1. Die Grundlagen zum Hardwaredesign

Zur Vorlesung Rechenanlagen

SS 2019

1.1 Grundlegende Begriffe

Zur Vorlesung Rechenanlagen

SS 2019

Notationen

$\mathbb{N}$	Bezeichne die natürlichen Zahlen	-- POSITIVE*)
$\mathbb{N}_0$	Bezeichne die natürlichen Zahlen mit 0	-- NATURAL*)
$\mathbb{Z}$	Bezeichne die ganzen Zahlen	-- INTEGER*)
$\mathbb{Q}$	Bezeichne die rationalen Zahlen	
$\mathbb{R}$	Bezeichne die reellen Zahlen	-- REAL*)

-- WüHDL

$\mathbb{B} := \{0,1\}$ Bezeichne die Menge der „booleschen“ Werte 0 (false,low) und 1 (true,high)

```
-- BOOLEAN (true,false) oder
-- BIT ('0','1')
```

-- WüHDL

*) mit Einschränkungen!

Mengen in WüHDL

Neben diesen Mengen haben beliebige endliche Mengen in der Informatik eine besondere Bedeutung. WüHDL ist als VHDL Subset und damit als Sprössling der Programmiersprache ADA "streng getypt". Endliche Mengen lassen sich als Typen durch Aufzählen der Elemente definieren:

TYPE Alphabet IS (a_1,a_2,...,a_n);

-- a_i Bezeichner für das i-te Zeichen

TYPE Color IS (red,blue,green);

Vordefinierte Typen dieser Art sind z.B.

TYPE CHARACTER IS (...,'A','B',...,'a','b',...,'0','1',...);

-- alle ASCII Zeichen

TYPE BIT IS ('0','1');

TYPE BOOLEAN IS (FALSE,TRUE);

Durch die Enumerationsreihenfolge ist zugleich auch eine Ordnung auf der Menge definiert, von Typbezeichner'low nach Typbezeichner'high.

1.1.1 Relationen und Funktionen

Seien A und B Mengen. Dann ist die Menge $A \times B$ die Menge aller geordneten Paare, die man aus Elementen aus A und B bilden kann:

$$A \times B := \{ (a,b) \mid a \in A \text{ und } b \in B \}$$

Eine Teilmenge $R \subseteq A \times B$ heißt **Relation**. Wir schreiben auch

aRb statt $(a,b) \in R$ (Infixschreibweise)

und nennen

$A := Q(R)$ die **Quelle** oder den **Quellbereich** der Relation,

$B := Z(R)$ das **Ziel** oder den **Zielbereich** der Relation.

Für eine Teilmenge $X \subseteq A$ sei

$$R(X) := \{b \mid (a,b) \in R \text{ und } a \in X \}$$

das **Bild** von X unter R

und es sei

$R^{-1} := \{(b,a) \mid (a,b) \in R\}$ die **Umkehrrelation** zu R.

Funktionen

Identifizieren wir einelementige Mengen *{x}* mit dem Element *x*, so ergibt dies für Elemente *x* aus *A*

$$R(x) := R(\{x\}) := \{b \mid (x,b) \in R\}$$

Eine Relation *R* heißt **(partielle) Funktion**

$: \Leftrightarrow \ \forall a \in Q(R): \#R(a) \leq 1$ -- #M := Anzahl der Elemente in M (Kardinalität)

Sei *R* eine partielle Funktion: Wir nennen

$D(R) := \{ a \mid \#R(a) = 1\}$ den **Definitionsbereich** von *R* und

$W(R) := R(D(R))\ (= R(Q(R))\ )$ den **Wertebereich** von *R*

R heißt **totale Funktion** oder **Abbildung**

$: \Leftrightarrow$ *R* ist Funktion und $D(R) = Q(R)$

Vereinbarungen

Für eine Funktion *R* und ein Element *a* ist offenbar *R(a)* einelementig oder leer.

Ist *R(a)* leer sagen wir auch **R(a) ist undefiniert**.

Ist *R* leer nennen wir *R* eine **überall undefinierte Funktion**, (meist notiert als Ω).

