EDIF

Electronic Design Interchange Format

Version 2 0 0

Holger Willmer (3847690)

29.9.1994

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist EDIF?

EDIF (Electronic Design Interchange Format) ist keine Programmiersprache, sondern lediglich ein Format, welches zur Übertragung von Designdaten zwischen verschiedenen Anwendern dient. Diese Übertragung erfolgt mittels eines Textformates, dessen Syntax sehr einfach ist.

1.2 Eigenschaften

EDIF besitzt atomare Datentypen wie Strings, Signale, Ports, Schichten, Zahlen und Bezeichner. Aus ihnen werden die komplexen Datenstrukturen (Listen) zusammengesetzt.

2 Die verschiedenen Level

2.1 Level 0

Dies ist die unterste Ebene. Daten werden hier nur durch statische Ausdrücke beschrieben.

2.2 Level 1

Dieser Level wird dadurch charakterisiert,daß er, aufbauend auf Level 0, die Möglickkeit bietet, das dynamische Verhalten beschreiben zu können. Dies wird durch die Erweiterung der Konstrukte von Level 0 erreicht. Es ist möglich, Funktionen zu verwenden, welche Funktionswerte aus den Wertebereichen boolean, float, string und FigureGroup (siehe später) liefern. Weiterhin sind einfache Schleifen mit Hilfe einer repeat-Anweisung realisierbar.

2.3 Level 2

Ein wichtiger Bestandteil dieser Ebene ist die Möglichkeit, prozedurale Vorgänge beschreiben zu können. Eine Level 2 Beschreibung wird im wesentlichen verwendet, um einen Datentransfer der Designdaten zu einer 'high-level' CAD Datenbank zu realisieren.

3 Struktur einer EDIF-Datei

3.1 Regeln der Meta-Syntax

- In der Spezifikation von EDIF werden Nichtterminale in *Italic*, Terminale in **fett** und smbolische Konstanten in normaler Schrift (Roman) gedruckt.
- Optionale Konstrukte werden durch [ ] geklammert
- Beliebig oft wiederholbare Konstrukte werden durch { } geklammert
- Bezeichner müssen definiert werden, bevor auf sie verwiesen wird, bzw. bevor sie benutzt werden.
- Durch ... wird eine geschlossene Folge benannt. Beispiel: A...Z benennt alle Großbuchstaben
- Ein Bezeichner erhält als Postfix enweder den Zusatz Def oder Ref, je nachdem, ob er definiert, oder auf ihn verwiesen werden soll.

3.2 Das EDIF-File

(EDIF EDIFFileNameDef
status { Hier werden Angaben über die verwendete EDIF–Version, den EDIF–Level, und den Autor gemacht.}

{ keyword } { Hier können neue Schlüsselwörter definiert werden (siehe Kapitel 4.2) .}

{ design } { Name des Designs.}

{ library } { Definition einer eigenen Bibliothek. }

{ external } { Referenz auf eine externe Bibliothek. }

[rename] { Umbenennungsblock }

{ comment } { Kommentarblock }

{ cell } { Einzelheiten siehe Kapitel 3.3 }

)

3.3 Aufbau eines Zellblocks

(CELL cellNameDef

cellType	{ Typ der Zelle. }
[status]	{ Autor des Zellblocks. Version und Level werden aus der EDIF–Datei übernommen}
[rename]	{ Umbenennungsblock }
[viewMap]	{ Beziehungen }
{ comment }	{ Kommentare }
{ property }	{ Fähigkeiten/Eigenschaften (siehe Kapitel 4.4 }
{ view }	{ Einzelheiten siehe Kapitel 3.4 }

)

Tabelle 1: Zellsichten und ihre Verwendbarkeit

Sicht	Verwendbarkeit
physikalische Sicht	Masken-Layout
topologische Sicht	symbolisches uns schematisches Layout, Netzlisten
Verhaltenssicht	Testdaten
dokument. Sicht	Dokumentation
Fremdsicht	Freie Verwendbarkeit für den Benutzer

