View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Università degli studi di Pisa

Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica

Tesi di laurea

Progetto e realizzazione di un traduttore per il linguaggio EDIF orientato a sistemi FPGA

Relatori

Prof. Andrea Domenici
Prof.ssa Cinzia Bernardeschi

Candidato

Massimiliano Itria

Anno accademico 2010-2011

ai miei genitori

Indice

1 Introduzione 5

2 Il linguaggio EDIF 9
2.1 Sintassi di EDIF 9
2.2 Struttura di EDIF 11
2.2.1 Library 13
2.2.2 Cell 13
2.2.3 View 13
2.2.4 Content 14
2.2.5 Interface 15
2.2.6 Declarations 17
2.2.7 Instance 18
2.3 Routing 19
2.3.1 Il costrutto Joined 19
2.3.2 I costrutti Mustjoin, Criticalsignal e Weakjoined 20
2.3.3 Tecnologia 20
2.3.4 Geometria 22
2.4 Visione d'insieme di un file EDIF 23

3 Flex 25
3.1 Espressioni regolari 26
3.1.1 Espressioni regolari in Flex 26
3.2 Funzionamento di Flex 28
3.2.1 Struttura dei file per Flex 29
3.2.2 Sezione definitions 30
3.2.3 Sezione rules 31
3.2.4 Sezione user code 32
3.2.5 Analisi dell'input 33
3.2.6 Generazione dell'analizzatore sintattico 34
3.2.7 Condizioni di start 35
3.3 Esempio di file per Flex 37

4 Bison 39
4.1 Concetti fondamentali di Bison 39
4.1.1 Grammatiche libere da contesto. 39
4.1.2 Simboli in Bison 40
4.1.3 Valori semantici 40
4.1.4 Azioni semantiche 41
4.2 Struttura di un file input per Bison 42
4.2.1 Il prologo 42
4.2.2 Sezione declarations 43
4.2.3 Sezione rules 44
4.2.4 Epilogo 46

4.3 Azioni 46
4.3.1 Riferimenti a valori semantici 46
4.3.2 Tipi di dato e valori nelle azioni 47
4.4 Interazione tra Bison e Flex 49
4.4.1 Acquisizione dei token 48
4.4.2 Acquisizione dei valori semantici. 48
4.4.3 Interazione tra yyparse() e yylex() 49
4.4.4 Controllo di coerenza del contesto 51

5 Il Traduttore per EDIF 53
5.1 Il linguaggio oggetto 54
5.1.1 Il lessico 54
5.1.2 La sintassi 55
5.2 Realizzazione del traduttore 57
5.2.1 La struttura dati 58
5.2.2 L'analizzatore lessicale 63
5.2.3 L'analizzatore sintattico 65
5.3 La funzione logica 67
5.3.1 Sintesi della mappa di Karnaugh 70
5.3.2 Strutture per il calcolo della funzione logica 71
5.4 Fase di traduzione 73

6 Casi di studio 74
6.1 Circuito b01 74
6.2 Circuito b02 77
6.3 Circuito b06 80

7 Conclusioni 83

Appendice A Implementazione del Parser 85
A.1 Implementazione lessico (Flex) 85
A.2 Implementazione sintassi (Bison) 87

Appendice B Implementazione struttura dati e azioni 95
B.1 file ST.h 95
B.2 file karnaugh.c 105
B.3 file outfnct.c 115

Appendice C Utilizzo del traduttore 122

Bibliografia 123

Ringraziamenti 125

Capitolo 1
Introduzione

Un *parser* è un programma che esegue l'*analisi sintattica* di uno stream continuo in input, letto per esempio da un file o da tastiera, in modo da determinare la sua struttura grammaticale grazie ad una data grammatica formale. Di solito i *parser* non sono scritti a mano ma sono generati attraverso altri programmi detti *generatori di parser*. Tipicamente, il *parsing* di un linguaggio consiste in una *analisi lessicale* e una *analisi sintattica*.

L'*analisi lessicale* è un processo che prende in ingresso una sequenza di caratteri e produce in uscita una sequenza di *token.* Un *token* è un blocco di testo categorizzato, normalmente costituito da caratteri indivisibili chiamati lessemi. Un *analizzatore lessicale* legge i lessemi, li suddivide in categorie a seconda della loro funzione dando loro un significato ed infine genera i *token*.

L'*analisi sintattica* opera su una sequenza di *token* in modo da determinare la sua struttura grammaticale grazie ad una data grammatica formale. Il termine *parsing* può essere usato per indicare l'*analisi sintattica*.

Un *traduttore* per un linguaggio è un programma che traduce il codice sorgente scritto in un determinato linguaggio in codice appartenente ad un linguaggio oggetto differente. Tipicamente un *traduttore* utilizza un *parser* per effettuare l'*analisi sintattica* e dal risultato di quest'ultima ricava la traduzione.

Lo scopo finale di questo progetto è la realizzazione di un *traduttore* per un sottoinsieme del linguaggio EDIF (EDIF 2000), un linguaggio usato come formato neutro per la memorizzazione e l'interscambio di *netlist* e schemi elettrici, in un linguaggio oggetto intuitivo e molto sintetico.

Il sottoinsieme del linguaggio EDIF considerato è quello utilizzato dal tool ISE di Xilinx (ISE 2010) ovvero un potente strumento software finalizzato alla

programmazione di dispositivi FPGA, Field Programmable Gate Arrays. Quest'ultimi sono dispositivi prefabbricati, costituiti da componenti le cui interconnessioni possono essere programmate elettricamente.

Un FPGA è composto da vari blocchi logici implementati utilizzando più reti multilivello di porte logiche. Le interconnessioni possono essere viste come switch programmabili elettricamente, queste rendono l'FPGA un dispositivo completamente diverso dai comuni circuiti integrati. La possibilità di essere programmato e riprogrammato più volte, conferisce all'FPGA notevoli vantaggi rispetto le altre tecnologie. I microcontrollori e i processori General Purpose offrono infatti grande flessibilità grazie alla loro programmabilità ma hanno generalmente prestazioni ridotte in termini di tempo computazionale e consumo energetico rispetto un FPGA. I circuiti integrati ASIC, Application Specific Integrated Circuit, offrono invece migliori prestazioni in termini di tempo computazionale e consumo energetico rispetto la tecnologia FPGA, ma hanno un maggior costo economico a causa di lunghi tempi di progettazione. Detto questo, si capisce che la tecnologia ASIC è vantaggiosa solo se utilizzata per la produzione di dispositivi su larga scala mentre per una produzione su media e piccola scala è possibile risparmiare denaro utilizzando la tecnologia FPGA.

Il tool ISE consente la progettazione delle funzioni che un FPGA deve svolgere mediante l'utilizzo di *netlist* e schemi elettrici. Il termine *netlist* è utilizzato nel campo della progettazione elettronica per indicare l'insieme delle connessioni (*net*) elettriche di un circuito elettronico, tipicamente riassunte in un file. Salvo rari casi, la *netlist* viene generata automaticamente a partire da uno schema elettrico.

Un file EDIF rappresenta in realtà una *netlist*. Il *traduttore* realizzato traduce le *netlist* EDIF prodotte da Xilinx ISE in un linguaggio oggetto molto semplice e compatto.

Il processo di traduzione si compone di più fasi, o sottoprocessi, che passano i loro risultati alla fase successiva:

- L'***analisi lessicale*** che produce in uscita una sequenza di *token* a partire dai caratteri contenuti nel file d'ingresso.
- L'***analisi sintattica*** che riceve in ingresso i *token* prodotti dall'analizzatore lessicale e li raggruppa nelle espressioni regolari definite dalla grammatica.

- L'***analisi semantica*** che esamina le espressioni regolari formate e associa ad esse un valore appartenente ad un determinato tipo.
- Generazione della ***struttura dati*** che mantiene le informazioni estratte dal file sorgente.
- ***Traduzione*** che si ottiene dall'elaborazione della struttura dati e produce come uscita il file nel linguaggio oggetto.

Al fine di implementare il *parser* che è stato utilizzato per realizzare il *traduttore*, sono stati usati due strumenti software, *Flex* e *Bison*. *Flex* è un programma che genera l'*analizzatore lessicale* per un determinato linguaggio a partire dalla dichiarazione delle espressioni regolari che definiscono il suo lessico. *Bison* è un programma che genera l'*analizzatore sintattico* per un linguaggio tramite la dichiarazione della sua grammatica. Entrambi generano codice sorgente C++. L'uso di questi due strumenti facilita al programmatore l'implementazione di un *parser* perché grazie ad essi la sua programmazione non è direttamente effettuata in C o C++ ma si riduce in gran parte alla dichiarazione delle espressioni regolari proprie del linguaggio da tradurre.

Altri importanti passi di questo lavoro sono stati la progettazione e la realizzazione della struttura dati (per memorizzare le informazioni relative ai circuiti elettronici) e la realizzazione dei metodi che producono la traduzione. L'implementazione della struttura dati e della traduzione sono completamente affidate al programmatore poiché non possono essere automatizzate da *Flex* e *Bison*.

Nel secondo capitolo di questa tesi viene analizzata dettagliatamente la sintassi del linguaggio EDIF per chiarire la maggior parte dei suoi aspetti ed illustrare le sue potenzialità.

Nel terzo e nel quarto capitolo sono esaminati invece gli strumenti *Flex* e *Bison.*In particolare vengono esaminate le metodologie per generare un generico parser utilizzando questi due strumenti.

Nel quinto capitolo vengono descritti: il linguaggio oggetto in cui sono tradotti i file EDIF, la struttura dati utilizzata per memorizzare le informazioni estratte da EDIF, i metodi utilizzati per ottenere la traduzione.

Nel sesto capitolo infine, sono mostrati alcuni casi d'uso tramite le traduzioni di alcune *netlist* le cui caratteristiche si ritrovano spesso in circuiti elettronici di reale utilizzo.

Il linguaggio di programmazione utilizzato per realizzare il *traduttore* è il C, le cui potenzialità sono pienamente sufficienti per consentire la programmazione del *traduttore* alla pari del C++.

Xilinx ISE 12.1 è stato utilizzato per lo studio dei circuiti elettronici e le loro rispettive *netlist* generate in EDIF.

Capitolo 2
Il linguaggio EDIF

EDIF (EDIF 2000) è l'acronimo di *Electronic Design Interchange Format*, ed è stato usato in modo predominante come formato neutro per la memorizzazione e l'interscambio di netlist e schemi elettrici. È stato uno dei primi tentativi di stabilire un formato neutro per lo scambio dei dati per il settore industriale dell'Electronic Design Automation (EDA). L'obiettivo era stabilire un formato comune dal quale potessero derivare i formati proprietari dei sistemi EDA. Quando gli utenti avevano bisogno di trasferire dati da un sistema ad un altro, era necessario scrivere dei traduttori da un formato all'altro. Mano a mano che il numero di formati si moltiplicava, le problematiche dei traduttori aumentavano esponenzialmente. L'aspettativa era che grazie all'EDIF il numero di traduttori si potesse ridurre al numero dei sistemi coinvolti. Le versioni 2.0 3.0 e 4.0 di EDIF sono tuttora ancora in uso.

2.1 Sintassi di EDIF

La sintassi di EDIF è simile a quella del linguaggio di programmazione Lisp e al linguaggio Postscript. Ciò rende umanamente difficile la comprensione di EDIF e ancor di più la sua scrittura a mano ma in realtà EDIF è da considerare un formato di interscambio tra programmi e non proprio un strumento di design.