Funktionen notieren wir fortan als kleine oder griechische Buchstaben, Relationen durch Operationssymbole oder große Buchstaben.

Schreibe $f{:}A \to B$ für eine (partielle) Funktion $f \subseteq A \times B$.

Definition

Sei $f: A \to B$ eine (partielle) Funktion.

Dann ist die Einschränkung von f auf die Menge X, $f|_X$, definiert durch $D(f|_X) := D(f) \cap X$

und

$$f|_X(a) := \begin{cases} f(a) : a \in D(f) \cap X \\ \infty \quad : \text{sonst} \end{cases}$$

Definition

Eine partielle Funktion $f: A \to B$ heißt

- **injektiv:** $\Leftrightarrow$ ($f(a)$, $f(b)$ def. und $f(a) = f(b)$) $\Rightarrow a=b$
- **surjektiv:** $\Leftrightarrow f(A) = B$
- **bijektiv:** $\Leftrightarrow f$ ist total, injektiv und surjektiv

Definition

Sei $f: A \to B$ eine partielle Funktion. Dann heißt $g: A \to B$ eine Erweiterung von f

$$f \subseteq g :\Leftrightarrow g|_{D(f)} = f$$

Weitere Notationen:

$F(A,B)$ Menge der Funktionen von A nach B

$Abb(A,B)$ Menge der totalen Funktionen von A nach B

Relationen auf einer Menge *M*

Definition

Eine Relation *R* auf einer Menge *M*, d.h. $R \subseteq M \times M$, heißt

- **reflexiv** $:\Leftrightarrow \forall a \in M: aRa$
- **symmetrisch** $:\Leftrightarrow \forall a,b \in M: aRb \Rightarrow bRa$
- **antisymmetrisch** $:\Leftrightarrow \forall a,b \in M: aRb$ und $bRa \Rightarrow a = b$
- **transitiv** $:\Leftrightarrow \forall a,b,c \in M: aRb$ und $bRc \Rightarrow aRc$

Eine reflexive, transitive und symmetrische Relation heißt **Äquivalenzrelation**.

Eine reflexive, transitive und antisymmetrische Relation heißt **partielle Ordnung**.

Beispiele:

A = Menge aller Menschen B = Menge aller Ortschaften

$$R \subseteq A \times B := \{(a,b) \mid a \text{ wohnt in } b\}$$

R ist keine Funktion (Neben bzw. Zweitwohnsitze)

A = B = Menge aller Ortschaften

$$R \subseteq A \times B := \{(a,b) \mid a \text{ zu Fuß erreichbar von } b\}$$

R ist
- reflexiv
- symmetrisch
- transitiv

R ist Äquivalenzrelation

Beispiele:

A= Menge aller PKW Typen
B={0,1}

$$R = \{(a,0) \mid V_{max}(a) \leq 110 km/h\}$$
$$\cup \{(a,1) \mid V_{max}(a) \geq 160 km/h\}$$

R ist eine Funktion, aber nicht total.

Operationen auf Relationen

Seien R,S Relationen. Dann ist

$$R \circ S \text{ definiert} :\Leftrightarrow Z(R) = Q(S)$$

und wir setzen:

$$R \circ S := \{(a,c) \mid \exists b : (a,b) \in R \wedge (b,c) \in S\}$$

Sei $R \subseteq A \times A$ Wir setzen

$$R^0 := \{(a,a) \mid a \in A\}$$ Identität auf A

und $R^1 := R$

$$R^{n+1} := R^n \circ R$$

$$R^* := \bigcup_{n \in \mathbf{N_0}} R^n$$ heißt reflexiver, transitiver Abschluss

1.1.2 Zeichen und Worte

Zeichen: Elemente einer endlichen Menge *A*
Wir nennen *A* auch Alphabet

Worte: (endliche) Folgen von Zeichen

$$A^+ := \{w \mid w : [1 : n] \to A, n \in \mathbf{N}\}$$

heißt Menge der Worte über dem Alphabet *A*. Für *i* ist *w(i)* das *i*-te Zeichen im Wort *w*.
Wir nennen *n* auch die Länge von *w*, und schreiben

$$|w| = n$$ -- WüHDL: w'length

Demnach ist ein Wort *w* also stets eine Abbildung

$$w : [1 : |w|] \to A$$

Worte, Vektoren und Arrays

In Programmiersprachen werden Worte im Sinne dieser Definition in vielerlei Hinsicht benutzt. Wir illustrieren dies an unserer Beispielsprache WüHDL:

- **Strings**: Betrachtet man Folgen von Zeichen mit beliebigen, nichtnegativen Längen, dann kann man diese zusammenfassen in einem Typ String. In WüHDL gibt es einen vordefinierten Typ dieser Art, allerdings nur für Folgen von CHARACTER:

```
TYPE STRING IS ARRAY(POSITIVE RANGE <> ) OF CHARACTER;
```

Hier ist der Bereich der Folge undefiniert. Konkrete Deklarationen von Strings müssen den Bereich aber spezifizieren:

```
STRING(1 TO 5);
```

Es gibt auch vordefinierte Strings über Bits, diese können aber mit Position 0 als Zuordnungsposition beginnen und heißen

```
TYPE BITVECTOR IS ARRAY(NATURAL RANGE <> ) OF BIT;
```

Worte, Vektoren und Arrays

➢ **Vektoren**:

Betrachtet man Folgen von Zeichen mit stets fester Länge *n*, so kann man diese wie Vektoren aus einem Vektorraum der Dimension *n* auffassen. Solche Vektormengen deklariert man als

```
TYPE Xvector32 IS ARRAY(0 TO 31) OF X;
```

Operationen auf Worten

Konkatenation (Hintereinanderreihung)

$$conc : A^+ \times A^+ \to A^+$$

mit $conc(u,v) : [1 : |u| + |v|] \to A$

und $$conc(u,v)(i) = \begin{cases} u(i) : 1 \le i \le |u| \\ v(i - |u|) : |u| < i \le |u| + |v| \end{cases}$$

Einfachere Infixschreibweise: $u \cdot v := conc(u,v)$

-- WüHDL: u & v

Operationen auf Worten

Teilwort

$$sstr : A^+ \times \mathbf{N} \times \mathbf{N} \rightarrow A^+$$

mit $sstr(u,i,j) : [1 : j - i + 1] \rightarrow A$

und $sstr(u,i,j)(l) = u(i + l - 1)$

sstr(u,i,j) ist undefiniert, wenn $i > j$ oder $j > |u|$

Einfachere Schreibweise: *u[i:j] := sstr(u,i,j)*

Anmerkung: *u[i:j]* liefert *u(i)·…·u(j)* -- WüHDL: u(i TO j)

Leeres Wort $\varepsilon : \{\} \rightarrow A \quad (\varepsilon : [1 : 0] \rightarrow A)$

Anmerkung: $u \cdot \varepsilon = u = \varepsilon \cdot u$ -- WüHDL: Kein Pendant!

Die Algebra der Worte

A^+ zusammen mit dem leeren Wort bildet unter der Konkatenation ein *Monoid*

$$A^* := A^+ \cup \{\varepsilon\}$$

Axiome:

- $\forall a,b,c \in A^*: \; a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- $\forall a \in A^*: \; a \cdot \varepsilon = a = \varepsilon \cdot a$

Weitere Operationen auf Worten

Fortsetzung einer binären Operation auf Mengen:

Seien $U, V \subseteq A^*$ dann setze

$$U \cdot V := \{a \cdot b \mid a \in U, b \in V\}$$

Definition

Seien $v, w \in A^*$

v heißt **Präfix** von w: $\Leftrightarrow \exists\, u \in A^*: v \cdot u = w$

v heißt **Suffix** von w: $\Leftrightarrow \exists\, u \in A^*: u \cdot v = w$

v heißt **Infix** von w $\Leftrightarrow \exists\, a, b \in A^*: a \cdot v \cdot b = w$

Ein Xfix v von w heißt **echt** $:\Leftrightarrow v \neq w$

1.1.3 Kodierungen

Häufig muss man Zeichen selbst über anderen Zeichensätzen darstellen.