3.4 Definition einer Zellsicht

(VIEW

ViewType	{ Typ der Sicht (siehe unten) }
ViewNameDef	{ Name der Sicht }
[status]	{ Autor der Sicht (Level und Version werden aus der EDIF–Datei übernommen) }
[rename]	{ Umbenennungsblock }
{ comment }	{ Kommentarblock }
{ property }	{ Fähigkeiten/Eigenschaften }
[contents]	{ Einzelheiten siehe Kapitel 3.5 }
interface	{ Einzelheiten siehe Kapitel 3.8 }

)

ViewType gibt den Typ der Sicht an. Zur Verfügung stehen die Typen MASKLAYOUT, NETLIST, SCHEMATIC, SYMBOLIC, BEHAVIOUR, DOCUMENT und STRANGER.
Die verschiedenen Sichten und ihre Verwendbarkeit werden in Tabelle 1 zusammengefasst.

3.5 Der Inhaltsblock

(CONTENTS

{ define }	{ Definitionen }
{ unused }	{ nicht angeschlossene Ports }
{ global }	{ Ausweitung des Definitionsbereiches }
{ figureGroup }	{ Definition von Figurengruppen }
{ joined }	{ Verbundene Ports }
{ mustJoin }	{ noch zu verbindende Ports }
{ annotate }	{ Text auf Figurengruppen }
{ wire }	{ Verbindungsdraht }
{ section }	{ Konstrukt für Dokumentationsschicht }
{ criticalSignal }	{ für routing–Zwecke }
{ required }	{ für tming–Zwecke }
{ measured }	{ für tming–Zwecke }
[logicModel]	{ Verhalten der Zelle }
{ UserData }	{ für benutzerdefinierte Datentypen }
{ Comment }	{ Kommentarblock }
{ instance }	{ Einzelheiten siehe Kapitel 3.6 }

)

Es ist möglich, Typen zu deklarieren. Ports, geometrische Figuren oder Signale müssen deklariert werden, bevor sie verwendet werden können. Dies geschieht durch den Befehl
(**define** *Richtung Typ declareObject* {*rename*}).
Richtung kann INPUT, OUTPUT, INOUT, LOCAL, oder UNSPECIFIED sein, *Typ* ist entweder PORT, SIGNAL oder FIGUREGROUP.
declareObject besteht entweder aus:

- dem zu definierenden Namen (declareObjectNameDef)
- einem Array, (siehe Kapitel 4.2)
- (multiple {*declareObjectNameDef* | *arrayDefinition* }) Es dürfen keine gleichen Namen in einem Gültigkeitsbereich auftreten.

3.6 Die Instanz einer Zellsicht

(INSTANCE

CellReference	{ Referenz auf eine Zelle }
oder:	
multiple { CellReference }	{ Refernz auf eine Zelliste }
viewNameRef	{ Verweis auf eine Sicht }
instanceNameDef	{ Name der Instanz }
[rename]	{ Umbenennungsblock }
{ parameter }	{ Parameter }
{ property }	{ Eigenschaften/Fähigkeiten }
{ portInstance }	{ Festlegung von properties }
[transform]	{ Einzelheiten siehe Kapitel 3.7 }

)

3.7 Transformationen

(TRANSFORM

- [(scaleX integer integer)]
  Skalierung bzgl. X–Achse

- [(scaleY integer integer)]
  Skalierung bzgl. Y–Achse

- [Orientierung
  Möglich sind:
  R0 | R90 | R180 | R270 | MX | MY | MYR90 | MXR90

- [translate] : Angaben über eine Verschiebung des Objekts

)

R bedeutet hier Rotation, M bedeutet Spiegelung.
MYR90 ist demnach eine Rotation um 90 Grad und anschliesßend eine Spiegelung an der Y–Achse.

