO;
protected
  D.Interfaces.Logic
           auxiliary[n](each start=L.'U');
equation
  auxiliary[1] = x[1];
  for i in 1:n - 1 loop
    auxiliary[i + 1] =
    D.Tables.AndTable[auxiliary[i],x[i + 1]];
  end for;
  y = pre(auxiliary[n]);
end And;
```

The And model inherits the MISO partial model (c.f. 3.1). Within a loop to the first two input signals the *and* operation is applied. To the result and the next input signal the *and* operation is applied again, until all inputs are combined. Like in the Not model the pre-operator is used . The and-operator is realised using the AndTable. The code numbers of the input signals define the position (both row and line number) in the matrix where the result can be found. Written in an abbreviated form the AndTable is:

```
input1   U  X  0  1  Z  W  L  H  -

i  U     U  U  0  U  U  U  0  U  U
n  X     U  X  0  X  X  X  0  X  X
p  0     0  0  0  0  0  0  0  0  0
u  1     U  X  0  1  X  X  0  1  X
t  Z     U  X  0  X  X  X  0  X  X
2  W     U  X  0  X  X  X  0  X  X
   L     0  0  0  0  0  0  0  0  0
   H     U  X  0  1  X  X  0  1  X
   -     U  X  0  X  X  X  0  X  X
```

In the models Nand, Nor, and Xnor the NotTable is applied to the result of the And-, Or-, and Xor-tables respectively.

3.3 Delays

In the library there are three delay models. The *transport delay* model is an application of the Modelica delay operator. The input signal is delayed by delayTime exactly as it is. The output of the model can be specified for the time interval between zero and delayTime. The algorithm section of the TransportDelay model is:

```
algorithm
  x_delayed := integer(delay(x, delayTime));
  y := if delayTime > 0 then
         if time >= delayTime then x_delayed
         else y0
       else x;
```

Another type of delay models is the *inertial delay.* In the InertialDelay model the input value must keep constant for the delayTime interval before it is passed on the output. The Modelica code of the inertial delay is:

```
block InertialDelay
  import D = Modelica.Electrical.Digital;
  import I = D.Interfaces;
  import L = D.Interfaces.LogicValue;
  extends DI.SISO;
  parameter Modelica.SIunits.Time
            delayTime=0 ;
  parameter DI.Logic y0=L.'U';
protected
  DI.Logic y_auxiliary(start=y0, fixed=true);
  DI.Logic x_old(start=y0, fixed=true);
  discrete Modelica.SIunits.Time
  t_next(start=delayTime, fixed=true);
algorithm
  when delayTime > 0 and change(x) then
    x_old := x;
    t_next := time + delayTime;
  elsewhen time >= t_next then
    y_auxiliary := x;
  end when;
  y := if delayTime > 0 then y_auxiliary
       else x;
end InertialDelay;
```

If the input signal x changes its value, the variable t_next is set to that time at which the output should change, that means at time + delayTime. If the time reaches t_next without another input change then the input change becomes active at the output. Otherwise if x changes before t_next, t_next is increased due to the new input change. In **Fig. 2** an example of an inertial delay with delayTime=1s is shown. Input changes smaller than 1s are neglected.

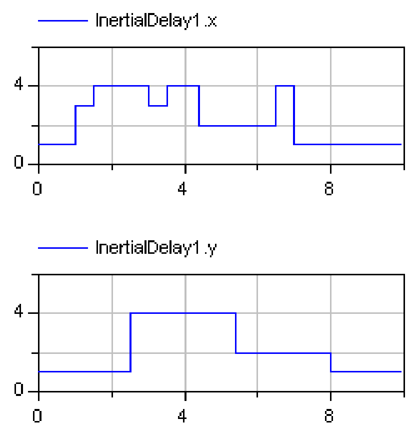


Figure 2: Inertial delay example

A generalization of the inertial delay is the sensitive intertial delay *InertialDelaySensitive*. For rising and falling edges different delay times can be specified. With a delay table it is decided whether a signal changing is regarded as rising (1) or falling (-1) or indifferent (0). Indifferent changes are not delayed. The delay table used in this library is:

