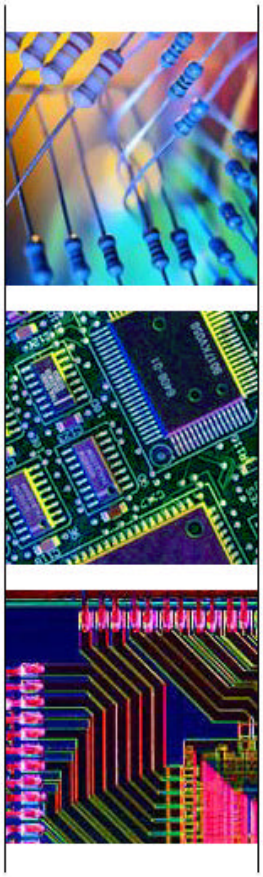


Modellierung und Simulation von Mixed-Signal-Systemen mit VHDL-AMS

Fachhochschule Esslingen

Hochschule für Technik

Fachbereich: Informationstechnik

Studienarbeit von

Mirko Pfitzner

Adrian Wöhr

INHALTSVERZEICHNIS

VORWORT

Damit der Leser in etwa weiß, was ihn erwartet, soll hier eine kurze Erklärung zum inhaltlichen Aufbau gegeben werden.

- Der **erste Teil** befasst sich mit **Mixed-Signal-Systemen** und der Sprache **VHDL-AMS**. Es soll einer knapper Überblick über die VHDL-AMS-Welt gegeben werden, sowie auf verfügbare **Literatur** und **Websites** kurz eingegangen werden.

- Im **zweiten Teil** werden verschiedene einfache digitale, digital/analoge und analoge **VHDL-AMS-Modelle** als konkrete Beispiele in Bezug auf Sprache und Simulation mit Hilfe des Tools **hAMSter** vorgestellt.
  Dabei sind die **ersten zwei Beispiele** bezüglich der sprachlichen Elemente besonders ausführlich, um die Grundstrukturen der Sprache VHDL vorzustellen (XOR-Beispiel, Struktur), und die wichtigsten AMS-Elemente zu erlernen (Einfaches Mixed-Mode-Beispiel), sowie den praktischen Umgang mit dem **Modeleditor** von hAMSter kurz vorzustellen.
  Die **nachfolgenden Beispiele** veranschaulichen **weitere Sprachelemente** und zeigen verschiedenartige **Möglichkeiten** von VHDL-AMS auf, wobei auch nach und nach Features von **hAMSterView**, dem Simulations-Viewer von hAMSter, kurz erläutert werden, die das praktische Arbeiten erleichtern.

  Alle Beispiele sind von der hAMSter-Software und können natürlich selber getestet werden.

- Im **Anhang** sind die wichtigsten **Daten, Tabellen, Listen** und **Dateien** auf einen Blick zu finden.

1 Einführung

1.1 VHDL-AMS

1.1.1 Was ist VHDL-AMS?

VHDL ist eine seit 1987 vom IEEE[1] standardisierte überaus mächtige Hardware-Beschreibungssprache mit sehr vielen Modellierungsmöglichkeiten, welche ihre Wurzeln, wie so oft bei neuen Konzepten, im Militär hat. Der bisher aktuellste VHDL-Stand wurde 1993 als IEEE 1076-Standard festgelegt, welcher bis zu diesem Zeitpunkt nur rein digitale Modellierung und Simulation erlaubte. Dieser wurde 1999 um die Erweiterung AMS, dem IEEE 1076.1-Standard ergänzt. Beide Standards zusammen genommen sind als VHDL-AMS bekannt. Dieses neue Konzept wird von der Automotiv-Branche besonders vorangetrieben.

VHDL-AMS steht für

> **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage – **A**nalog- and **M**ixed-**S**ignal Extensions

Die AMS-Erweiterung macht es möglich, Mixed-Signal- und Mixed-Technology-Systeme in eine einzige Sprache mit einzubeziehen, wobei die bisherigen VHDL-Festlegungen natürlich weiterhin gelten. Mit **Mixed-Signal-Systemen** ist gemeint, dass

- digitale und
- analoge

Elemente in einer Beschreibung enthalten sind. Mit „analog“ sind **zeit- und wertkontinuierliche** Signale gemeint. Derartige Systeme kommen sehr oft vor, einfachstes Beispiel ist ein A/D- oder D/A-Umsetzer.

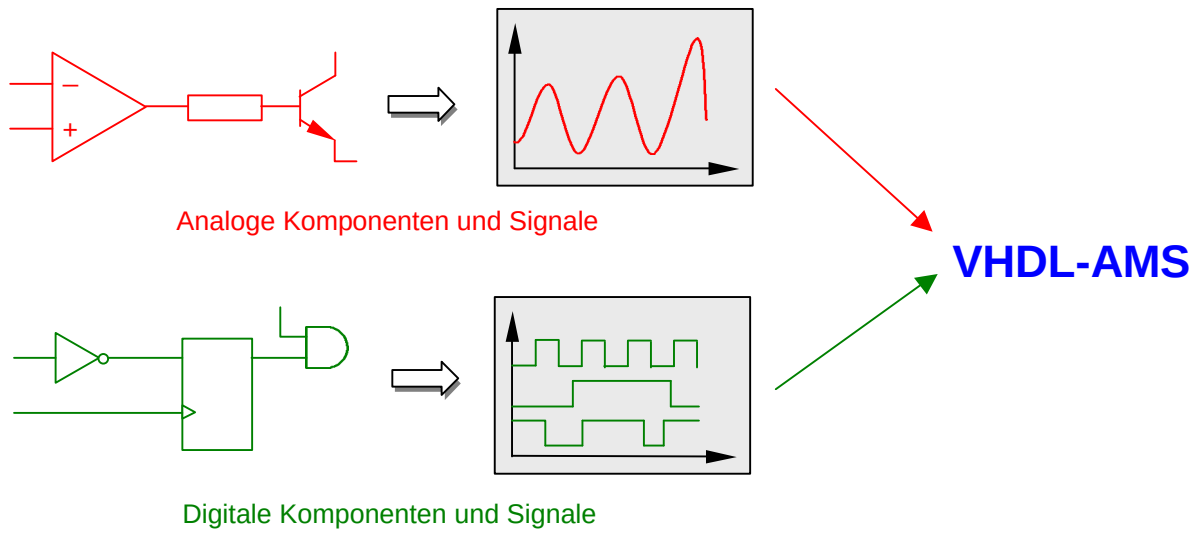
Handelt es sich um ein **Mixed-Technology-System**, so sind Elemente verschiedener physikalischer Gebiete enthalten, die natürlich auch mit elektronischen Signalen kombiniert werden können. Das heißt, diese Systeme, auf welche hier nicht

[1] Institute of Electric and Electronic Engineers

weiter eingegangen werden soll, können beispielsweise aus einem oder mehreren folgender Bereiche bestehen:

- Mechanik
- Fluid-Technik
- Hydraulik
- Thermik
- Mikrosysteme
- Chemie

So ist es möglich, auch komplexe Systeme im Gesamten zu modellieren, simulieren und analysieren, was ansonsten nur getrennt verwirklicht werden konnte.
Wichtigster Aspekt ist wohl die **Zusammenführung der analogen mit der digitalen Welt** sowie die größeren Möglichkeiten der Sprache im Analog-Bereich. Hier soll auch der Fokus dieser Ausarbeitung liegen. Obwohl Schaltkreise im zunehmenden Maße digital werden, wird die analoge Elektronik nie aussterben. Wir leben schließlich in einer „analogen Welt“. Darum ist diese Analog/Digital-Schnittstelle nach wie vor von hoher Wichtigkeit bei der Schaltungs-Simulation.
Folgende Abbildung verdeutlicht die Forderung und den Anspruch von VHDL-AMS [1]:

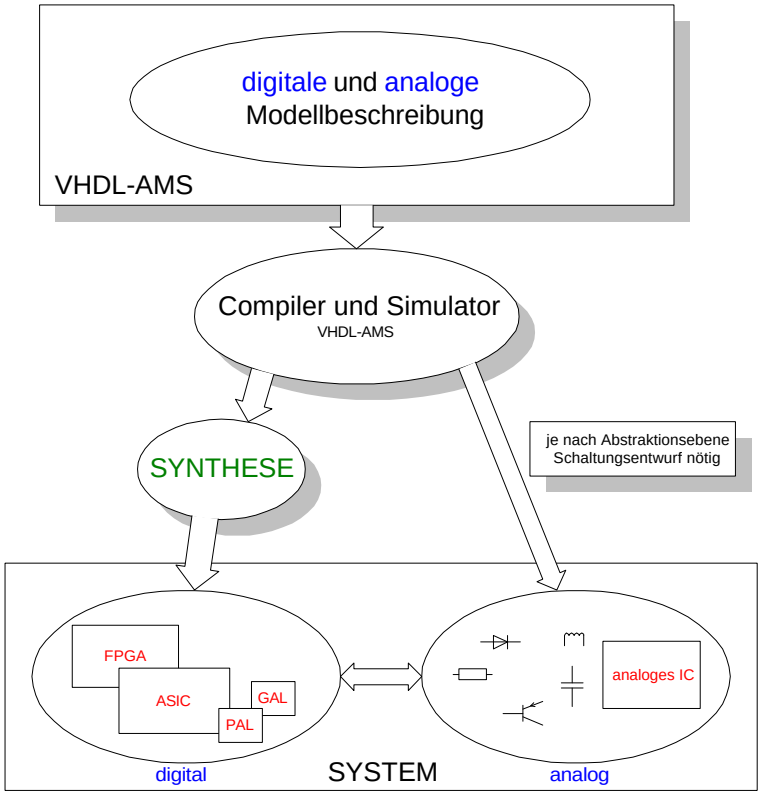


1.1.1.1 Modellierung und Synthese

Zu beachten ist, dass **Modellierung** und **Synthese** zwei verschiedene Paar Schuhe sind. Für die Modellierung (Editor) und die Simulation (Simulator) gibt es Mixed-Signal-Tools, wie z.B. das hier verwendete hAMSter, wohingegen eine Synthese

bislang nur digitalen Systemen vorbehalten ist. Der Grund dafür ist, dass Analog-Synthese eine mit vielen Randbedingungen verbundene hochkomplizierte Angelegenheit ist, für die man in naher Zukunft keine Lösungen erwarten kann. Das heißt also, den Analog-Part kann man zwar durch Gleichungen modellhaft beschreiben und dann auch, und das ist das Wichtige, zusammen mit dem Digital-Part zusammen **simulieren** und **analysieren**, aber es kann keine Schaltung synthetisiert, also „generiert“ werden. Dieses Generieren ist ja für digitale Systeme heute kein Problem. Ein fertig kompiliertes VHDL-Modell wird zusammen mit verschiedenen Bibliotheken einem Synthese-Tool zugeführt, welches dann direkt den Code zum Beschreiben von FPGAs[2] oder zum Herstellen von ASICs[3] erzeugt.

Die Zeichnung zeigt, wie die Realisierung eines Mixed-Signal-Systems aussehen könnte.

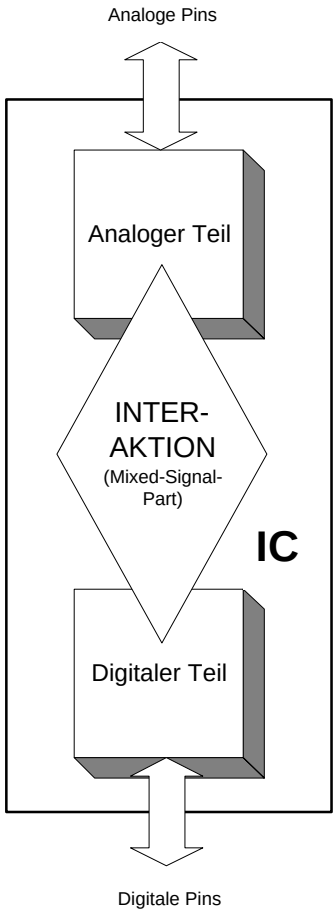


Ein **analoges Modell**, das in der Simulation funktioniert, muss jedoch erst schaltungstechnisch realisiert bzw. entwickelt werden. Relativ einfach ist es dann, wenn normale passive Bauelemente wie Widerstände, Kondensatoren, oder auch aktive, wie Transistoren verwandt wurden. Ist die analoge Beschreibung jedoch hauptsächlich aus abstrakt formulierten Gleichungen aufgebaut, die das gewünschte noch nicht existierende System darstellen sollen, so ist natürlich nach erfolgreicher Simulation noch viel Arbeit mit der physikalischen Realisierung verbunden. Man spricht dann auch von einem **Nicht-Konservativen**

[2] Field Programmable Gate Array

[3] Application Specific Integrated Circuit

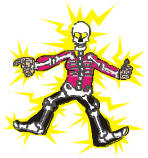
System, wenn also nur das Verhalten beschrieben wird, jedoch nicht unbedingt auf physikalische Zusammenhänge eingegangen wird.
Mit anderen Worten: Modelliert man beispielsweise einen Operationsverstärker mit bestimmtem geforderten Verhalten mit Hilfe von Gleichungen und es gibt genau diesen OP als IC – die Vorgehensweise könnte übrigens auch anders herum sein – so stellt das ja kein Problem dar. Was aber, wenn man auf einer sehr hohen analogen Abstraktionsebene arbeitet, d.h. das was man beschreibt gibt es beispielsweise noch nicht als IC?



Mixed-Signal-IC

Ähnliche Überlegungen könnte man auch für den Digital-Teil anstellen, jedoch liegt es in der Natur der Digitaltechnik – das ist ein entscheidender Vorteil – dass eben komplexe digitale Baugruppen einfach aus elementaren Baugruppen zusammengesetzt werden können; dafür gibt es erschöpfend viele **Bibliotheken**. Sogar für Modelle, die mit einer Verhaltensbeschreibung modelliert wurden, gibt es heutzutage leistungsfähige Synthese-Werkzeuge.

Es ist auch möglich, gemäß nebenstehender Abbildung einen sogenannten **Mixed-Signal-Chip** zu entwickeln. Bei dieser Technologie findet sich der analoge und digitale Teil auf einem einzigen IC wieder. Hier wird der Vorteil einer gemeinsamen Entwicklungs- und Simulationsumgebung besonders deutlich.



Man erkennt also, dass man mit VHDL-AMS auf sehr vielen verschiedenen **Abstraktionsebenen**[4] arbeiten kann. Vom CMOS-Transistor bis zur kompletten ALU oder CPU, vom einzelnen Widerstand bis zur fertigen PLL. Die jeweilige Ebene wird von den Anforderungen und Voraussetzungen bestimmt.

[4] vgl. **Y-Diagramm nach Gajski-Walker.** (Zu finden in jeder Literatur, in der es um den Entwurf elektronischer Systeme geht.)

Dieses „Black-Box-Prinzip“, also das Arbeiten mit fertigen Unterbaugruppen und dessen Schnittstellen, ist bei der enormen Komplexität heutiger elektronischer Systeme unerlässlich. So spielt z.B. auch die **Wiederverwendung** von Entwurfsdaten eine wichtige Rolle.

1.1.1.2 VHDL-AMS versus SPICE

PSPICE war bisher immer noch das bevorzugte Werkzeug, wenn es um die Simulation analoger Baugruppen und Systeme geht. Da wie schon erwähnt, die einheitliche Simulation und Verifizierung von analog/digital-kombinierten Systemen immer wichtiger wird, wird sich wohl jedes Entwicklungsteam einmal die Frage stellen, ob es sich lohnt, ganz auf VHDL-AMS umzusteigen. Denn jeglicher SPICE-Code kann zu VHDL-AMS-Code konvertiert werden, jedoch nicht ohne weiteres umgekehrt.

VHDL-AMS kann als eine **Art Nachfolger** von SPICE angesehen werden und bietet eine viel höhere Flexibilität.

Das äußert sich auch durch die Möglichkeit der Verwendung von **algebraischen Gleichungen und Differenzialgleichungen** in VHDL-AMS, es können also viel höhere Abstraktionsebenen erreicht werden, wohingegen bei SPICE nur mit Elementen modelliert werden kann und eine Verhaltensmodellierung sehr schwierig hinzubekommen ist.

Einzig nachteilig wirken sich bei VHDL-AMS die Tatsachen aus, dass der Code oft länger ist als bei SPICE und dass die Rechenzeiten vom Modellierungsstil abhängen.

1.1.1.3 Alternativen und die Zukunft

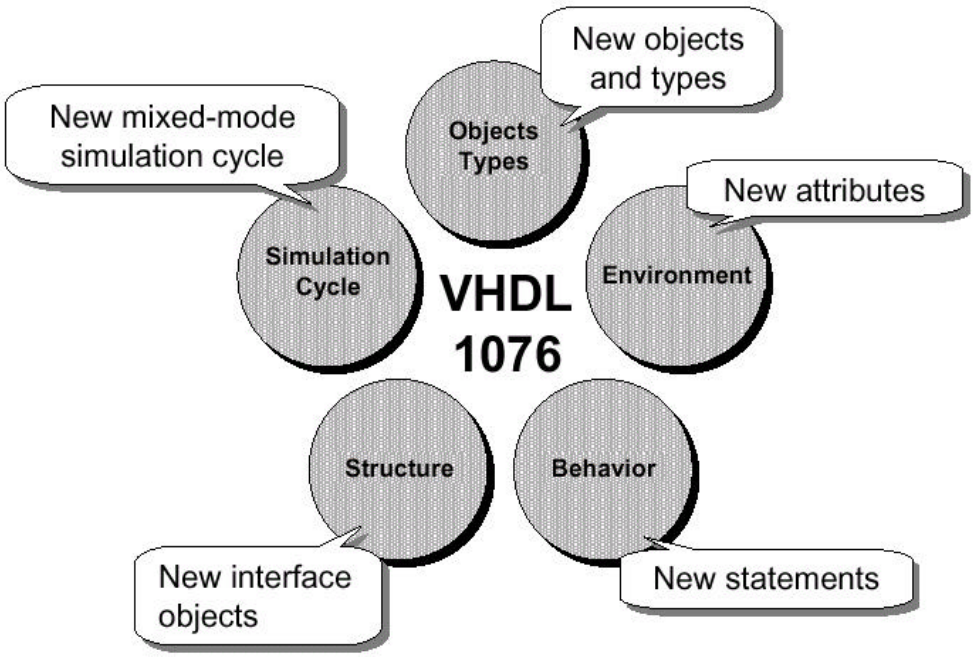
Gemischte Signale in einer einzigen Sprache zu beschreiben ist ein relativ neues Konzept, welches jedoch nach und nach immer mehr Anwendung findet. Neben VHDL-AMS gibt es noch andere Ansätze, wie z.B. Verilog-AMS oder Analog/Digital SPICE.

Für die Simulation analoger Systeme gäbe es viele altbewährte Tools, wie z.B. SPICE, ASTAP, Accusim und Spectre. Auch rein mathematische

Simulationswerkzeuge wie Matlab oder MATRIXx, die lediglich der Simulation von Nicht-Konservativen Systemen dienen, sind sehr verbreitet.
Doch nur VHDL-AMS deckt ein wirklich großes Spektrum von Einsatzgebieten gleichzeitig ab. Zusammen mit der Tatsache, dass das reine VHDL schon eine sehr große Verbreitung aufweist, rechnet man dem VHDL-AMS Standard für die Zukunft große Chancen zu.
Allein entscheidend für den Erfolg der Sprache ist jedoch die **Akzeptanz der Entwickler**. Es liegt an ihnen, eine neue Richtung einzuschlagen, und ihre bisherigen Analog-Modelle in AMS zu integrieren.

1.1.2 Kurzer Überblick über die wichtigsten neuen Elemente

Eine gute Übersicht über die Erweiterung des VHDL-Standards zu VHDL-AMS gibt die folgende Grafik:



Erweiterung von VHDL-AMS [1]

Dabei stellen die Kreise den Umfang des bisherigen Standards dar und die Sprechblasen die verschiedenen AMS-Erweiterungen.
Es gibt beispielsweise neben den bisher bekannten **Objektklassen** wie `CONSTANT`, `VARIABLE`, `SIGNAL` und `FILE` jetzt **`QUANTITIES`** und **`TERMINALS`** und mit den

Aspekten ACROSS und THROUGH können physikalische Gesetzmäßigkeiten veranschaulicht werden.
Auf einige Elemente wird später in den Beispielen näher eingegangen. Hier jedoch eine kurze Vorinformation wichtiger Neuerungen:

- QUANTITIES

  Mit ihnen definiert man verschiedenste **physikalische Einheiten**, wie z.B. Spannung und Strom. Sie werden nochmals unterteilt in
  - BRANCH Quantities
  - FREE Quantities
  - SOURCE Quantities
  - INTERFACE Quantities

- TERMINALS

  definieren allgemein **Anschlüsse**, z.B. analoge Pins und stellen einen internen Knoten dar.

- NATURES

  Sie definieren **Merkmale** eines Terminals, z.B. „electrical“. Es gibt
  - Scalare Natures
  - Composite Natures

- SIMULTANEOUS STATEMENTS

  (Simultane Anweisungen) werden z.B. gebraucht um **algebraische**- oder **Differenzialgleichungen** zu formulieren. Es gibt mehrere Arten von Simultanen Anweisungen, so können z.B. in der Verhaltensbeschreibung **if-** und **case-**Bedingungen erstellt werden.

- TOLERANCES

  ist eine **frei** durch den Anwender definierbare Gruppe und lässt sich so projekt-spezifisch einsetzen um die **Qualität** der Lösung zu bestimmen.

- ATTRIBUTS

  Viele neue Attribute verschiedener Arten wurden eingeführt. So gibt es sie für Terminals, Natures, Quantities oder Signals. Beispielsweise wird mit `Q'DOT` eine Quantity abgeleitet oder mit `S'RAMP` die Anstiegszeit eines Signals nachgebildet.

1.1.3 hAMSter und andere VHDL-AMS Simulatoren

Simulatoren für Mixed-Signal-Systeme sind ein großes Kapitel für sich. Soviel sei gesagt, dass besonders die Schnittstelle und Interaktivität zwischen internem analogen und digitalem Simulator besonders problembehaftet und kompliziert ist. Analog- und Digital-Simulatoren sind vollkommen verschieden aufgebaut. Da ein Digital-Simulator von stark vereinfachten Annahmen ausgehen kann, wie z.B. nur bestimmte Pegel und festdefinierte An- oder Abstiegszeiten, braucht er nur einen Bruchteil (etwa 1/1000) der Rechenzeit, verglichen mit der, die ein Analog-Simulator für eine äquivalente Schaltung bräuchte. Digitale Systeme sind häufig auch getaktet, also nicht nur wertdiskret sondern auch zeitdiskret, wohingegen analoge Systeme wert- und zeitkontinuierlich sind. Man erkennt schnell, das ein idealer Analog-Simulator unendlich viele Werte verarbeiten müsste und das die Genauigkeit eines realen Simulators dann maßgeblich von der Abtastfrequenz abhängt.
Ein analoger Simulator wäre für digitale Signale höchst ungeeignet, weil er für viele verschiedene Signalformen ausgelegt ist. Über den umgekehrten Fall braucht man natürlich gar nicht erst nachzudenken.
Es wird also offensichtlich, dass die Koordination und die Kopplung beider Simulatoren auch in Zukunft wohl die größte Herausforderung der Hersteller bleibt.

Es gibt zwar über ein Dutzend Anbieter von Mixed-Signal-Tools, jedoch nur drei dieser Tools **unterstützen bis heute VHDL-AMS**, wobei die Anzahl in den nächsten Jahren wohl wachsen wird, gerade auch auf dem Gebiet der grafisch-unterstützten Editoren, welche auch heute schon vereinzelt erhältlich sind.