3.8 Der Interface–Teil

(INTERFACE

{ portImplementation }	{ Aussehen }
{ unused }	{ Unbenutzte Ports }
[symbol]	{ symbolische Darstellung }
[protectionFrame]	{ abstrakte Darstellung }
[arrayRelatedInfo]	{ Angaben über Gatearrays }
{ joined }	{ Verbundene Ports }
{ mustJoin }	{ noch zu verbindende Ports }
{ weakJoined }	{ Verbindung zu einer Portgruppe }
{ permutable }	{ Vertauschung von Ports }
{ define }	{ Definitionen }
{ timing }	{ Verzögerungszeiten }
{ simulate }	{ Testdaten }
{ comment }	Kommentarblock }
{ userData }	{ Benutzerdefinierte Daten }

)

4 EDIF-Syntax

4.1 Grundlegende EDIF-Konstrukte

Im Folgenden werden einige EDIF–Konstrukte näher beschieben. Eine vollständige Dokumentation ist in [1] auf den Seiten 45–63 angegeben. Folgende Bezeichnungen werden im Folgenden benutzt:

- *Identifiers* sind alphanumerische Zeichenketten der Länge 1 – 255. Erlaubt sind alphabetische Zeichen (A...Z,a...z), Zahlen (0..9) und Underscores (_). Das erste Zeichen eines Identifiers muß ein alphabetisches Zeichen sein.
- *primitiveData* können sein:
  - True
  - False
  - ein String
  - eine Integer–Zahl
  - eine Float–Zahl.
    Beispiel : 1.234 wird als e 12324 -3 angegeben.

4.2 Definition eines Identifiers

Um den Identifier 'keyword' zu definieren, wird folgende Form benutzt:

(keywordDef *form*),

wobei form

- eine symbolische Konstante
- ein *identifier*
- ein *primitiveData* oder
- ein *EDIF–Statement*

sein kann.

Das MiNoMax – Konstrukt gibt 3 Werte einer Variablen an: Den mimimalen, den nominellen und den maximalen.

```
Beispiel:

  ( Hallo ( MiNoMax 3 5 8 ) )
  ( comment "Der minimale Wert von Hallo ist 3,
             der nominelle ist 5, der maximale ist 8" )
```

Mit dem multiple-Konstrukt kann einer Variablen eine Liste von Werten zugewiesen werden.

```
Beispiel:

  ( Hallo (multiple 1 2 3 4) )

Hallo besitzt nun den Wert 1 2 3 4 ( Eine Liste von Integer-Werten ).
```

Definition eines Arrays: Durch
(**arrayDefinition** declareObjectNameDef {*integer*})
wird ein Array definiert.

```
Beispiel:

  (arrayDefinition Hallo 38)

definiert das Array        (member Hallo 0)...(member Hallo 37).
```

4.3 Kommentare

Kommentare können an jeder Stelle eingegeben werden. Sie haben die Form:

```
( comment  "string" )
```

4.4 Properties

Eigenschaften werden wie folgt definiert:
(**property** *PropertyType PropertyNameDef* [*propertyValue*]

{ *property* }

[*rename*]

{ *comment* }

)
Der *propertyValue* kann ein *primitiveData-*, oder ein *MiNoMax*-Konstrukt sein. Der *PropertyType* ist eine symbolische Konstante, welche den Ursprung der property angibt.
Folgende *PropertyTypes* sind denkbar:

- **EDIF**: Einige Properties sind bereits vordefiniert und können nicht umdefiniert werden. Beispielsweise ist ACLOAD vordefiniert. ACLOAD kann einen Zahlenwert (integer oder float) sowie einen miNoMax – Wert annehmen.
- **USER**: Properties können von einer Arbeitsgemeinschaft vordefiniert werden. Sie dürfen vom einzelnen Benutzer nicht neu definiert werden.
- Name: Jeder Benutzer kann sich eigene properties frei definieren.

4.5 Rename

Das *rename*-Konstrukt wird in allen Fällen verdendet, in denen es nötig erscheint, einem verwendeten Konstrukt einen alias–Namen zu geben. Dies kann zum Beispiel der Fall sein, wenn ein in EDIF illegaler Bezeichner aus einer externen Bibliothek benutzt wird.