Definition

Gegeben sei $\varphi : A \to B^*$

Dann ist die Fortsetzung von φ auf A^*

$$\varphi : A^* \to B^*$$

gegeben durch

$$\varphi(w) := \varphi(w(1)) \cdot \cdots \cdot \varphi(w(|w|)), \text{ sowie } \varphi(\varepsilon) := \varepsilon$$

Anmerkung: Eine solche Fortsetzung nennt man auch Homomorphismus, da

$$\varphi(u \cdot v) = \varphi(u) \cdot \varphi(v), \text{ und } \varphi(\varepsilon) = \varepsilon$$

Kodierungen -- ff

Definition

Eine Abbildung $\varphi : A \to B^*$ heißt **Kodierung** dann und nur dann, wenn die Fortsetzung $\varphi : A^* \to B^*$ injektiv ist.

Bezeichnung: Wir nennen $X \subseteq A^*$ **präfixfrei**

$\forall a, b \in X$: a ist nicht Präfix von b

Lemma:

$\varphi : A \to B^*$, injektiv, liefert eine Kodierung, wenn

(a) $\exists n \in \mathbf{N}\ \forall a \in A : |\varphi(a)| = n$ Blockcode

(b) $\varphi(A)$ ist präfixfreie Menge präfixfreier Code

1.1.4 Sprachen

Worte sind zunächst Zeichenreihen ohne jede Bedeutung.
Man bildet daraus **Sprachen,** indem man sagt

- welche Zeichenreihen (Worte) Bedeutungen haben
- welche Aneinanderreihungen erlaubt sind.

D.h. man legt die Struktur, die **Syntax**, fest.

Danach definiert man die Bedeutung

- der Worte
- und der zulässigen Aneinanderreihungen,

d.h. man legt auf der Struktur dann die Bedeutung, die **Semantik**, fest.

Syntaxdefinition in BNF

Um die Syntax von Sprachen festzulegen, benutzen wir selbst eine einfache Sprache, die wir natürlich über unserem alphanumerischen Zeichensatz definieren müssen: die **Backus Naur Form** (BNF).

Die Grundlage bilden Symbole, die interpretiert werden als Wortmengen, d.h. als Teilmengen von A*. Wir nennen sie **Wortmengenbezeichner** (Nichtterminale) und lassen dazu beliebige Buchstabenfolgen, eingefasst durch < >, zu.

Ferner benutzen wir das **Definitionssymbol** "::=". Mit diesem Symbol definieren wir die Wortmenge zu einem Wortmengenbezeichner unter Zuhilfenahme anderer Wortmengenbezeichner und Operatoren.

Anmerkung: Da wir zur Definition auch Zeichen benutzen müssen und diese meist aus dem gleichen Zeichensatz stammen, über dem die Zielsprache definiert wird, kennzeichnen wir die Zeichen der Zielsprache durch fette Schrift.

Syntaxdefinition -- ff

Die eigentlichen Wortmengen definieren wir nun durch Ausdrücke über Wortmengenbezeichnern, Zeichen und folgenden
Operatoren:

Hintereinanderschreiben: Konkatenation der Wortmengen

| Vereinigung von zwei Wortmengen

{ } Klammern für Ausdrücke

* *Abschluss einer Wortmenge *X*, d.h. man betrachtet die Menge aller Worte, die man durch Aneinanderreihung von Worten aus *X* bilden kann, einschliesslich des leeren Worts.

+ +Abschluss: Wie *Abschluss aber ohne leeres Wort.

Häufig benutzt man noch weitere Operatoren:
So bedeutet etwa { **a** | **ab** } **c**, die Menge {ac, abc} = {a,ab} {c}.
Das Zeichen **b** folgt also **a** "optional".
Dies kann man auch durch einen weiteren Operator [ ] ausdrücken, mit

X [Y] gleichbedeutend mit {X | XY }.