La sintassi di EDIF consiste in una serie di statement nel seguente formato:

```
(keywordName {form})
```

La parentesi tonda aperta è sempre seguita da una *keyword* che è a sua volta seguita da più *form* che terminano con una parentesi tonda chiusa. Un form è una sequenza di identificatori, dati primitivi, costanti simboliche o statement. La semantica di EDIF è definita mediante le sue keyword le quali sono sempre nomi seguiti da una parentesi

aperta. Un identificatore rappresenta in EDIF il nome di un oggetto o un raggruppamento di dati. Gli identificatori sono utilizzati per definire nomi, riferimenti a nomi, keyword, e costanti simboliche. Gli identificatori validi in EDIF sono formati da caratteri alfanumerici, compreso underscore, e devono essere preceduti da una e commerciale '&' se il primo carattere non è alfabetico. La e commerciale non è considerata parte dell'identificatore. La lunghezza di un identificatore varia da 1 a 255 caratteri e non è case sensitive per cui i seguenti identificatori

```
&clok   Clock     clock     &Clock
```

sono equivalenti.

I numeri in EDIF sono interi a 32 bit con segno. Per rappresentare i numeri reali viene utilizzato il sistema a virgola mobile; ad esempio il numero reale 1,4 è rappresentato come `(e 14 -1)` dove 14 è la mantissa e -1 è l'esponente. Possono essere rappresentati reali appartenenti all'intervallo ±1 ∞ 10 ^ ±35. Le unità di misura utilizzate per quantificare le grandezze tramite i numeri non sono mai esplicite in EDIF ma sono determinate in base al contesto del linguaggio e alla posizione occupata dagli stessi numeri all'interno del file. Coordinate e lunghezze sono distanze e devono essere relazionate a metri, ogni coordinata è convertita in metri usando un fattore di scala. Tutte le *library* di EDIF posseggono per questo una sezione individuata dalla keyword *technology* tramite la quale si può relazionare i numeri presenti nel file a unità fisiche.

Una stringa in EDIF consiste in una sequenza di caratteri ASCII racchiusa tra virgolette. E' permesso qualunque carattere alfanumerico, compresi *spazio* e *tab*, come ad esempio: ! # $ & ' () * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ , alcuni caratteri come gli apici " e il percento % fanno però eccezione e devono essere preceduti dal carattere *escape* \.

La keyword *rename* è usata per creare un nuovo identificatore EDIF come segue:

```
(cell (rename TEST_1 "test$1") ...
```

In questo esempio la stringa contiene il nome originale, `test$1`, mentre il nuovo identificatore creato è `TEST_1`. Ciò può generare delle ambiguità perché in questo modo due differenti identificatori rappresentano lo stesso oggetto. (Smith 1997)

2.2 Struttura di EDIF

Le elevate potenzialità di design di EDIF derivano dalla sua struttura. In un file EDIF sono presenti una o più *library* contenenti le descrizioni dei componenti. Ogni componente è descritto mediante una *cell* e più *cell* possono appartenere alla stessa *library* che contiene anche le informazioni riguardanti la tecnologia usata. Ogni *cell* contiene una o più *view*, definite da un proprio particolare tipo *viewtype*, una view contiene a sua volta una *interface* che definisce come possono essere connesse le *cell* definendo la tipologia delle sue porte, *port*. Infine una *view* contiene i *contents* che comprendono le dichiarazioni dei componenti e le loro interconnessioni.

Complessivamente la struttura di un file EDIF è fatta come nel seguente esempio:

```
(edif name
    (status information)
    (design where-to-find-them)
    (external reference-libraries)
    (library name
        (technology defaults)
        (cell name
            (viewmap map)
            (view type name
                (interface external)
                (contents internal)
            )
        )
    )
)
```

Lo statement *status* è utilizzato per tracciare il progresso di design e contenere i nomi degli autori, date e versioni del programma. Lo statement *design* indica invece dove può essere trovato il modello descrittivo completo di cui fa parte il file.

La struttura gerarchica di EDIF può essere evidenziata tramite il diagramma entità-relazione in figura 2.1 (Hillawi Bennett 1986)

In questo diagramma ogni rettangolo rappresenta una entità che può essere ad esempio una *cell*, una *view*, una *instance* o una *port*. Le linee di interconnessione rappresentano le relazioni tra le entità. Il simbolo *uno a molti* accanto un'entità indica che una o più entità dello stesso tipo possono appartenere all'entità indicata della freccia. Come si nota dal diagramma, un *file* EDIF è composto da una o più *library* e da uno o più

design. Una *library* contiene più una o più *cell* e le informazioni relative alla tecnologia in uso. Una *cell* è composta da una o più *view* in cui troviamo una o più *netlist*.

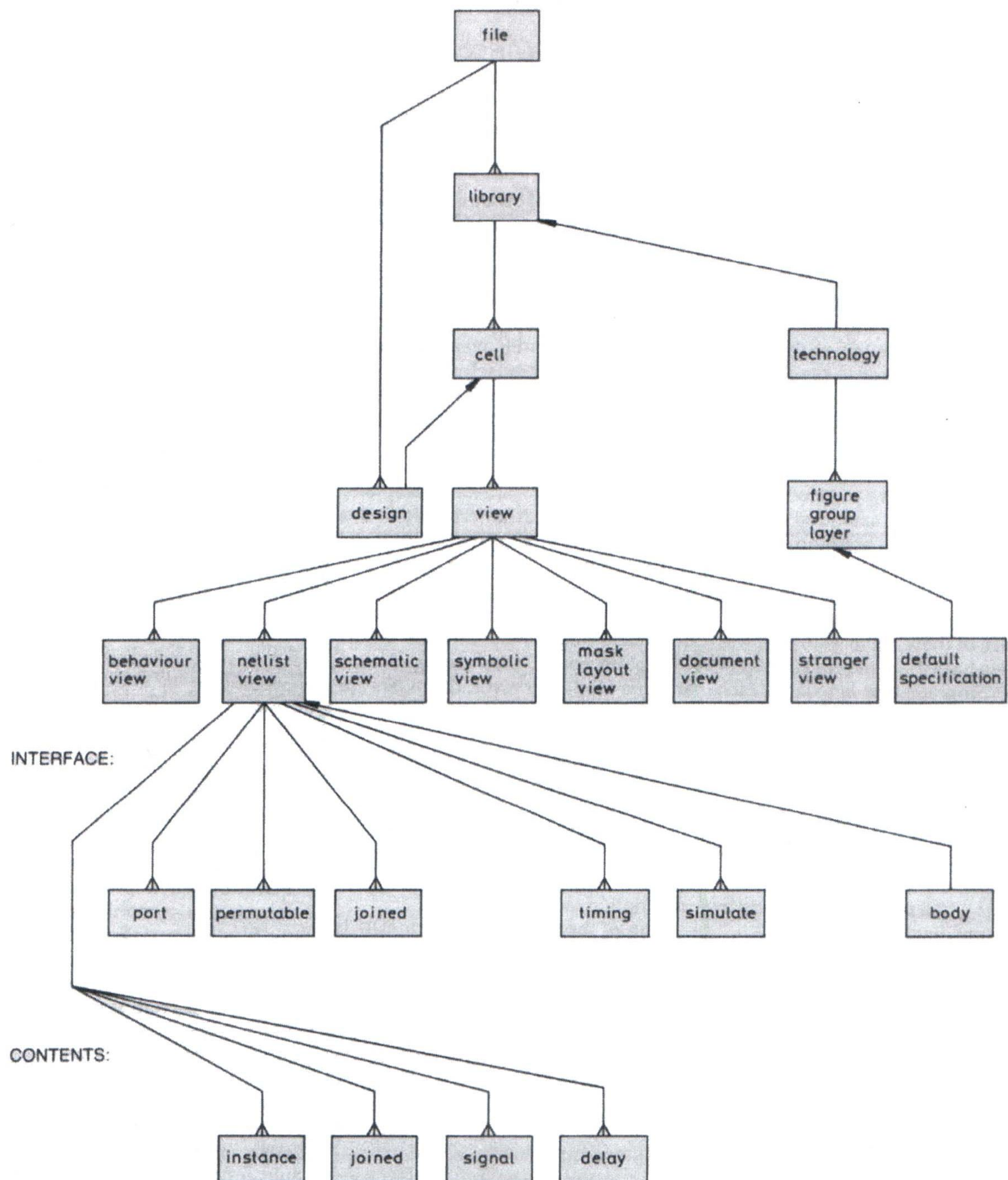

Figura 2.1:Diagramma entità-relazione della struttura del linguaggio EDIF

2.2.1 Library

Una *library* è l'oggetto che in EDIF definisce un intero circuito elettronico o un blocco di esso costituito da uno o più componenti. Ogni *library* contiene di seguito alle informazioni relative alla tecnologia utilizzata, un insieme di *cell* che descrivono le caratteristiche dei componenti presenti nel circuito. Alcune *library* sono dichiarate con lo statement *external* per indicare che la libreria è usata ma non è descritta all'interno del file EDIF.

2.2.2 Cell

Le *cell* sono le definizioni dei tipi di componenti presenti nel circuito, ogni componente nel circuito appartiene ad una *cell* già definita e può essere utilizzato solo rispettando le specifiche presenti nelle *view* della *cell* a cui appartiene. Lo statement *celltype* esplicita il tipo a cui la *cell* appartiene, ad esempio *celltype* GENERIC.

2.2.3 View

Le informazioni che modellano una *cell* sono ottenute tramite le sue *view*, ogni *view* di una *cell* descrive la stessa da differenti punti di vista e diversi livelli di astrazione. Il tipo di *view* può essere uno dei seguenti:

- ***Behaviour*** (funzionamento) specifica il circuito ed il suo funzionamento dal punto di vista logico.
- ***Netlist***, specifica le informazioni di connettività e timing.
- ***Schematic*** fornisce informazioni di natura grafica relativa allo schematico della *cell,* connettività compresa.
- ***Mask Layout*** descrive la geometria del chip e la fabbricazione della scheda.
- ***Symbolic***, fornisce dettagli grafici della configurazione della *cell*.
- ***Document*** descrive la *cell* mediante informazioni testuali e grafiche.
- ***Stranger*** utilizza dati che non seguono le convenzioni degli altri tipi di view ma possono essere espressi tramite la sintassi di EDIF

Ogni tipo di *view* consente vari costrutti del linguaggio. I tipi di view *netlist*, *schematic* e *symbolic* sono essenzialmente molto simili perché descrivono la topologia dei circuiti.

Tramite le *view* di tipo *netlist* si descrive con precisione le modalità di collegamento che i componenti appartenenti ad un determinato tipo di *cell* devono rispettare quando sono connessi agli altri componenti del circuito.

2.2.4 Content

I *content* contengono informazioni sui singoli componenti del circuito e le loro interconnessioni. Essi sono dichiarati in modi differenti in base al tipo di *view* utilizzato, ogni oggetto che un *content* rappresenta è in realtà una *view* e possono essere utilizzati per dichiararlo tutti e sette i tipi di *view* sopra descritti.

Ogni tipo di *view* utilizzata per la descrizione di un *content* consente però solo alcuni costrutti del linguaggio e solo alcuni statement sono permessi per ogni tipo di *view*. I tipi di view *netlist*, *schematic* e *symbolic* sono molto simili perché descrivono la topologia dei circuiti. Gli statement permessi in queste *view* sono quelli per le dichiarazioni `define, unused, global, rename`, ed `instance;` quelli per le specifiche di interconnessione `joined, mustjoin`, e `criticalsignal`; e quelli per le specifiche della temporizzazione `required` and `measured.`

Per i tipi di view *schematic* e *symbolic* sono anche permessi gli statement `annotate` and `wire`. Per il tipo *mask layout* sono previsti tutti gli statement di dichiarazione, qualcuno per le in interconnessioni e lo statement `figuregroup` per far riferimento a immagini reali. Il tipo *behavior* permette solo poche dichiarazioni e lo statement `logicmodel`. Il tipo *document* permette solo i costrutti `instance` e `section`. Infine il tipo *stranger* permette qualunque costrutto ma in realtà non supporta nulla. (Rubin 1994)

Un *content* dichiara generalmente una serie di istanze dei componenti appartenenti ad una data *cell* usando lo statement *instance*. Ogni *instance* definisce un unico componente nel circuito e attribuisce ad esso un nome univoco ed il corrispondente tipo definito da una *cell*.