```
after     U  X  0  1  Z  W  L  H  -

b  U      0  0 -1  1  0  0 -1  1  0
e  X      0  0 -1  1  0  0 -1  1  0
f  0      1  1  0  1  1  1  0  1  1
o  1     -1 -1 -1  0 -1 -1 -1  0 -1
r  Z      0  0 -1  1  0  0 -1  1  0
e  W      0  0 -1  1  0  0 -1  1  0
   L      1  1  0  1  1  1  0  1  1
   H     -1 -1 -1  0 -1 -1 -1  0 -1
   -      0  0 -1  1  0  0 -1  1  0
```

3.4 Gates

In the Gates subpackage there are collected the InvGate, AndGate, NandGate, OrGate, NorGate, XorGate, XnorGate, and the BufGate. Each of the gates is graphically composed by a basic logic model whose output is delayed by a sensitive inertial delay. The InvGate consists of a Not model with delayed output. As a special case the BufGate consists only of a sensitive inertial delay. For the sake of completeness the BufGate should belong to that subpackage. In **Fig. 3** the composition of Gates is demonstrated considering the AndGate as an example. The strange connecor at the left hand site is an interim solution of painting vectors of connectors.

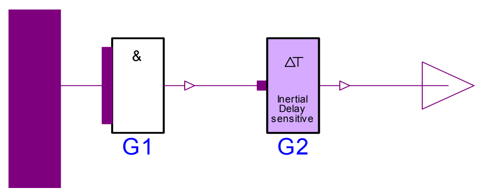


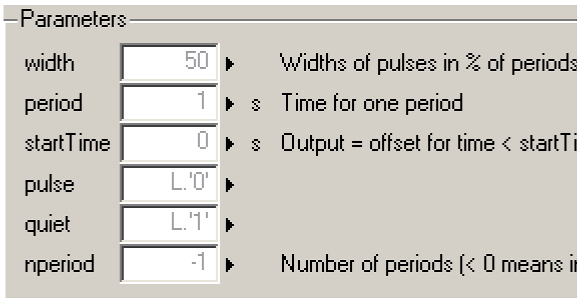
Figure 3: AndGate

3.5 Sources

The sources Set, Step, Table, Pulse, and Clock are not borrowed from the standard but written as nice-to-have sources. The *Set* source simply sets a logic value. *Step* steps one-time from one value to a second value at a given time. The *Table* source follows a user specified value-time-table. The essential part of the Modelica code of the Table model is (after checking the acceptance of parameters):

```
algorithm
  y := y0;
  for i in 1:n loop
    if time >= t[i] then
      y := x[i];
    end if;
  end for;
end Table;
```

With the *Pulse* source arbitrary pulsing between two values can be created. The Pulse parameters are shown in **Fig. 4**.



Parameters			
width	50		Widths of pulses in % of periods
period	1	s	Time for one period
startTime	0	s	Output = offset for time < startTi
pulse	L.'0'		
quiet	L.'1'		
nperiod	-1		Number of periods (< 0 means i

Figure 4: Pulse Source Parameters

The *Clock* source is a simplified Pulse source without counting the number of periods which pulses between ’0’ and ’1’. The code for generating the pulsing behaviour of Clock is:

```
  Modelica.SIunits.Time t_i
    (final start=startTime)
    "Start time of current period";
  Modelica.SIunits.Time
     t_width=period*width/100;
algorithm
  when sample(startTime, period) then
    t_i := time;
  end when;
  y := if time < startTime or
          time >= t_i + t_width
        then L.'0' else L.'1';
end Clock;
```

3.6 Converters

The IEEE 1164 like converters *LogicToUX01*, *LogicToX01Z*, and *LogicToX01* map the nine-valued digital logic to the limited sets of values {’U’, ’X’, ’0’, ’1’}, {’X’, ’0’, ’1’, ’Z’}, or {’X’, ’0’, ’1’} respectively. The mapping is done with conversion tables. E.g. the conversion table for LogicToX01 is:

```
input:   U  X  0  1  Z  W  L  H  -
output:  X  X  0  1  X  X  0  1  X
```

The following converters are not from the IEEE 1164 standard.

The *BooleanToLogic* converter maps the Boolean input to Logic according to the following table (t - true, f - false):

```
input:   t  f
output:  1  0
```

The LogicTo*Boolean* converter maps the Logic input to the Boolean output according to the following table (t - true, f - false):

```
input:   U  X  0  1  Z  W  L  H  -
output:  f  f  f  t  f  f  f  t  f
```

Further conversions are possible between Real and Logic values. In the *LogicToReal* converter the Real output jumps to a real number wich can be defined by the user for each of the nine logic values. The default values are:

```
input:    U  X  0  1  Z  W  L  H  -
output:  .5 .5  0  1 .5 .5  0  1 .5
```

The RealToLogic converter has two limits: an upper limit, and a lower limit. If the input x is x > upper limit, an upper_value is chosen, If x < lower limit, a lower value is chosen, otherwise the middle_value is chosen. The limits and the values are parameters of the converter. In Figure **Fig. 5** a sine curve is converted to logic using the default parameters (lower_limit=0, upper_limit=1, lower_value=’0’, upper_value=’1’, middle_value =’X’).

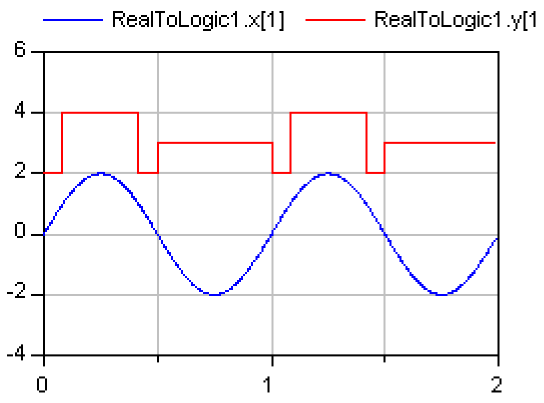


Figure 5: Default Real to Logic Conversion

4 Usage

The components of the electrical digital library can be combined to form more complex models. This is possible on the text level, or in a graphical way.
Since complex devices like flipflops, multiplexers, memories, ... are still missing, such components have to be composed using the set of basic gates. In the example package some of these components are available.
Furthermore, the user can modify the models by changing the description, adding pins, introducing parameters, fixing parameters...
The signal strengh according to the IEEE 1164 strength table is not modeled yet, since no resolution function is implemented. This will be added in a later version of that library.

Sometimes it is possible that algebraic loops occur which can be not solved by the simulator. In such cases often the inclusion of additional delay components helps.
The whole variety of the possibilities of the library usage is not presented. Some aspects of the library usage are demonstrated in the examples.

5 Examples

The examples are part of a validation suite, some of them are furthermore part of the library example subpackage. They show some of the possibilities of the library. Since the library is developed recently further tests e.g. with 'large' logic designs are necessary. Test examples were developed using wellknown textbooks [9], [10], [11]. All examples presented here were simulated using the simulator Dymola5.3a [12].

5.1 Logic Equivalence

This simple example tests the logic equivalence $AB \vee \bar{A}C \vee BC = AB \vee \bar{A}C$ by modeling both sides $X = AB \vee \bar{A}C \vee BC$ and $Y = AB \vee \bar{A}C$ of the logic equation with basic components.

The following Modelica text shows the circuit description without graphical instructions. The instantiation of library components can be seen as well as the usage of parameters. Once instantiated the devices are connected in the equation part.