4.6 Benutzerdaten

Das *UserData*-Konstrukt wird verwendet, um komplexe Daten zu beschreiben, für die es momentan noch keine andere EDIF-Ausdrucksmöglichkeit gibt. UserData werden wie folgt definiert:
(**UserData** userDataNameDef

{ *form* } (siehe 4.2)

)

4.7 Neue Schlüsselwörter

Neue Schlüsselwörter können wie folgt definiert werden:
(**keyword** keywordNameDef

{ *formal* }

{ *optional* }

{ *extra* }

{ *parameterNameRef* | *string* | *integer* }

{ *build* }

{ *buildName* }

{ *forEach* }

)

- *formal*:
  Mit diesem Konstrukt werden formale Parameter angegeben.

  ```
  Beispiel:

       (keyword Hallo
               (formal a b c)
       )

   (Hallo 1 2 3)   bedeutet nun:  a=1; b=2; c=3;
  ```

- *optional*:
  Das Optional-Konstrukt dient zur Einbindung von formalen Parametern, die nicht bei jedem neuen Schlüsselwort eingefügt werden müssen.

  ```
  Beispiel:

      (keyword Hallo
              (formal a b)
              (optional Start 5)
              (optional Ende Start)
      )
  ```

```
Nun bedeutet    (Hallo 2 4 6 8) :   a=2; b=4; Start=6; Ende=8;
                (Hallo 2 4)     :   a=2; b=4; Start=5; Ende=5;
```

- *extra*:
  In diesem Konstrukt werden die überschüssigen Übergabeparameter zusammengefaßt. Wird das extra-Konstrukt ohne Variablen aufgerufen, werden alle überschüssigen Parameter ignoriert. Wird das extra-Konstrukt mit mehr als einem Parameter aufgerufen, so werden die überschüssigen Werte alternierend auf die angegebenen Parameter verteilt.

```
    Beispiel:

        (keyword Hallo
                (formal a b)
                (extra s)
        )
        (keyword Bello
                (formal x)
                (extra)
        )
        (keyword Cello
                (formal y)
                (extra n p r)
        )

Nun bedeutet (Hallo 2 4 18 24)       : a=2; b=4; s=(multiple 18 24);
             (Hallo 3 6 5 3 9 )      : a=3; b=6; s=(multiple 5 3 );
             (Bello 2 4)             : x=2; 4 wird ignoriert
             (Cello 1 2 3 4 5 6 7 8): y=1; n=(multiple 2 5 8);
                                           p=(multiple 3 6);
                                           r=(multiple 4 7);
```

- *build*:

```
Beispiel:

    ( keyword Hallo
            (formal a b)
            (build Cello b a)
    )

  Nun erzeugt   (Hallo 5 8) die Ausgabe (Cello 8 5).
```

- *buildName*:
  Dieses Konstrukt erlaubt die Verwendung von sybolischen Konstanten in Schlüsselwörtern. Zum Beispiel erzeugt (buildname r90) die symbolische Konstante r90.

- *forEach*:
  Dieses Konstrukt wird verwendet, um eine Schleife zu simulieren.

```
Beispiel:

    ( Hallo (multiple 2 4 6) )
    ( foreach Hallo (build Bello Hallo) )
    ( foreach Hallo (build Cello Hallo (index Hallo) ) )

generiert (Bello 2), (Bello 4), (Bello 6)
bzw.      (Cello 2 0), (Cello 4 1), (Cello 6 2).
```

4.8 Grafik

- *point*:
  Mit dem Befehl (point *integer integer*) wird ein Punkt definiert, wobei der erste Interger–Wert die X–Koordinate, und der zweite Wert die Y–Koordinate kennzeichnet.

- *path*:
  Mit dem Befehl (path {*point* }) wird ein Streckenzug duch die angegebene Punktliste gezeichnet.

Hier ein Beispiel zu den angegebenen Konstrukten:

```
(keyword liste
        (extra x y)
        (foreach (multiple x y) )
                (build point x y)
        )
)
```

`Wird nun  (path (liste 10 10 10 20 50 20 50 10) )  aufgerufen, so wird derselbe Streckenzug gezeichet, wie im folgenden Befehl:`

```
(path
        (point 10 10)
        (point 10 20)
        (point 50 20)
        (point 50 10)
)
```

Literatur

[1] EDIF Steering Committee, EDIF Specification (1985)

[2] Digest of Technical Papers, EDIF Uder Group Workshop (1985)

[3] K.R. Bennett, D.L. Chalmers, An analysis of EDIF Version 2 0 0 – How it has changed from Version 1 1 0 – (1987)

[4] Gert Kachel, Zusammenfassung: EDIF (1986)

[5] John D. Crawford, An Electronic Design Interchange Format (1984)