Die drei Produkte sind:

- ADVance MS von Mentor Graphics
- TheHDL (VeriasHDL-AMS Simulator) von Analogy
- hAMSter von SIMEC/Ansoft

Wir werden hAMSter für unsere Beispiele benutzen. Die zwei ersteren sind große und teure Entwicklungsumgebungen. hAMSter hingegen ist schlank und für die Evaluierung sogar kostenlos. So ist es gerade für Studenten und alle, die sich mit VHDL-AMS vertraut machen wollen, bestens geeignet.
Hier die Vorteile von hAMSter auf einen Blick:

- ☑ preisgünstig
- ☑ flexibel
- ☑ einfach
- ☑ für Windows-Plattformen

hAMSter kann von „www.hamster-ams.com“ heruntergeladen und ohne Problem auf Windows-Plattformen installiert werden. Das Tool benötigt keinen schnellen PC und belegt weniger als 20MB Festplatten-Speicher.
Mir dem Editor von hAMSter kommt man sehr schnell zurecht. Er bietet eine sehr hilfreiche Syntax-Kolorierung. Der farbig gestaltete Simulator offeriert einige Optionen und stellt die analogen und digitalen Simulationsdiagramme anschaulich übereinander dar.
Doch dazu mehr bei den jeweiligen Beispielen.
Man beachte jedoch: Es gibt Einschränkungen bei hAMSter, nicht der gesamte Standard ist implementiert. So stehen z.B. Features wie `TOLERANCES`, `COMPOSITE NATURES` oder `POINTER` nicht zur Verfügung. Eine Übersicht über alle unterstützten und nicht unterstützten VHDL-AMS Strukturen gibt es auf der Website unter 'about hAMSter, hAMSter implementation'.

1.2 Internetübersicht und Dokumentationen

Die Durchforstung des Internets ist immer noch die beste und schnellste Methode, sich aktuelle Daten, Infos und Dokumentationen zu einer relativ jungen Technologie zu beschaffen.

Die Suchmaschinen reagierten recht unterschiedlich auf Suchverknüpfungen mit den zwei Hauptelementen 'VHDL' und 'AMS'. Von knapp 100 Hits bis über 3000 (Google) war alles vertreten. Wie üblich wiederholen sich viele Einträge auf die ein oder andere Weise. So ist es relativ schwer, gute allgemeine Informationen zu VHDL-AMS herauszufiltern. Wenn man jedoch Informationen zu speziellen Details sucht, sind die Suchmaschinen sehr hilfreich.

Nach ausführlicher Recherche konnten wir zum Schluss kommen, dass die Website-Sammlung in der hAMSter-Dokumentation die beste Basis rund um das Thema VHDL-AMS darstellt.

Im Folgenden sind diese und noch andere aufschlussreiche **WWW-Links mit Inhaltsüberblick** sowie kurzem Eindruck aufgelistet [siehe auch Anhang I.]:

- http://www.hamster-ams.com/
  - Die Site von SIMEC. Hier findet man alles über hAMSter und kann z.B. auch die hier behandelten Beispiele sowie hAMSter selbst downloaden.
  - Kostenloser E-Mail-Support soll gewährleistet sein und Modellierungsprobleme können auf deren Newsserver diskutiert werden.

- http://www.eda.org/vhdl-ams/
  - Offizielle Homepage der IEEE1076.1-Workinggroup der EDA[5], die sich mit der Analog- und Mixed-Signal-Extension von VHDL befasst.
  - Hier gibt es die 'IEEE VHDL 1076.1 Language Reference Manual', allerdings nicht kostenlos.
  - Ausführliches Tutorial: VHDL-AMS Tutorial.pdf [siehe weiter unten].

- http://www.ececs.uc.edu/~dpl/
  - Page der University of Cincinnati, die sich viel mit VHDL-AMS befasst.

[5] Electronic Design Automation

- Sehr gut: Hier gibt es frei downloadbare VHDL-Modelle verschiedener Kategorien.

➢ http://www.ti.informatik.uni-frankfurt.de/grimm/hybrid/
- Hier geht es allgemein um Hybrid/Analog/Digital-Systeme
- Insbesondere gibt es einen VHDL-AMS-Parser in Java

➢ http://www.syssim.ecs.soton.ac.uk/
- Southampton VHDL-AMS-Validierungs-Suite.
- Interessant: Online-VHDL-AMS-Scanner, welcher den mit dem Browser upgeloadeten beliebigen Code in Text- oder Datei-Form, scannt und analysiert. Funktioniert allerdings nicht immer.
- Auch hier sind diverse Mixed-Mode-Modelle in VHDL-Sourcecode frei zu bekommen.

➢ http://www.vhdl-ams.com/
- Allgemeine VHDL-AMS-Page mit Links zu Simulator- und Tool-Herstellern.
- Umfangreicher Online-Workshop (VeriasHDL–Simulator): Step-by-Step-Beispiel eines Modell-Design-Prozesses anhand eines Audio-Systems, bestehend aus acht einzelnen Design-Blöcken.

➢ http://www.vhdl-online.de/
- Kein VHDL-AMS ohne VHDL: Diese Website befasst sich zwar nicht mit der AMS-Erweiterung, gehört jedoch zu einer der besten Basis-Seiten, wenn es um VHDL geht.
- Sehr gut, für alle die autodidaktisch VHDL lernen möchten, mit Beispielen und Tutorials.
- Zentrale Austauschplattform für die „VHDL-Community“.

➢ http://www.ee.duke.edu/research/impact/vhdl-ams/index.html/
- Sammlung von vielen kleinen, vor allem physikalischen Modellen.

➢ http://wwwsoft.nf.fh-nuernberg.de/~stlabor/me-seminare/VHDL_AMS/ index.htm/
- Sehr gute Zusammenfassung über den neuen Standard, auch als Word-Dokument erhältlich (Zusammenfassung VHDL-AMS FH-Nürnberg.doc)

Über VHDL-AMS ist bislang nur ein Buch erhältlich: „**The VHDL-Reference**“ [siehe Anhang V.], dessen AMS-Teil eher bescheiden ausgefallen ist. Nichtsdestotrotz ist das Buch sehr gut strukturiert, d.h. jeder Unterpunkt beginnt auf einer neuen Seite. Zusammen mit den Tabellen und prägnant hervorgehobenen Einzelaspekten ist es so eine wertvolle Hilfe für das praktische Arbeiten mit VHDL-AMS.
Das Buch besteht aus vier großen Teilen: Einem VHDL und VHDL-AMS Tutorial, einem interessanten VHDL-Workshop, bei dem es um ein Kamera-Kontrollsystem geht, und der Referenz (nur VHDL).
Das Buch wird mit drei CDs geliefert. Eine enthält eine benutzerfreundliche VHDL-Referenz in HTML inklusive Workshop, wobei die Referenz der AMS-Erweiterung leider fehlt. Die anderen zwei CDs enthalten VHDL-Entwicklungsumgebungen von Mentor Graphics[6] und Xilinx[7].

Zum größten Teil aus den oben vorgestellten Links stammen die nun im Folgenden kurz vorgestellten **VHDL-AMS-Dokumentationen**, welche meist in Form eines Tutorials verfasst wurden. Diese Dateien befinden sich auf der zu dieser Dokumentation gehörigen CD [siehe Anhang III.].

- Zusammenfassung VHDL-AMS FH-Nürnberg.doc
  Sehr gute verständliche Einführung in das Thema VHDL-AMS mit vielen Grafiken. Von einer Einführung über verschiedene Simulatoren bis hin zu den wichtigsten VHDL-AMS-Syntax- und Grammatik-Elementen ist alles vertreten, wobei auch die Ergänzungen und Unterschiede zum VHDL-Standard deutlich werden.

- VHDL-AMS Tutorial.pdf
  Umfangreiches, zum Teil sehr detailliertes Tutorial in Folienform, welches auf einer Design Automation Conference in New Orleans entstanden ist. Den größten Nutzen mit diesem 199-seitigen Dokument werden wohl Personen haben, die sich schon öfters mit Hardwarebeschreibungssprachen bzw. VHDL beschäftigt haben.

[6] Großer Hersteller von Simulations- und Synthesewerkzeugen z.B. für ASICs (Application Specific Integrated Circuits)

[7] Einer der führenden Hersteller von FPGAs (Field Programmable Gate Arrays)

- VHDL-AMS Fraunhofer.ps

  Sehr guter Folienvortrag vom Fraunhofer Institut Integrierte Schaltungen mit vielen anschaulichen Grafiken.

- Erfahrungen mit VHDL-AMS bei der Simulation heterogener Systeme (Fraunhofer).pdf

  Ebenfalls vom Fraunhofer Institut ist diese deutsche Ausarbeitung, bei welcher das Thema gut mit zahlreichen Beispielen umrissen wird.

- Introduction to VHDL-AMS.html

  Diese kurze in HTML verfasste Einführung veranschaulicht die zusätzlichen Möglichkeiten von AMS vor allem mit Code-Beispielen.

- VHDL-AMS-Syntax.htm

  Da VHDL ein sehr hohes Maß an Typkonformität aufweist, kann diese interne mit Links verbundene Syntax-Liste in bestimmten Fällen eine schnelle Hilfe sein.

- Einführung in VHDL-AMS (Folien Brodowski).pdf

  Sehr guter farbig illustrierter Vortrag, welcher dem Interessierten verständlich alle Aspekte von grundsätzlichen VHDL-Konstrukten bis hin zum AMS-Überblick mit den wichtigsten neuen Elementen nahe bringt.

- Einführung in VHDL-AMS (Ausarbeitung Brodowski).pdf

  Das ist die ausgearbeitete Textversion der vorherig erwähnten Dokumentation.

1.2.1 Was gibt es schon an frei erhältlichen AMS-Modellen?

Im Gegensatz zu der großen Anzahl der erhältlichen VHDL-Modelle, die frei im Internet abrufbar sind, fällt die Menge der VHDL-AMS-Modelle doch recht mager aus. An dieser Stelle soll dem Leser ein kleiner Überblick über die zur Zeit downloadbaren AMS-Modelle gegeben werden, um allgemein das technisch Machbare zu vermitteln und vielleicht sogar Anregungen zu schaffen.

Die Modelle reichen von einfachen Elementen, wie analoge Bauteile, bis hin zu komplexen Systemen. Manche der Modelle sind recht gut kommentiert, andere leider unzureichend.

Die hier aufgeführten Modelle liegen direkt als Quellcode im Text- bzw. vhd-Format vor, wobei der Fokus der Auswahl auf den Mixed-Mode-, den physikalischen und analogen Modellen liegt. Die Bezugs-URLs, welche auch schon aufgeführt wurden, sind im Anhang zu finden [I.].

Hier nicht aufgeführt sind die Beispiele, welche in Kapitel 2 beschrieben werden sollen, sowie alle anderen hAMSter-Beispiele. Diese Auflistung befindet sich im Anhang [II.].

Mixed-Mode- und Analogsysteme

- Level3 MOS transistor
- Bipolar transistor with thermal effects
- Voltage controlled oscillator (VCO)
- Phase locked loop (PLL)
- Switch-mode power regulator
- 4-bit flash ADC
- 8-bit DAC
- Analog multiplier
- Binary tree comparator
- Sigma-Delta converter
- Butterworth filter
- Different signal generators
- Switched-capacitor demodulator

Physikalische und andere Syteme

- Pressure sensor
- Cantilever explosive sensor
- Combdrive microresonator
- Micropump
- Microvalve
- Acceleration sensor
- Accelerometer system model
- U-Beam
- Combdrive microresonator
- Lorenz Chaos
- Hodgkin Huxley Neuron
- Carpenter-Grossberg ART2 neuronal network

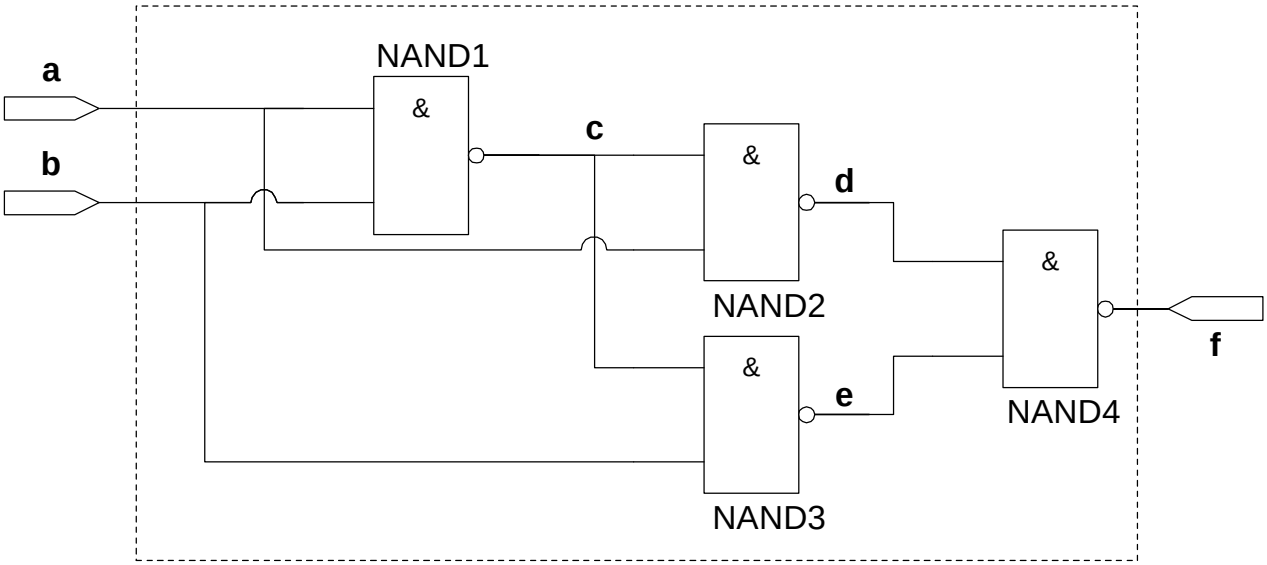
2 Modell- und Simulationsbeispiele

2.1 XOR: Einleitendes VHDL-Beispiel mit Simulation

Vor dem Einstieg in die Welt der analogen Signale, soll anhand eines aus NAND-Gattern aufgebauten Antivalenz-Gliedes kurz auf die wichtigsten Bestandteile der VHDL-Syntax eingegangen und der Aufbau eines VHDL-Modells erklärt werden. Darüber hinaus sollen mit diesem einfachen Beispiel die ersten Gehversuche auf dem VHDL-AMS Simulationsprogramm hAMSter unternommen werden. Die folgende Beschreibung gliedert sich deshalb in zwei Teile – VHDL-Code und Vorstellung der Simulationssoftware.

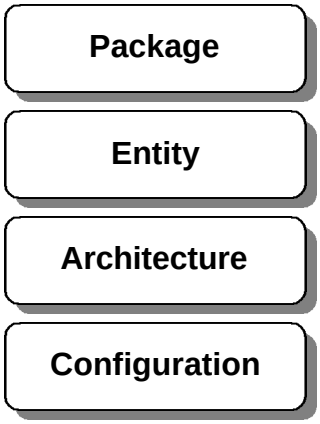
2.1.1 Aufgabenstellung des Beispiels

Wie oben erwähnt, soll aus einzelnen NANDs ein XOR-Glied nachgebildet werden. Es sind **vier NAND-Bausteine mit jeweils zwei Eingängen** zu verwenden. Als zusätzliche Eigenschaft sollen die logischen Gatter eine **optionale Signal-Durchlaufszeit** besitzen.



a	b	c	d	e	f
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

2.1.2 Aufbau eines VHDL-Modells



Bevor wir loslegen, noch ein paar Worte über den Aufbau eines VHDL-Modells.
Ein komplettes VHDL-Modell besteht aus mehreren Teilen (Designeinheiten). Zu den Grundbausteinen einer VHDL-Beschreibung gehören

- die **Schnittstellenbeschreibung – Entity**
  Über die Entity, können einzelne Modelle eines komplexen VHDL-Entwurfs miteinander kommunizieren. Sogenannte `PORTs` stellen die Verbindung 'nach außen' her.
  Zudem hat man in einer Schnittstellenbeschreibung die Möglichkeit, das VHDL-Modell über Parameter, sogenannte `GENERICS`, zu konfigurieren (z.B. Verzögerungs- bzw. Set-Up Zeiten).

- eine oder mehrere **Architekturen – Architecture**
  Die Architektur enthält die Beschreibung der Modelleigenschaften bzw. der Funktionalität eines Modells. Hierfür gibt es verschiedene Möglichkeiten. Das Modell kann über sein Verhalten, seinen Aufbau oder einer Kombination aus beidem beschrieben werden. Es ist durchaus möglich für eine Entity mehrere Architekturen zu definieren.

- eine oder mehrere **Konfigurationen – Configuration**
  In dieser Design-Einheit wird festgelegt, welche Architektur für die Entity des Modells zu verwenden ist. Vor allem bietet sie bei der Beschreibung der Struktur (Aufbau) eines Modells zahlreiche Konfigurationsmöglichkeiten.

Zusätzlich können dem Modell über **Bibliotheken** (**Libraries**) und **Packages**, häufig benötigte Datentypen, Objekte und Funktionen zugänglich gemacht werden. Implizit sind alle Elemente des Packages `std.standard` sichtbar und müssen nicht in die Modellbeschreibung eingebunden werden.

2.1.3 Implementation

Fangen wir bei der Systembeschreibung des Grundbausteins unseres XOR-Modells an, einem einfachen NAND.

2.1.3.1 PORT

Wie in der Aufgabenstellung gefordert, hat unser NAND zwei Eingänge und einen Ausgang. Deshalb müssen in der Entity drei **Schnittstellensignale** festgelegt werden. Jeder dieser sogenannten **Ports** besitzt dabei einen Namen, eine Signalflussrichtung (Modus) und einen bestimmten Datentyp, optional kann der Schnittstelle auch ein Defaultwert übergeben werden.

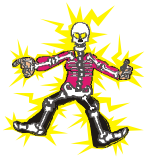
```
PORT ( port_name_1 {, port_name_2} : modus type_name [ := default_value ]
       ...
       { ; port_name_n : modus    type_name [ := default_value ] } );
```

Elemente, die in eckigen Klammern stehen, können entfallen.

modus	Bedeutung
IN	Eingang
OUT	Ausgang
INOUT	bidirektional
BUFFER	Ausgang, kann nur von einer Quelle beschrieben werden

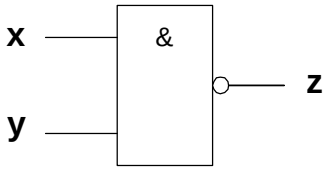
Folgende Datentypen sind im Package `std.standard` vordefiniert. Es können selbstverständlich eigene Datentypen definiert werden (Bsp. mehrwertige Logik).

type_name	Wertebereich
BOOLEAN	true, false
BIT	'0','1'
INTEGER	min. -2147483647 … +2147483647
REAL	min. -1.0E38 … +1.0E38
CHARACTER	VHDL'93 256 Zeichen: 'a', 'b', …
TIME	femtosec … min
SEVERITY_LEVEL	note, warning, error, failure



VHDL unterscheidet nicht zwischen Groß – und Kleinschreibung, es wird aber empfohlen VHDL-Schlüsselwörter groß zu schreiben (Lesbarkeit, etc.)

Zurück zu unserem NAND. Die Portbeschreibung müsste demnach folgendermaßen aussehen.



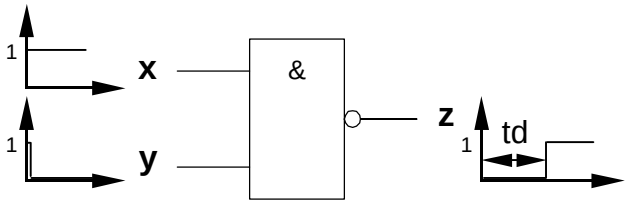

```
PORT ( x,y : IN BIT;
       z   : OUT BIT );
```

2.1.3.2 GENERIC

Zusätzlich wird in der Aufgabe eine Verzögerungszeit für das NAND verlangt. Diese Eigenschaft und sonstige Charakteristika, können dem Modell als Parameter mit der **GENERIC**-Anweisung übergeben werden. Wie Ports haben die Übergabeparameter einen Namen, einen Datentyp und einen optionalen Defaultwert.

```
GENERIC (  param_1 {, param_2} : type_name [ := default_value ]
           ...
           { ; param_n :        type_name [ := default_value ] } );
```

Bei unserem Modell wird kein Wert für den Parameter `td` angegeben. Wie wir noch sehen werden, erfolgt die Zuweisung in der Architekturbeschreibung .




```
GENERIC ( td : time );
```

Neben Ports und Generics können in der Schnittstellenbeschreibung eines VHDL-Modells weitere Deklarationen gemacht werden, die für die Entity und damit auch für alle zugehörigen Architekturen Gültigkeit besitzen.

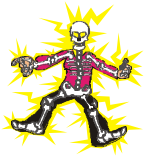
2.1.3.3 ENTITY

Somit haben wir alle Bausteine und wir können uns an die **komplette Schnittstellenbeschreibung** des NANDs wagen. Vorab die allgemeine Syntax für eine Entity.

```
ENTITY entity_name IS
    [ GENERIC...; ]
    [ PORT...; ]
    ...      -- Deklarationen von
    ...      -- Konstanten, Signale, Typen,...
    [ BEGIN
    ...      -- passive Befehle, Assertions
    ... ]
END [ENTITY] [entity_name] ;
```

Ein doppelter Bindestrich leitet ein Kommentar ein, der bis zum Ende der Zeile reicht.

Was jetzt noch fehlt, ist ein geeigneter Name für die Entity. Was bietet sich da besser an als 'nand' - aber Halt, NAND ist ein Schlüsselwort - nennen wir es deshalb besser 'nand_' !



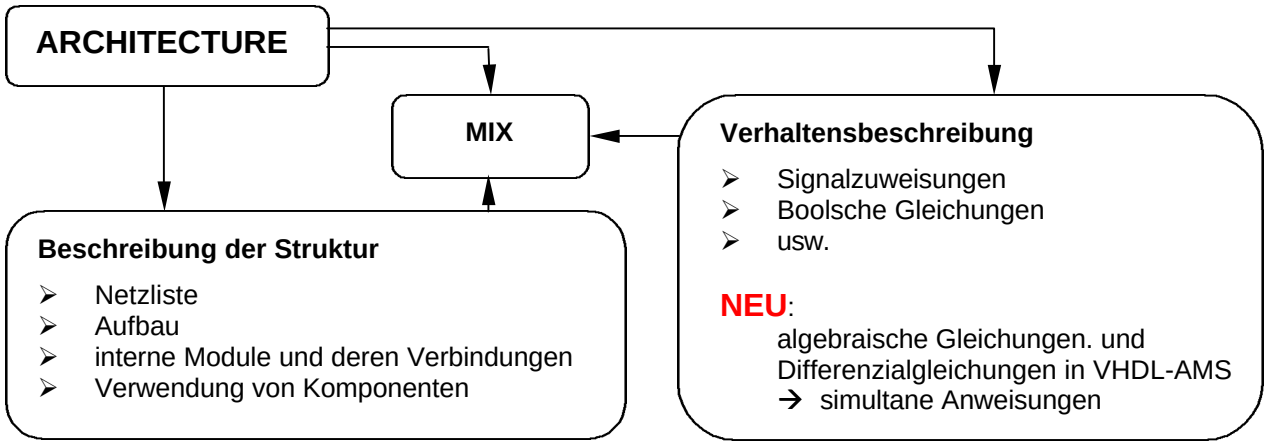
Es ist zu beachten, dass Bezeichner (z.B.Variablennamen, Entity-Name) nur aus Buchstaben, Ziffern und einzelnen Unterstrichen bestehen dürfen. Leerzeichen und Sonderzeichen sind nicht erlaubt. Das erste Zeichen des Bezeichners muss dabei immer ein Buchstabe sein.

Jetzt fehlt noch die Beschreibung, wie es in dem System, d.h. in unserem NAND aussieht. Was geschieht mit den Signalen, die es über seine Ports bekommt und wie sind sie im Inneren verschaltet ? Die Funktionalität eines VHDL-Modells wird in der **Architektur** beschrieben.

```
ENTITY nand_ IS
    GENERIC ( td : time );
    PORT    ( x,y : IN BIT;
              z : OUT BIT );
END ENTITY nand_;
```

2.1.3.4 ARCHITECTURE

Die Architektur, d.h. die Eigenschaften eines Modells, kann auf verschiedene Weisen beschrieben werden. Je nachdem auf welcher Abstraktionsebene man sich befindet und welches Verfahren dafür besser geeignet ist, wählt man entweder die Verhaltens-, die strukturale Beschreibung oder einem Mix aus beidem. Um diesen Sachverhalt auf unser Beispiel zu beziehen, heißt das folgendes:
Wir beschreiben zuerst ein einfaches NAND mit Hilfe seines Verhaltens. Zwei Eingänge werden miteinander 'NAND'-verknüpft und führen zum entsprechenden Ergebnis am Ausgang. Das Antivalenz-Glied, bestehend aus vier NANDs, wird durch seinen Aufbau, d.h. seinen internen Verbindungen, beschrieben. Dies ähnelt dem Anlegen einer Netzliste in PSpice.