```
model LogicEquivalence
  import DD=Modelica.Electrical.Digital;
  DD.Basic.And And1, And2, And3;
  DD.Basic.Or Or1, Or2(n=3);
  DD.Basic.Not Not;
  DD.Sources.Table TabB
   (x={3,3,4,4,3,3,3,3,3,4,4,4,4,3,3},
   t={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14});
  DD.Sources.Table TabA
   (x={3,3,4,4,4,4,4,3,3},
   t={0,1,2,3,4,5,6,7,8},);
  DD.Sources.Table TabC
   (x={3,3,4,3,3,4,3,3,3,3,4,3,3,3,4,3,3},
   t={0,1,2,3,4,5,6,7,8,9,10,11,
      12,13,14,15,16});
  DD.Interfaces.Logic X, Y;
equation
  connect(TabA.y, And1.x[2]);
  connect(TabA.y, Not.x);
  connect(TabB.y, And1.x[1]);
  connect(TabB.y, And3.x[2]);
  connect(TabC.y, And2.x[1]);
  connect(TabC.y, And3.x[1]);
  connect(And1.y, Or1.x[2]);
  connect(And1.y, Or2.x[3]);
  connect(And2.y, Or2.x[2]);
  connect(And2.y, Or1.x[1]);
  connect(AndB3.y, Or2.x[1]);
  connect(Not.y, And2.x[2]);
  X = Or2.y;  Y = Or1.y;
end LogicEquivalence;
```

More instructive is the graphical representation like **Fig. 6** which is normally used to model digital circuits:

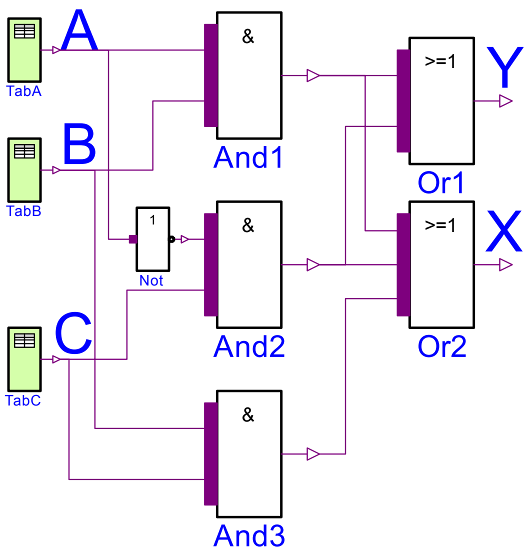


Figure 6: Logic Equivalence Circuit

The simulation result are the outputs X and Y of both Or components which are equivalent. Furthermore the input values of TabA, TabB, and TabC are shown which correspond to A, B, and C:

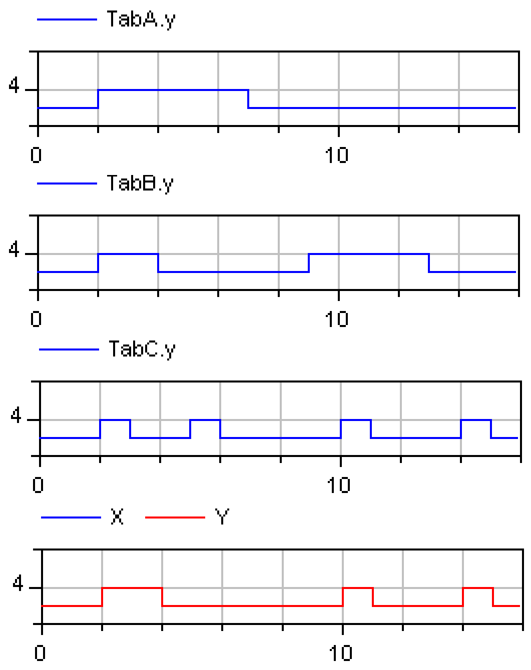


Figure 7: Simulation results of the equivalence circuit

5.2 Half-Adder