Syntaxdefinition -- ff

Beispiele: Die Syntax von ganzen Dezimalzahlen kann man in BNF direkt wie folgt angeben

<IntegerKonstante>
::= [**+** | **-**] {**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** }$^{+}$

Die Syntax von Bitvektorkonstanten sogar noch einfacher durch

<BitvektorKonstante> ::= **"**{**0** | **1** }$^{+}$**"**

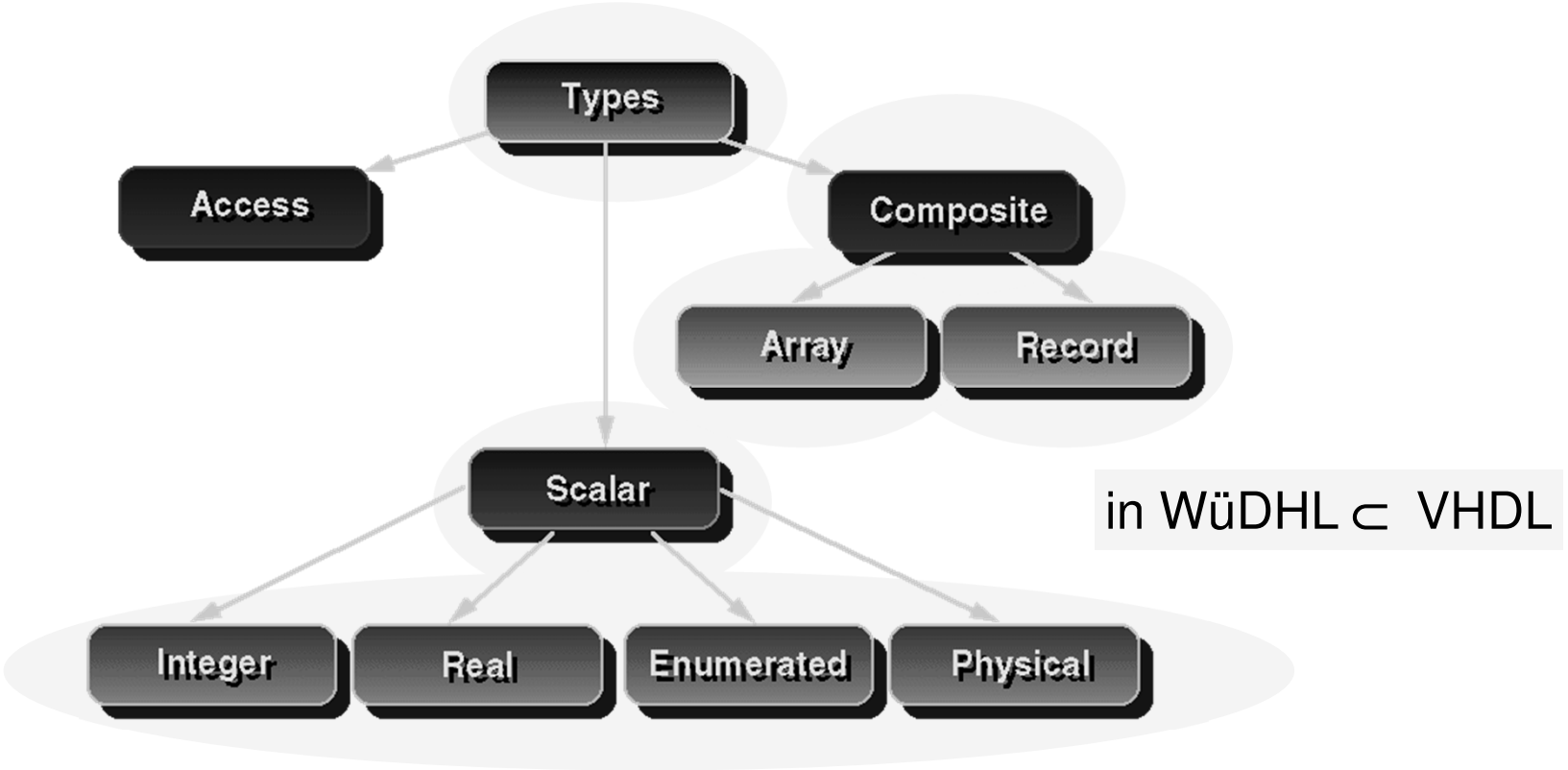
Arithmetische Ausdrücke kann man durch

<ArithExpr> ::= <ArithExpr> {**+** | **-** } <Term> | <Term>
<Term> ::= <Term> {***** | **/** } <Factor> | <Factor>
<Factor> ::= <ValueIdent> | **(** <ArithExpr> **)**
<ValueIdent> ::= <Constant> | <Identifier> | <ArrayExpr>

1.1.5 Typen in WüHDL

Typen stehen für Wertebereiche, über denen andere Objekte Werte annehmen können. Sie werden in Typdeklarationen definiert.