Eine Entity kann eine, keine oder mehrere Architekturen besitzen.

Vor der Funktionsbeschreibung eines NANDs - der prinzipielle Aufbau einer Architektur.

```
ARCHITECTURE architecture_name OF entity_name IS
   ...      -- Deklarationsteil
   ...      -- d.h. Konstanten, Signale, Typen,...
   BEGIN
   ...      -- Anweisungen zur
   ...      -- strukturalen- bzw. Verhaltensmodellierung
END [ARCHITECTURE] [architecture_name] ;
```

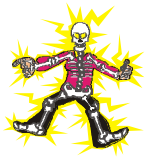
Im Unterschied zur Entity-Beschreibung ist das Schlüsselwort `BEGIN` nach dem Deklarationsteil hier nicht mehr optional.

Die Verhaltensbeschreibung für das NAND besteht aus einer einfachen Signalzuweisung bzw. einer logischen Verknüpfung der beiden Eingangssignale. Sie steht wie oben ersichtlich zwischen `BEGIN` und dem Ende `END` der Architektur. Um die Architekturbeschreibung zu vervollständigen, geben wir ihr den Namen `nand_behav`.

Die Bezeichner x,y und z kommen uns bekannt vor, sie sind die Namen der Schnittstellensignale. Die Verzögerungszeit td zeigt hier sehr schön ihre Wirkung.

```
ARCHITECTURE nand_behav OF nand_ IS
BEGIN
   z <= x NAND y AFTER td;
END ARCHITECTURE nand_behav;
```

Im Allgemeinen werden bei der Verhaltensmodellierung hauptsächlich **Operatoren** (z.B. `NAND`, `+`, `-`, `MOD`, `>=`), **Signalzuweisungen** (z <= ...) und **Anweisungen** (z.B. `WAIT`, `IF-ELSEIF-ELSE`, `CASE`, `LOOP`) eingesetzt.



In der Literatur findet man ab und zu neben den beiden vorgestellten Beschreibungsarten zusätzlich die Datenflussbeschreibung, die hier aber der Verhaltensbeschreibung zugeordnet wird.

2.1.3.5 Signalzuweisungen und Verzögerungsmodelle

Die Grundsyntax für Signalzuweisungen lautet:

```
signal_name <= [ TRANSPORT ] value_expr_1 [ AFTER time_1 ]
          { , value_expr_n: AFTER time_n } );
```

Das Argument `value_expr` kann entweder wieder ein **Signal** oder ein **Ausdruck** (z.B. `x NAND y`) sein. Wichtig ist, dass beide dem Datentyp von `signal_name` entsprechen. Die Signalzuweisung erfolgt dabei stets, wenn das Argument seinen Wert ändert. Optional kann eine **Verzögerungszeit** über das Schlüsselwort AFTER angegeben werden. Es existieren zwei verschiedene **Verzögerungsmodelle** in VHDL, TRANSPORT kennzeichnet das 'Transport-Verzögerungsmodell'. Fehlt dieses Schlüsselwort wird das 'Inertial-Verzögerungsmodell' angewandt. Bei der Erklärung dieser Thematik wird auf die Fachliteratur verwiesen.

2.1.3.6 Logische Operatoren

Logische Operatoren wirken auf einzelne Signale oder Vektoren vom Typ BIT oder BOOLEAN. Das Ergebnis von logischen Operatoren ist entweder gleich '1' bzw. 'true' oder gleich '0' bzw. 'false'. Folgende logische Operatoren sind VHDL bekannt.

AND	**NOT**
NAND	**XOR**
OR	**XNOR**
NOR	

2.1.3.7 Nebenläufige und sequentielle Anweisungen

Innerhalb des Anweisungsteils einer Architektur sind alle Sprachkonstrukte **nebenläufig** (**concurrent**). D.h. anders als in Programmiersprachen, wie z.B. 'C', erfolgt die Abarbeitung der einzelnen Zeilen gleichzeitig oder besser ausgedrückt 'quasi' gleichzeitig, was vom Simulationsalgorithmus des Programms abhängt. Die Reihenfolge der Zeilen ist daher ohne Bedeutung.

Sequentielle Anweisungen werden nacheinander abgearbeitet und dürfen nur innerhalb von Prozessen, Funktionen oder Prozeduren stehen. Die beiden zuletzt genannten Ausdrücke gleichen Unterprogrammen in höheren Programmiersprachen. Im Gegensatz zu nebenläufigen Anweisungen kommt es hier auf die Reihenfolge an.

EXKURS

Kurze Gegenüberstellung:
FUNCTIONS
PROCEDURES
PROCESSES

FUNCTION:

- Übergabeparameter NUR vom Typ IN
- Exakt EIN Rückgabewert

PROCEDURE:

- IN, OUT, INOUT als Übergabeparameter möglich
- Beliebig viele Rückgabewerte
- Eigenständige sequentielle oder nebenläufige Anweisung

PROCESS:

- Umgebung für sequentielle Anweisungen
- Liste sensitiver Signale im Prozesskopf
- WAIT-Anweisungen

2.1.3.7.1 Prozesse

Innerhalb einer Architektur können mehrere Prozesse definiert sein, die untereinander gleichzeitig, d.h. nebenläufig, abgearbeitet werden. Prozesse können durch zwei Möglichkeiten gestartet/aktiviert und gestoppt werden,

- durch **Triggersignale** im Prozess-Kopf

  Prozesse dieser Art werden bei der Modellinitialisierung einmal komplett durchlaufen, danach nur noch, wenn sich eines der Triggersignale ändert.

- durch **WAIT**-Anweisungen

  Derartige Prozesse werden bei der Initialisierung bis zur ersten WAIT-Anweisung abgearbeitet und erst dann wieder gestartet bis die

Bedingung der WAIT-Anweisung erfüllt ist oder die dort angegebene Zeit verstrichen ist.

Obwohl wir in unserem Beispiel keine Prozess-Anweisung benötigen, soll hier dennoch kurz, im Vorgriff auf die anschliessenden Beispiele, die Syntax beider Varianten dargestellt werden.

```
[ process_label : ] PROCESS (signal_1 {, sig_n})
    ...      -- Deklarationsteil
    ...      -- d.h. Konstanten, Signale, Typen,...
    BEGIN
    ...      -- sequentielle Anweisungen ohne WAIT
END PROCESS [ process_label ] ;
```

```
[ process_label : ] PROCESS
    ...      -- Deklarationsteil wie oben
    BEGIN
    ...      -- sequentielle Anweisungen
    WAIT  [  ON signal_name_1 {, signal_name_n} ]
          [  UNTIL condition ]
          [  FOR time_expression ] ;
END PROCESS [ process_label ] ;
```

`WAIT ON`: es wird solange gewartet bis sich mindestens eines der Signale ändert (entspricht sozusagen Variante mit Triggersignal).

`WAIT UNTIL`: unterbricht die Prozessabarbeitung solange bis die Bedingung (`condition`) erfüllt ist

`WAIT FOR`: stoppt den Prozessablauf maximal für diese Zeitdauer.

Es ist durchaus eine Kombination aus den verschiedenen WAIT-Anweisungen möglich.

2.1.3.8 Modellierung des XOR-Gliedes

Fassen wir zusammen: In den vorangegangenen Seiten haben wir gelernt, wie man ein NAND-Glied in VHDL durch seine Schnittstellenbeschreibung und seine Architektur vollständig modellieren kann. Wir kennen jetzt die Unterschiede zwischen nebenläufigen und sequentiellen Anweisungen und können bei Bedarf Prozessbeschreibungen in die Architektur aufnehmen. Was uns im Folgenden erwartet ist, wie man einzelne NANDs zu einem XOR verknüpfen kann. Dazu wird näher auf das Strukturmodell der Architektur eingegangen, genauer gesagt auf die Komponentendeklaration und -instantiierung.

Zur Modellierung des XOR-Gliedes benötigt man wie bei jedem VHDL-System

- eine Schnittstellenbeschreibung
  - nun ohne zusätzliche Schnittstellensignale, diese werden von den einzelnen Komponenten (NANDs) selbst geliefert.
- eine Architekturbeschreibung
  - Signaldefinition
  - Komponentendeklaration
  - Komponenteninstantiierung
  - Takterzeugung (wird zur späteren Simulation benötigt)

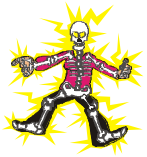
Die Beschreibung der Schnittstelle ist verhältnismäßig einfach, es bedarf lediglich einer Namensgebung (→ exor).

```
ENTITY exor IS
END ENTITY exor;
```

Für die Architektur müssen zuerst die internen und externen Signale des Modells definiert werden. Das sind wie man im Schaltbild des Antivalenzgliedes [siehe 2.1.1] sehen kann die Signale a bis f.

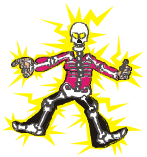
2.1.3.9 Signale

Signale dienen in VHDL dazu, die Eigenschaften elektrischer Systeme nachbilden zu können. Ihnen kann wie bei Variablen jederzeit ein neuer Wert zugewiesen werden. Zusätzlich haben sie jedoch die Eigenschaft, den zeitlichen Verlauf der Wertzuweisungen zu speichern, d.h. man hat die Möglichkeit, auch auf Werte in der Vergangenheit zuzugreifen, und ihnen kann ein Wert in der Zukunft zugewiesen werden (Bsp.: a <= AFTER 2ns, siehe Signalzuweisungen und Verzögerungsmodelle).



Signaldeklarationen dürfen im Deklarationsteil der Entity und der Architecture gemacht werden.

```
SIGNAL signal_name_1 { , signal_name_n } : type_name [ := default_value ] ;
```



Wird kein Default-Wert angegeben, wird der bei der Angabe des Wertebereichs für einen Datentyp (siehe PORT-Beschreibung) erstgenannte Wert zugewiesen. Beispielsweise wäre der Standardwert für den Datentyp BIT '0'.

Da es sich um eine digitale Logikschaltung handelt, haben bei unserem Beispiel alle Signale den Datentyp BIT.

```
SIGNAL a,b,c,d,e,f  : BIT ;
```

2.1.3.10 Komponentendeklaration/instantiierung

Strukturale Modellierung bedeutet im allgemeinen die Verwendung (=Instantiierung) und das Verdrahten von Komponenten in Form einer Netzliste. [2]
Bevor weiter auf das Beispiel eingegangen wird, zuerst ein allgemeiner Teil, um die nachfolgende 'Konstruktion' des XOR-Gliedes verständlicher zu machen. Grundsätzlich geht man bei der strukturalen Modellierung in zwei Schritten vor.

1.Schritt: Komponentendeklaration

Bei der Komponentendeklaration wird der **Prototyp der Komponente** im Deklarationsteil der Architektur eingeführt. Unter dem Prototyp versteht man ein **Abbild der Entity**, d.h. die Schnittstellensignale und die zu übergebenden Parameter (Generics), des einzusetzenden Modells.

```
COMPONENT component_name [IS]
      [ GENERIC (...) ; ]
      [ PORT (...) ; ]
END COMPONENT [ component_name ]
```

Deklarieren wir nun das NAND-Glied als Komponente des XORs. Wie geschildert übernehmen wir dazu die Ports und Generics aus der Entity des NANDs

```
COMPONENT nand_
    GENERIC ( td : time );
    PORT    ( x,y : IN BIT; z : OUT BIT );
END COMPONENT;
```

2.Schritt: Komponenteninstantiierung

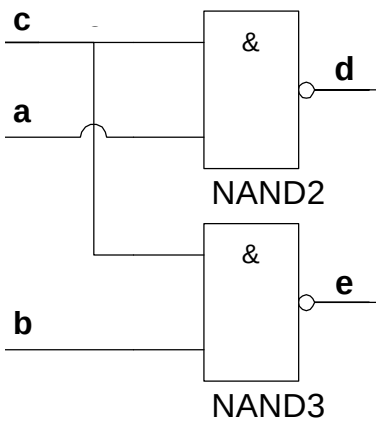
Bei der Komponenteninstantiierung werden einzelne Exemplare der Komponente erzeugt bzw. instantiiert und gleichzeitig miteinander verdrahtet.

```
inst_name : component_name [ GENERIC MAP (...) ] [ PORT MAP(...) ] ;
```

Das Verdrahten der einzelnen Komponenten geschieht mit Hilfe der **PORT MAP**. Dort werden den Schnittstellen (Ports) der Komponente Signale zugewiesen und zwar in der Reihenfolge wie sie in der entsprechenden Entity deklariert wurden. Die Port Map besteht aus diesem Grund lediglich aus einer durch Komma getrennten Liste von Signalnamen. Die **GENERIC MAP** enthält Übergabewerte für die Parameter der Komponente. Dabei werden eventuelle Defaultwerte der Generics überschrieben. Mit Hilfe der Generics können sehr rasch Modelle mit verschiedenen Parameterwerten simuliert werden. Werden die Übergabeparameter nach dem

Kompilieren geändert, ist es nicht nötig den gesamten Quellcode erneut zu übersetzen.

Als Verdrahtungsbeispiel soll folgender Ausschnitt dienen. NAND2 und NAND3 sollen die Namen der NAND-Instanzen sein. Selbstverständlich klein geschrieben, da sie keine VHDL-Schlüsselwörter sind. Es soll keine Verzögerungszeit wirken. Die Modellierung ist nun recht einfach. Betrachten wir die Komponente NAND2. In die Port Map werden die Eingangssignale c und a, und als letztes das Ausgangsignal d eingetragen. Die Generic Map erhält den Wert 0 ns. Somit ergibt sich für die Instanz NAND2 folgende Darstellung (die Reihenfolge c vor a oder c nach a ist bei diesem Beispiel beliebig).

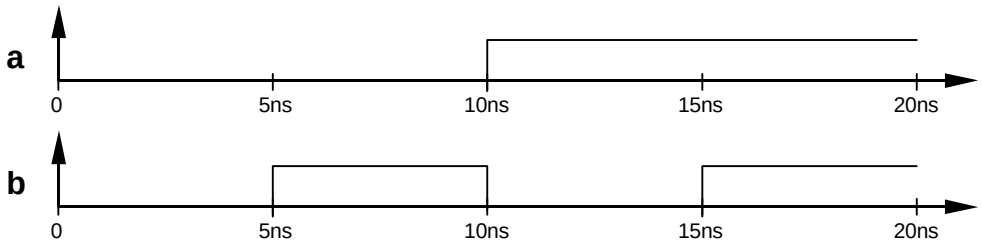



```
nand2 : nand_ GENERIC MAP (0 ns) PORT MAP (c,a,d);
```

In gleicher Weise werden die restlichen NANDs instantiiert (NAND4 weist eine Verzögerungszeit von 2 ns auf).

2.1.3.11 Architektur des XOR-Gliedes

Um die Schaltung testen und das Ergebnis am Simulator betrachten zu können fehlen noch Stimuli an den Eingängen a und b des Antivalenz-Gliedes. Das geschieht durch einfache Signalzuweisungen im Anweisungsteil der Architektur. Der zeitliche Verlauf von a und b soll wie folgt aussehen.



Im VHDL-Code entsprechend

```
a <= '0', '1' AFTER 10 ns;
b <= '0', '1' AFTER 5 ns, '0' AFTER 10 ns, '1' AFTER 15 ns;
```

Noch einmal das Erarbeitete in Kurzform:
Für die vollständige Architekturbeschreibung benötigt man im **Deklarationsteil**

- Signaldefinitionen (vgl. Schaltbild des XORs)
- Komponentendeklarationen (Prototyp des NANDs)

im **Anweisungsteil** werden die Signale den Ein- und Ausgängen der erzeugten Komponenten zugewiesen und miteinander verbunden, zusätzlich werden für die spätere Simulation Stimulus-Signale erzeugt. Das ganze fällt unter die Begriffe

- Signalzuweisungen
- Komponenteninstantiierung

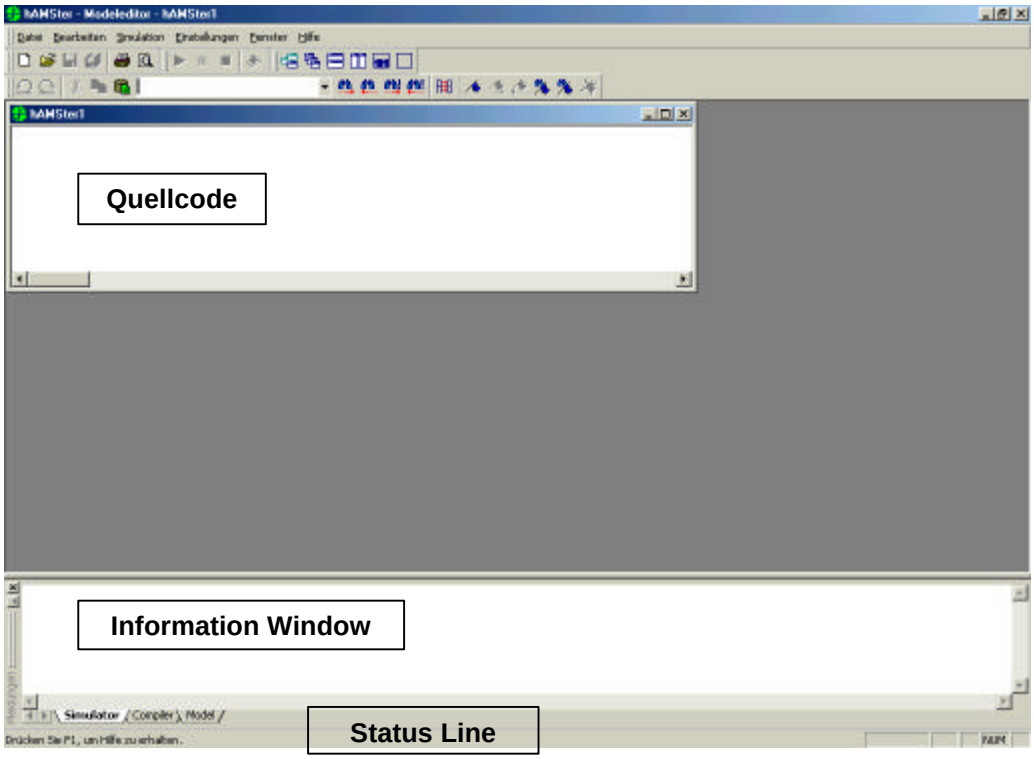
Alle Bestandteile zusammengefasst ergeben die Netzliste (Strukturbeschreibung der Architektur) des XORs mit der Bezeichnung `xor_struct`..

```
ARCHITECTURE exor_struct OF exor IS
   SIGNAL a,b,c,d,e,f    : BIT ;

   COMPONENT nand_
       GENERIC    ( td : time );
       PORT       ( x,y : IN BIT; z : OUT BIT );
   END COMPONENT;

BEGIN

   a <= '0', '1' AFTER 10 ns;
   b <= '0', '1' AFTER 5 ns, '0' AFTER 10 ns, '1' AFTER 15 ns;

nand1 : nand_ PORT MAP (a,b,c);
nand2 : nand_ GENERIC MAP (0 ns) PORT MAP (c,a,d);
nand3 : nand_ GENERIC MAP (0 ns) PORT MAP (b,c,e);
nand4 : nand_ GENERIC MAP (2 ns) PORT MAP (d,e,f);

END;
```

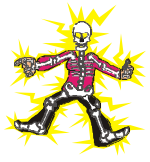
2.1.4 Simulation

Im Folgenden soll am Beispiel des Antivalenz-Gliedes eine kurze Einführung in das Arbeiten mit hAMSter gegeben werden. Für eine detailliertere Beschreibung, speziell die von verschiedenen Fehlermeldungen, kann die PDF-Dokumentation zu Rate gezogen werden. Diese ist im hAMSter-Verzeichnis, im Unterordner \Help\hamster.pdf zu finden.

Nach dem Starten des Programms sieht man den Bildschirm des **hAMSter-Model-Editors**. Dieser besitzt alle Fähigkeiten eines herkömmlichen Text-Editors. Zusätzlich werden VHDL-AMS Schlüsselwörter erkannt und farblich hervorgehoben. Dies geschieht jedoch erst nach dem Abspeichern des Quellcodes. Die Farben für die einzelnen VHDL-Elemente können nicht verändert werden.

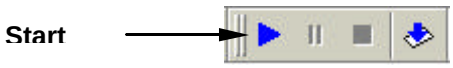


Der erarbeitete VHDL-Code kann direkt in das Quellcode-Fenster eingegeben werden oder kann in etwas veränderter Form geöffnet werden. Die entsprechende Datei ist ...\hAMSter\Examples\digital\exor_struct.vhd.

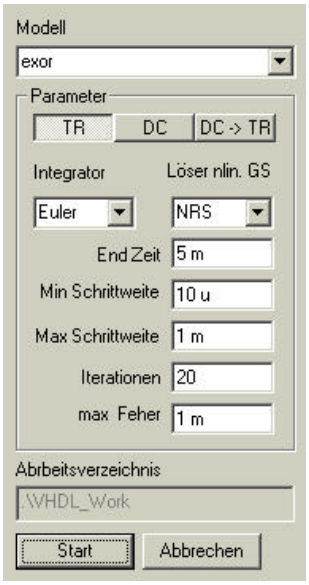


hAMSter speichert VHDL-Modelle mit der Dateiendung *.vhd ab und kann nur Dateien mit dieser Dateikennzeichnung kompilieren.

Nachdem der Quellcode des VHDL-AMS Modells vollständig editiert worden ist, kann er kompiliert, d.h. in eine für den Simulator lesbare Form gebracht werden. Der Simulator wird nach dem Kompilieren automatisch gestartet, wenn auf den Start-Button in der Toolbar gedrückt wird (alternativ: F12 oder über das Menü).

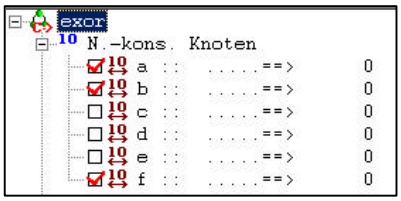


Eventuelle Fehler oder Warnungen des Compilers und des Simulators erscheinen im **Information Window**. Bei Doppelklick auf die Fehlermeldung springt der Cursor zur entsprechenden Stelle im Code, an dem der Fehler aufgetreten ist.



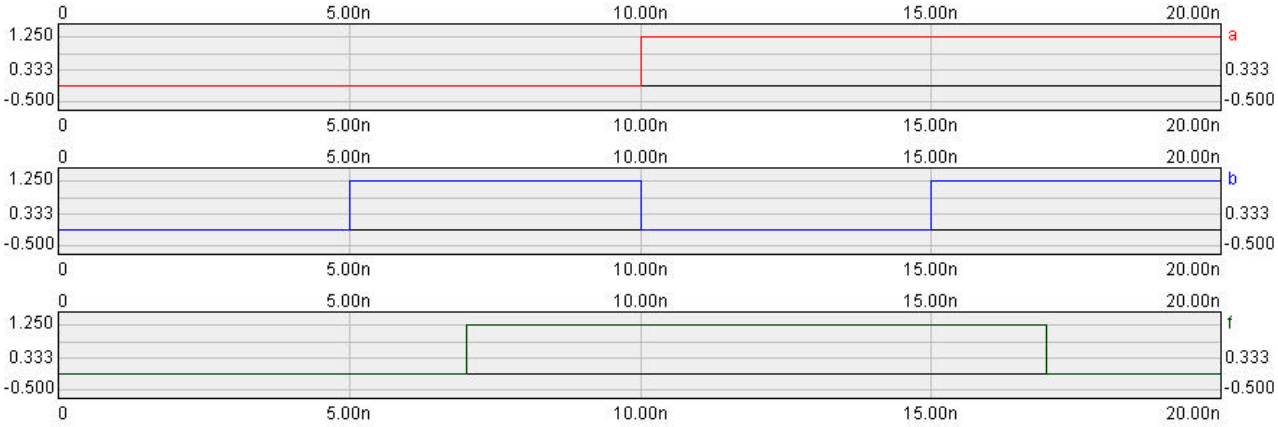
Hat das Kompilieren fehlerfrei funktioniert – eine häufige Fehlerursache ist das Vergessen von Strichpunkten – erscheint das Simulationsfenster. In diesem Dialogfenster können die Simulationsparameter und die Signale, die man beobachten möchte, festgelegt werden. Beim ersten Simulieren sind Default-Werte eingetragen, die jedoch an unser Modell angepasst werden müssen. Die Einstellungen für den Simulator (TR → Simulation im Zeitbereich), die Integrationsmethode (Euler) und die Lösungsmethode für nichtlineare Gleichungssysteme (NRS) können übernommen werden. Unsere Simulation soll höchstens 20ns Sekunden dauern, da unsere Eingangssignale auch nicht länger sind. Die minimale Schrittweite legen wir auf 1ns fest, die maximale auf 10ns. Somit sind alle Einstellungen für die Simulation getroffen und es müssen lediglich noch Signale selektiert werden, die man beobachten möchte. Auf der rechten Seite des Dialogfensters kann die Auswahl getroffen werden.