A half-adder can be found in the Examples.Utility package. It is composed according to **Fig. 8** using gates with a delay of 0.5s.

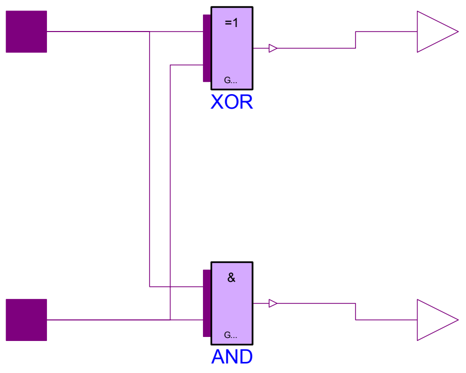


Figure 8: Half-Adder

Starting with 'Unknown' at the signal inputs a and b, and testing all combinations with '0' and '1' the behaviour is as expected, c.f. **Fig. 9**.

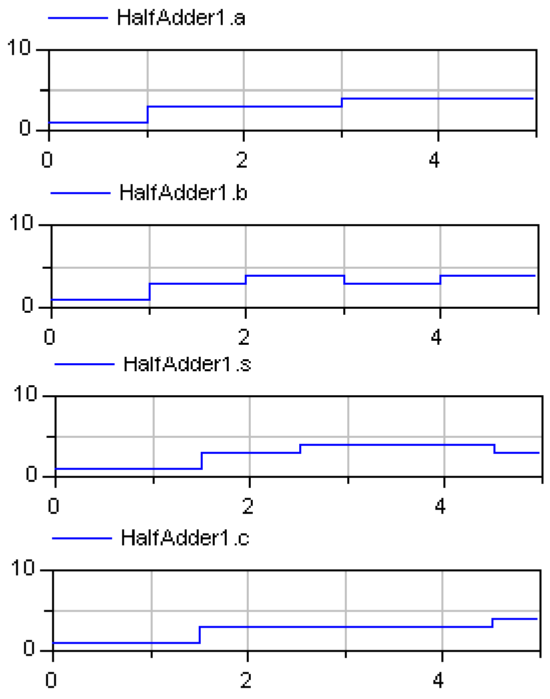


Figure 9: Half-Adder, Results

5.3 JK-Flip-Flop

A JK-Flip-Flop (with inputs j, k, and clock) is composed according to **Fig. 10**. It uses a static RS-Flip-Flop which is shown in **Fig. 11**. Both components are in the Examples.Utitlies package of the Digital library.

The results in **Fig. 12** show the behaviour of the JK-Flip-Flop: If J is '0' (coded by 3), the output q follows K, if both inputs are '1' (coded by 4), the output is clocked, if J is '1' and K is '0' the output becomes '0'.

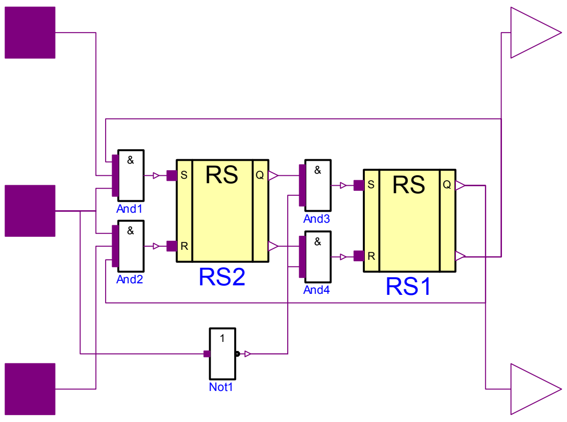


Figure 10: JK-Flip-Flop

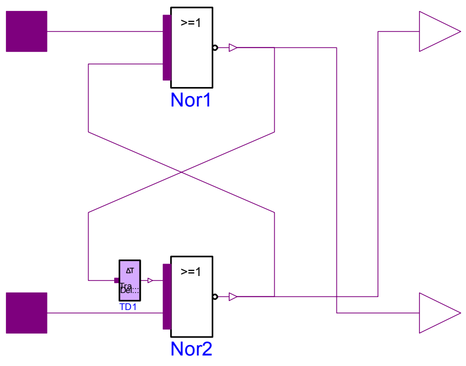


Figure 11: RS-Flip-Flop

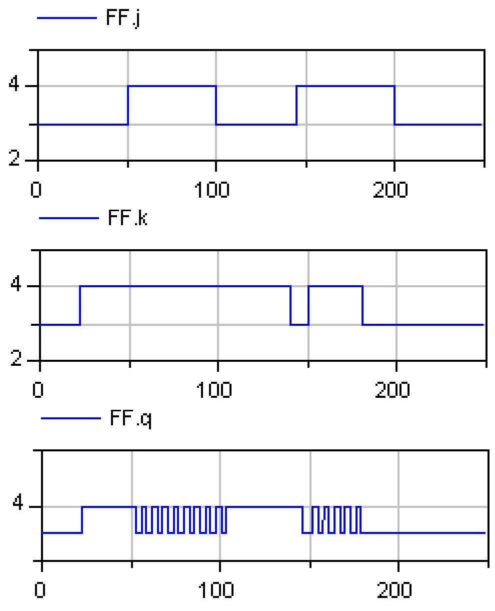


Figure 12: Results of the JK-Flip-Flop

5.4 Adder with Counter

Two half-adders described in 5.2 and an or gate can be combined to a full-adder which is able to add two digits including a carry bit from a preceding full-adder. In **Fig. 12** the schematic of the full-adder is shown.

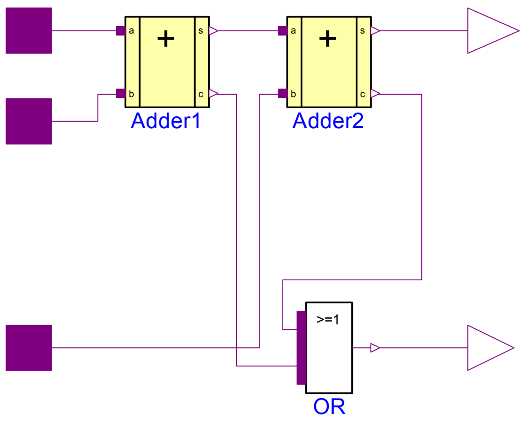


Figure 12: Full-Adder Schematic

The JK-Flip-Flop described in **Fig. 10** can be combined to a counter. Depending on the number of Flip-Flops the number of digits can be chosen. Figure **Fig. 13** shows the schematic of a three-bit-counter.