	Netlist	Schematic	Symbolic	Mask Layout	Behavior	Document	Stranger
`define`	X	X	X	X	X		X
`unused`	X	X	X	X			X
`global`	X	X	X	X			X
`rename`	X	X	X	X	X		X
`instance`	X	X	X	X		X	X
`joined`	X	X	X	X			X
`mustjoin`	X	X	X	X			X
`criticalsignal`	X	X	X				X
`required`	X	X	X				X
`measured`	X	X	X				X
`logicmodel`					X		X
`figuregroup`				X			X
`annotate`		X	X				X
`wire`		X	X				X
`section`						X	X

Figura 2.2: Statement permessi per ogni tipo di view

2.2.5 Interface

Tutte le *cell* contengono una *interface*, ovvero la descrizione che individua dove e come la *cell* stessa può essere connessa. Ogni *interface* contiene infatti la descrizione delle porte della *cell* e la loro direzione, *input* o *output.* E’ possibile specificare la *interface* di una *cell* utilizzando opportunamente i sette metodi previsti per le *view* tenendo conto però che, come accade per le *view* all’interno dei *contents*, per ogni tipo di view utilizzato non tutti gli statement sono permessi come si può notare dalla seguente figura.

	Netlist	Schematic	Symbolic	Mask Layout	Behavior	Document	Stranger
`define`	X	X	X	X	X		X
`rename`	X	X	X	X	X		X
`unused`	X	X	X	X	X		X
`portimplementation`	X	X	X	X	X		X
`body`	X	X	X	X	X		X
`joined`	X	X	X	X	X		X
`mustjoin`	X	X	X	X	X		X
`weakjoined`	X	X	X	X	X		X
`permutable`	X	X	X	X	X		X
`timing`	X	X	X	X	X		X
`simulate`	X	X	X	X	X		X
`arrayrelatedinfo`			X	X			X

Figura 2.3: Statement permessi per ogni tipo di interface

Un importante statement per le *interface* è *portimplementation*, che descrive le porte e i loro rispettivi componenti associati, le caratteristiche grafiche, il timing e altre proprietà. Nonostante le porte siano dichiarate mediate lo statement *define*, *portimplementation* permette una descrizione più dettagliata e con un numero di informazioni maggiore. Il formato è il seguente:

```
(portimplementation portname figuregroups instances properties)
```

dove *portname* è banalmente il nome della porta corrispondente a quella definita nella *cell* corrispondente. *Figuregroup* descrive ogni elemento grafico connesso alla porta, le *instance* specificano ogni *subcell* che descrive la porta e le sue proprietà. Ovviamente, le porte che sono definite dalle *instance* delle altre *cell* non necessitano di *figuregroup* per essere descritte per cui lo statement *portimplementation* è opzionale.

Lo statement *body* è utilizzato per descrivere l'aspetto esterno di una *cell* o la sua interfaccia. Nella maggior parte dei casi questo statement è semplicemente usato per dare una apparenza esterna alle *instance* della *cell* associandola ad una immagine.

Ciò permette in pratica agli editor grafici di poter visualizzare le immagini associate ai componenti all'interno degli schematici dei circuiti, facendo riferimento a librerie di immagini grafiche.

Il formato dello statement *body* è:

```
(body figuregroups instances)
```

dove *instances* sono le *subcells* che possono essere usate per descrivere il body.

Infine, per concludere il discorso che riguarda le *interface*, è necessario menzionare i costrutti *timing* e *simulate*. Lo statement *timing* è utilizzato per attribuire un ritardo del segnale tra le porte dei componenti interessati. *Simulate* elenca i dati relativi a test e i risultati attesi.

2.2.6 Declarations

Le *declarations* stabiliscono alcuni oggetti presenti all'interno di una *cell,* inclusi i segnali e porte. I segnali interni sono definiti tramite lo statement

```
(define direction type names)
```

dove *direction* può essere *input*, *output*, *local* oppure *unspecified*. Solo *local* e *unspecified* hanno significato all'interno della sezione dei *contents*, gli altri sono usati all'interno di sezioni *interface*. Il tipo di dichiarazione può essere *port*, *signal* o *figuregroup*, dove *port* è destinata alla sezione *interface*, *signal* per la sezione *contents* e *figuregroup* alla sezione *technology* di una *library*. I nomi possono essere dichiarati dando un singolo nome od una lista di nomi aggregati. E' anche possibile definire array usando il costrutto `(arraydefinition name dimensions)` che può essere indicizzato usando il costrutto `(member name indexes)`. Si riportano di seguito tre semplici esempi d'uso dello statement *define*:

```
(define input port Clk)
(define unspecified signal (multiple a b c))
(define local signal (arraydefinition i-memory 10 32))
```

Lo statement *unused* ha la seguente forma:

```
(unused name)
```

e indica che il nome non è utilizzato nella *view* della *cell* in cui è dichiarato per evitare di incorrere in segnalazioni di errori durante l'analisi del file EDIF.

La dichiarazione *global* ha invece la seguente forma:

```
(global names)
```

Essa definisce i segnali usati nella *view* della *cell* ed anche ad un livello più basso, cioè nei sottocomponenti situati all'interno della *view*. Quando questi sottocomponenti hanno porte che corrispondono a nomi dichiarati *global* essi sono implicitamente equiparati al livello del componente superiore.

Infine la dichiarazione *rename*, permette di associare un qualunque nome EDIF ad una stringa arbitraria. Ciò permette ad EDIF di utilizzare una convenzione in generale illegale per qualunque altro linguaggio.

2.2.7 Instances

Ogni *instance* definisce un unico componente nel circuito, attribuisce ad esso un nome univoco e il corrispondente tipo definito da una *cell* e infine fornisce informazioni di natura tecnica come ad esempio la tabella di verità del componente in forma di attributo testuale (attributo INIT). Ogni istanza deve essere messa in relazione con le altre specificando le interconnessioni tramite lo statement *joined*.

```
  (instance (rename rstpot_renamed_9 "rstpot")
      (viewRef view_1 (cellRef LUT6 (libraryRef UNISIMS)))
      (property XSTLIB (boolean (true)) (owner "Xilinx"))
      (property INIT (string "BFBFBF0000BF0000") (owner
"Xilinx")
  )
```

In questo esempio è dichiarato il componente `rstpot`, rinominato come `rstpot_renamed_9`. Esso è una istanza della *cell* LUT6 e possiede attributo INIT BFBFBF0000BF0000 che descrive la sua particolare funzione all'interno del circuito.

La struttura gerarchica di EDIF è realizzata dichiarando le *instance* di una *cell* all'interno di *contents* di un'altra *cell*. Il formato dello statement *instance* è il seguente:

```
(instance cell view name transform)
```

Il nome dell'altra *cell* è in `cell` e la particolare *view* utilizzata è in `view`. Questo meccanismo permette di utilizzare più tipi di *view* per un unica *cell* quando ciò ha un senso.

2.3 Routing

Per un circuito elettronico, con il termine *routing* si intende la descrizione dettagliata delle interconnessioni dei suoi componenti e delle connessioni ai segnali esterni al circuito. La descrizione della connettività è indicata da vari costrutti come *joined, mustjoin* e *criticalsignal.*

2.3.1 Il costrutto joined

Il costrutto *joined* identifica i segnali e le porte connesse per ogni componente.

```
(net reset_IBUF
  (joined
    (portRef CLR (instanceRef next_state_renamed_0))
    (portRef PRE (instanceRef count_0))
    (portRef PRE (instanceRef count_1))
    (portRef PRE (instanceRef count_2))
    (portRef O (instanceRef reset_IBUF_renamed_3))
    (portRef I0 (instanceRef output_rstpot_renamed_9))
    (portRef I (instanceRef reset_inv1_INV_0))
    (portRef I3 (instanceRef faulty_output_rstpot_renamed_10))
   )
)
```

Nel precedente esempio vengono definiti i collegamenti tra il pin di uscita del buffer chiamato `reset_IBUF` e i pin di ingresso dei componenti che sono direttamente

collegati a *reset_IBUF*. Più specificatamente lo statement `(portRef PRE (instanceRef count_0))` indica che la porta *PRE* relativa al componente *count_0* è collegata all'uscita di *reset_IBUF*. Lo statement `(portRef O (instanceRef reset_IBUF_renamed_3))`, che può in apparenza sembrare ambiguo, è in realtà solo un modo di EDIF di specificare il nome della porta di ingresso del buffer *reset_IBUF* e l'altro nome che è stato associato ad esso quando è stato istanziato e rinominato.

2.3.2 I costrutti Mustjoin, Criticalsignal e Weakjoined

Altri due costrutti che incrementano le potenzialità descrittive del linguaggio EDIF sono *Mustjoin*, *Criticalsignal* e *Weakjoined*.

Il primo dei tre, *Mustjoin*, definisce un segnale non ancora connesso ma che sarà connesso più avanti durante la descrizione del routing. Ciò serve a evitare messaggi di errore durante l'analisi del file EDIF. Il costrutto *criticalsignal* stabilisce invece alcune priorità del routing. Infine *Weakjoined* è utile per definire un insieme di connessioni, questo costrutto può essere utilizzato solo all'interno delle sezioni *interface* ed è in realtà un modo alternativo di specificare le connessioni ma che si rivela più sintetico in alcuni casi. Esistono in EDIF anche altri costrutti di routing ma non verranno presi in considerazione in questa sede.

2.3.3 Tecnologia

La sezione *technology* prevede una serie di specifiche riguardanti la descrizione delle *library*. In questa sezione si può stabilire ad esempio quale set di figure utilizzare per la descrizione grafica degli elementi. Si possono specificare inoltre le unità di misura di tempo, spazio, potenza e così via. Il formato della sezione technology è il seguente:

```
(technology name
    defines renames
    figuregroupdefaults
    numberdefinitions gridmaps
    simulation
)
```

dove *name* è l'identificatore della tecnologia in uso. Può essere utilizzato un insieme di statement *define* per dichiarare le *figuregroup* per vari tipi di segnali, e vari statement *rename* per stabilire o cambiare i corrispondenti nomi che verranno utilizzati nella library. Lo statement *figuregroupdefault* carica una lista di parametri *pathtype*, *width*, *color*, *fillpattern e borderpattern* per stabilire il default da associare alla library. Uno statement molto importante è *numberdefinition* perché definisce le unità di misura che sono utilizzate nella library.

Lo statement *numberdefinition* è del tipo seguente:

```
(numberdefinition SI
    (scale distance edif real)
    (scale time edif real)
    (scale capacitance edif real)
    (scale current edif real)
    (scale resistance edif real)
    (scale voltage edif real)
    (scale temperature edif real)
)
```

L'esempio mostra come viene dichiarato il set di unità di misura denominato SI. Ogni clausola *scale* di *numberdefinition* dichiara che le grandezze: spazio, tempo, capacità, intensità di corrente, resistenza, tensione e temperatura vengono espresse secondo il fattore di scala `edif:real`. Per convenzione, in EDIF l'unità di misura della distanza è il metro ciò vuol dire che la clausola

```
(scale distance 1000000 1)
```

indica che le distanze nella *library* in uso sono specificate in unità pari ad un milionesimo di metri ovvero in micron.

L'unità di tempo è invece il secondo, per la capacità si usa il farad, la corrente si misura in ampere, la resistenza in ohm, la tensione in volt e infine la temperatura in gradi celsius.

La clausola *gridmap* è utilizzata invece nella sezione technology per dichiarare una scala non uniforme degli assi cartesiani, ad esempio:

```
(gridmap 3 4)
```

stabilisce che le unità di misura delle ascisse siano tre volte più grandi del *numberdefinition* che definisce l'unità di misura dello spazio, mentre le unità di misura delle ordinate sono quattro volte più grandi del *numberdefinition*. Questo tipo di scaling ha comunque un numero di applicazioni limitate.