Wir wollen die wichtigsten Typen in WüHDL bzw. VHDL zusammenfassen:



Skalare Datentypen

Die Enumerationstypen BIT, BOOLEAN, CHARACTER haben wir schon behandelt.

Zu weiteren vorgegebenen Typen

INTEGER:

Garantierter Bereich **bei jeder VHDL Implementierung:**
-2.147.483.647 bis +2.147.483.647

REAL **:**

Garantierter Bereich
-1.0e38 bis +1.0e38
mit wenigstens 6 Dezimalstellen Genauigkeit

Skalare Datentypen: physikalische Typen

Physikalische Typen (physical) sind Skalare Typen, die zusätzlich mit Einheiten versehen sind.

Zur Definition eines physikalischen Typs gehört

- ein **Bereich** eines anderen skalaren Typs
- eine Definition der **Einheiten**.

Beispiel

```
TYPE TIME IS RANGE -9223372036854775807 TO +9223372036854775807
UNITS
        fs;                 -- femtosecond
        ps  = 1000 fs;      -- picosecond
        ns  = 1000 ps;      -- nanosecond
        us  = 1000 ns;      -- microsecond
        ms  = 1000 us;      -- millisecond
        sec = 1000 ms;      -- second
        min = 60 sec;       -- minute
        hr  = 60 min;       -- hour
END UNITS;
```

```
Process (X)
  variable a: TIME;
begin
  a := 1;  -- bad
  a := 1 ps;  -- OK
end process;
```

Vordefiniert **in WüHDL ist nur der physikalische Typ** TIME

Vordefinierte Typen und Funktionen

Package Standard

```
PACKAGE STANDARD IS
    TYPE BOOLEAN IS (FALSE, TRUE);
    TYPE BIT IS ('0', '1');
    TYPE CHARACTER IS (
                    NUL, SH, STX, EOT, ENQ, ACK, BEL,
                                ....
                                'x', 'y', 'z', '{', '|', '}', '~', DEL );
    TYPE SEVERITY_LEVEL IS
          (NOTE, WARNING, ERROR, FAILURE);
    TYPE INTEGER IS RANGE -2147483648 TO +2147483647;
    TYPE REAL IS RANGE -0.179769313486 2E+309 TO
    +0.179769313486 2E+309;
    TYPE TIME IS ….
```

Package Standard (2)

```
FUNCTION NOW  RETURN  TIME;
SUBTYPE NATURAL  IS INTEGER RANGE 0 TO INTEGER'HIGH;
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;
TYPE STRING IS ARRAY ( POSITIVE RANGE <> ) OF
CHARACTER;
TYPE BIT_VECTOR IS ARRAY ( NATURAL RANGE <> ) OF BIT;
END STANDARD
```

- Die Realisierung des Standard Packages ist versteckt. Es ist immer vorhanden.
- Standardisiert ist auch die IEEE Library, die auf dem Standard Package aufsetzt, und weitere Packages bereitstellt.

1.1.6 Was ist Hardware?

Bevor wir uns weiter mit der Einführung von Begriffen und einer Hardwarebeschreibungssprache beschäftigen, sollten wir uns zunächst diese Frage stellen!

Definition:

Hardware besteht aus einer Menge B parallel arbeitender **Bausteininstanzen** und einer Menge S von **Signalen**, die der Kommunikation zwischen den Bausteininstanzen und mit der Außenwelt dienen.

Eine Bausteininstanz $b \in B$ wirkt auf Signale über ihre Ports $P(b)$, ein Signal $s \in S$ verbindet eine Menge $P(s)$ von Ports.