Unter '10' kann man sämtliche digitalen Signale des Systems finden. Wir wählen die Eingangssignale a bzw. b und das Ausgangssignal f aus.



hAMSter speichert die Simulatoreinstellungen und die darzustellenden Signale eines VHDL-AMS-Modells in Configurations-Dateien (*.cfg) ab.

Die Simulation kann nun gestartet werden. Nach dem Klick auf 'Start' öffnet sich das hAMSter View Tool.



Hier sieht man sehr schön den Einfluss der Verzögerungszeit von 2ns. Erst zum Zeitpunkt 7ns und 17ns reagiert der Ausgang auf dem Wechsel an den Eingängen des XORs.

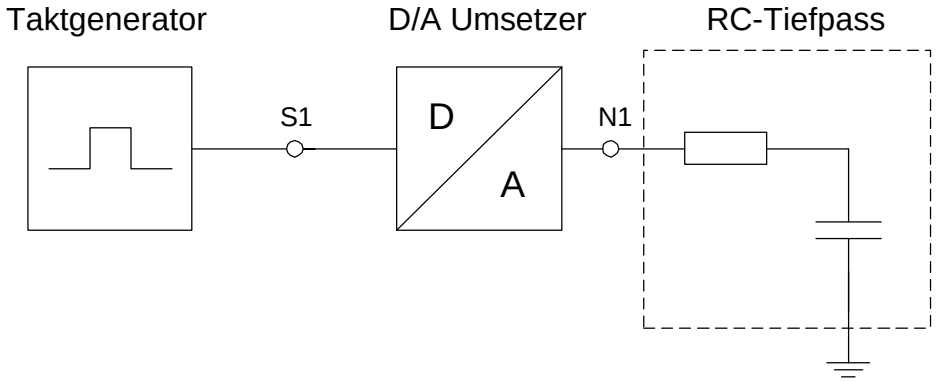
Wie man sieht, ist die Bedienung von hAMSter sehr einfach und fast selbsterklärend - eine größe Stärke von hAMSter.

2.2 Einfaches Mixed-Mode System

Mit diesem Beispiel sollen die wichtigsten Elemente der VHDL Erweiterung AMS anhand eines einfachen Mixed-Mode Systems erklärt werden. Danach sollte man in der Lage sein, die darauffolgenden Beispiele besser zu verstehen. Darüber hinaus soll das nötige Rüstzeug vermittelt werden, um eigene einfache Modelle zu generieren.

Vor der Implementation erfolgt unter [2.2.2] eine Beschreibung der Objekte und Sprachkonstrukte, die mit VHDL-AMS neu hinzugekommen sind. Diese Beschreibung soll gleichzeitig als Referenz aller VHDL-AMS Beispiele dienen.

2.2.1 Aufgabenstellung

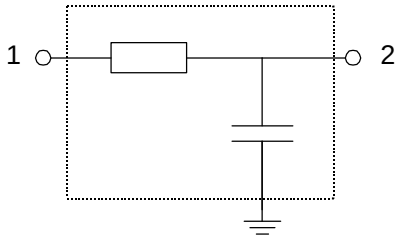


Bei diesem Beispiel soll der digitale Bitstrom vom Taktgenerator im D/A-Umsetzer in eine analoge Spannung umgesetzt werden. Die umgesetzte Spannung soll am Eingang eines RC-Glieds (Tiefpass) angelegt werden. Der **D/A-Umsetzer** soll folgende Spezifikation haben

Bitwert am Eingang	Spannung am Ausgang
0	-2.0 V
1	+2.0V

Für die Bauteile des **RC-Gliedes** werden die Werte: R = 1 k und C = 1µF gefordert. Auf das zu erzeugende Bitmuster des **Taktgenerators** wird später eingegangen.

2.2.2 Einführung in VHDL-AMS am Beispiel eines RC-Gliedes



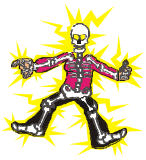
Betrachten wir dieses RC-Glied und halten unser bisheriges Wissen über digitale Modelle im Hinterkopf. Was muss VHDL hinzugefügt werden, um ein Modell eines Tiefpasses zu beschreiben ? Dieser Frage soll nun nachgegangen und gleichzeitig die wichtigsten hinzugekommenen Begriffe erwähnt werden. Das RC-Glied sei zuerst eine Black-Box. Von aussen sieht man lediglich die Anschlüsse 1,2 und Masse. Da an diesen Punkten wertkontinuierliche Signale anliegen können, bezeichnet man sie als **`TERMINALS`**, um sie von digitalen Schnittstellensignalen zu unterscheiden. Da es mit VHDL-AMS möglich ist, verschiedene physikalische Systeme zu beschreiben, muss den `TERMINALS` zusätzlich die Eigenschaft (**`NATURE`**) **`ELECTRICAL`** zugewiesen werden, um auf das elektrische Verhalten im Innern hinzuweisen. An den Terminals liegen keine digitalen Signale an, vielmehr besteht in diesem Beispiel die Möglichkeit, das Verhalten an den Anschlüssen mit Hilfe von physikalische Größen (Spannung und Strom) zu beschreiben. Physikalische Größen werden als **`QUANTITIES`** deklariert und behandelt. Um das Verhalten zu modellieren, fließen diese `QUANTITIES` in mathematische Gleichungen (algebraische- aber auch Differenzialgleichungen) ein. Simultane Anweisungen (**`SIMULTANEOUS STATEMENTS`**) drücken Beziehungen zwischen Quantities aus, d.h. sie sind mathematische Gleichungen.

In dieser kurzen Erklärung ist auf die wichtigsten VHDL-AMS Schlüsselwörter hingewiesen worden. Wie sie im Detail verwendet werden, soll im folgenden näher beleuchtet werden, bevor wir uns schließlich an die Modellierung des gesamten Systems wagen.

2.2.2.1 TERMINALS

Terminals definieren Anschlüsse. Präziser ausgedrückt sind Terminals festgelegte Punkte in der Struktur eines physikalischen Modells. Stets damit verbunden ist die Angabe der physikalischen Domäne, d.h. mit welchem physikalischen System man

es zu tun hat. Aus diesem Grund werden Terminals nie ohne ihre **Natures**, ihren speziellen Eigenschaften genannt. Aus physikalischer Sicht besitzt jeder dieser Punkte ein gewisses Potenzial (siehe RC-Glied). Wie damit umgegangen werden kann, wird wiederum durch die dem Anschluss zugewiesene Nature festgelegt.



Terminals können sowohl in der Schnittstellenbeschreibung der Entity, als auch im Deklarationsteil der Architektur definiert werden (interne Knoten).

Als Beispiel soll der Widerstand des RC-Gliedes dienen. Die Portbeschreibung würde folgendermaßen aussehen.


```
PORT ( TERMINAL t1, t2  : ELECTRICAL );
```

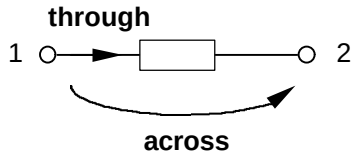
Aufgepasst ! Die Bezeichnung der Terminals mit 1 bzw. 2 ist unzulässig. Wie schon darauf hingewiesen, müssen Bezeichner stets mit einem Buchstaben beginnen. An der Syntax der Portbeschreibung ändert sich dabei nichts, deshalb sei noch einmal darauf betont, dass es sich bei VHDL-AMS lediglich um eine Erweiterung des bisherigen VHDL-Standards handelt.

2.2.2.2 NATURES

Natures definieren die Attribute bzw. die physikalischen Eigenschaften von Terminals. Sie geben an, in welcher Form das Potenzial am Verbindungspunkt (Terminal) behandelt werden kann. Treten zum Beispiel Potenzialdifferenzen zwischen zwei Punkten auf, kann die Beziehung zwischen beiden auf zwei Arten beschrieben werden

- durch den **Fluss** zwischen beiden Punkten, erzeugt durch die Potenzialdifferenz.
- durch die **Potenzialdifferenz** selbst.

Um diesen Sachverhalt in VHDL-AMS ausdrücken zu können, wurden zwei neue Begriffe eingeführt: **ACROSS** und **THROUGH**. 'Across' gibt dabei die Potenzialdifferenz, 'through' den Fluss zwischen zwei Punkten an. Zur Veranschaulichung soll ein Widerstand dienen.



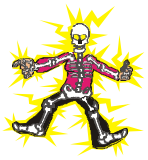
Es ist ersichtlich, dass in diesem Beispiel mit 'through' der Strom, der durch den Widerstand fließt, und mit 'across' die Spannung, die am Widerstand anliegt, gemeint ist. In elektrischen Systemen drückt 'through' die Spannung und 'across' den Strom aus, immer bezogen auf zwei Punkte. Hat man daher zwei Terminals mit der Eigenschaft (Nature) **ELECTRICAL** definiert, können die elektrischen Größen am Widerstand folgendermaßen beschrieben werden (Branch Quantity Beschreibung).

```
...
PORT ( TERMINAL t1, t2  : ELECTRICAL );
...
QUANTITY r_voltage ACROSS resistor_i THROUGH t1 TO t2;
...
```

Das Schlüsselwort **QUANTITY** definiert eine physikalische Größe (Spannung, Strom) und wird später detailliert beschrieben. Mit dem Syntaxelement **TO** wird die Potenzialdifferenz zwischen den Punkten t1 und t2 ausgedrückt. Fehlt dieses Element ist die Masse Referenzpotenzial. Es ist keine Pflicht, Spannung und Strom stets zusammen zu deklarieren. Je nach Bedarf kann auch nur eine der beiden Größen in Beziehung mit Terminals gebracht werden. In den anschließenden Beispielen wird der Gebrauch dieser Sprachelemente verständlicher.

Natürlich gibt es neben elektrischen auch weitere physikalische Systeme, in denen die Schlüsselwörter 'across' und 'through' mit anderen physikalischen Größen in Verbindung gebracht werden.

NATURE	ACROSS	THROUGH
ELECTRICAL	voltage	current
THERMAL	temperature	heat flow rate
MECHANICAL	force	velocity



Natures werden in **PACKAGES** beschrieben. Dort sind für jede Nature die entsprechenden **ACROSS**- und **THROUGH**-Typen und das Bezugspotenzial **REFERENCE** festgelegt. Vor Verwendung von Natures müssen deshalb die Packages der entsprechenden **LIBRARY** geladen werden.

Folgender Abschnitt zeigt wie Libraries bzw. Packages in den Quellcode eingebunden werden können.

2.2.2.3 Bibliotheken und Packages

Bibliotheken dienen als Aufbewahrungsort für kompilierte und häufig benötigte Datentypen, Objekte, Funktionen und seit VHDL-AMS auch Natures. In VHDL-AMS gibt es zahlreiche Bibliotheken, wie z.B. für mechanische, elektrische, thermische oder gar chemische Systeme.

Möchte man bestimmte Elemente einer Bibliothek nutzen, muss zuerst die verwendete Bibliothek bekannt gemacht werden. Das geschieht durch die **LIBRARY**-Anweisung.

```
LIBRARY library_name_1 { , library_name_n } ;
```

Mit Hilfe der **USE**-Anweisung werden Bibliotheksobjekte schließlich im Quellcode sichtbar gemacht bzw. eingebunden.

```
USE library_name.ALL ;
USE library_name.element_name ;
USE library_name.package_name.ALL ;
USE library_name.package_name.element_name ;
```

Sind die Objekte, die man verwenden möchte, in einem Package definiert, muss der Bibliothek zusätzlich der Name des entsprechenden Packages nachgestellt werden. **ALL** wählt den Inhalt der gesamten Bibliothek bzw. des Packages aus.

In hAMSter sind folgende Pakete (Packages) in übergeordneten Bibliotheken zusammengefasst.

DISCIPLINES	IEEE	STD
ELECTROMAGNETIC_SYSTEM	MATH_REAL	STANDARD
FLUIDIC_SYSTEM	STD_LOGIC_1164	
KINEMATIC_SYSTEM		
PHYSICAL_CONSTANTS		
ROTATIONAL_SYSTEM		
THERMAL_SYSTEM		

Standardmäßig stehen einer Modellbeschreibung die Elemente des Packages **STD.STANDARD** zur Verfügung und müssen nicht explizit genannt werden.

2.2.2.4 QUANTITIES

Sucht man im Wörterbuch den Begriff 'quantity', so findet man zwei Übersetzungen. Die erste ist trivial, nämlich die Menge bzw. Anzahl, die zweite Übersetzung führt uns näher auf die Bedeutung in VHDL-AMS hin, die Größe. D.h. in VHDL-AMS sind **physikalische Größen** gemeint, wenn von 'quantities' die Rede ist. Mit quantities können somit Größen wie Spannung, Strom oder Temperatur definiert werden.

Allgemein lassen sich die **grundlegenden Eigenschaften** wie folgt zusammenfassen. Quantities

- sind vom Datentyp **REAL** (Gleitkommazahlen),
- sind die Unbekannten einer Differenzialgleichung oder algebraischen Gleichung,

- repräsentieren zeitlich veränderbare Werte und können in Form eines Graphen dargestellt werden,
- können überall dort eingesetzt werden, wo Signale vereinbart werden können.

Es gibt verschiedene Typen von Quantities:

- NORMALE / FREE QUANTITIES
- BRANCH QUANTITIES
- IMPLICIT QUANTITIES
- INTERFACE QUANTITIES

Quantities beschreiben nicht nur **physikalische Eigenschaften** (normale/free quantity), wie z.B. die Kapazität eines Kondensators, vielmehr besteht die Möglichkeit in Verbindung der schon erwähnten `THROUGH`- UND `ACROSS`-Komponenten, **physikalische Zusammenhänge** (branch quantity) zwischen zwei `TERMINALS` auszudrücken. Zur **Beschreibung von mathematischen Operationen** an Quantities (wie z.B. die Spannung abgeleitet nach der Zeit) bedient man sich implicit quantities. Da Quantities **Signalcharakter** haben, können sie zur Verbindung zweier Entities neben Terminals und Signalen in einer Port-Beschreibung stehen (interface quantity).

2.2.2.4.1 Normale/Free Quantities

Durch Free bzw. Normale Quantities werden lediglich abstrakte Größen definiert, die z.B. als Hilfsgrößen in der Architektur verwendet werden können. Als Beispiel sei die Kapazität eines Kondensators genannt. Wird diese Größe nicht in einem physikalischen Zusammenhang definiert, drückt sie lediglich eine physikalische Eigenschaft aus.

Hier die Syntax für die Definition freier Quantities.

```
QUANTITY quantity_name_1 { , q_name_n } : type_name [:= default_value ] ;
```

Der Typ von Quantities ist stets `REAL`. Durch die Angabe von `type_name` besteht die Möglichkeit, den Datentyp der Quantity weiter zu spezifizieren. In Packages sind

für das jeweilige physikalische System Sub-Typen definiert. Für elektronische Systeme sind z.B. folgende Typen deklariert.

VOLTAGE	**CURRENT**
RESISTANCE	**CAPACITANCE**
CHARGE	**INDUCTANCE**
FLUX	**MMF**

Als Beispiel werden als Hilfsgrößen die elektrische Ladung Q und die Kapazität eines Kondensators aufgeführt.

```
QUANTITY   q_help : CHARGE
QUANTITY   c_help : CAPACITANCE
```

2.2.2.4.2 Branch Quantities

Branch Quantities eignen sich dafür, physikalische Zusammenhänge zwischen Terminals zu beschreiben. Aus diesem Grund, werden sie mit den Nature-Eigenschaften ACROSS und THROUGH des entsprechenden physikalischen Systems definiert. Mit ihnen ist es möglich die komplette Netzliste einer elektronischen Schaltung zu entwerfen (daher auch die Bezeichnung 'Branch', d.h. Teil bzw. Zweig eines aus Knoten - Terminals - aufgebauten Netz).

Branch Quantities werden wie folgt definiert.

```
QUANTITY    [ across_quantity_1 { , across_quantity_n } ACROSS ]
            [ through_quantity_1 { , through_quantity_n } THROUGH ]
              plus_terminal_name [ TO minus_terminal_name ] ;
```

Gegebenenfalls kann den einzelnen Quantities mit einer einfachen Wertzuweisung (`:= Wert`) ein Default-Wert gegeben werden. Mit dem Terminal-Aspekt **TO** kann die Polarität einer Potenzialdifferenz bestimmt werden, d.h. dadurch kann die Richtung eines Spannungspfeils vorgegeben werden. Fehlt der Aspekt **TO**, bezieht man sich

auf den Referenzpunkt bzw. auf das Referenzpotenzial des jeweiligen physikalischen Systems (definiert in der Naturebeschreibung unter **REFERENCE**).

Zahlreiche Anwendungsbeispiele sind in den weiteren Beispielen zu finden.

2.2.2.4.3 Interface Quantities

Interface Quantities haben die Modi `IN` bzw. `OUT` und können durch Definition in der Portbeschreibung als Verbindung zwischen zwei Entities dienen.

```
PORT ( QUANTITY name_1 {,name_n} : modus  type_name [ := default_value ] );
```

2.2.2.4.4 Implicit Quantities

Implizite Quantities dienen zur Beschreibung mathematischer Operationen und werden auch als Quantity-Attribute bezeichnet.

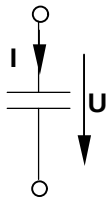
Attribut	Beschreibung	Ergebnistyp
Q'DOT	1.Ableitung von Q nach der Zeit	Wie Q
Q'INTEG	Integral von Q über der Zeit von O bis zum aktuellen Zeitpunkt	Wie Q
Q'SLEW [(MAX_RISING_SLOPE [,MAX_FALLING_SLOPE])**]**	Maximale Slew Rate einer Quantity	Basistyp von Q
Q'DELAYED (T**)**	Verzögerung von Q um die Zeit T	Wie Q
Q'LTF	Laplace-Transformation von Q	Basistyp von Q
Q'ZTF (NUM,DEN,T,[,INIT_DELAY]**)**	'Z-Domain Transfer Function'	Wie Q
Q'ZOH (T [, INIT_DELAY]**)**	Q abgetastet	Basistyp von Q
Q'ABOVE (E**)**	Ist Q unter/oberhalb eines Schwell-werts ?	BOOLEAN
Q'TOLERANCE	Toleranzwert einer Quantity	Zeichenkette

Erklärung der Parameter:

`MAX_RISING_SLOPE`
`MAX_FALLING_SLOPE`
} Folgt Q, aber die Ableitung nach der Zeit ist begrenzt durch diese Flanken

T	Einfache Zeitgröße (Typ Real)
INIT_DELAY	Erste Messung nach INIT_DELAY (Typ Real)
E	Schwellwert (Typ der Quantity)

Auf die Erklärung der anderen Parameter soll hier nicht weiter eingegangen werden. Für das Verständnis der folgenden Beispiele sind die oben genannten Parameter völlig ausreichend. Die Beschreibung des Stromes durch einen Kondensator sieht demnach folgendermaßen aus:




```
I == C * U'DOT
```

Das doppelte Gleichheitszeichen symbolisiert eine algebraische Gleichung bzw. eine simultane Anweisung.

2.2.2.5 SIMULTANEOUS STATEMENTS

Simultane Anweisungen drücken auf mathematische Weise Beziehungen zwischen physikalischen Größen bzw. Quantities aus. Dabei handelt es sich in erster Linie um algebraische und Differenzialgleichungen. Kennzeichnend für diese mathematischen Gleichungen ist das doppelte Gleichheitszeichen. Bei der Aufstellung simultaner Anweisungen werden neben Quantities hauptsächlich Signale und Konstanten verwendet. Speziell bei Differenzialgleichungen kommen nicht selten Quantity-Attribute zum Einsatz (z.B. `Q'DOT`, erste Ableitung).