2.3.4 Geometria

Nelle *view* di tipo *mask-layout* può essere specificata la geometria tramite il costrutto *figuregroup* come mostrato nel seguente esempio

```
(figuregroup groupname
    pathtype width color fillpattern borderpattern
    signals figures)
```

Quando esiste un *figuregroupdefault* dichiarato nella sezione technology non è necessario definire esplicitamente le caratteristiche grafiche *pathtype*, *width*, *color*, *fillpattern*, e *borderpattern.* Queste cinque caratteristiche grafiche sono opzionali ed hanno il seguente formato:

```
(pathtype endtype cornertype)
(width distance)
(color red green blue)
(fillpattern width height pattern)
(borderpattern length pattern)
```

Il *pathtype* descrive come devono essere disegnati le terminazioni dei fili e i loro angoli. *Width* specifica invece lo spessore del filo e *color* il colore dei fili nei diversi modi in cui essi stessi possono essere utilizzati. *Fillpattern* definisce che tipo di reticolo deve riempire l'interno della figura. Infine *borderpattern* stabilisce il modello da usare per i bordi della figura dei componenti. Si riporta di seguito, senza scendere nei particolari, un breve esempio di definizione degli attributi di *figuregroup*.

```
(pathtype round round)
(width 200)
(color 0 0 100)
(fillpattern 4 4 "1010010110100101")
(borderpattern 2 "10")
```

Esistono anche costrutti per disegnare oggetti di forma particolare come: rettangoli, quadrati, cerchi, archi e molti altri. Ad esempio per un poligono :

(polygon *points*)

per dare i punti del poligono basta dichiarare tutti i suoi punti con (point *x y*)

2.4 Visione d'insieme di un file EDIF

Si riporta di seguito un breve esempio che mostra in pratica la struttura complessiva di un file EDIF ed i vari livelli di astrazione. (Rubin 1994)

```
(edif my-design
    (status
        (edifversion 1 0 0)
        (ediflevel 0)
        (written
            (timestamp 1985 4 1 11 16 6)
            (accounting author "Steven Rubin")
            (accounting location "Palo Alto")
            (accounting program "Electric")
            (comment "timestamp contains year, month, day,
hour,")
            (comment "minute, and second")
        )
    )
    (design hot-dog-chip
        (qualify hot-dog-library top-cell)
        (comment "look for top-cell in hot-dog-library")
    )
    (external pad-library pla-library)
    (library hot-dog-library
        (technology 3-micron-nMOS
            ...
        )
        (cell top-cell
            (viewmap ... )
            (view masklayout real-geometry
                (interface ... )
                (contents ... )
            )
            (view schematic more-abstract
                (interface ... )
                (contents ... )
            )
        )
```

```
      )
  )
)
```

Come si può facilmente notare la descrizione è chiamata *my-desin*. Tramite il costrutto *status* sono state inserite la versione dell'EDIF in uso e le informazioni relative all'autore, compresi alcuni suoi commenti. Sono state dichiarate due library, `pad-library` e `pla-library`, che saranno utilizzate ma non descritte in *my-desin.* Di seguito è invece definita e descritta la *library* `hot-dog-library` che possiede due *view*: `real-geometry` di tipo *masklayout* e `more-abstract` di tipo *schematic,* con relative *interface* e *contents*.

Capitolo 3
Flex

Flex (Flex 2007), è un potente strumento utilizzato per la costruzione di analizzatori lessicali, detti anche brevemente *scanner*. Attraverso questi oggetti è possibile suddividere un flusso di caratteri in *token*, cioè in elementi base a cui sono associate specifiche operazioni predeterminate. L'uso in combinazione con altri strumenti come *Bison* ,un generatore di parser, permette di implementare analizzatori sintattici anche molto complessi.

L'analisi lessicale, come già precedentemente anticipato, consiste nel raggruppare insiemi di caratteri e simboli all'interno di unità lessicali dette *token* che corrispondono a specifiche espressioni regolari.

L'analizzatore lessicale è lo strumento attraverso il quale questo processo viene attuato, può essere visto secondo il modello a black box, come una scatola nera avente in ingresso una sequenza caratteri e in uscita un flusso di *token* generati a partire dagli elementi forniti, associando o meno ad ognuno di essi una operazione specifica. L'analizzatore lessicale è dunque uno strumento in grado di riconoscere ed estrapolare da un flusso di dati gli elementi all'interno di esso, e trattarli nel modo corretto.

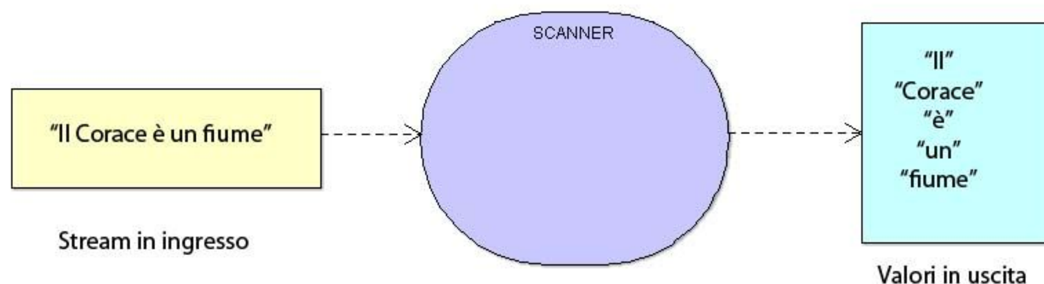

Figura 3.1: Un generico analizzatore lessicale

3.1 Espressioni regolari

Le espressioni regolari sono importanti notazioni utilizzate per specificare modelli per le stringhe di caratteri. (Aho, Lam, Sethi, Ullman 2006)

Dato un linguaggio, è possibile definire una rappresentazione per la sua grammatica in modo tale da consentire ad un interprete di interpretare correttamente le frasi del linguaggio utilizzando tale rappresentazione. L'interprete dovrà utilizzare una classe per rappresentare ogni regola della grammatica.

La grammatica descritta dal seguente diagramma è realizzata da cinque classi. Una espressione regolare può essere un solo carattere, *Literal expression*, o una sequenza di caratteri, *sequence expression*, ma allo stesso tempo può essere una serie di espressioni formate da sottosequenze che sono a loro volta delle espressioni. (Gamma, Helm, Johnson, Vlissides 1998)

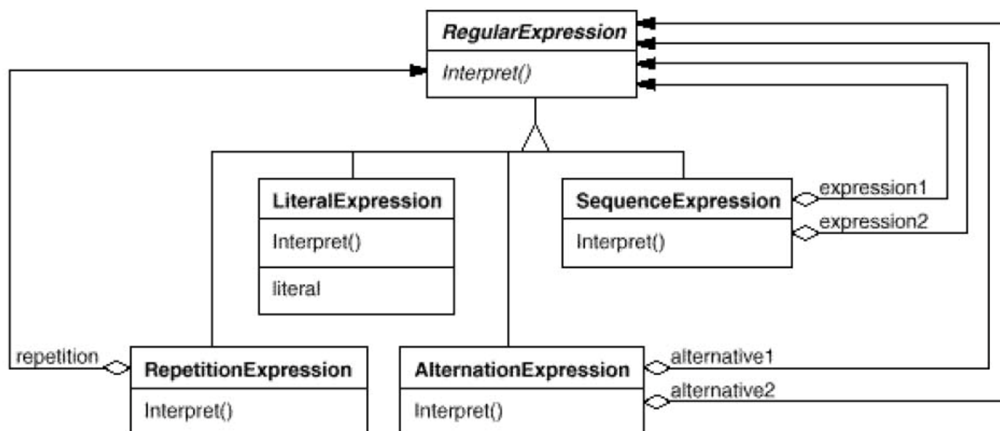

Figura 3.2: Struttura di una espressione regolare

3.1.1 Espressioni regolari in Flex

L'analizzatore lessicale riceve in ingresso uno stream di caratteri in cui sono presenti le espressioni di un determinato linguaggio. Se si volesse individuare ogni singola parola delle espressioni del linguaggio si potrebbe definire ogni parola come *token*.

Si prenda in considerazione la stringa "*sono nato nel 1974*".

Intuitivamente, essa può essere suddivisa in svariati modi, ma per individuarne le parole bisogna stabilire le regole che consentano di riconoscerle definendo le espressioni regolari che descrivono le caratteristiche di ogni parola. In pratica, le espressioni regolari definiscono un lessico in grado di rappresentare un determinato insieme di stringhe a partire da un flusso di caratteri alfanumerici. Questo permette di estrapolare specifici elementi in base alle loro caratteristiche.

Per formare le espressioni regolari Flex utilizza una sua particolare notazione che si avvale alcuni operatori:

“ \ [] ^ - ? . * + | () $ / {} % <>

Lettere e numeri del file di ingresso sono descritti mediante se stessi. I caratteri non alfabetici vengono rappresentati racchiudendoli tra doppi apici, per evitare ambiguità con gli operatori, l’espressione xyz“++” rappresenta ad esempio la sequenza ‘x’ ‘y’ ‘z’ ‘+’ ‘+’. I caratteri non alfabetici possono essere anche descritti facendoli precedere dal carattere \, l’espressione xyz\+\+ rappresenta quindi la sequenza ‘x’ ‘y’ ‘z’ ‘+’ ‘+’.

Le classi di caratteri vengono descritte mediante gli operatori [], l’espressione [0123456789] rappresenta una cifra. Nel descrivere classi di caratteri, il segno - indica una gamma di caratteri, per cui l’espressione [0-9] rappresenta una cifra. Per includere il carattere - in una classe di caratteri, questo deve essere specificato come primo o ultimo della serie, l’espressione [-+0-9] rappresenta una cifra o un segno. Nelle descrizioni di classi di caratteri, il segno ^ posto all’inizio indica una gamma di caratteri da escludere. L’espressione [^0-9] rappresenta un qualunque carattere che non sia una cifra.

L’insieme di tutti i caratteri eccetto ritorno carrello viene descritto mediante il simbolo “.”. Il carattere di ritorno carrello viene descritto dal simbolo \n e il carattere di tabulazione viene descritto dal simbolo \t.

Altri esempi che saranno utilizzati nel prosieguo sono riportati di seguito:

- `'x'` : senza apici, corrisponde al carattere `x`

- '.' : senza apici, indica un qualsiasi carattere ad eccezione del carattere di nuova linea
- [abc], [^abc] e [b-o] : indicano rispettivamente uno fra i caratteri a, b e c, ogni carattere ad eccezione di a, b e c e uno dei caratteri compresi fra b e o
- [a|b] : corrisponde ad un carattere a oppure ad un carattere b, indipendentemente
- r*, r+ e r? : indicano rispettivamente un numero nullo o illimitato di caratteri r, uno o più caratteri r e infine uno o nessun carattere r
- <<EOF>> : definisce il carattere di fine file EOF
- '\0' il carattere NUL (ASCII code 0)
- '\x2a' il carattere con valore esadecimale 2a

Questi sono solo alcuni esempi delle possibili strutture utilizzabili per la costruzione di proprie espressioni in grado di catturare elementi di testo specifici. Attraverso questi elementi di base si possono costruire espressioni più complesse e potenti, in grado di estrapolare dal flusso di dati i diversi *token*. Si riporta qualche esempio:

- [a-zA-Z] : indica un qualsiasi carattere alfabetico, maiuscolo o minuscolo
- [a-z]+ : definisce una stringa di uno o più caratteri alfabetici minuscoli
- [+|-]?[0-9]+ : rappresenta la classe dei numeri interi con o senza segno, senza limiti di cifre decimali a patto che esse siano in numero maggiore o uguale a una
- [+|-]?[0-9]*"."[0-9]+[[e|E][0-9]+]? : è una espressione più complessa ma che esprime bene le potenzialità delle espressioni regolari, la quale permette di esprimere numeri in virgola mobile con o senza segno in forma esponenziale e non.