Demnach ist die Menge aller Ports

$$P = \bigcup_{b \in B} P(b) = \bigcup_{s \in S} P(s)$$

wobei $\forall\, b,b' \in B: b \neq b' \Rightarrow P(b) \cap P(b') = \emptyset$ --Partition von P

und $\forall\, s,s' \in S: s \neq s' \Rightarrow P(s) \cap P(s') = \emptyset$ --Partition von P

Bausteine

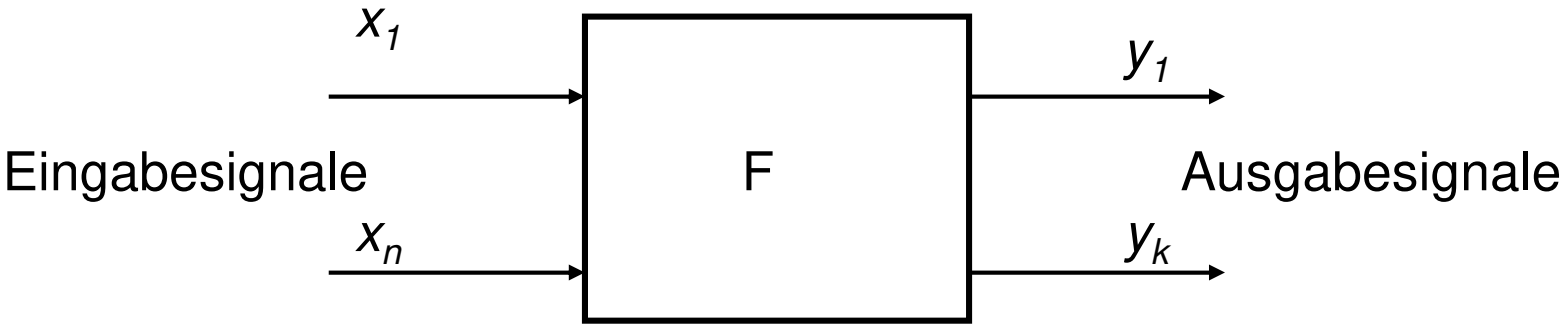
Man wird natürlich viele Bausteininstanzen benutzen, die das gleiche Verhalten haben (Transistoren, Gatter, Latches, ...). Will man ein System über Bausteininstanzen beschreiben, so ist es sinnvoll, nicht für jede Instanz eines Bausteins Ports und Verhalten zu definieren, sondern Bausteine in ihren Eigenschaften (Ports, Verhalten) allgemein zu definieren, und dann Instanzen von ihnen zu benutzen.

Wir werden sehen, dass WüHDL dazu 3 Mechanismen bereitstellt:

- entity: Definition einer Bausteinschnittstelle (Ports).
- component: Definition eines Bausteins zur Benutzung.
- Instanziierungen: Erzeugung von Instanzen von Komponenten und deren Verdrahtung durch Anweisungen.

Systeme und Signale

Ein System ist eine Ansammlung von Komponenten die in ihrem Verhalten zueinander in Wechselwirkung stehen. Das Verhalten des Systems selbst ist seine Wechselwirkung mit seiner Umgebung.



System als Black Box: Verhalten $y = F(x)$

Systeme und Signale

Was sind Signale?

Signale sind Darstellungen von Mitteilungen durch zeitliche Veränderung physikalischer Grössen. Sie liegen an physikalischen Orten vor, haben eine physikalische Repräsentation, können zwischen Komponenten kommuniziert werden und ggf. als Ein- oder Ausgabe des Systems beobachtet werden.

In der Regel unterliegt der Wert eines Signals einer Abstraktion.

Was ist ein Signalwert?

Allgemein betrachtet, ist ein **Signalwert (Waveform)** eine Abbildung

$$\beta(s): T \rightarrow V,$$

wobei T eine total geordnete Menge ist, die für die Zeit steht, und V eine Menge ist, die für die Werte steht.

Digitale Signale

Betrachte einen Signalwert (Waveform)

$$\beta(s): T \rightarrow V,$$

eines Signals *s*.