Hier die Syntax zu einer einfachen simultanen Anweisung

```
expression_1 == expression_2 ;
```

EXKURS

Einfaches Modell einer DIODE unter Verwendung des SIMULTANEOUS STATEMENTS

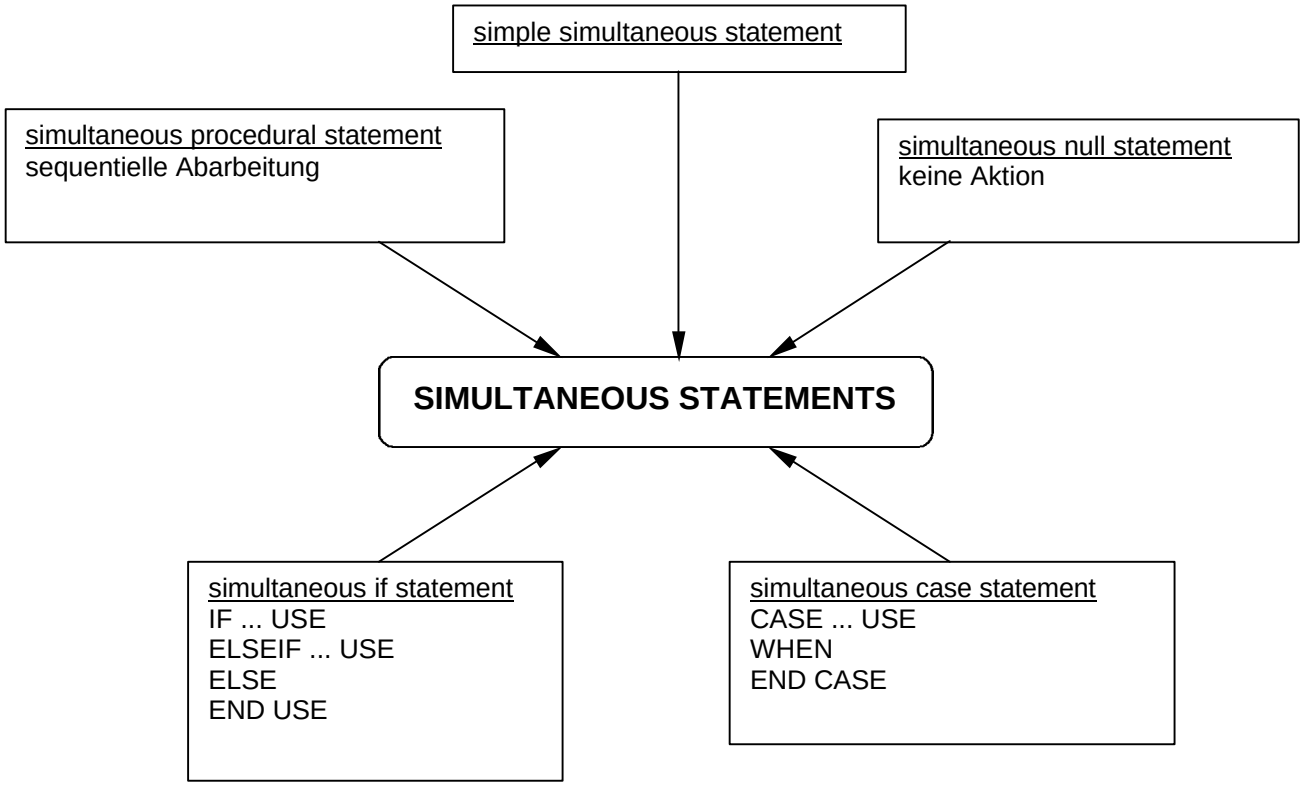
Um die Vielfältigen Möglichkeiten der nebenläufigen Anweisungen in Form von Gleichungen zu demonstrieren, soll hier ein beliebtes Beispiel, die Modellgleichung einer **einfachen abstrakten Diode**, herangezogen werden:

Id == Is*(exp((Vd-Rs*Id)/Vt)-1.0);

wobei:

Id :	Strom durch die Diode (Quantity)
Is :	Dioden-Sperrstrom (Real-Constant)
Vd :	Äußere DiodenSpannung (Quantity)
Rs :	Innerer Serienwiderstand der Diode (Real-Constant)
Vt :	Temperatur-Spannung (Real-Constant)

Man unterscheidet folgende simultane Anweisungen:



2.2.2.5.1 SIMULTANEOUS IF-STATEMENT

Hier erfolgt zuerst eine Auswertung von verschiedenen Bedingungen. Nach der ersten Bedingung nach IF bzw. ELSEIF, die den Ergebniswert TRUE liefert, wird die simultane Anweisung ausgewertet und die weiteren ELSE-IF-Zweige nicht mehr berücksichtigt. Sind alle Bedingungen falsch wird die Anweisung im ELSE-Zweig ausgeführt.

```
IF condition USE
      simultaneous_statement_part ;
{ ELSEIF condition USE
      simultaneous_statement_part ; }
[ ELSE
      simultaneous_statement_part ; ]
END USE;
```

2.2.2.5.2 SIMULTANEOUS CASE-STATEMENT

Damit eine simultane Anweisung ausgewertet werden kann, muss der Wert des Ausdrucks (expression) mit einem Wert (choice) des entsprechenden WHEN-Zweiges übereinstimmen.

```
CASE expression USE
      WHEN choices =>
            simultaneous_statement_part ;
      [ WHEN choices =>
            simultaneous_statement_part ; ]
      [ WHEN =>
            simultaneous_statement_part ; ]
END CASE;
```

2.2.2.5.3 SIMULTANEOUS PROCEDURAL STATEMENT

Dieser Typ Simultaner Anweisungen erlaubt eine sequentielle Beschreibung algebraischer- und Differenzialgleichungen. Er eignet sich besonders für Schleifen und die Berechnung rekursiver Funktionen.

```
PROCEDURAL [ IS ]
      { procedural_declarative_part ; }
BEGIN
      { sequential statement part ; }
END PROCEDURAL ;
```

2.2.2.5.4 SIMULTANEOUS NULL STATEMENT

Die 'Simultane-Null-Anweisung' führt keine Aktion aus. Sie ist kennzeichnet für aktionslose Fälle in IF- und CASE- Anweisungen

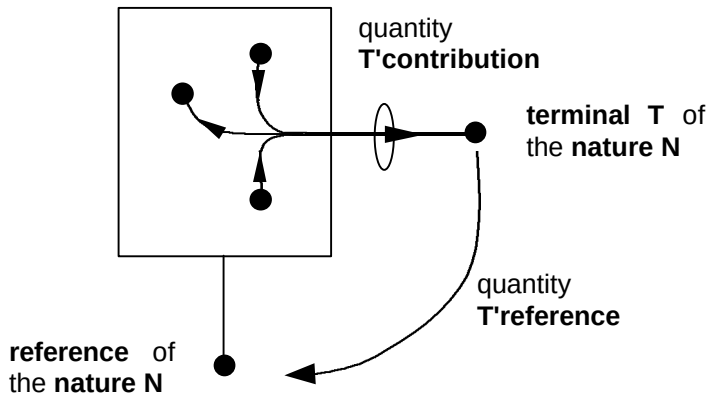
```
NULL;
```

2.2.2.6 Terminal- und Signal Attribute

In VHDL-AMS sind nützliche Signalattribute hinzugefügt worden und es wurden Attribute für Terminals eingeführt. Mit Hilfe dieser Attribute kann zum Beispiel die ACROSS- und THROUGH Quantity eines Terminals bestimmt werden.

Attribut	Beschreibung	Ergebnistyp
S'RAMP [(TRISE [,TFALL])]	Quantity t, die einem Signal S folgt. Dabei wird die Anstiegs- und die Abfallverzögerung berücksichtigt	Basistyp von S
S'SLEW [(RISING_SLOPE [, FALLING_SLOPE])]	Quantity, die einem Signal S folgt und durch die angegebenen Steigungen begrenzt ist.	Basistyp von S
T'REFERENCE	Across-Quantity vom angegebenen Terminal zum Referenzpunkt.	Across-Typ der Nature von T (z.B. VOLTAGE)
T'CONTRIBUTION	Through-Quantity, entspricht der Summe aller Teilströme in diesem Punkt.	Through-Typ der Nature von T (z.B. CURRENT)

Beispiel:



2.2.2.7 Toleranzen

Toleranzen sind notwendig, da ein numerischer Algorithmus zur Lösung von Gleichungen nur Näherungen der exakten Lösung ermitteln kann. Deshalb werden Toleranzen angeben, die eine Aussage darüber geben, wie gut man sich an die Lösung einer Gleichung annähern soll. Jedes Quantity und jede simultane Anweisung gehört zu einer Toleranzgruppe, welche durch einen Stringausdruck angezeigt wird. Weiter soll auf diese Thematik nicht eingegangen werden.

```
QUANTITY vol : real TOLERANCE “voltage”;
```

2.2.2.8 Architektur- bzw. Schnittstellenbeschreibung unter VHDL-AMS

Da VHDL-AMS nur eine Erweiterung des bisherigen VHDL-Standards darstellt, bleibt der Aufbau bzw. die Syntax der Entity und der Architecture erhalten. Beide Beschreibungen werden lediglich ergänzt.

a) Auswirkungen auf die ENTITY

Im **Port-Deklarationsteil** hinzugefügt werden können

- Terminal-Deklaration mit Zuweisung einer Nature

```
PORT ( TERMINAL n1, n2  : ELECTRICAL );
```

- Interface-Quantity-Deklaration

```
PORT ( QUANTITY A, B : IN REAL );
```

b) Auswirkungen auf die ARCHITECTURE

Im **Deklarationsteil** hinzugefügt werden können

- Free-Quantity-Deklaration

```
QUANTITY q : CHARGE ;
```

- Branch-Quantity-Deklaration

```
QUANTITY v ACROSS i THROUGH p TO m ;
```

- Terminal-Deklaration

```
TERMINAL t1, t2  : ELECTRICAL );
```

Im **Anweisungsteil** hinzugefügt werden können

- alle Arten simultaner Anweisungen

```
U == R*I;
```

2.2.3 Implementation

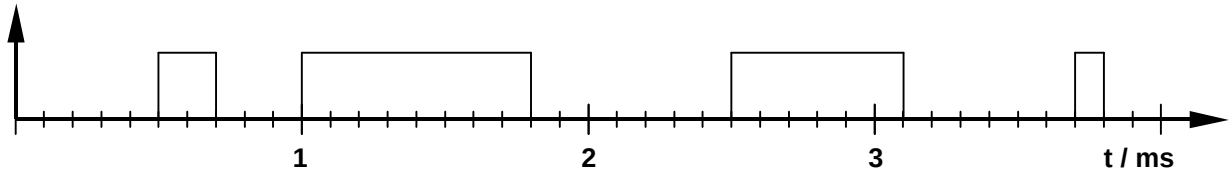
Die Reihenfolge bei der Beschreibung soll sein
Digitaler Taktgenerator - Digital /Analog-Umsetzer - analoges RC-Glied.

2.2.3.1 Digitaler Taktgenerator

Da der Taktgenerator nur einen Ausgang vom Typ BIT besitzt, ist seine Schnittstellenbeschreibung relativ einfach.

```
ENTITY flipflop IS
       PORT ( output : OUT BIT );
END;
```

Das an den D/A-Umsetzer angelegte Taktsignal soll folgenden Verlauf haben.



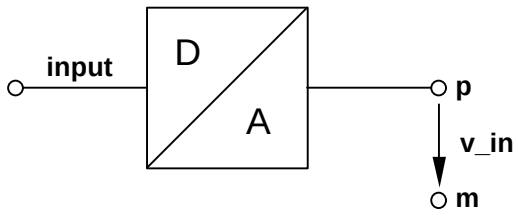
Modelliert wird dieser Signalverlauf durch einfache Signalzuweisungen im Anweisungsteil der Architektur. Anschließend wird es dem Ausgang des Taktgenerators zugewiesen.

```
ARCHITECTURE behav OF flipflop IS
        SIGNAL a : BIT ;
BEGIN
        a <=  '1' AFTER 0.5 ms,
              '0' AFTER 0.7 ms,
              '1' AFTER 1.0 ms,
              '0' AFTER 1.8 ms,
              '1' AFTER 2.5 ms,
              '0' AFTER 3.5 ms,
              '1' AFTER 3.7 ms,
              '0' AFTER 3.8 ms ;
        output <= a;
END;
```

Man hätte den Signalverlauf natürlich gleich dem Ausgang zuweisen können (z.B. `output <= '1' AFTER 0.5 ms,...`). Der Vorteil über das Zwischensignal a zeigt sich, wenn man außer mit Signal a zusätzlich eine Simulation mit einem anderen Signal durchführen möchte. Dafür muss lediglich ein neues Signal mit entsprechendem Kurvensignal definiert und dem Ausgang eines der beiden Signale zugewiesen werden (`output <= a oder b;`).

2.2.3.2 Digital/Analog-Umsetzer

Folgende Abbildung zeigt das allgemeine Schaltbild eines D/A-Umsetzers. Die Bezeichnungen der Anschlüsse bzw. der Ausgangsspannung sind schon eingezeichnet und sollen so übernommen werden.



Da bei der Modellierung Terminals verwendet werden, die elektrische Eigenschaften aufweisen, muss vor der Schnittstellenbeschreibung das Package geladen werden,

in dem das entsprechende Nature definiert worden ist. Verwendet man zur Simulation das VHDL-AMS Tool hAMSter, müssen die ersten Zeilen wie folgt aussehen.

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
```

Nun kann die Definition der Schnittstellensignale und die Beschreibung der gesamten Entity erfolgen.

```
ENTITY da_converter IS
     PORT ( TERMINAL p,m : ELECTRICAL ;
            SIGNAL input : IN BIT );
END;
```

Auffällig bei der Portbeschreibung ist, dass dem Eingangssignal `input` explizit der Objekttyp `SIGNAL` vorangestellt wird. Defaultmäßig haben, außer den Terminals, alle Schnittstellensignale den Objekttyp `SIGNAL`. Aus Gründen der Lesbarkeit ist diese Darstellungsweise aber sehr zu empfehlen.

In der Architektur-Beschreibung muss einerseits eine Spannung `v_in` am Ausgang des D/A-Umsetzers eingeführt werden, andererseits muss eine Möglichkeit gefunden werden, das dargestellte Wandlungsverhältnis mathematisch auszudrücken. Diese zwei Forderungen fallen unter die Schlagworte Branch-Quantity und Simultaneous IF-Statement. Die Realisierung zeigt folgende Architekturbeschreibung.

```
ARCHITECTURE behav OF da_converter IS
    QUANTITY v_in ACROSS i_out THROUGH p TO m;
BEGIN
    IF ( input= '0' ) USE
       v_in == -2.0;
    ELSE
       v_in == 2.0;
    END USE;
END;
```

Zusätzlich wird hier der Strom `i_out` zwischen den Terminals p und m definiert.

2.2.3.3 RC-Glied

Das RC-Glied soll aus Komponenten aufgebaut werden (strukturale Modellierung). Dazu müssen zuerst die Komponenten 'Widerstand R' bzw. 'Kondensator c' beschrieben und anschließend entsprechend verschaltet werden.

2.2.3.3.1 Modellierung des Widerstandes

Ein Widerstand ist durch seinen (Widerstands-)Wert und zwei Anschlüsse schnell beschrieben. Die elektrischen Größen an seinen Anschlüssen lassen sich mit Hilfe des Ohmschen Gesetzes ausdrücken.

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
ENTITY resistor IS
     GENERIC ( val  : REAL );
     PORT    ( TERMINAL p,m   : ELECTRICAL );
END resistor;

ARCHITECTURE behav OF resistor IS
     QUANTITY u_r ACROSS i_r THROUGH p TO m;
BEGIN
     i_r == u_r / val;
END behav;
```

Mit Hilfe des Parameters (Generics) `val` ist es nun möglich, Widerstände mit den unterschiedlichsten Widerstandswerten zu erzeugen bzw. zu instantiieren. Da beide Komponenten (R und C) des Tiefpasses in Reihe geschaltet werden, wird mit der

einfachen simultanen Anweisung (`i_r == u_r / val`), der Strom und nicht die Spannung beschrieben.

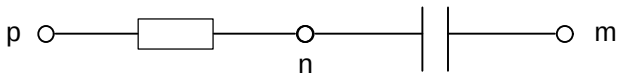
2.2.3.3.2 Modellierung des Kondensators

Die Beschreibung des Kondensators verläuft analog zu der des Widerstandes. Einziger Unterschied ist die mathematische Beschreibung des Stromes, der durch das Bauteil fließt. Wie bekannt, wird der Strom durch einen Kondensator durch folgende Gleichung zum Ausdruck gebracht: $i_C = C\frac{du(t)}{dt}$

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
ENTITY capacitance IS
    GENERIC ( val        : REAL);
    PORT    ( TERMINAL p,m    : ELECTRICAL);
END;

ARCHITECTURE behav OF capacitance IS
    QUANTITY u_c ACROSS i_c THROUGH p TO m;
BEGIN
    i_c == val * u_c'dot;
END;
```

Das Quantity-Attribut `u_c 'dot` kennzeichnet die erste Ableitung der Kondensator-Spannung nach der Zeit.

2.2.3.3.3 Modellierung des RC-Tiefpasses



Bei der Zusammenführung der einzelnen Komponenten benötigt man in der Architekturbeschreibung den internen Verbindungspunkt n. Die Komponenten werden hier nicht, wie im vorangegangenen Beispiel beschrieben, nach der Vorgehensweise Komponentendeklaration - Komponenteninstantiierung zusammengesetzt, vielmehr benutzt man hierbei die **direkte Instantiierung**, die keiner Komponentendeklaration bedarf (die direkte Instantiierung wird unter [2.2.3.4] genauer beschrieben).

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
ENTITY rc IS
    PORT( TERMINAL p,m: ELECTRICAL );
END;

ARCHITECTURE behav OF rc IS
    TERMINAL n    : ELECTRICAL;
BEGIN
   R1:  ENTITY resistor    (behav)
        GENERIC MAP (val => 1000.0) PORT MAP (p,n);
   C1:  ENTITY capacitance (behav)
        GENERIC MAP (val => 1.0e-6) PORT MAP (n,m);
END;
```

Die Angaben in der Generic Map der entsprechenden Komponente, erzeugen den RC-Tiefpass mit den geforderten Werten von R = 1 k und C = 1µF.

2.2.3.4 Direkte Instantiierung und eine neue Zuweisungsart

Wie der Name schon sagt, können bei der direkten Instantiierung Komponenten auf direktem Weg erzeugt werden, d.h. ohne vorhergehende Deklaration. Dafür muss

neben dem Namen der Komponente zusätzlich die verwendete Architektur des Modells angegeben werden. Die Handhabung der Port- bzw. Generic Map bleibt dieselbe. Die direkte Instantiierung stellt lediglich eine andere Schreibweise für die 'normale' Komponenteninstantiierung dar.

```
inst_name : ENTITY entity_name ( architecture_name )
            [ GENERIC MAP (...) ] [ PORT MAP(...) ] ;
```

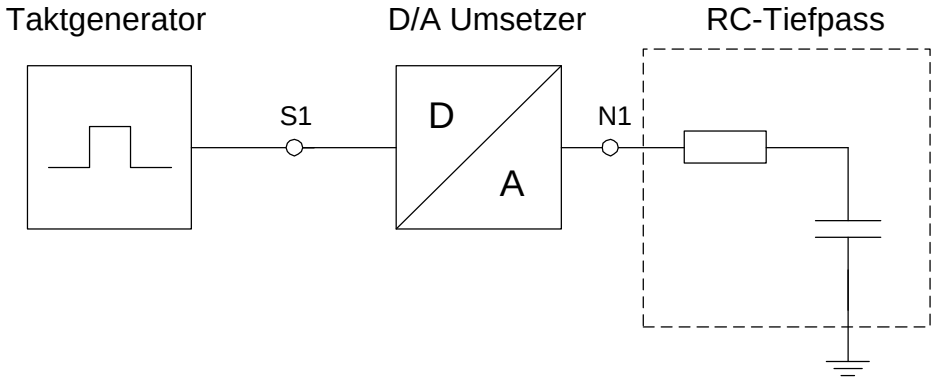
Dem einen oder anderen mag aufgefallen sein, dass der Name der Architekturbeschreibung der einzelnen Komponenten derselbe ist. VHDL kann sie bei der direkten Instantiierung auseinanderhalten, da für jedes Bauteil explizit die dazugehörige Schnittstellenbeschreibung angegeben werden muss.

Weiterhin fällt im vorangegangenen Beispiel des RC-Gliedes die besondere Wertzuweisung in der Generic Map auf. Man nennt sie 'named association'. Sie ist auch in der Port Map erlaubt. Port- bzw. Parameterbezeichnungen der Entity-Beschreibung werden hierbei namentlich genannt und durch den Operator => neue Werte oder Signale zugewiesen. Die Reihenfolge der Werte ist bei einer 'named association' im Vergleich zur herkömmlichen Methode (durch Komma getrennte Werte) vollkommen egal.

Daraus ergibt sich eine andere Schreibweise für die Port Map des Widerstands R1.

```
PORT MAP (m => n, p => p)
```

2.2.3.5 Modellierung des gesamten Systems



Mit Hilfe der beschriebenen Bestandteile (Taktgenerator - D/A-Umsetzer - RC-Glied) und der direkten Instantiierung, kann das Gesamtsystem schnell modelliert werden. Hierfür sind sinnvollerweise alle Schnittstellen- und Architekturbeschreibungen in der gleichen Datei abgespeichert. Als Schnittstellensignale werden S1(digital) und N1(analog) definiert.

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
ENTITY bench IS
END;

ARCHITECTURE behav OF bench IS
    TERMINAL N1    : ELECTRICAL;
    SIGNAL   S1    : BIT;
BEGIN
    RC1: ENTITY rc (behav)
         PORT MAP (n1,electrical_ground);
    DA1: ENTITY da_converter (behav)
         PORT MAP (n1,electrical_ground,S1);
    FF1: ENTITY flipflop (behav) PORT MAP (S1);
END;
```

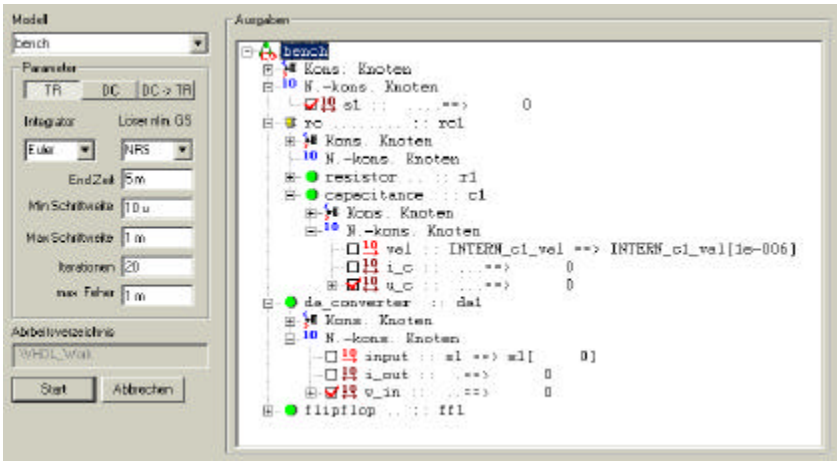
In der Nature `ELECTRICAL`, wird neben den `ACROSS`- und `THROUGH`-Typen, ein Bezugs- bzw. Referenzpotenzial mit dem Namen `electrical_ground` definiert, das hier in der Port Map Verwendung findet.

Zusammenfassend ist anzumerken, dass ein System wie es uns gerade vorliegt, schnell und recht einfach konstruiert und simuliert werden kann, wenn man es in mehrere Teilsysteme zerlegt. Vor allem der Übergang von digitaler zur analoger Hardwarebeschreibung kann hier sehr schön beobachtet werden.

2.2.4 Simulation

Zur Simulation geben wir die erarbeitete VHDL-AMS Beschreibung des Mixed-Mode-Modells ein oder öffnen die Datei mit dem Editor (...\hAMSter\Examples\mixed_mode\mixed_mode_simple.vhd). Dem einen oder anderen werden an manchen Stellen Unterschiede zwischen beiden Dateien auffallen, das ist weiter nicht schlimm, da sich beide Versionen fehlerfrei kompilieren lassen und bei der Simulation zum selben Ergebnis führen. Zum Kompilieren und Simulieren wird der Play-Button gedrückt, daraufhin erscheint zuerst das Dialogfenster des Simulators.

Hier werden sämtliche Simulationsparameter übernommen. Neben den Potenzialen an den Punkten S1 und N1 (v_in) lassen wir uns zusätzlich die Spannung am Kondensator anzeigen.



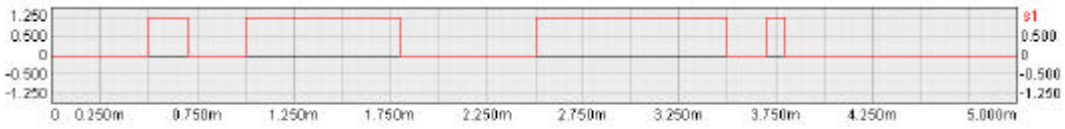
Nach 'Start' sieht man im View Tool von hAMSter die Auf- und Entladevorgänge am Kondensator in Abhängigkeit vom Eingangssignal. Auffallend ist, dass hAMSter versucht, für alle dargestellten Signale eine einheitliche Skalierung der Y-Achse

anzustreben. Aus diesem Grund reicht die Spannung an N1 (Ausgang des D/A-Umsetzers) nur von -1 ... +1V. Abhilfe schafft ein Klick auf 

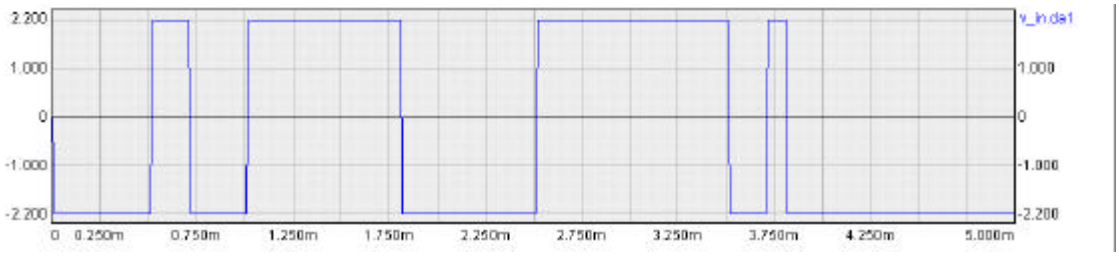
unter dem Menüpunkt „Bearbeiten“ oder das entsprechende Icon in der Toolbar. Bei der Frage, welche Diagramme optimal dargestellt werden sollen, ist es ratsam, immer alle Diagramme auszuwählen.

Bei diesem Beispiel wird der Gebrauch und der Nutzen von `Generics` sehr deutlich. Der Quellcode muss bei einer Änderung eines Generics nicht neu übersetzt werden, die modifizierten Werte werden ohne erneutes Kompilieren übernommen. Im Folgenden wird für verschiedene Kapazitätswerte jeweils die Spannung am Kondensator aufgenommen.

Ausgangsspannung Flipflop (Eingangsspannung D/A-Umsetzer)

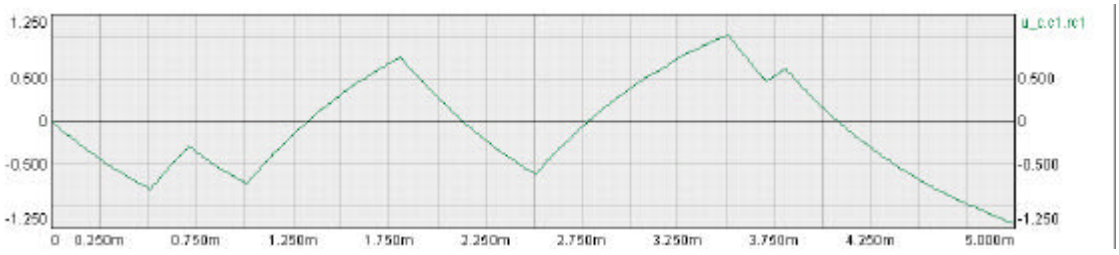


Ausgangsspannung D/A-Umsetzer (Eingangsspannung Tiefpass)

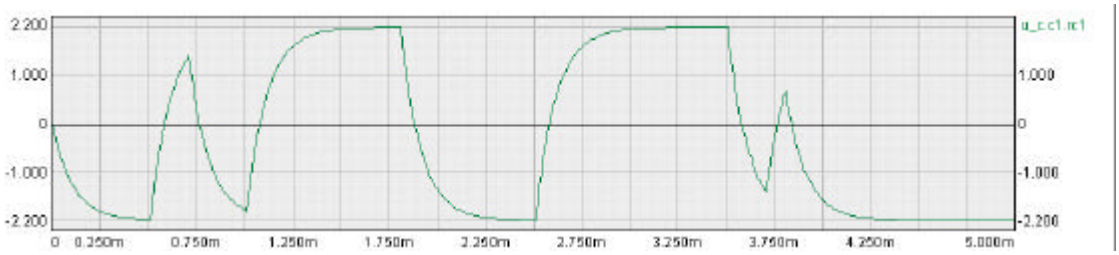


Spannung am Kondensator

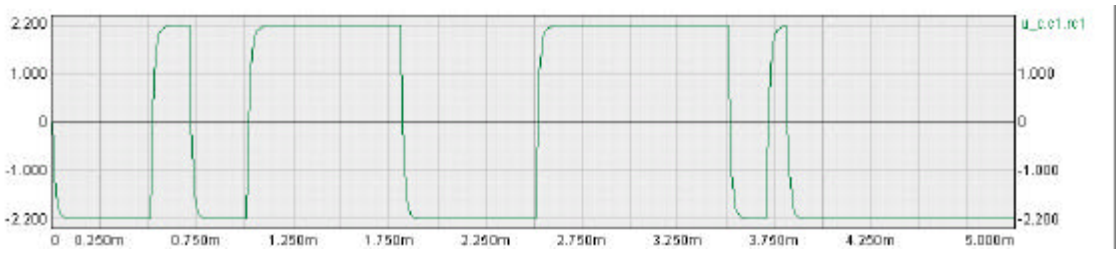
Kapazitätswert: C = 1µF



Kapazitätswert: C = 100nF



Kapazitätswert: C = 10nF



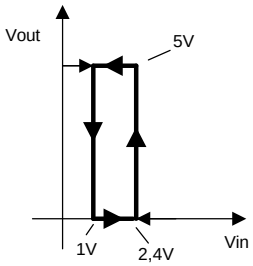
2.3 Schmitt-Trigger

2.3.1 Aufgabenstellung

In diesem Beispiel soll ein analoger Schmitt-Trigger mit Hysterese vorgestellt und simuliert werden. Nach dem ersten Simulationsdurchlauf werden dann einige Parameter geändert und nochmals simuliert, um verschiedene Funktionen zu verdeutlichen. Dieser Schmitt-Trigger hat ein analoges Spannungssignal als Ausgang, man hätte aber auch prinzipiell einen Digitalausgang implementieren können.

Das Verhalten des Schmitt-Trigger soll in unserem Beispiel das in der Tabelle gezeigte Verhalten aufweisen, welches nebenstehender **Hysterese** entspricht:

Eingangsspannung	**Ausgangsspannung**
Vin > 1.0V	Vout = 0V
Vin < 2.4V	Vout = 5.0V



Als Eingangssignal verwenden wir eine Sinusspannung mit 5 Volt Amplitude. Wird sie von Nullpunkt kommend zu 1V, so wird getriggert und die Ausgangsspannung geht auf 0V. Nachdem der Sinus seinen Scheitel erreicht hat und dann unter den Wert 2,4V fällt, wird die Triggerung erneut aktiviert und setzt das Ausgangssignal auf 5V. Solange der Sinus weiter in den negativen Bereich fällt, sein Minimum erreicht und wieder aufsteigt bis 1V, ändert sich nichts.

2.3.2 Implementation

Zunächst werden wieder die mathematischen und elektromagnetischen Packages eingebunden.

Das Modell beginnt mit der Entity, welche hier wieder sehr primitiv ausfällt, da keine externen Schnittstellen benötigt werden.