3.2 Funzionamento di Flex

Come già specificato sopra, Flex è uno strumento per la generazione analizzatori lessicali. Flex legge i file forniti in ingresso, o da standard input se nessun file viene indicato, per ottenere una descrizione dello scanner da generare. Tale descrizione è costituita da un insieme di coppie, chiamate regole, comprendenti una espressione regolare seguita da codice C. Flex genera in uscita un file di codice sorgente C, *lex.yy.c*,

che definisce una funzione *yylex()*. Questo file è compilato e collegato con alcune librerie per produrre un eseguibile. Quando l'eseguibile viene lanciato, analizza il proprio ingresso in cerca delle occorrenze di espressioni regolari e ogni volta che ne individua una, viene eseguito il corrispondente codice C della regola.

La funzione *yylex()* può essere utilizzata direttamente dalla funzione *main* di un parser o di un altro tipo di software per il recupero dei *token*.

E' la funzione *yylex()* che analizza il flusso di dati in ingresso ed estrapola i vari *token* secondo le regole imposte. Questa, salvo diversa indicazione, non ritorna al chiamante fino a quando non ha esaurito i suoi dati in lettura. In realtà i dati in ingresso vengono analizzati un blocco per volta: questo è il caso di uso concomitante con strumenti come *Bison* per la generazione di parser.

3.2.1 Struttura dei file per Flex

Un file di ingresso per Flex si divide in tre sezioni separate fra loro con una riga contenente solamente la coppia di caratteri *%%*. Queste tre sezioni rappresentano rispettivamente:

- *definitions*, nomi utilizzati per semplificare le descrizioni dell'analizzatore lessicale ma anche opzioni e porzioni di codice utente
- *rules*, le regole che determinano la suddivisione del flusso in *token*
- *user code*, una sezione che riporta opzionalmente funzioni di supporto sviluppate dall'utente

Nelle sezioni *definizioni* e *regole,* tutto ciò che viene a trovarsi racchiuso fra la sequenze di caratteri *%{* e *}%* viene copiato interamente nel file prodotto da Flex. All'interno di essi il programmatore può inserire dichiarazioni e funzioni di supporto.

```
definitions
%%
rules
%%
User code
```

3.2.2 Sezione definitions

Questa sezione contiene principalmente le dichiarazioni di alcuni *token* che saranno utilizzati per dichiarare le espressioni regolari dell'analizzatore lessicale nella sezione *rules*. Ogni definizione è del tipo: *nome definizione*.

Il *nome* sarà lo stesso che, nella sezione *rules*, potrà essere utilizzato per riferirsi alla specifica definizione attraverso la dicitura *{nome}* .Alcuni esempi di definizione sono:

- ID [a-zA-Z]+
- DIGIT [0-9]
- STRING ["][^"]*["]
- EXP [e|E][0-9]+
- SIGN [-|+]

La prima riga definisce gli identificatori contenenti lettere maiuscole e minuscole. La seconda definisce una cifra numerica che può essere una tre 0 e 9. La terza definisce invece una stringa, cioè un identificatore racchiuso tra virgolette. La quarta descrive numeri reali (questi ultimi espressi con la notazione esponenziale). L'ultima definisce infine semplicemente il segno.

In questa sezione tutto ciò che si trova all'interno della coppia di identificatori *%{* e *%}* viene copiato direttamente nel file prodotto da Flex senza alcuna modifica. Nell'esempio che segue, questa particolarità è sfruttata per includere alcune librerie C.

```
%{
/*
                                Librerie C
*/
#include <string.h> /* per strdup */
#include <stdlib.h>  /* per atoi */
#include "edif.tab.h" /* per le definizioni dei token e yylval
*/
%}
```

Questa sezione permette anche il passaggio a Flex di opzioni che altrimenti si dovrebbero passare da riga di comando, mediante lo statement *%option*. Ad esempio, tramite la seguente direttiva:

```
%option noyywrap
```

si fa in modo che non venga chiamata la funzione *yywrap()* dopo che è stato raggiunto un carattere di *end-of-file* (fine file), ma piuttosto sia assunto che non esistano più file da analizzare e ottenere la terminazione dello scanner.

3.2.3 Sezione rules

La sezione *rules* contiene una lista di regole nella forma: *modello azione*. Precisamente, il *modello* non è altro che un'espressione regolare che permette di selezionare determinate stringhe di caratteri. Una caratteristica peculiare delle *regole* è che esse utilizzano i *nomi* associati alle diverse *definizioni* della sezione *definitions*, ciò le rende più compatte ma purtroppo anche meno comprensibili.

Per quanto riguarda le *azioni* invece, queste sono in realtà porzioni di codice C. Precisamente, ogni *azione* è associata ad un *modello* specifico e sarà eseguita integralmente solo dopo che tale *modello* è stato riconosciuto dall'analizzatore lessicale.

Di seguito sono riportate e descritte singolarmente, alcune coppie di *modello-azione*:

Facendo riferimento alla definizione di identificatore data in precedenza ID [a-zA-Z]+ il seguente statement permette di estrapolare identificatori (sotto forma di stringhe contenenti caratteri maiuscoli e/o minuscoli) dal flusso in ingresso.

```
{ID} { printf("IDENTIFICATORE\n"); }
```

Il prossimo permette invece riconoscere le stringhe:

```
{STRING} { printf("STRINGA\n"); }
```

Infine il seguente statement fa riconoscere numeri interi con o senza segno:

```
{SIGN}?{DIGIT}+ { printf("INTERO\n"); }
```

Il simbolo "+" a destra di `{DIGIT}+` indica un sequenza di *DIGIT*, ovvero una sequenza di cifre da 0 a 9, come specifica la dichiarazione di `DIGIT` data sopra. Inoltre il segno del numero è reso opzionale dal punto interrogativo.

Per ignorare durante l'analisi lessicale i caratteri spazio "\t" e ritorno carrello "\n" basta non associare alcuna azione alla definizione che comprende tutte le loro combinazioni in modo che esse siano riconosciute ma ignorate dall'analizzatore.

```
[ \t\n]+  /* nessuna azione per ignorare spazi e invii */
```

Questa forma è equivalente ad aprire e chiudere parentesi graffe, ovvero associare un'*azione* nulla al *modello*.

3.2.4 Sezione user code

La sezione *user code* contiene semplicemente codice utente che sarà copiato direttamente nel file prodotto da Flex. Questa sezione è opzionale e viene spesso usata per contenere funzioni chiamate dallo *scanner* tramite le *azioni*. Se non esiste la necessità di alcuna funzione di supporto la sezione *user code* può essere lasciata completamente vuota.

Se si desidera generare uno *scanner* utilizzando esclusivamente Flex, sarà necessario inserire comunque in questa sezione le funzioni che gli permettono di funzionare, compresa la funzione main. Quando però, si implementa un parser utilizzando software di supporto come Bison, la sezione *user code* di Flex si lascia generalmente vuota in quanto le funzioni che implementano l'intero parser possono essere inserite nell'apposita sezione di Bison.

Nell'esempio proposto di seguito, la funzione *yylex()* è chiamata all'interno della procedura principale e scorrerà l'intero insieme di dati fin quando incontrerà il carattere di fine file *EOF* per poi ritornare alla funzione chiamante. Nella sezione di codice utente sarà riportato ciò che segue:

```
int
main (int argc, char** argv)
```

```
{
  ++argc; --argv;
  if(argc > 0)
    yyin = fopen(argv[0], "r");
  else yyin = stdin;
  yylex();
  return 0;

}
```

Nello specifico, si associa il flusso di ingresso dello *scanner* ad un file passato come argomento da riga di comando oppure direttamente allo standard input se il primo non è presente. Questo lo si fa semplicemente assegnando alla variabile globale *yyin* il corretto flusso di ingresso dal quale lo *scanner* estrapolerà un elemento alla volta fino al carattere EOF, a seguito del quale ritornerà il valore 0.

3.2.5 Analisi dell'input

Quando lo scanner è attivo, esamina il proprio input alla ricerca delle stringhe che corrispondono ai *modelli (match)*. Se una stringa corrisponde a più di un *modello* l'analizzatore considererà la regola che compare per prima nel file di Flex.

Appena è determinato un *match*,l'indirizzo del testo corrispondente al match (il *token*) viene copiato nel puntatore a caratteri *yytext* e la sua lunghezza è riportata nella variabile globale *yyleng*. Successivamente viene eseguita l'azione corrispondente al *match,* al termine della quale l'analizzatore lessicale torna quindi ad esaminare l'input alla ricerca di un altro *match*. Nel caso non sia rilevato alcun *match* viene eseguita una regola di default: i caratteri che non corrispondono ai modelli sono considerati *match* e inseriti in output uno alla volta. Se ad esempio, in Flex non viene definita alcuna regola si otterrà uno scanner che restituisce in output tutti i singoli caratteri contenuti nell'input uno alla volta.

Si noti che *yytext* può essere definita in due differenti modi: come puntatore a caratteri oppure come array di caratteri. E' possibile scegliere uno tra i due modi utilizzando le direttive %pointer o %array nella sezione *definitions* di Flex. Se non si specifica alcuna direttiva tra queste, viene selezionata per default la modalità %pointer. Utilizzando la direttiva %pointer si ottiene uno scanner più veloce ma diventa più difficoltoso

utilizzare o modificare *yytext*. La direttiva %array non è utilizzabile se si usano classi C++ per implementare lo scanner.

3.2.6 Generazione dell'analizzatore sintattico

L'output di Flex è un file chiamato lex.yy.c che contiene la funzione *yylex*() ed altre ruotine ausiliarie. Per default *yylex()* è dichiarata come segue:

```
int yylex()
{
... definizioni varie e azioni ...
}
```

Questa definizione può essere cambiata definendo la macro YY_DECL. Ad esempio si potrebbe utilizzare

```
#define YY_DECL float lexscan( a, b ) float a, b;
```

In questo modo il nome della funzione *yylex()* diventerà *lexscan*, avrà in ingresso due float e restituirà un float.

Quando *yylex()* è chiamata, essa esamina il file puntato dal puntatore *yyin* e continua finchè viene raggiunto *end-of-file*, a questo punto restituisce il valore 0. Se *yylex()* termina la scansione a causa dell'esecuzione di un return all'interno di una azione, lo scanner può essere richiamato e potrà riprendere il suo lavoro dal punto in è stato interrotto. Questo meccanismo rende possibile l'interazione tra Flex e Bison come si vedrà in seguito.

Al termine del suo input, *yylex()* chiama la funzione *yywrap()*. Tale funzione ha il compito di decidere se terminare completamente l'analisi lessicale o se continuare con la scansione di un altro file modificando la variabile *yyin*. Se *yywrap()* ritorna 1 l'analisi termina definitivamente, altrimenti la *yywrap()* può modificare *yyin* e ritornare 0 per comunicare *a yylex()* di continuare a leggere da un altro file. *yyin* sarà sempre accessibile sia a *yylex()* che a *yywrap()* perché è una variabile globale.