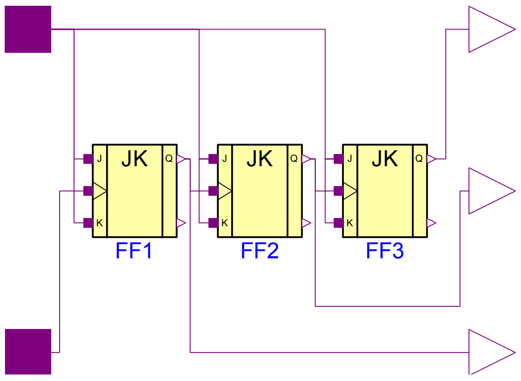


Figure 13: Three-Bit-Counter

Within the schematic of **Fig. 14** the three-bit-counter output is taken as input of a full-adder. The full-adder sums up the three outputs of the counter. The result can be found in **Fig. 15**.

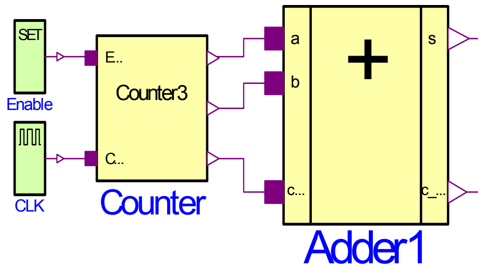


Figure 14: Counter with Adder

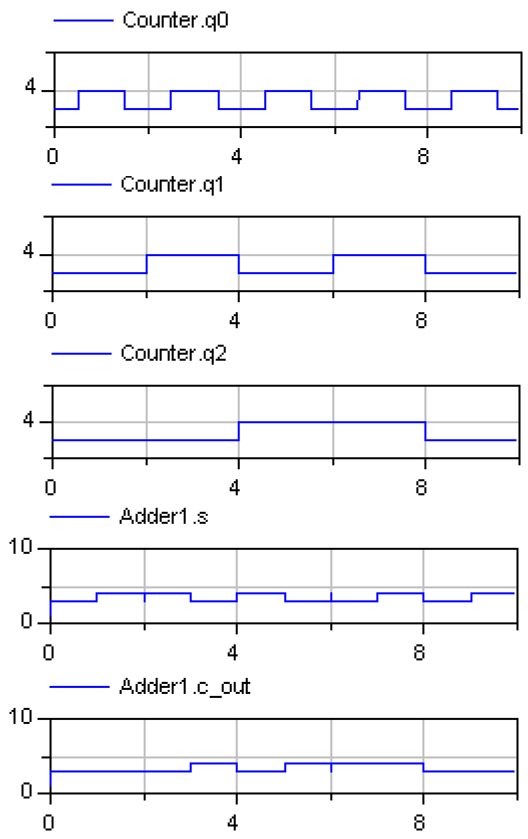


Figure 15: Results of the Counter with Adder

6 Summary

The digital electric library presented is part of the Modelica standard library. In this paper the devices and their principles of implementation are explained. Some examples show the usage of this library.

Although tested during development a wide usage is desirable to get extensive experiences. Especially large designes are needed as well as mixed applications with other physical domains.

Once the first version of the digital library is accepted it will be extended by behavioural models of flip-flops, latches, transfer gates, tristate devices, multiplexers, and memories. A discussion on principles will be expected concerning the introduction of a resolution function at general nodes.

This work was supported by the Sonderforschungsbereich 358, Teilprojekt D4 of the Deutsche Forschungsgemeinschaft. We are grateful to Dr. Peter Schwarz, and Dr. Martin Otter for encouraging this work, and to our students Teresa Schlegel, Liane Jacobi, and Hagen Staemmler for implementing and testing.