```
ENTITY AnalogueSchmitt is
END ENTITY AnalogueSchmitt;
```

Wenden wir uns nun dem Deklarationsteil der Architekturbeschreibung zu. Da es nur einen Eingang und einen Ausgang gibt, werden nur zwei elektrische Terminals n1

```
ARCHITECTURE Hysteresis OF AnalogueSchmitt IS
        TERMINAL n1,n2    : ELECTRICAL;
        SIGNAL   State : REAL := 0.0;
        QUANTITY Vin  ACROSS Iin THROUGH n1;
        QUANTITY Vout ACROSS Iout THROUGH n2;
BEGIN
...
END ARCHITECTURE Hysteresis;
```

und n2 deklariert. Der aktuelle Zustand des Schmitt-Triggers soll in einem `SIGNAL` mit Modus `REAL` zwischengespeichert werden. Es wird mit Null initialisiert.

Nun müssen wir zwei Quantities definieren, um die elektrischen Größen Spannung und Strom mit den Anschlüssen (Terminals) zu koppeln. Die Zuordnungen sind nun eindeutig, '`Vin`' wurde also für die Eingangsspannung und '`Iin`' für den Eingangsstrom gewählt, entsprechendes gilt für den Ausgang. Zur Erinnerung: Wenn nach Angabe des ersten Terminals nichts mehr folgt, ist die Größe auf die Reference (Bezugspotenzial) bezogen, hier also Masse.

Jetzt fehlt noch das wichtigste an dem Ein-Block-System, nämlich der Beschreibungsteil der Architecture. Hier werden ein neues AMS-Sprachelement sowie 2 AMS-Attribute eingeführt. Zunächst soll der Eingangs-Sinus generiert werden. Und wieder einmal

```
Vin == 5.0 * sin(2.0 * math_pi * 0.5E3 * NOW);
```

wird der Vorteil der AMS-Erweiterung, Gleichungen einzusetzen, ersichtlich. Denn man braucht nur die allgemeine Sinusgleichung ($y_{Sinus}=A \cdot \sin(2\pi ft)$) anzuschreiben. Die Sinus-Funktion ist selbstverständlich im mathematischen Package enthalten. Hier soll die Amplitude also den Wert 5 Volt aufweisen und die Frequenz beträgt 500Hz. Das Wort **`NOW`** nimmt ein besondere Stellung in VHDL-AMS ein. Es ist die Zeit zum aktuellen Simulationszeitpunkt.

TIP: Statt '`2.0*math_pi`' kann auch **'`math_2_pi`'** geschrieben werden.

VHDL ist sehr typstreng[8]. Die sin-Funktion und eine `Quantity` verlangen einen Float-Wert (REAL), so müssen auch alle anderen Werte dieses Ausdruckes diesem Typ entsprechen. Real-Werte müssen stets mit einem **Dezimalpunkt** versehen sein!

2.3.2.1 Nebenläufige Break-Anweisung und das Attribut `'ABOVE()`

Nun kommen wir zur eigentlichen Triggerfunktion. Sie ist mit Hilfe einer „Nebenläufigen Break-Anweisung“ (**Concurrent Break Statement**) und dem Quantity-Attribut **`'above()`** implementiert. Dieses Sprachkonstrukt erlaubt es, den Ablauf eines Analog-Modells an einer bestimmten Stelle zu **unterbrechen**. Das kann passieren, wenn entweder ein physikalischer Umstand dies erfordert, oder wenn ein bestimmtes Event beim Digital-Teil auftritt, das den Analog-Teil beeinflussen soll. In solch einem Fall wird der Analog-Solver, also der Teil des Simulators, der die

```
BREAK State => 0.0 WHEN Vin'ABOVE(1.0);
BREAK State => 5.0 WHEN NOT Vin'ABOVE(2.4);
```

analogen Gleichungen bearbeitet und löst, neu initialisiert und wird veranlasst, die betreffenden Gleichungen neu zu berechnen. Die Neuberechnungen berücksichtigen dann die neuen physikalischen Umstände. So sieht die nebenläufige Break-Anweisung allgemein aus:

```
[ label: ] BREAK [ break_list ]
[ ON sensitivity_list] [ WHEN condition ];
```

Das Besondere an diesem Concurrent Break Statement ist, das es eben nebenläufig ist, das heißt, es muss nicht in einem Prozess stehen, sondern kann in ganz normalen Anweisungsteilen stehen und muss somit ständig vom Analog-Solver überwacht bzw. abgefragt werden.

Ein dementsprechendes Statement in einem Prozess würde folgendermaßen aussehen:

```
PROCESS
BEGIN
      BREAK [ break_list ] [ WHEN condition ]
```

[8] Wichtiger **Vorteil** der Typstrenge: Programmierfehler sind schneller detektierbar.

```
      WAIT [ on sensitivity_list] | [ ON name ];
END PROCESS;
```

Es ist also ersichtlich, dass es dieses neue Statement erlaubt, ein bisher den Prozess-Teilen vorbehaltenes Sprachelement in der Analog-Umgebung bequem und intuitiv einzusetzen.

Das Attribut **`Q'ABOVE(real_wert)`** ist sehr wichtig bezüglich der **Analog/Digital-Schnittstelle**. Das 'Q' steht für Quantity und stellt somit beispielsweise einen analogen Signalverlauf dar. Wächst dieses Signal nun an und **übersteigt** einen bestimmten Schwellwert 'real_wert' +Toleranz[9], so nimmt der Ausdruck den Wert TRUE an. Untersteigt die Quantity diese Schwelle (-Toleranz), so ist der Ausdruck FALSE. Der Ergebnistyp ist also Boolean. Zu beachten ist, dass nicht der Momentanzustand zählt, sondern ausschließlich der **Wechsel** das Ereignis darstellt und somit entscheidend ist.

```
quantity_name'ABOVE(real_wert);
```

EXKURS

Übersicht und Vergleich wichtiger Operatoren

:	Typ/Modus-Zuweisung; nach Labels
:=	Variablenzuweisung; Initialisierung
=	Vergleichsoperator
==	Zuweisung für Simultaneous Statements
<=	Signalzuweisung, !!! Kann auch Vergleichsoperator sein, kontextabhängig) !!!
=>	Verschiedenenes: Z.B. CASE-Anweisung; Zuweisung beim Mapping; Concurrent oder Sequential Break Statement

Und nun zu unserem Fall: Dem `SIGNAL` 'State' wird '0.0' zugewiesen, wenn **`Vin'ABOVE(1.0)`** ist, also wenn die Eingangspannung den Wert 1.0 (Volt) übersteigt. Wenn die Spannung dann später wieder unter 1.0 sinkt, nimmt der „above-Ausdruck“ den Zustand FALSE ein, jedoch beeinflusst das nicht das Break-Statement. Es reagiert nur beim Wechsel auf TRUE. Genau diese „Speicherwirkung“ benötigt unser Schmitt-Trigger-Modell, es wird also hier nur das Break-Ereignis ausgelöst, wenn der Wert von unten nach oben wechselt und nicht umgekehrt!
In der zweiten Zeile wird das above-Attribut invertiert, d.h. wenn die Eingangsspannung von oben kommend 2.4V unterschreitet wird der Ausdruck

[9] Es ist eine kleine Toleranz und damit eine kleine Hysterese eingebaut, die ständiges „Umschalten“ verhindern soll. Für Näheres soll hier aber auf entsprechende Fachliteratur verwiesen werden.

FALSE und durch die Invertierung TRUE, was wiederum das Break-Statement auslöst und 'State' den Wert 5.0 zuweist. Damit ist die Hysterese abgeschlossen.

2.3.2.2 Sequentielle Break-Anweisung

Wie wir auch im reinen digitalen VHDL zwischen Nebenläufigkeit und Sequenz unterschieden haben, können wir das auch für das AMS-seitige Break-Statement machen.

Deshalb soll hier noch die sequentielle Break-Anweisung (**Sequential Break Statement**) erwähnt werden. Sie werden vornehmlich für die Initialisierung des Analog-Solvers genutzt. Die break_list besteht dann aus einer sogenannten selector_clause und einer Zuweisung, wobei die selector_clause eine abgeleitete oder integrierte `QUANTITY` enthalten muss.

```
BREAK [ break_list ] [ WHEN condition ];
```

Wenn keine <condition> benutzt wird, so werden die betreffenden Quantities zu Beginn der Simulation initialisiert.

Für nähere Informationen sei auf Fachliteratur verwiesen.

2.3.2.3 Das Attribut `'RAMP()`

Im letzten Teil der Architecture ist folgende Zeile zu finden:

```
Vout == State'RAMP(10.0e-6, 10.0e-6);
```

Die Übergabewerte dieser Code-Zeile sind ausnahmsweise abgeändert worden, warum, wird im nächsten Abschnitt noch erklärt.

Wie der Name schon verrät, ist die Rampen-Funktion mit im Spiel. **`'RAMP`** ist ein Attribut für Signale, d.h. wird Signalen angehängt. Der Ergebnistyp ist der Basistyp vom Signal, also meist wie auch in diesem Beispiel eine Quantity.

`'RAMP` erwirkt eine Verzögerung eines Signals in Form einer Rampe. Dem Attribut kann eine Anstiegszeit und eine Abstiegszeit übergeben werden, wobei letztere auch weggelassen werden kann. Beide Zeiten müssen Real-Werte sein und verstehen sich in Sekunden.

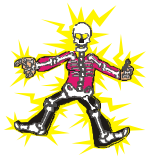
```
signal_name'RAMP(t_rise [,t_fall]);
```

In unserem Beispiel wird also dem `SIGNAL` 'State', welches den internen Schmitt-Trigger-Zustand speichert, auferlegt, keine abrupten Signaländerungen zu vollziehen, sondern Änderungen stets mit einer Anstiegsflanke der Dauer 10µs und einer Abstiegsflanke, die ebenfalls 10µs dauert, zu berücksichtigen. Der ganze Ausdruck wird dann der `QUANTITY` 'Vout', also der Ausgangsspannung zugewiesen. Nun sieht man auch, dass es sinnvoll war, den Zustand zunächst als SIGNAL zu definieren, um es dann modifiziert in eine Spannung zu „verwandeln“.
Mit anderen Worten: `S'RAMP(tr,tf)` ist eine `QUANTITIY`, die dem `SIGNAL` S folgt unter Berücksichtigung der An- (t_rise) und Abstiegsverzögerung (t_fall).
Mit diesem Attribut kann man, wie auch hier, sehr gut das reale **analoge Verhalten** eines Modells nachahmen. Es sind natürlich immer unendlich schnelle Sprünge und abrupte Änderungen erwünscht, nur ist das in der Realität nicht möglich. Um Trugschlüsse bei der Simulation, d.h. dann auch Fehler im realen System, zu vermeiden, sollte man mit solchen Mitteln nicht sparen. Es müssen also **Verzögerungszeiten** eingesetzt werden, die denen der im realen System verwendeten Komponenten genau oder zumindest näherungsweise entsprechen.
Abgesehen davon werden diese Verzögerungen gebraucht, um dem Simulator eine Diskontinuität im Signal anzukündigen, also abrupte Änderungen zu vermeiden.
Die Funktion und Auswirkungen dieses Attributes werden im nächsten Abschnitt (Simulation des Schmitt-Triggers) noch mal ganz deutlich.

2.3.3 Simulation

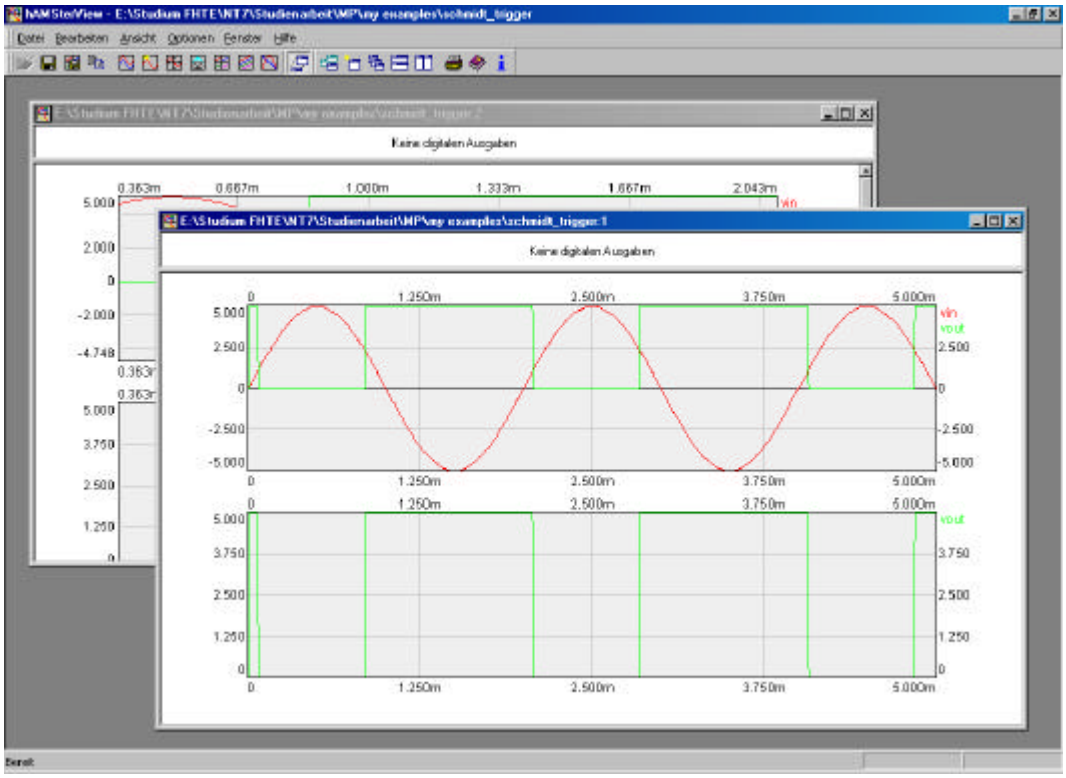
Um den Schmitt-Trigger nun zu simulieren, öffne man im Editor die Datei 'schmidt_trigger.vhd' (hier fälschlicherweise mit 'dt') und drücke den Play-Button. Wenn wir die Default-Parameter vom bestehenden cfg-File im Fenster 'Simulationsparameter' beibehalten wollen, müssen wir etwas im Code des Schmitt-Triggers ändern. Wie schon erwähnt, wurde die An- und Abstiegszeit beim RAMP-Attribut geändert. Wir haben die Werte von den gegebenen 1ns auf 10µs erhöht. Der Grund dafür ist, dass als minimale Schrittweite 1µs angegeben ist, so könnte eine Flanke, die nur 1ns dauert natürlich nicht dargestellt werden, sähe also bestenfalls wie eine 1µs-Flanke aus. (Kann leicht ausprobiert werden.)

Gut, nachdem also alle Simulationsparameter übernommen und mit 'Start' quittiert wurden, öffnet sich **'hAMSterView'**, der Simulations-Viewer von hAMSter. In den nachfolgenden Beispielen sollen die wichtigsten Features dieses Programms erläutert werden.
Analog- und Digital-Teil sind deutlich voneinander getrennt, die Abgrenzungslinie kann nach Bedarf einfach verschoben werden.

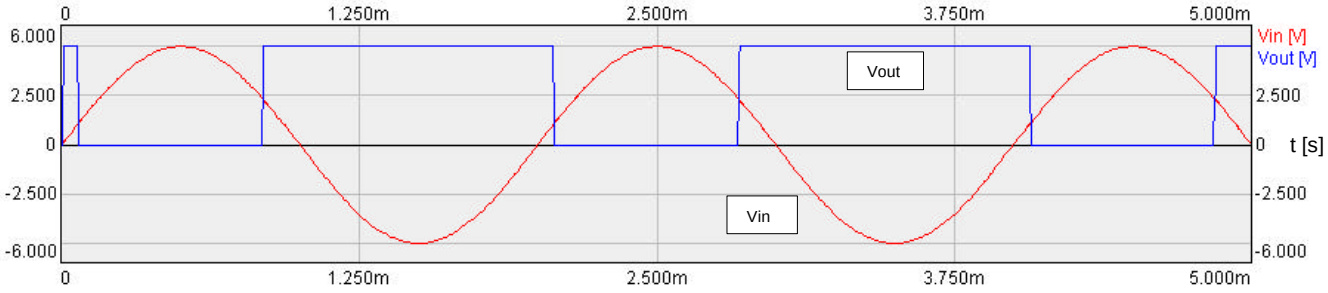


Mit hAMSter lassen sich nur Transientenanalysen, also Simulationen nach der Zeit, vornehmen. Analysen im Frequenzbereich sind nicht möglich.

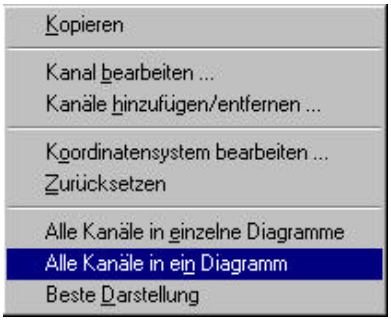
So sieht der Arbeitsoberfläche von hAMSterView aus:



Es können leicht mehrere Fenster erzeugt werden, indem man beispielsweise einen interessanten Bereich aufzoomt. Einfach mit der Maus ein Rechteck aufspannen, es werden automatisch jeweils neue Fenster erzeugt. So können enorme Detailansichten bis zur Berechnungs-Auflösung dargestellt werden.
Folgende Abbildung zeigt nun das Simulationsergebnis:

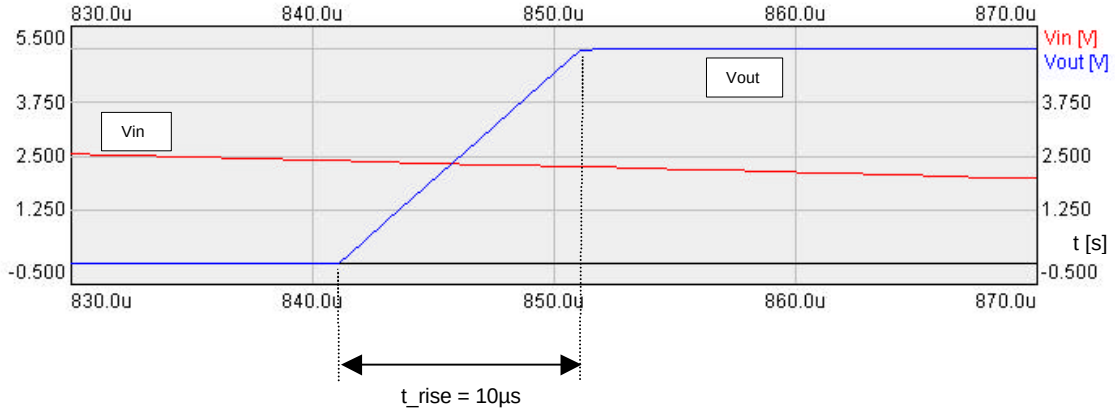


Wenn beide Kurven in einem Diagramm dargestellt werden, erkennt man schön die **Triggerpunkte** bei 1V bzw. 2,4V. Um das zu bewerkstelligen, wählt man im



Kontextmenü (rechte Maustaste) auf dem Diagramm einfach 'Alle Kanäle in ein Diagramm'. Umgekehrt können alle Kurven auch wieder in einzelnen Diagrammen dargestellt werden. Das **Kontextmenü** bietet noch weitere interessante Features: In 'Koordinatensystem bearbeiten' kann man die x- sowie die y-Achse getrennt voneinander konfigurieren. Wem die Default-Einstellung des Maximalwertes der y-Achse von 5 Volt zu knapp ist, kann sie leicht z.B. auf 6 Volt skalieren, wie auch hier dargestellt.

Außerdem können die Achsen auf logarithmische Darstellung umgeschaltet werden.



An obiger **Zoom-Darstellung** lässt sich schön die Auswirkung des RAMP-Attributes beobachten. Die Dauer der Anstiegsflanke entspricht also der eingestellten Rise-Time von 10µs.

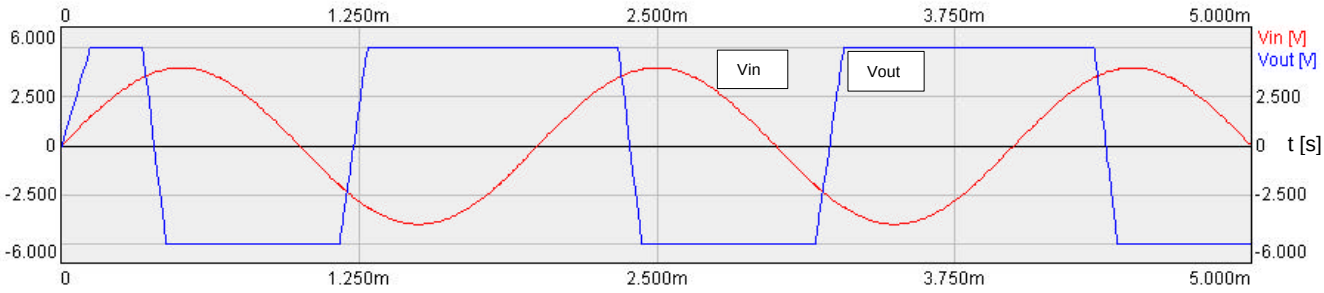
2.3.3.1 Simulation mit geänderten Parametern

Nun sollen verschiedene Parameter vor der Simulation wie folgt abgeändert werden:

- Amplitude des Sinus: 4 Volt
- Schaltschwelle bei **ansteigender** Kurve: 3,5 Volt → Schalten auf –5 Volt
- Schaltschwelle bei **absteigender** Kurve: -2 Volt → Schalten auf +5 Volt
- Anstiegszeit des Ausgangssignals: t_rise = 120µs
- Abstiegszeit des Ausgangssignals t_fall = 100µs

Das sähe dann so aus:

```
...
Vin == 4.0 * sin(math_2_pi * 0.5E3*NOW);
BREAK State => -5.0 WHEN Vin'ABOVE(3.5);
BREAK State => 5.0 WHEN NOT Vin'ABOVE(-2.0);
Vout == State'RAMP(120.0e-6, 100.0e-6);
...
```



Die Änderungen kann man anhand der Simulationsgrafik nun gut verfolgen. Aufgrund der verlängerten An- und Abstiegszeiten fallen die rampenartigen **Flanken** der Ausgangsspannung jetzt auch in dieser Einstellung deutlich auf.

2.4 Sample&Hold

2.4.1 Aufgabenstellung

Ein Signal soll abgetastet und der Wert für eine Abtastperiodendauer gehalten werden. Wir verwenden hier wieder ein Sinussignal; man kann sich also vorstellen, dass man ein treppenartiges Ausgangssignal bekommt.
Sample&Hold-Einheiten sind heute von hoher Wichtigkeit, werden beispielsweise als Eingangstufen von Analog/Digital-Umsetzern eingesetzt und sind allgemein zur Digitalisierung von Signalen unentbehrlich. Da **Abtasten** allein nicht sinnvoll ist, d.h. für die Weiterverarbeitung meist nicht zu gebrauchen ist, wird der zum Abtastzeitpunkt „gelesene“ Wert auf seinem Niveau **gehalten** bis der nächste Wert erfasst wird.
In diesem Beispiel ist auch eine sogenannte Testbench integriert. In der Implementation [siehe 2.4.2] soll eine einfache Anwendungsmöglichkeit von einigen existierenden VHDL-Testbench-Strategien kurz vorgestellt werden.

2.4.2 Implementation

Begonnen wird wieder mit der `ENTITY`. Da diesmal als äußere Umgebung eine Testbench vorhanden ist, besitzt die `ENTITY` Schnittstellen nach außen, also Ports.

```
ENTITY sample_and_hold IS
     GENERIC (Tsampl : REAL := 0.05e-3;
              delay  : REAL := 0.0);
     PORT (TERMINAL inp, outp : ELECTRICAL);
END ENTITY sample_and_hold;
```

Neben den elektrischen Ein- und Ausgängen (`'inp'` und `'outp'`) werden zwei Real-Generics definiert: Die Sample-Periodendauer `'Tsampl'` (mit 50µs initialisiert) und die Verzögerung `'delay'` (zunächst auf Null gesetzt).

2.4.2.1 Das Attribut 'ZOH()

In der 'ideal'[10] genannten Architektur des Modells findet sich nun das Attribut **'ZOH()** wieder, welches sehr einfach eine **Abtastung** mit anschließendem Halten eines Signals ermöglicht. ZOH ist die Abkürzung für „zero-order hold“, es geht also um das Halten ersten Grades, welches der gewöhnlichen treppenförmigen Abtastung/Haltung entspricht, bei der nachträglich keine Interpolation oder Veränderung des Signals erfolgt.

```
ARCHITECTURE ideal OF sample_and_hold IS
     QUANTITY vinsh ACROSS              inp;
     QUANTITY v_in  ACROSS i_in THROUGH outp;
BEGIN
     v_in == vinsh'ZOH(Tsampl, delay);
END ARCHITECTURE ideal;
```

Es werden zunächst zwei Quantities deklariert, 'vinsh' für das Eingangssignal und 'v_in' für das gewandelte, also getastete Signal.

Nun wird das ZOH-Attribut angewandt, indem es mit dem Eingangssignal gekoppelt und 'v_in' zugewiesen wird. 'v_in' ist dann die abgetastete Version von 'vinsh'. Nebenstehend ist wieder die allgemeine Anwendungs-Form des Attributes dargestellt. Der erste frei konfigurierbare Parameter ist die Periodendauer der Abtastfrequenz. Mit dem zweiten, jedoch nicht obligatorischem Übergabewert, kann eine Zeit definiert werden, wobei erst nach dessen Ablauf der erste abgetastete Wert ausgegeben wird. Diese Zeit kann die Initialisierungsphase einer entsprechenden Komponente simulieren.

```
quantity_name'ZOH(T_Sample [,init_delay])
```

2.4.2.2 Die Testbench

Testbenches (im deutschen Testumgebung) dienen vielerlei Zwecke. Ein entworfenes Modell muss natürlich getestet werden, d.h. es muss eine Umgebung

[10] Es handelt sich um ein ideales Modell, da z.B. die Flanken des abgetasteten Signals im Prinzip unendlich steil sind, also keine Verzögerungszeiten berücksichtigt werden.

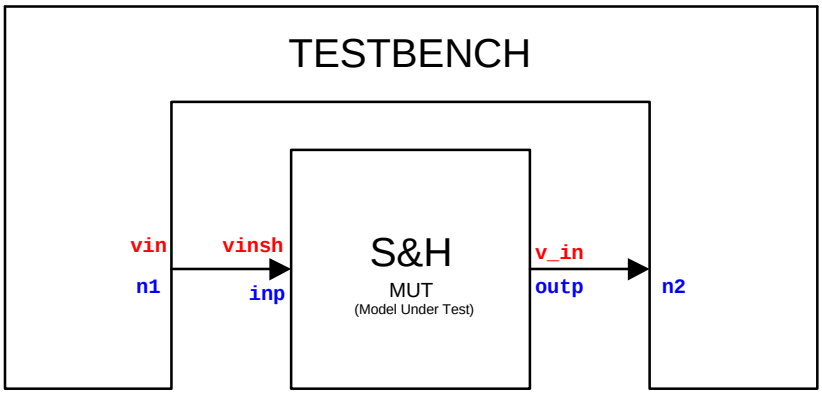
geschaffen werden, in der Eingangstestsignale (**Stimuli**) generiert werden und anschließend die am Ausgang auftretenden Signale erfasst und auf Richtigkeit **überprüft** werden. Ein andere wichtige Anwendung von Testbenches ist das Testen von verschiedenen Architekturen unter gleichen Umgebungsbedingungen. So lassen sich bequem Performancevergleiche durchführen.

Es gibt diverse Strategien, was die Modellierung von Testbenches betrifft, wobei das zu testende Modell stets auf die eine oder andere Art und Weise in die Bench eingebettet ist (siehe auch Zeichnung 'Model Under Test'). Hier ist nur eine sehr einfache Version angezeigt, welche nicht einmal eine Überprüfung des Ausgangssignals vornimmt, sondern lediglich an das Ausgangs-Interface anknüpft und dessen Signal übernimmt.

```
ENTITY bench IS END ENTITY bench;

ARCHITECTURE sample OF bench IS
     TERMINAL n1, n2 : ELECTRICAL;
     QUANTITY vin ACROSS iin THROUGH n1;
     CONSTANT f : REAL := 1.0E3;
BEGIN
     vin == sin(2.0 * math_pi * f * NOW);
     UUT : ENTITY sample_and_hold(ideal)
        GENERIC MAP (Tsampl => 0.1e-3, delay => 0.1e-3)
        PORT MAP (inp => n1, outp => n2);
END ARCHITECTURE sample;
```

An dieser Stelle muss nicht mehr auf jede Einzelheit des VHDL-AMS-Codes eingegangen werden, es sollte nur folgendes zur Erinnerung kurz erwähnt werden: Bei der Instantiierung des sample_and_hold-Moduls, die ja sinnvollerweise innerhalb der Testbench erfolgt, handelt es sich hier um die Version der **direkten Instantiierung** [vgl. 2.2.3.4]. Als Periodendauer der Abtastfrequenz werden 100µs gemapped und für die Initialisierungszeit ist ebenfalls 100µs vorgesehen. Die Sinusgleichung (`'vin'`) dient als Eingangssignal mit 1V Amplitude und 1kHz.

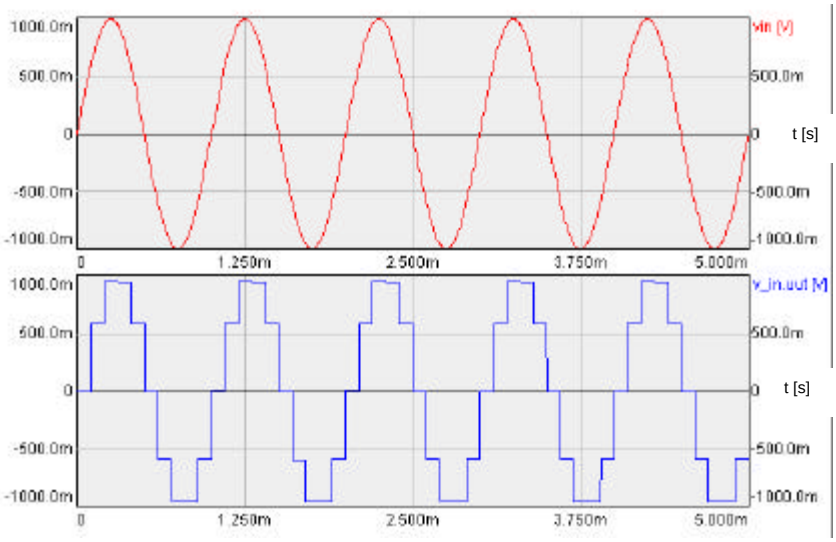
Sehr deutlich wird die Struktur der Testbench-Modell-Kombination sowie die Zusammenhänge aller benutzten Bezeichner in folgender Zeichnung:



Model Under Test

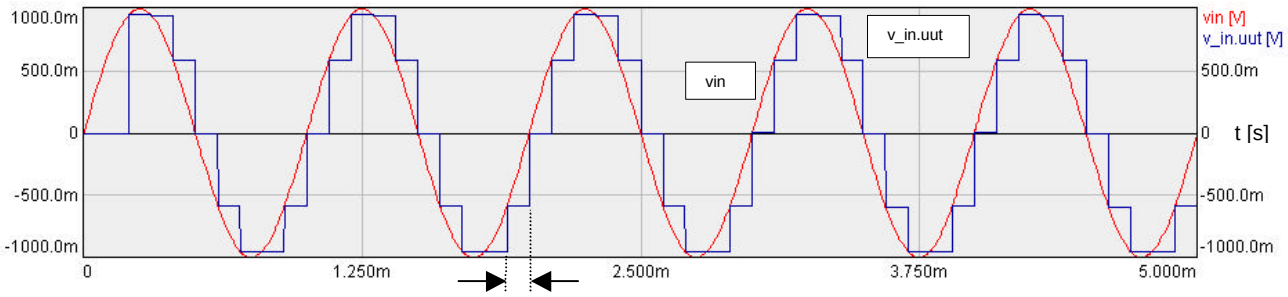
Hier kann man schnell überblicken, welche Quantities bzw. Spannungen und welche Terminals miteinander verbunden sind.

2.4.3 Simulation



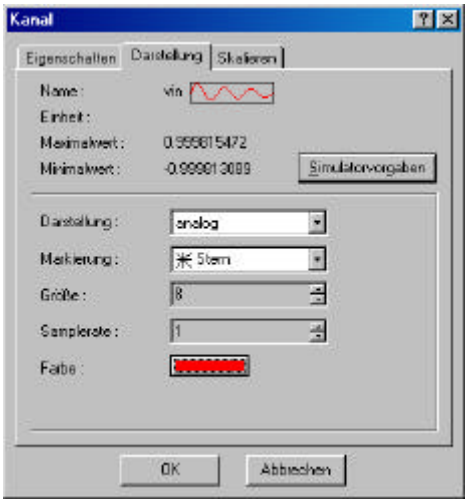
Hier ist das Eingangs-, sowie Ausgangssignal des S&H-Moduls dargestellt, welches sich nach der Simulation der Datei 'sample&hold.vhd' ergibt. Wie zu erkennen ist, war die Wahl der Initialisierungszeit (`'delay'`) von 100µs zu kurz, da während dieser Zeit der Abtastwert ohnehin Null beträgt.

Doch erst im nächsten Diagramm vermag man das **Arbeitsprinzip** einer Sample&Hold-Schaltung gut nachzuvollziehen. Um den Einfluss der Initialisierung zu zeigen, wurde hier `delay` 200µs zugewiesen. Eine wichtige Eigenschaft ist die **Phasenverschiebung**. Es ist klar zu beobachten, dass die Ausgangs-Kurve 'v_in.uut' etwa um ½T_Sample nach rechts verschoben ist.



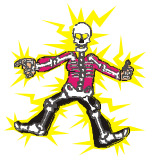
T_Sample=100µs

Die farbig gestalteten Kurven in **hAMSterView** sind mit ihrem dem Code entsprechendem Namen rechts am Diagrammrand aufgeführt. Ruft man über diese das **Kontextmenü** auf, bieten sich vielerlei Optionen. So kann man z.B. einen Cursor auf der Kurve entlang wandern lassen, wobei der aktuelle x/y-Diagrammpunkt in der Statuszeile angezeigt wird. Über den Eintrag 'Eigenschaften' lässt sich



nebenstehendes Dialogfeld aufrufen, in welchem der entsprechende **Kanal** (Signal) direkt beeinflusst werden kann, sei es nun die Farbeinstellung, die Beschriftung oder die Skalierung, mit welcher u.a. verschiedene Zoom-Darstellungen direkt einstellbar sind. Unter dem Punkt 'Markierungen' können verschiedene Kurven in einem Diagramm durch unterschiedliche Muster voneinander abgegrenzt werden.

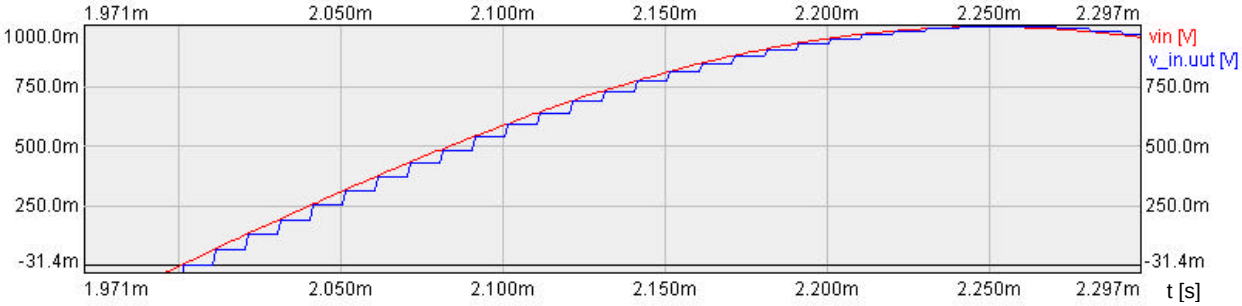
Über diesen farblich gekennzeichneten Namen am Rande des Diagramms, kann man auch den Kanal per Drag&Drop einfach in andere Diagramme einzeichnen. Die intuitive Bedienung des Programms mittels gerade genannter Funktion sowie die Möglichkeiten aller Kontextmenüs, machen die Buttons der Tool-Bar fast überflüssig. Über diese lassen sich zwar auch alle Funktionen steuern, jedoch muss beispielsweise immer erst der gewünschte Kanal angegeben werden.



VORSICHT: Unsere Erfahrungen im Umgang mit dem **Simulator** zeigten, dass ein Modell, welches ohne Fehler kompilierbar ist noch lange nicht **simulierbar** sein muss. Es kann sein, dass die Simulation ohne weitere Begründung abgebrochen wird, weil z.B. ein Parameter fehlt. Oder aber sie kann nicht durchgeführt werden, weil das Modell **unter- oder überbestimmt** ist, also semantische Fehler beinhaltet.

2.4.3.1 Simulation mit hoher Abtastfrequenz

Folgendes Diagramm zeigt das Simulationsergebnis, wenn das Modell mit sehr **niedriger Abtastperiodendauer (10µs)** betrieben wird. Zu sehen ist nur ein kleiner Ausschnitt des Sinussignals.



Die abgeschrägten Flanken des (idealen) Sample&Hold-Signals sind darauf zurückzuführen, dass zuvor bei den Simulationsparametern die 'Minimale Schrittweite' bei 1µs belassen wurde, die also nur ein Zehntel der Tastdauer beträgt. Würde man diese Schrittweite verringern, vergrößerte sich zwar die Rechenzeit, die Flanken wären dann aber wieder (ideal) senkrecht.

2.5 Peak-Detection

2.5.1 Aufgabenstellung

Mit diesem Modell soll es möglich sein, ein Eingangssignal so zu erfassen, dass immer nur dem momentan höchsten Wert gefolgt wird. D.h. nimmt das Signal einen fallenden Verlauf ein, so **hält** die Peak-Detection den bis dahin höchsten Wert fest. Nach Durchlauf einer Signal-Sequenz wird also der darin enthaltende Spitzenwert ermittelt.
Außerdem wird mit diesem Beispiel eine spezielle Anwendung des Attributes **`'SLEW()`** aufgezeigt.

2.5.2 Implementation

2.5.2.1 Das Attribut `'SLEW()`

Der Peak-Detector besteht aus dem Modul `'peakdetector'` samt Architecture `'ideal'` und einer Workbench `'bench'` dessen Architecture `'peak'` genannt wurde.

```
ENTITY peakdetector IS
     PORT (TERMINAL inp, outp : ELECTRICAL);
END ENTITY peakdetector;

ARCHITECTURE ideal OF peakdetector IS
     QUANTITY vin  ACROSS              inp;
     QUANTITY v_in ACROSS i_in THROUGH outp;
BEGIN
     v_in == vin'SLEW(1.0E38, -1.0e-38);
END ARCHITECTURE ideal;
```

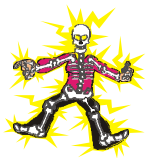
So wie auch im vorigem Beispiel werden in der `Entity` des Detectors Ein- und Ausgangs-Terminals definiert. Auch die Definition der Eingangsquantität ('vin') und

der (vielleicht nicht ganz glücklich genannten) Ausgangsspannung ('v_in') verläuft ähnlich.

Das **SLEW-Attribut**, welches für `QUANTITIES` und `SIGNALS` existiert, liefert als Ergebnistyp den jeweiligen Basistypen zurück. Es beschränkt die Steilheit der

```
quantity_ | signal_name'SLEW(max_rising_slope [,max_falling_slope]);
```

Anstiegs- bzw. Abstiegsflanken einer Quantität oder eines Signals um die Steigungswerte, die als Parameter übergeben werden[11]. In unserem Fall handelt es sich um die `Quantity 'vin'`. Bei den Übergabewerten handelt es sich um wirkliche **Steigungen**, also $\Delta y/\Delta x$ für die steigende bzw. $-\Delta y/\Delta x$ für die fallende Flanke. Wenn das mit dem Attribut verknüpfte Signal sich also **schneller ändert**, als es die Steigungen zulassen, wird im Ergebnis nicht mehr der Signalverlauf nachgebildet, er wird stattdessen durch eine **Gerade** mit dem entsprechenden Steigungswert ersetzt, bis die Änderungsgeschwindigkeit wieder innerhalb des „erlaubten“ Bereiches ist.



Mit **`'SLEW()`** lassen sich also bestimmte Beschränkungen, die auf der Trägheit realer Systeme beruhen, leicht nachbilden. Einleuchtendes Beispiel ist die namensverwandte **Slew-Rate** von Operationsverstärkern, welche das Folgen der Eingangsspannung auf eine maximale Flankensteilheit am Ausgang beschränkt.

In diesem Beispiel wird die eigentliche Funktion des Attributes natürlich missbraucht. So ist es eben möglich, mit dessen Hilfe eine Peak-Detection zu realisieren, indem man willkürliche, aber betragsmäßig extrem große bzw. kleine Werte übergibt. Die Anstiegsflanke ist somit quasi unendlich steil, so dass dem Eingangssignal exakt gefolgt werden kann. Die fallende Flanke jedoch wird mit dem sehr großen negativen Wert künstlich so flach gehalten, dass sie als waagerecht angesehen werden kann. Somit wird immer der am Eingang höchste auftretende Wert gehalten.

[11] Laut Standard kann der zweite Parameter (sogar beide, hier nicht gekennzeichnet) weggelassen werden, hAMSter verlangt für die Simulation jedoch beide Parameter.

2.5.2.2 Bench

Um den Detector zu betreiben und zu testen wird wieder eine Testbench eingesetzt. Hier wird ein geeignetes Eingangssignal (`'vin'`) generiert.

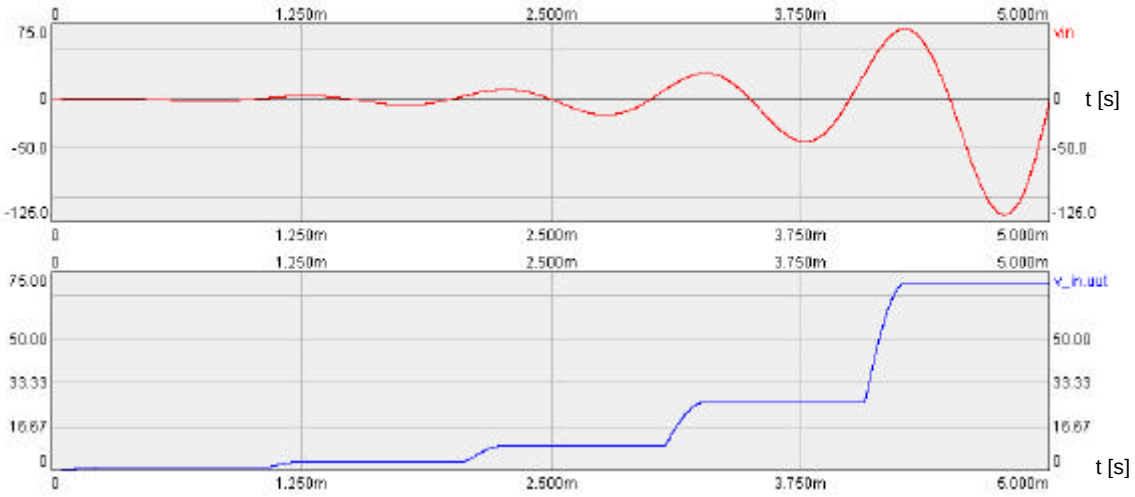
```
ENTITY bench IS END ENTITY bench;