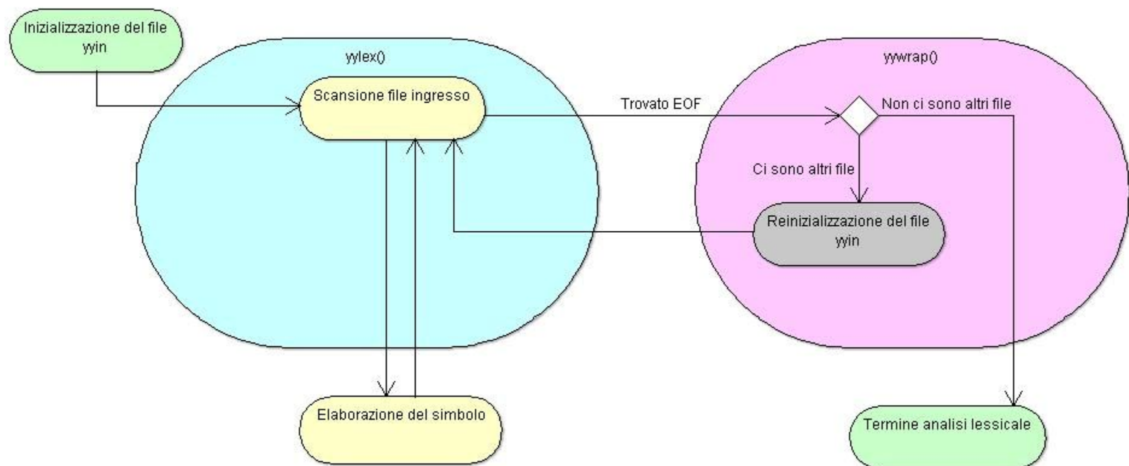

Figura 3.3: interazione tra yylex() e yywrap()

3.2.7 Condizioni di Start

Flex prevede un meccanismo per attivare le regole in base ad differenti condizioni. Ogni regola, dichiarata con un prefisso del tipo "<start_condition>", viene eseguita solo se lo scanner si trova nella condizione nominata start_condition. Ad esempio:

```
<STRING>[^"]* { /* fa qualcosa se si è in condizione STRING */}
```

In questo caso quando lo scanner incontra una stringa esegue l'azione corrispondente solo se la condizione attiva in quel momento è STRING, altrimenti la regola non viene eseguita.

Le condizioni di start devono essere dichiarate nella prima sezione di Flex facendole precedere dalla direttiva `%s` oppure `%x`. La prima di queste due direttive indica che la condizione di start dichiarata è di tipo *inclusive* mentre la seconda si riferisce al tipo *exclusive*. La condizione è attivata utilizzando l'azione BEGIN. Quando una condizione è attiva in modo *exclusive* lo scanner eseguirà soltanto le regole precedute da quella specifica condizione, mentre se la condizione attiva è *inclusive* lo scanner eseguirà le regole precedute dalla specifica condizione ed anche quelle che non hanno condizione, saranno quindi ignorate solo le azioni che appartengono a condizioni diverse da quella attiva. Ad una regola si possono attribuire anche più di una condizione.

Questo meccanismo permette la coesistenza di più tipi di scanner assieme in quanto in funzione di quale sia la condizione attiva può essere eseguita una analisi sintattica specifica diversa. Per illustrare come ciò possa essere realizzato in pratica si riporta un esempio di uno scanner che prevede due differenti tipi di interpretazione di una stringa come “555.321”. Per default la stringa sarà riconosciuta come costituita da tre *token*: l’intero “555”, un carattere “.” e un itero “321”. Se però lo scanner incontra la stringa “expect-float” allora sarà attivata la condizione expert e l’interpretazione di “555.321” sarà un unico token, il numero reale 55.321. (Paxson 1995)

```
%{
#include <math.h>
%}
%s expect
%%
expect-floats BEGIN(expect);
14
<expect>[0-9]+"."[0-9]+ {
printf( "found a float, = %f\n",
atof( yytext ) );
}
<expect>\n {
/* that’s the end of the line, so
* we need another "expect-number"
* before we’ll recognize any more
* numbers
*/
BEGIN(INITIAL);
}
[0-9]+ {
Version 2.5 December 1994 18
printf( "found an integer, = %d\n",
atoi( yytext ) );
}
"." printf( "found a dot\n" );
```

L’azione BEGIN(INITIAL), che equivale a BEGIN(0), riporta lo scanner nella condizione di partenza. Ogni volta che è chiamata un’azione BEGIN, lo scanner genera una serie di variabili dinamiche per gestire le condizioni, se tali chiamate sono eccessive esiste il rischio di saturare la memoria dinamica del processo che causerebbe inevitabilmente l’aborto del parser.

3.3 Esempio di file per Flex

Quanto detto sopra può essere messo in pratica con un piccolo esempio (Paxson 1995) finalizzato a evidenziare la struttura di un file per Flex. Il seguente input file per Flex genera uno scanner giocattolo per il linguaggio di programmazione Pascal in grado di riconoscere un insieme molto ristretto di keyword del linguaggio.

```
/* scanner for a toy Pascal-like language */
%{
/* need this for the call to atof() below */
#include <math.h>
%}
DIGIT [0-9]
ID [a-z][a-z0-9]*
%%
{DIGIT}+ {printf("An integer:%s (%d)\n", yytext, atoi(yytext));}
{DIGIT}+"."{DIGIT}*
      {
      printf( "A float: %s (%g)\n", yytext,atof( yytext ) );
      }
if|then|begin|end|procedure|function
      {
      printf( "A keyword: %s\n", yytext );
      }
{ID} printf( "An identifier: %s\n", yytext );
"+"|"-"|"*"|"/" printf( "An operator: %s\n", yytext );
"{"[^}\n]*"}" /* eat up one-line comments */
[ \t\n]+ /* eat up whitespace */
. printf( "Unrecognized character: %s\n", yytext );
%%
main( argc, argv )
int argc;
char **argv;
{
++argv, --argc; /* skip over program name */
if ( argc > 0 )
yyin = fopen( argv[0], "r" );
else
yyin = stdin;
yylex();
}
```

Nella prima sezione (*definitions)*, inizialmente si inserisce tra i simboli “%{“ e” %}”, il codice che deve essere copiato integralmente nel file sorgente C dello scanner, in questo

caso viene inclusa la libreria math.h . Di seguito vengono definite le espressioni regolari per individuare cifre e identificatori, DIGIT e ID.

Nella seconda sezione (*rules*), dopo il primo simbolo “%%”, vengono invece dichiarate le regole per l’analisi lessicale e le corrispondenti azioni. Come si può facilmente notare una keyword viene definita con lo statement:

```
if|then|begin|end|procedure|function
```

ciò significa che una keyword può essere una tra if, then, begin, end, procedure o function. Il simbolo “|” svloge la funzione di operatore logico or. Gli operatori sono inseriti tra doppi apici per trattarli come caratteri non alfabetici:

```
"+"|"-"|"*"|"/"
```

Ai modelli `"{"[^}\n]*"}"` e `[ \t\n]+` non corrisponde alcuna *azione* perché si vuole deliberatamente ignorare i commenti, con il primo, e caratteri vuoti (spazi) e invii, con il secondo.

Infine nella terza sezione (*user code*), si inserisce il main dello scanner che inevitabilmente deve aprire il file input in lettura, attribuire il suo indirizzo alla variabile globale *yyin* e successivamente chiamare yylex() che inizierà l’analisi sintattica.

Capitolo 4
Bison

Un parser è un programma che determina se il proprio input è sintatticamente valido e determina la sua struttura intraprendendo alcune azioni. Come avviene per gli scanner, il parser può essere scritto a mano oppure generato automaticamente da un generatore di parser a partire dalla descrizione delle strutture sintattiche del suo input. Tale descrizione è data in forma di grammatica libera da contesto. Bison (Bison 2010) è un programma che genera parser per una data grammatica libera da contesto a partire dalla descrizione della grammatica stessa.

4.1 Concetti fondamentali di Bison

In questo capitolo sono introdotti alcuni concetti fondamentali senza i quali sarebbe impossibile dare una descrizione dettagliata di Bison.

4.1.1 Grammatiche libere da contesto

La trattazione di questo argomento potrebbe essere così vasto da richiedere un intero capitolo, ma in questa sede saranno discussi esclusivamente gli aspetti necessari alla comprensione di Bison.

Una grammatica libera da contesto è una grammatica formale in cui ogni regola sintattica è espressa sotto forma di derivazione di un simbolo a partire da uno o più simboli già definiti. Queste grammatiche sono abbastanza potenti da descrivere la sintassi della maggior parte dei linguaggi di programmazione; al tempo stesso, sono abbastanza semplici da consentire un parsing molto efficiente.

Per effettuare un'analisi sintattica di un linguaggio è necessario descriverlo mediante una grammatica libera da contesto. Ciò significa che è necessario specificare uno o più gruppi sintattici e fornire le regole per costruirli in tutte le loro parti. Il sistema più comune di descrivere le regole di una grammatica libera da contesto è la forma Backus Naur che permette di definire l'intera grammatica in modo ricorsivo. Bison usa un formalismo che deriva da quest'ultima e ne conserva molti aspetti.

Nelle grammatiche formali si definisce *simbolo* ogni unità sintattica o raggruppamento di elementi sintattici. I *simboli* costituiti da più unità sintattiche o da costrutti sintattici sono detti *simboli non terminali*, mentre i simboli formati da un'unica unità sintattica indivisibile sono detti *simboli terminali*, questi ultimi sono i *token*. Bison legge una sequenza di *token* e li raggruppa in base alle regole della grammatica, se l'input è valido il risultato finale è la riduzione ad un unico *simbolo* che racchiude in sé l'input complessivo dell'intera sequenza di token. Tale *simbolo* è chiamato *simbolo di partenza* o *start symbol*.

4.1.2 Simboli in Bison

Per definire una grammatica formale in Bison è necessario utilizzare la specifica sintassi di Bison. I *simboli non terminali* sono rappresentati in Bison come identificatori in C a caratteri minuscoli. I *simboli terminali* (i token) sono invece rappresentati come identificatori a lettere maiuscole per distinguerli dai *simboli non terminali*. Un *simbolo terminale* è generalmente associato ad una keyword del linguaggio o ad un singolo carattere, come ad esempio: parentesi tonde e graffe, operatori, terminatori di righe. Quando un simbolo è costituito da un unico carattere è opportuno, quando possibile, rappresentare il *token* corrispondente tramite lo stesso carattere.

4.1.3 Valori semantici

In generale una grammatica formale elabora i token in base alla loro classificazione che può essere ad esempio un numero intero, "1234", una espressione aritmetica "2+3" o una stringa "sono una stringa". Conoscere la classificazione dei token è necessario e

sufficiente per ottenere il parsing di una sequenza di dati, ma per ottenere dei risultati significativi è necessario conoscere il preciso valore che il token rappresenta.
Ad esempio i token "1974" e "2011" sono entrambi numeri interi, appartengono perciò allo stesso tipo ma posseggono due significati ben diversi perché indicano anni differenti. Il parser può assegnare ad essi la corretta collocazione all'interno si un *simbolo* descritto dalle regole di grammatica sapendo che essi sono numeri interi senza sapere quale sia il loro valore. In realtà però, è quasi sempre necessario attribuire ad alcuni *token* il loro valore al fine di elaborarli per ottenere dei risultati, come può essere la traduzione di un determinato linguaggio.
I valori che sono attribuiti ai *token* sono chiamati *valori semantici*. Tali *valori semantici* possono appartenere a qualunque tipo predefinito del linguaggio C. Il valore semantico del token "1974" sarà dunque l'intero 1974 mentre, il token "x+1" potrebbe avere come valore semantico 3 se x=2.
E' possibile definire anche valori semantici di tipo stringa da associare a token come "*sono una stringa*", in questo caso si potrebbe avere come valore semantico un puntatore a caratteri che individua tale stringa in memoria.
Anche valori semantici composti sono permessi da Bison, basta utilizzare strutture o unioni del linguaggio C.
Per implementare un traduttore di un linguaggio, oltre che effettuare l'analisi lessicale e sintattica di questo, occorre processare i *valori semantici* ottenuti dall'analisi poiché è da essi che si ottiene la traduzione.

4.1.4 Azioni semantiche

Come accade per Flex, le regole di grammatica in Bison possono essere associate ad azioni che sono in realtà porzioni di codice C. Ogni volta che il parser riconosce un match per una regola viene eseguita l'azione corrispondente a tale regola. Il parser esegue un'azione semantica quando è necessario produrre un valore semantico da associare ad un ad un *simbolo*.

Si consideri ad esempio la seguente regola: l'espressione *somma* è la somma di due espressioni. Quando il parser riconosce tale regola, può generare il valore semantico da associare all'espressione *somma* a partire dai *valori semantici* delle due sottoespressioni

facendone la somma aritmetica. In questo caso il parser ha eseguito un'azione semantica.