Ist *T* abzählbar, dann nennt man *s* auch **zeitdiskret**,

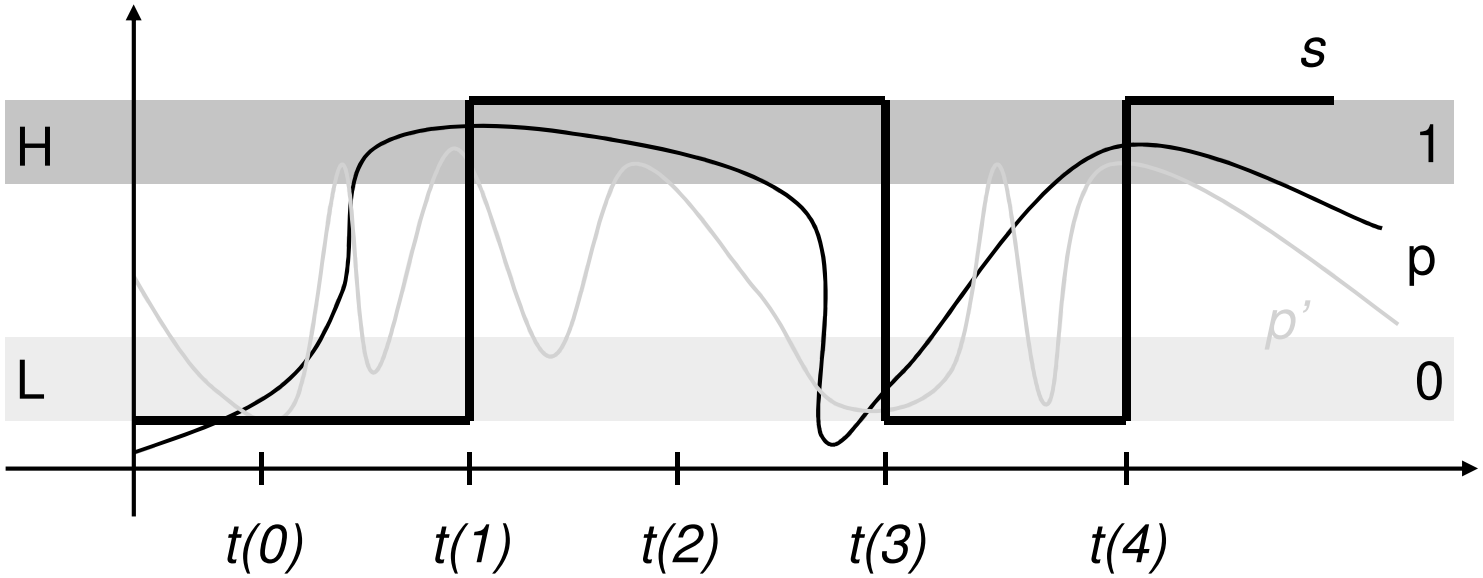
ist *V* abzählbar, nennt man *s* auch ein **wertdiskretes** Signal.

Ein **digitales Signal** ist zeit- und wertediskret mit endlichem Wertebereich. Wir betrachten zunächst nur Signalwerte der Form

$$\beta(s): \mathbf{N_0} \rightarrow V$$

-- in der Physik betrachtet man hingegen $\beta(s): \mathbf{R} \rightarrow \mathbf{R}$

Abstraktion zu einem digitalen Signal



$$s(i) := \begin{cases} 1 & \text{falls} \quad p(t(i)) \in H \\ 0 & \text{falls} \quad p(t(i)) \in L \\ \infty & \text{sonst} \end{cases}$$

p' liefert die gleiche Abstraktion als digitales Signal!

Klassifikation digitaler Systeme

Ein digitales System ist ein System, dessen Verhalten über digitalen Signalen beschrieben ist.

Es heißt **kombinatorisch** $:\Leftrightarrow \forall t: y(t) = F(x(t))$

und **sequentiell** $:\Leftrightarrow \forall t: y(t) = F(x(0:t))$

Wir werden uns im Verlauf der Vorlesung zunächst mit dem Entwurf kombinatorischer und dann mit dem Entwurf sequentieller Systeme beschäftigen.