ARCHITECTURE peak OF bench IS
     TERMINAL n1, n2 : ELECTRICAL;
     QUANTITY vin ACROSS iin THROUGH n1;
     CONSTANT Amp : REAL := 1.0;
     CONSTANT a   : REAL := 1.0E3;
     CONSTANT f   : REAL := 1.0E3;
BEGIN
     vin == Amp*EXP(a*NOW)*sin(2.0*math_pi*f*NOW);
     UUT: ENTITY peakdetector(ideal)
        PORT MAP (inp => n1, outp => n2);
END ARCHITECTURE peak;
```

Größen wie die Amplitude (`'Amp'`), den „Wachstumsparameter“ (`'a'`) und der Frequenz (`'f'`) werden als Konstanten aufgelistet und initialisiert. Die Form des Test-Eingangssignals wird wieder mit Hilfe eines Simultanous Statements formuliert. Da ein im Amplitudenausschlag sich **ständig änderndes Signal** für einen sinnvollen Test unabdingbar ist, bietet sich hier z.B. eine exponentiell anklingende Sinusschwingung an. Die sonst bei Sinus-Testsignalen übliche konstante Amplitude wird einfach zusätzlich mit einer e-Funktion (`'EXP'`) multipliziert.
Jetzt wird noch eine Instanz des Peak-Detectors namens `UUT` innerhalb der Bench vereinbart und die Ein- und Ausgänge auf die entsprechenden Anschlüsse gemapped.

2.5.3 Simulation

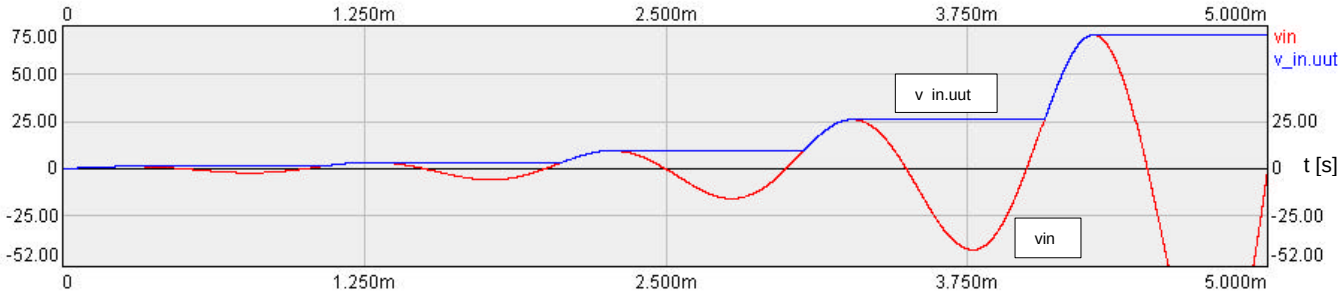
Die VHDL-Sourcecode-Datei heißt 'peak.vhd'. Um die Kurven nach der Simulation voll sichtbar zu machen, bietet sich hier die Funktion 'Beste Darstellung' an, welche bereits während der Simulation betätigt werden kann. Das Ergebnis zeigen die

beiden Grafiken. Zunächst sind Ein- und Ausgangssignal in jeweils einem Diagramm dargestellt:

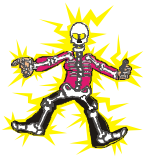


Da das Modell keine konservativen Bauelemente wie z.B. Widerstände enthält, sind die Einheiten der Ein- und Ausgangsspannungen nicht bestimmt und somit frei interpretierbar.

Nachfolgende Grafik zeigt beide Kurven zusammen in einem Diagramm. Hier kann man deutlich erkennen, dass sich der Ausgang `'v_in.uut'` nur erhöht, wenn der Eingang sein bisheriges Maximum überschreitet.

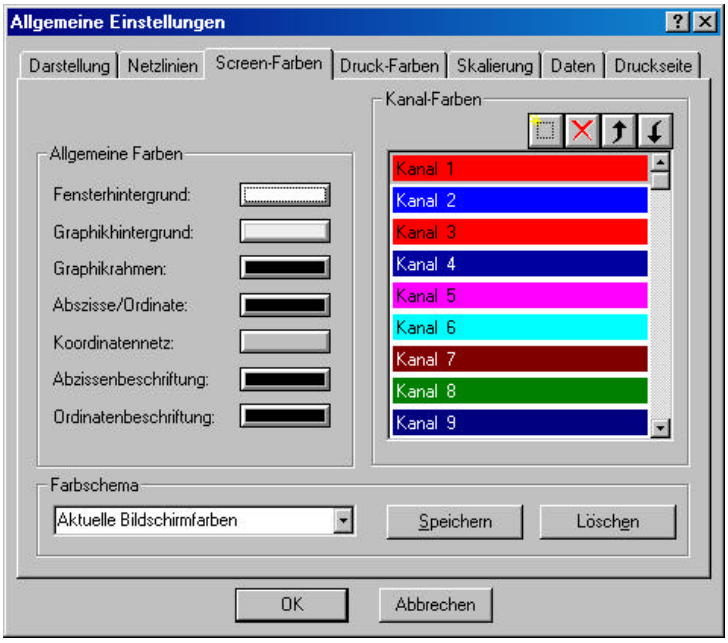


Wenn man beide Kurven nun in einem Diagramm dargestellt hat und möchte diese Ansicht für darauffolgende Simulationen beibehalten, gibt es in hAMSterView die Möglichkeit die **Konfiguration**, d.h. auch alle Einstellungen wie Skalierung etc., zu speichern. Dazu braucht man nur den Button 'Konfiguration speichern' anzuklicken. Nun sind alle Einstellungen automatisch in der Datei <modell_name>.vtc gespeichert.



Zusammenfassung hAMSter-Konfigurationsdateien: In der Datei **'*.cfg'** lassen sich die vor der Simulation eingestellten Simulationsparameter speichern; sie wird aktualisiert bei jeder Simulation. In der Datei **'*.vtc'** sind die Einstellungen für die Simulationsausgabe in hAMSterView gespeichert. Sie wird durch Betätigung von 'Konfiguration speichern' aktualisiert.

Unter 'Optionen, Einstellungen...' können alle **allgemeinen Einstellungen** für hAMSterView getätigt werden, wie beispielsweise einzelne Details der Darstellung bezüglich Achsen, Rahmen oder Ränder, Konfiguration der Netzlinien, verschiedene Skalierungsoptionen sowie detaillierte Einstellungen für Ausdrucke. Außerdem erlaubt dieser Dialog die **Farben** aller Kanäle und anderer Grafikelemente frei zu bestimmen, sowohl für den Bildschirm als auch für Ausdrucke.



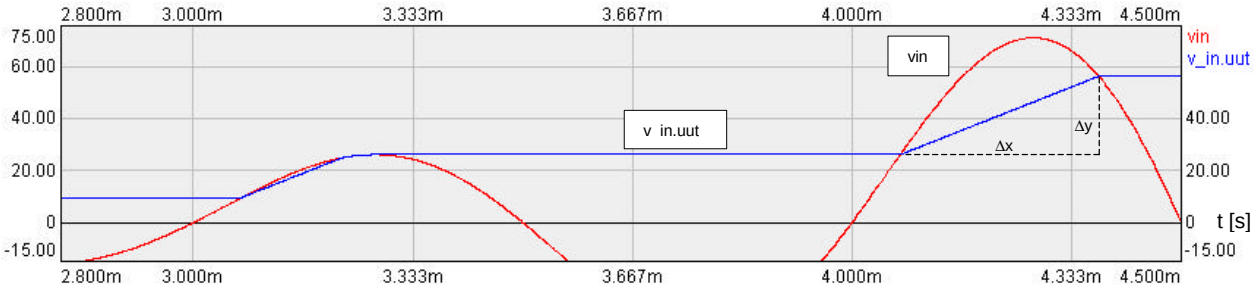
hAMSter erlaubt das **Exportieren** der Simulationsdaten in diverse ASCII-Formate, sowie in das Access™- und Excel™-Dateiformat. So können alle errechneten Simulationswerte in Tabellenform in anderen Anwendungen weiterverarbeitet werden. Das unter 'Daten speichern' erreichbare Dialog-Window offeriert für jeden Kanal eine Exportauswahl. Hier können auch Formatoptionen und Dateinamen sowie dessen Speicherort bestimmt werden.

Dieser Export umfasst lediglich die Ergebnisse in Form von Zahlenwerten. Möchte man die **Grafik** exportieren, so kann sie bequem mittels der Funktion 'Kopieren' in andere Anwendungen integriert werden.

2.5.3.1 Simulation mit langsamem Anstieg

Um den Effekt des SLEW-Attributs zu verdeutlichen, wird nun dem Parameter der maximalen Anstiegssteigung statt 10^{38} der hier **niedrige Wert** 100 000 zugewiesen. Folgende Zoom-Darstellung wählt einen Simulationsausschnitt von 2,8 bis 4,5 Millisekunden.

Beim ersten Maximum erfüllt das Modell noch fast seine Aufgabe, aber wie deutlich zu erkennen ist, bereitet schon der folgende Maximalwert Schwierigkeiten: Aufgrund des begrenzten 'max_rising_slope' kann 'v_in.uut' nicht „mitziehen“ und der neue Spitzenwert wird nicht erfasst.



ANHANG

I. Weblink- und Dokumentationen-Liste

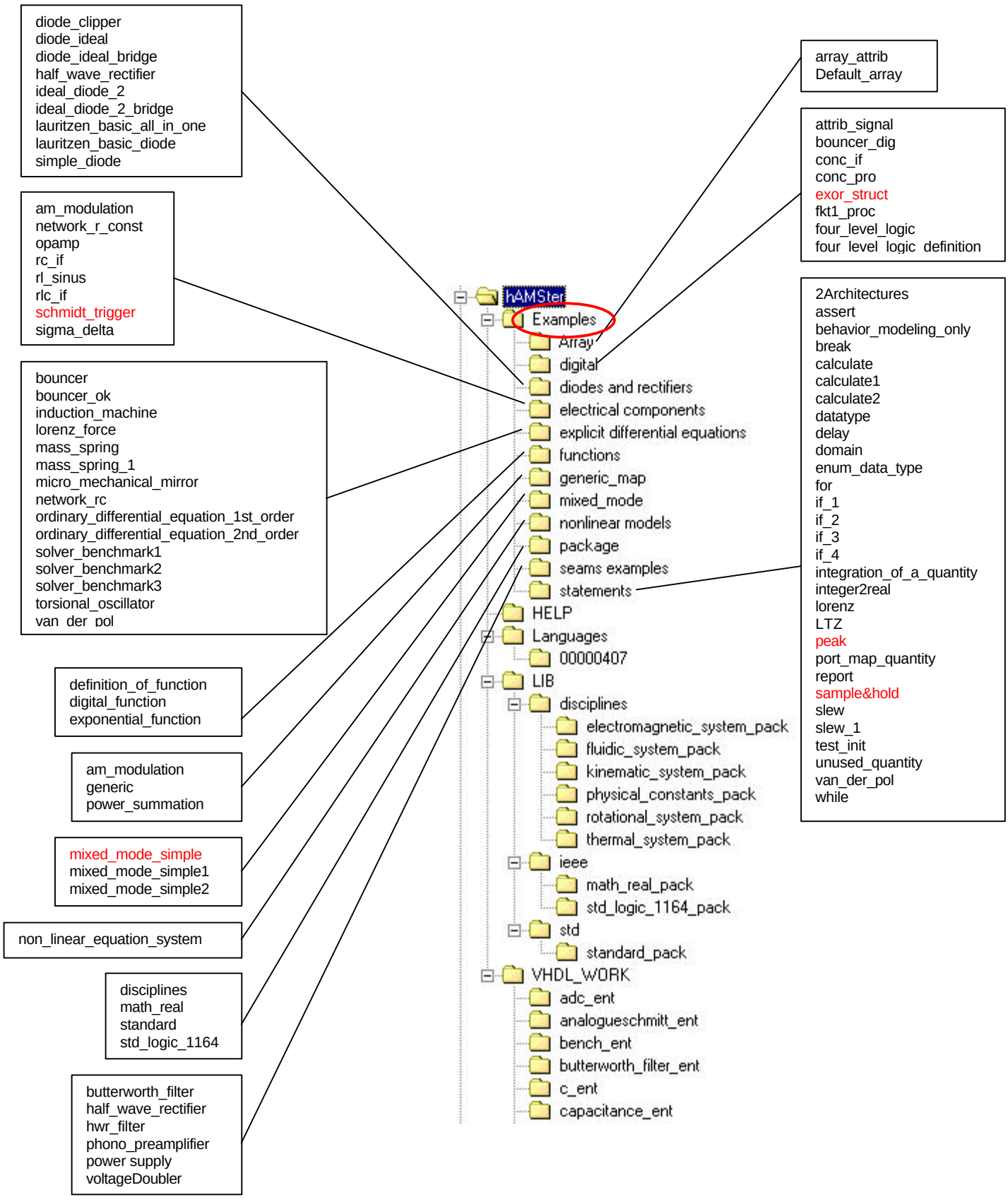
WEBLINKS

- http://www.hamster-ams.com/
- http://www.eda.org/vhdl-ams/
- http://www.ececs.uc.edu/~dpl/
- http://www.ti.informatik.uni-frankfurt.de/grimm/hybrid/
- http://www.syssim.ecs.soton.ac.uk/
- http://www.vhdl-ams.com/
- http://www.vhdl-online.de/
- http://www.ee.duke.edu/research/impact/vhdl-ams/index.html/
- http://wwwsoft.nf.fh-nuernberg.de/~stlabor/me-seminare/VHDL_AMS/index.htm/

DOKUMENTATIONEN

- Zusammenfassung VHDL-AMS FH-Nürnberg.doc
- VHDL-AMS Tutorial.pdf
- VHDL-AMS Fraunhofer.ps
- Erfahrungen mit VHDL-AMS bei der Simulation heterogener Syst.pdf
- Introduction to VHDL-AMS.html
- VHDL-AMS-Syntax.htm
- Einführung in VHDL-AMS (Folien Brodowski).pdf
- Einführung in VHDL-AMS (Ausarbeitung Brodowski).pdf

II. hAMSter: Liste aller Beispiele, Ordnerstruktur



III. CD-Inhalt

IV. Source-Code der Modellbeispiele

EXOR (Datei: exor_struct.vhd)

```
-- Modell of an exor gate consisting of 4 nand gates

-- nand gate with delay
ENTITY nand_ IS
       GENERIC (td:time);                   -- delay time
       PORT (x,y : IN BIT; z : OUT BIT);    -- connections of nand
END ENTITY nand_;

--behaviour of nand gate
ARCHITECTURE nand_behav of nand_ IS
BEGIN
       z <= x NAND y AFTER td;
END nand_behav;

-- modell of exor gate
ENTITY exor IS
END ENTITY exor;

ARCHITECTURE exor_struct_ OF exor IS
       SIGNAL a,b,c,d,e,f : BIT;
-- using of sub model nand- by default binding to entity nand_
       COMPONENT nand_
              PORT (x,y : IN BIT; z: OUT BIT);
              GENERIC (td:time);
       END COMPONENT;
BEGIN
a <='0', '1' AFTER 10 ns;                                   -- simple stimulis generation
b <='0', '1' AFTER  5 ns, '0' AFTER 10 ns, '1' AFTER 15 ns;

nand1: nand_  GENERIC MAP (0 ns) PORT MAP (a,b,c);          --instantiation of sub model nand
                                                            -- without delay
nand2: nand_  GENERIC MAP (0 ns) PORT MAP (a,c,d);
nand3: nand_  GENERIC MAP (0 ns) PORT MAP (b,c,e);
nand4: nand_  GENERIC MAP (2 ns) PORT MAP (d,e,f);          --instantiation of sub model nand
                                                            -- with delay
END;
```

EINFACHES MIXED-MODE-SYSTEM (Datei: mixed_mode_simple.vhd)

```
ENTITY flipflop IS
       PORT ( output : OUT BIT );
END;

ARCHITECTURE behav OF flipflop IS
       SIGNAL a : BIT ;
BEGIN
       a <=    '1' AFTER 0.5 ms,
               '0' AFTER 0.7 ms,
               '1' AFTER 1.0 ms,
               '0' AFTER 1.8 ms,
               '1' AFTER 2.5 ms,
               '0' AFTER 3.5 ms,
               '1' AFTER 3.7 ms,
               '0' AFTER 3.8 ms;
       output <= a;
END;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY da_converter IS
       PORT ( TERMINAL p,m : ELECTRICAL ;
       SIGNAL input : IN BIT );
END;

ARCHITECTURE behav OF da_converter IS
       QUANTITY v_in ACROSS i_out THROUGH p TO m;
BEGIN
       IF ( input= '0' ) USE
              v_in == -2.0;
       ELSE
              v_in == 2.0;
       END USE;
END;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY resistor IS
       GENERIC ( val  : REAL );
       PORT    ( TERMINAL p,m: ELECTRICAL );
END resistor;

ARCHITECTURE behav OF resistor IS
       QUANTITY u_r ACROSS i_r THROUGH p TO m;
BEGIN
```

```
        i_r == u_r / val;
END behav;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY capacitance IS
        GENERIC ( val : REAL);
        PORT    ( TERMINAL p,m : ELECTRICAL);
END;

ARCHITECTURE behav OF capacitance IS
        QUANTITY u_c ACROSS i_c THROUGH p TO m;
BEGIN
        i_c == val * u_c'dot;
END;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY rc IS
        PORT( TERMINAL p,m: ELECTRICAL );
END;

ARCHITECTURE behav OF rc IS
        TERMINAL n    : ELECTRICAL;
BEGIN
        R1:     ENTITY resistor (behav)
                GENERIC MAP (val => 1000.0) PORT MAP (p,n);
        C1:     ENTITY capacitance (behav)
                GENERIC MAP (val => 1.0e-7) PORT MAP (n,m);
END;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY bench IS
END;

ARCHITECTURE behav OF bench IS
        TERMINAL N1 : ELECTRICAL;
        SIGNAL   S1 : BIT;
BEGIN
        RC1:    ENTITY rc (behav)
                PORT MAP (n1,electrical_ground);
        DA1: ENTITY da_converter (behav)
                PORT MAP (n1,electrical_ground,S1);
        FF1: ENTITY flipflop (behav) PORT MAP (S1);
END;
```

SCHMITT-TRIGGER (Datei: schmidt_trigger.vhd)

```
LIBRARY DISCIPLINES;
LIBRARY IEEE;

USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;

ENTITY AnalogueSchmitt IS
END ENTITY AnalogueSchmitt;

ARCHITECTURE Hysteresis OF AnalogueSchmitt IS
       TERMINAL n1,n2      : ELECTRICAL;

-- declare a signal to memorise the hysteresis state:
       SIGNAL   State : REAL := 0.0;                      -- initial state is low
       QUANTITY Vin ACROSS Iin THROUGH n1;

-- declare a through branch for the output voltage source
       QUANTITY Vout ACROSS Iout THROUGH n2;

BEGIN
-- the architecture consists of two architecture statements:
--   a conditional concurrent signal assignment to implement the hysteresis
--   a simultaneous statement to implement the equation for the output source

       Vin == 5.0 * sin(2.0 * math_pi * 0.5E3 * NOW);

-- hysteresis:
       BREAK State => 0.0 WHEN Vin'above(1.0);            -- trigger event when Vin > 1.0
       BREAK State => 5.0 WHEN NOT Vin'above(2.4);        -- trigger event when Vin <  2.4

-- output voltage source equation:
       Vout == State'RAMP(10.0e-6,10.0e-6); -- the use of ramp assures that
                                            -- when a discontinuity in State arises, it is
                                            -- announced to the simulator
end architecture Hysteresis;
```

SAMPLE&HOLD (Datei: sample&hold.vhd)

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY sample_and_hold IS
       GENERIC (Tsampl   : REAL := 0.05e-3;
                delay    : REAL := 0.0);
       PORT (TERMINAL inp, outp : ELECTRICAL);
END ENTITY sample_and_hold;

ARCHITECTURE ideal OF sample_and_hold IS
       QUANTITY vinsh  ACROSS              inp;
       QUANTITY v_in   ACROSS i_in THROUGH outp;
BEGIN
       v_in == vinsh'ZOH(Tsampl, delay);
END ARCHITECTURE ideal;

-- Sample ZOH function
LIBRARY DISCIPLINES, IEEE;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;

ENTITY bench IS END ENTITY bench;

ARCHITECTURE sample OF bench IS
       TERMINAL n1, n2     : ELECTRICAL;
       QUANTITY vin ACROSS iin THROUGH n1;
       CONSTANT f   : REAL := 1.0E3;
BEGIN
       vin == sin(2.0 * math_pi * f * NOW);
UUT: ENTITY sample_and_hold(ideal)
       GENERIC MAP (Tsampl => 0.1e-3, delay => 0.10e-3)
       PORT MAP (inp => n1, outp => n2);
END ARCHITECTURE sample;
```

PEAK-DETECTION (Datei: peak.vhd)

```
LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

ENTITY peakdetector IS
       PORT (TERMINAL inp, outp : ELECTRICAL);
END ENTITY peakdetector;

ARCHITECTURE ideal OF peakdetector IS
       QUANTITY vin  ACROSS              inp;
       QUANTITY v_in ACROSS i_in THROUGH outp;
BEGIN
       v_in == vin'SLEW(1.0E38, -1.0e-38);
END ARCHITECTURE ideal;

-- SLEW statement for quantities; only the peaks of a sine signal will be adopt to the
-- quantity
LIBRARY DISCIPLINES, IEEE;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;

ENTITY bench IS END ENTITY bench;

ARCHITECTURE peak OF bench IS
       TERMINAL n1, n2     : ELECTRICAL;
       QUANTITY vin ACROSS iin THROUGH n1;
       CONSTANT Amp : REAL := 1.0;
       CONSTANT a   : REAL := 1.0E3;
       CONSTANT f   : REAL := 1.0E3;
BEGIN
       vin == Amp * EXP(a * NOW)*sin(2.0 * math_pi * f * NOW);
UUT: ENTITY peakdetector(ideal)
       PORT MAP (inp => n1, outp => n2);
END ARCHITECTURE peak;
```

V. Quellenangaben

[1] Zusammenfassung VHDL-AMS FH-Nürnberg.doc (auf CD)

[2] Schaltungsdesign mit VHDL (siehe unten)

ALLGEMEINE QUELLEN:

- **Schaltungsdesign mit VHDL** [Lehmann/Wunder/Selz; Franzis-Verlag] (auch auf CD)
- **The VHDL Reference** (Including VHDL-AMS) [Heinkel/Padeffke/Haas/Buerner/Braisz/Gentner/Grassmann; Wiley Verlag]
- sowie Informationen aus mehreren von den vorgestellten **Dokumentationen**, die auch im Anhang [I.] aufgelistet sind und sich auf der CD befinden [III.].

VI. Impressum

Modellierung und Simulation von Mixed-Signal-Systemen mit VHDL-AMS

Eine praktisch orientierte Einführung in das Arbeiten mit VHDL-AMS und dem Simulator hAMSter.

Dieses Tutorial wurde im Rahmen einer **Studienarbeit** für den Fachbereich Informationstechnik der FHTE erstellt.

Betreuender Professor:

Professor Dr.-Ing. Gerald Kampe

Autoren:

Mirko Pfitzner, NT7, mipfit02@fht-esslingen.de
Matrikel-Nr.: 721187

Adrian Wöhr, NT7, adwoit00@fht-esslingen.de
Matrikel-Nr.: 721362

Fachhochschule Esslingen, Hochschule für Technik, 2001
Wintersemester 2001/2002
Fachbereich: Informationstechnik
Studiengang: Nachrichtentechnik
Zeitraum: 01.10.2001 bis 07.01.2002
Prüfer-Nr.: 0119

Wir bedanken uns bei Herrn Professor Dr.-Ing. Gerald Kampe für die Unterstützung und für die gute Betreuung während der Durchführung der Studienarbeit.