4.2 Struttura di un file input di Bison

Il file in ingresso a Bison è un file che descrive una grammatica, in generale la sua forma è la seguente:

```
%{
Prologue
%}
Bison declarations
%%
Grammar rules
%%
Epilogue
```

Questo modello come si può facilmente notare, è molto simile a quello di Flex. Come in Flex la sequenza di caratteri "%%" indica la linea di confine tra le tre sezioni ed il codice inserito tra le sequenze di caratteri "%{" e "%}" viene copiato interamente nel file prodotto da Bison.
Il *prologo* può ospitare definizioni di tipi e variabili utilizzati all'interno delle azioni, oppure macro e #include per includere le librerie C che si vuole utilizzare per implementare le azioni. Nella sezione *declarations* vengono dichiarati nomi di *simboli terminali* e *non terminali*. La sezione *rules* definisce la grammatica del linguaggio da analizzare. Infine l'*epilogo* contiene codice opzionale. (Donnelly Stallman 2010)

4.2.1 Il prologo

La sezione *prologue* contiene definizioni di macro e dichiarazioni di funzioni e variabili utilizzate nelle azioni delle regole della grammatica. Queste sono copiate per intero all'inizio del file C prodotto da Bison, per cui è possibile in questa sezione usare la direttiva #include per definire la header del file prodotto. Si può avere più di una sezione *prologue*, anche inserendone qualcuna tra le dichiarazioni.

Questa sezione è destinata ad ospitare le dichiarazioni delle *union* che ospiteranno i *valori semantici* attribuiti ai simboli che si incontreranno durante l'analisi sintattica.

4.2.2 Sezione declarations

La sezione declarations definisce i simboli utilizzati per dichiarare la grammatica ed i tipi di dato dei *valori semantici*. Qui devono essere dichiarati tutti i tipi di token, eccetto quelli formati da singoli caratteri. I *simboli non terminali* devono essere dichiarati se esiste la necessità di specificare il tipo di dato da usare per il *valore semantico*. La prima regola della sezione deve obbligatoriamente specificare il *simbolo* di partenza, *start symbol*. Tutti i token sono dichiarati usando la direttiva `%token` con la sintassi:

```
%token nometoken
```

```
//Sezione declarations:

%union semrec /* La Semantica degli Identificatori */
{
int intval; /* Valori Interi */
char *id; /* Identificatori */
}
/*=================================================================
                                TOKENS
=================================================================*/

%start dispositivo
%token <intval> NUM /* Numero intero */
%token <id> IDENTIFICATORE /* Identificatore-Stringa */
%token CELL STATUS COMMENT KEYWORDMAP VERSION AUTHOR
KEYWORDLEVEL
%token EXTERNAL EDIF EDIFVERSION TECHNOLOGY VIEW EDIFLEVEL
PROGRAM
%token DIRECTION PORT ATONDA CTONDA CELLTYPE TIMESTAMP WRITTEN
%token INTERFACE TECHNOLOGYDEFINITION VIEWTYPE ACCOUNTING
%token INSTANCE DESIGN LOCATION QUALIFY STRINGA PROPERTY
DESIGNATOR
%token CONTENTS VIEWREF CELLREF LIBRARYREF NET PORTREF JOINED
%token RENAME ARRAY
%%

//Sezione rules...
```

Nell'esempio precedente, nella sezione *prologo* viene dichiarata l'unione che contiene i *valori semantici* dei token. Il *simbolo di start* che raggruppa l'intero insieme di simboli contenuti nell'ingresso del parser è chiamato *dispositivo*. I *token* ai quali vengono attribuiti *valori semantici* sono `NUM` e `ID`. Di seguito sono dichiarati tutti i token necessari per comporre le regole della grammatica.

4.2.3 Sezione rules

La sezione *rules* contiene una o più regole della grammatica e niente altro. Esistono tre differenti modi di inserire un *simbolo terminale* all'interno delle regole della grammatica:

- Se il *simbolo* è già dichiarato come token, si inserisce con il nome del token come se fosse un identificatore.
- Se il *simbolo* è un carattere, si inserisce nella grammatica usando la stessa sintassi del C senza essere dichiarato come token. Per convenzione un token che consiste in un unico carattere è usato solo per rappresentare quel carattere.
- Se un *simbolo* è una stringa può essere inserito come una costante stringa del C come ad esempio "!=". Un token stringa non ha bisogno di essere dichiarato se non si deve attribuire ad esso un valore semantico.

Una regola di grammatica ha in Bison la seguente forma:

```
result: components...
;
```

dove `result` è il *simbolo non terminale* che la regola descrive e `components` sono i vari *simboli terminali* e *non terminali* raggruppati assieme dalla regola. Ad esempio:

```
exp: exp '+' exp
;
```

significa che un'espressione, `exp`, può essere formata da altre due espressioni dello stesso tipo separate dal carattere "+". Gli spazi vuoti all'interno della regola sono utilizzati esclusivamente per separare i simboli. Quando una regola è riconosciuta, viene

automaticamente eseguita l'azione associata ad essa. Come avviene per Flex, un'azione si specifica all'interno di parentesi graffe e si posiziona al seguito della regola.

```
{ codice C }
```

La correttezza del codice specificato all'interno delle parentesi graffe non è controllata da Bison poiché esso si limita soltanto a copiarlo nel file C generato. Il compito di verificare il codice delle azioni è demandato al compilatore C. Più azioni possono essere associate ad una regola ma generalmente si usa associarne una sola.
Esiste un modo per associare allo stesso *simbolo* `result` più regole seguite dalle rispettive azioni; consiste nel separare le varie regole mediante il carattere barra verticale "|" che funge da operatore logico *or* tra espressioni regolari.

```
proprietà  :          /* nessuna */
                      |
                      STRINGA
                       |
                       IDENTIFICATORE INTERO { codice azione }
;
```

In questo esempio il simbolo `proprietà` può essere: una stringa, un identificatore seguito da un numero intero , nessun *simbolo*.

La potenza di questa notazione sta soprattutto nel fatto che con essa è possibile specificare regole in modo ricorsivo, ad esempio la seguente coppia di regole

```
infoseq:        info
                |
                infoseq info
;
```

definisce il *simbolo* sequenza di info, `infoseq`, che può essere un solo *simbolo* `info` oppure una sequenza di *simboli* `info` seguita da una `info`. Ciò significa che una `infoseq` può essere una sequenza di un qualsiasi numero di `info`.

4.2.4 Epilogo

L'*epilogo* contiene le funzioni implementate dal programmatore. Tutto il codice contenuto in esso è copiato interamente nel file generato da Bison come accade per il *prologo*. Generalmente si inseriscono nell'epilogo le funzioni di supporto al parser ed in particolare la funzione main.

4.3 Azioni

Un'azione accompagna una regola sintattica e contiene codice che viene eseguito ogni volta che la regola è riconosciuta. Quasi sempre, il compito delle azioni è computare il *valore semantico* del *simbolo* descritto dalla regola. Tale *valore semantico* è calcolato in funzione dei singoli *valori semantici* associati ai token raccolti dalla regola.

4.3.1 Riferimenti a valori semantici

Un'azione ha accesso ai *valori semantici* contenuti all'interno della sua corrispondente regola utilizzando il costrutto $n che indica l'ennesimo componente nella regola. Il valore semantico del *simbolo* risultante è invece accessibile con il costrutto $$.

Un tipico esempio può essere il seguente:

```
exp:        ...
            | exp1 '+' exp2
            { $$ = $1 + $3; }
```

Questa regola costruisce l'espressione `exp`, a partire da due espressioni più piccole unendole con il carattere "+". $1 e $3 che sono contenuti nell'azione, si riferiscono rispettivamente ai *valori semantici* del primo e del terzo *simbolo* contenuti nella regola. Precisamente, $1 indica il *valore semantico* di `exp1`, $3 indica il *valore semantico* di `exp2` e `$$` indica il *valore semantico* di `exp`. Bison scambia i costrutti $n e $$ con le espressioni di tipo appropriato, le sostituisce all'interno delle azioni ed infine riporta

il codice di quest'ultime nel file del parser. Come in pratica è realizzato questo meccanismo è discusso nella successiva sezione.

Quando non si specifica un'azione per una regola, Bison assegna comunque un valore semantico per default al *simbolo* risultante. Questo valore è quello che corrisponde al primo *token* nella regola, quindi $$=$1.

4.3.2 Tipi di dati e valori nelle azioni

Il parser generato da Bison memorizza i *valori semantici* dei token in una variabile globale chiamata *yyval*. Quando si utilizza un solo tipo di dato per i *valori semantici* dei token, la variabile *yyval* appartiene esclusivamente a quel tipo.

Se invece, si usano più tipi di dato, *yyval* è una unione strutturata come specificato dalla direttiva *%union* nella sezione *prologo*. Tale direttiva specifica l'intero insieme di tipi di dato necessari a gestire i *valori semantici*. `%union` è seguita da un blocco di codice molto simile alla dichiarazione di una union in C. Ad esempio, lo statement

```
%union {
int numero;
char* stringa;
}
```

dichiara che i token del parser utilizzano due tipi di dato: interi o stringhe di caratteri. I nomi delle variabili che costituiscono l'unione devono essere specificati quando si dichiara un token nella sezione *declaration* per informare Bison che il valore semantico di quel token deve essere contenuto in quella specifica variabile membro di *yyval*.

```
%token <numero> NUM /* Numero intero */
```

In questo caso si specifica che per il token `NUM` è previsto un *valore semantico* che deve essere memorizzato nella variabile membro *numero* dell'unione *yyval*. Il tipo del valore semantico deve essere ovviamente uguale al tipo della variabile membro di *yyval*. Se infatti si vuole utilizzare il token `IDENTIFICATORE` con valore semantico di tipo stringa occorre associarlo alla variabile `stringa` di *yyval*.

```
%token <stringa> IDENTIFICATORE /* Stringa */
```

4.4 Interazione tra Bison e Flex

L'interazione tra Bison e Flex è realizzata mediante la cooperazione tra la funzione *yylex* dello scanner e la funzione *yyparse* del parser . La funzione *yyparse* chiama più volte la funzione *yylex* dello scanner per ottenere i token. Ogni volta che *yylex* viene eseguita, restituisce a *yyparse* un valore che corrisponde ad un preciso tipo di token dell'input analizzato. *yyparse* esegue l'analisi sintattica esaminando la sequenza dei valori ottenuti da *yylex*. (Donnelly Stallman 2010)

4.4.1 Acquisizione dei token

Tutti i token utilizzati dal parser devono essere specificati nella sezione *declarations* di Bison. La lista dei token dichiarati viene convertita in un tipo enumerato per cui ad ogni token si fa corrispondere un numero intero positivo. Ogni volta che un'azione dello scanner esegue un return, è al ritornato parser il numero intero corrispondente ad un determinato token.

La seguente coppia *modello azione* di Flex fa in modo che sia ritornato il numero intero corrispondente a `PORT` ogni volta che nell'input d'ingresso è rilevata l'espressione "`porta`"

```
port { return(PORT); }
```

4.4.2 Acquisizione dei valori semantici

Per rendere disponibili i *valori semantici* dei token al parser, essi devono essere inseriti nella variabile globale *yyval* dallo scanner. Quando lo scanner riconosce un token a cui si deve attribuire un valore, l'azione corrispondente deve scrivere quel valore in *yyval* e poi ritornare l'intero rappresentante il token rilevato. L'esempio che segue mostra una porzione di codice in Flex che dichiara alcuni tipi di *modelli*, nella sezione *definitions*.

```
                  (sezione definitions di Flex)
//=============================================================
DIGIT [0-9]
ID [_A-Za-z][_A-Za-z0-9]*
STRING ["][^"]*["]
%%
//=============================================================
```

```
                     (sezione rules di Flex)
//===============================================================
{DIGIT}+ { yylval.intval = atoi( yytext ); return(NUM); }
...
```

Quando viene riconosciuta una sequenza di cifre, `{DIGIT}+`, l'azione corrispondente calcola il valore numerico corrispondente all'espressione trovata utilizzando la funzione *atoi()* e memorizza tale valore nella variabile *intval* membro *yyval*. Successivamente l'azione ritorna a *yyparse()* l'intero positivo `NUM` corrispondente al tipo di token che è definito nella sezione *definitions* di Bison come segue.

```
%token <intval> NUM /* Numero intero */
```

4.4.3 Interazione tra yyparse() e yylex()

Per comprendere il meccanismo di interazione tra lo scanner ed il parser è opportuno partire esaminando il main della funzione *yyparse*(). La funzione main non è generata automaticamente da Bison ma è implementata interamente dal programmatore. Il main deve eseguire obbligatoriamente tre azioni fondamentali.