7 References

[1] Elmqvist, H. et al.: Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Version 2.1, January 2004. http://www.Modelica.org

[2] Otter, M.; Elmqvist, H.; Mattsson, S.E.: Objektorientierte Modellierung physikalischer Systeme, Teil 8. at Automatisierungstechnik 47(1999)9

[3] IEEE Standard Multivalue Logic System for VHDL Model Interoperability. http://www.ieee.org

[4] IEEE 1076-1993: IEEE Standard VHDL Language Reference Manual (ANSI). 288 p. ISBN 1-55937-376-8. IEEE Ref. SH16840-NYF.

[5] IEEE 1164-1993: IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164). 24 p. ISBN 1-55937-299-0. IEEE Ref. SH16097-NYF.

[6] Lipsett, R.; Schaefer, C.; Ussery, C.: VHDL: Hardware Description and Design. Boston: Kluwer, 1989, 299 p. ISBN 079239030X.

[7] Navabi, Z: VHDL: Analysis and Modeling of Digital Systems. New York: McGraw-Hill, 1993, 375 p. ISBN 0070464723.

[8] Normen über graphische Symbole für die Elektrotechnik, Schaltzeichen. DIN-Taschenbuch 514, Beuth Berlin, Wien, Zürich, 1994

[9] Ashenden, P. J.: The Designer's Guide to VHDL. San Francisco: Morgan Kaufmann, 1995, 688 p. ISBN 1-55860-270-4.

[10] Horowitz, P.; Hill, W.: The Art of Electronics. Cambridge University Press, 1989, ISBN 0-521-37095-7

[11] Tietze, U.; Schenk, C.: Halbleiter-Schaltungstechnik. Springer-Verlag Berlin, Heidelberg, New York, 1980, ISBN 3-540-09848-8

[12] Dymola: http://www.Dynasim.se