- Aprire in lettura il file input del parser
- Attribuire alla variabile globale *yyin* l'indirizzo del file input
- Chiamare la funzione *yyparse()*

La funzione *yylex()* può accedere in qualsiasi momento al file input tramite la variabile globale *yyin* che viene inizializzata nel main del parser. *yyparse()* esegue l'analisi sintattica chiamando iterativamente più volte la funzione *yylex().* Ogni volta che *yylex()* riconosce un token, inserisce il suo *valore semantico* nella variabile *yyval* (se tale token possiede un *valore semantico*) e ritorna a *yyparse()* il numero intero positivo che corrisponde al token trovato. Tutte le volte che *yylex()* viene richiamata*,* ricomincia a leggere il file in ingresso dal punto in cui è stata interrotta.

Quando *yylex()* incontra il carattere *end-of-file*, *yylex()* termina restituendo zero a *yyparse*() che comprende che la lettura del file input è stata portata a termine. Dopo aver riconosciuto l'ultima espressione regolare ricevuta, l'analisi sintattica è conclusa e il

parser può terminare definitivamente o ricominciare ad esaminare un altro file in ingresso.

Si riporta di seguito un breve esempio di funzione main dichiarata nell'epilogo di Bison.

```
main( int argc, char *argv[] )
{
extern FILE *yyin;
++argv; --argc;
yyin = fopen( argv[0], "r" ); // yyin punta al file di ingresso
yyparse (); // esecuzione analisi sintattica
printf ( "Parse Completed\n" );
}
```

Il seguente diagramma mostra le fasi che si susseguono durante il processo di parsing e le interazioni di *yyparse()* e *yylex()* come sopra descritto.

Figura 4.1: diagramma a stati del parser

Si noti che quando il parser termina l'analisi sintattica del file in ingresso, controlla se ci sono altri file da analizzare. In tal caso viene modificato il puntatore a file *yyin* e l'analisi sintattica ricomincia con il file successivo. La funzione *yywrap(),* che svolge questo compito per lo scanner realizzato con Flex, non è quindi più utilizzata perché le sue funzioni sono demandate al main del parser.

4.4.4 Controllo di coerenza del contesto

Flex e Bison consentono di esaminare il file di input anche per rilevare alcuni errori che non possono essere rilevati mediante la sola applicazione delle regole sintattiche e grammaticali. Quando si deve realizzare un compilatore, è necessario ad esempio che ogni variabile dichiarata non sia mai dichiarata più di una volta. Inoltre è corretto che ogni variabile dichiarata non sia usata se non prima della sua inizializzazione.
Bison e Flex consentono di effettuare il controllo della coerenza di questo tipo di informazioni, dette informazioni di contesto.
Per effettuare un controllo su tali informazioni, esse devono essere prima di tutto inserite in una struttura chiamata *symbol table*. La *symbol table* è una struttura dati in cui viene memorizzato un insieme di identificatori che il parser individua durante l'analisi sintattica. Può essere implementata usando liste, tabelle hash, alberi o altro a discrezione del programmatore.

Il seguente esempio realizza una semplice *symbol table* che può essere utilizzata per verificare se nel file di input è dichiarato più volte lo stesso identificatore per una variabile.

```
struct symrec
{
char *name; /* name of symbol */
struct symrec *next; /* link field */
};
```

La struttura *symrec* definisce una lista di identificatori. Ogni volta che il parser incontra la dichiarazione di una variabile deve per prima cosa controllare che essa non sia già presente in questa lista. Se l'identificatore di tale variabile non è ancora in lista il parser chiama l'azione che provvede all'inserimento dell'identificatore altrimenti rileva l'errore di duplice dichiarazione oppure di uso di variabile non dichiarata.
Per controllare se l'identificatore non è ancora stato dichiarato si usa la funzione *getsym*() che cerca l'identificatore nella struttura *symrec*. Se l'identificatore non esiste nella lista, la funzione restituisce zero, altrimenti ritorna il puntatore relativo all'identificatore nella lista.

```
symrec *getsym ( char *sym name )
{
symrec *ptr;
for (ptr = sym table; ptr != (symrec *) 0;
ptr = (symrec *)ptr- >next)
if (strcmp (ptr- >name,sym name) == 0)
return ptr;
return 0;
}
```

Per l’inserimento dell’identificatore nella lista si usa la funzione *putsym*() riportata di seguito.

```
symrec *putsym ( char *sym name )
{
symrec *ptr;
ptr = (symrec *) malloc (sizeof(symrec));
ptrs >name = (char *) malloc (strlen(sym name)+1);
strcpy (ptr  >name,sym name);
ptrs >next = (struct symrec *)sym table;
sym table = ptr;
return ptr;
}
```

Capitolo 5
Il Traduttore per EDIF

L'obbiettivo raggiunto è la realizzazione di un traduttore per il sottoinsieme del linguaggio EDIF 2.0 utilizzato dal tool ISE di Xilinx (ISE 2000). In generale, un traduttore per un linguaggio è un programma che traduce il codice sorgente scritto in un determinato linguaggio in codice appartenente ad un linguaggio oggetto differente. Un traduttore utilizza un *parser* per effettuare l'*analisi sintattica* e dal risultato di quest'ultima ricava la traduzione.

La differenza sostanziale tra il linguaggio EDIF 2.0 ed il suo sottoinsieme utilizzato da Xilinx risiede nel fatto che quest'ultimo utilizza esclusivamente view di tipo netlist per la descrizione dei circuiti elettronici. Tale caratteristica rende l'EDIF di ISE sintatticamente meno complesso del suo fratello maggiore.

Partendo da un circuito elettronico descritto utilizzando il linguaggio EDIF, il traduttore realizzato elabora una traduzione che rappresenta completamente il circuito elettronico di partenza descritto con EDIF in un linguaggio sintetico e compatto, svincolato dall'utilizzo di identificatori di tipo stringa utilizzati da EDIF. Tale traduzione contiene soltanto le informazioni necessarie alla descrizione del circuito a differenza dei file EDIF che contengono anche un gran numero di commenti e informazioni non attinenti alle caratteristiche dei dispositivi.

Per la realizzazione del traduttore è stato implementato un parser costituito da un analizzatore lessicale generato utilizzando Flex ed un analizzatore sintattico generato utilizzando Bison. Il parser realizzato è stato utilizzato per memorizzare le informazioni del circuito in una struttura dati. Infine, sono stati implementati i metodi che consentono di ottenere la traduzione nel linguaggio oggetto utilizzando le informazioni memorizzate nella struttura dati.

5.1 Il linguaggio oggetto

Il linguaggio in cui si traducono le netlist EDIF è costituito da una serie di entry. Una entry è una linea di testo che termina con il carattere terminatore ';'. I campi di ogni entry sono separati da spazi. Una entry ha in generale la seguente forma:

```
<component_ID> <entry_type> <parameters>
```

Una o più entry rappresentano un componente. Il primo campo della entry è l'identificatore del componente, cioè un numero compreso tra 0 ed il numero dei componenti nel circuito meno 1. Il secondo campo è il tipo del componente rappresentato da una stringa di caratteri. Il terzo campo contiene i parametri che rappresentano le interconnessioni o le funzioni logiche specifiche dei componenti.

5.1.1 Il lessico

Ogni componente di un circuito appartiene ad una specifica categoria e svolge funzioni differenti dagli altri per cui è necessario che siano distinti innanzitutto in base al loro tipo. Il linguaggio oggetto utilizza per praticità gli stessi tipi di dato usati da ISE, alcuni tipi sono ad esempio:

- Per i Buffer
  ibuf, obuf

- Per i Flip flop
  fd, fdc, fdp, fdce, fdcp, fdce, fdcpe

- Per le Look up table
  lut

- Per i Multiplexer
  mux

Per gli identificatori dei componenti si usano semplicemente i numeri naturali, ovvero ogni componente nel circuito è individuato univocamente da un numero naturale.

I componenti di tipo *ibuf* e *obuf* sono rispettivamente buffer di ingresso e buffer di uscita del circuito elettronico. Essi interconnettono il circuito con l'esterno. I tipi di componenti *fd, fdc, fdp, fdce, fdcp, fdce, fdcpe* rappresentano vari tipi di flip flop. Le lettere che seguono la prima lettera *f* del nome indicano i pin che il flip flop possiede: *d* è il pin data, *c* clear, *p* pre-charge, *e* enable. Il tipo *lut* rappresenta le look up table con numero di ingressi variabili da uno a sei, mentre *mux* rappresenta i multiplexer.

Dato che ogni componente del circuito possiede una sola uscita, è possibile determinare tutte le interconnessioni del circuito considerando solamente le connessioni dei pin di ingresso dei singoli componenti.

5.1.2 La sintassi

La sintassi delle entry varia in base al tipo di componente che esse rappresentano.

Per i buffer di ingresso il primo campo è l'identificatore numerico del buffer. Il secondo campo è il tipo di componente cioè *ibuf*. Il terzo campo comprende un solo parametro, cioè il numero del pin del circuito collegato al buffer. Un semplice buffer di ingresso collegato al pin numero 1 del circuito è rappresentato dalla seguente entry:

```
1 ibuf 1;
```

Le entry *obuf* invece, hanno due parametri nel campo *parameters*, di cui il primo e`l'identificatore del componente collegato al buffer, e il secondo e` il numero del pin di uscita del circuito.

Ogni look up table è rappresentata da due entry: una di tipo *lut_fctn* ed una di tipo *lut*. La entry di tipo *lut_fctn* ha come unico parametro una stringa rappresentante la funzione logica della LUT. Tale stringa rappresenta l'uscita del componente in funzione dei suoi ingressi che compaiono sotto forma di implicanti. Gli ingressi delle LUT: I0, I1, I2, I3, I4, I5 sono rispettivamente indicati con i caratteri '0', '1', '2', '3', '4', '5'. Il simbolo utilizzato per indicare la negazione è il '!' mentre il simbolo '+' indica *or*. Per la entry *lut* invece, il primo parametro è il numero di piedini d'ingresso della look up table mentre i parametri successivi sono, ordinatamente, gli identificatori dei componenti

collegati ai piedini d'ingresso della LUT: il secondo parametro per il piedino I0, il terzo per il piedino I1 etc.

Le due entry che seguono specificano una LUT a quattro ingressi con identificatore 9. Il piedino d'ingresso I0 è collegato al componente 14, il piedino d'ingresso I1 è collegato al componente 16, e così via.

```
9 lut_fctn 02+013+!2!3+2!3+!0!1!2;
9 lut 4 14 16 15 17;
```

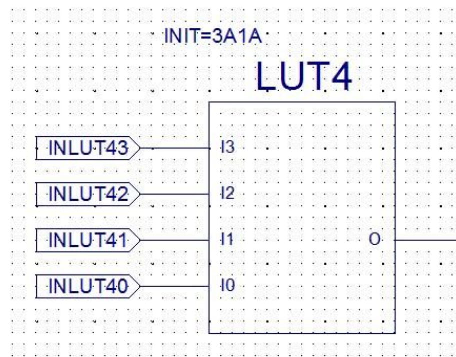

Figura 5.1: Look up Table a quattro ingressi

Per le entry dei tipi di flip flop si adotta la convezione di indicare, dal primo parametro in poi, i pin di ingresso del componente in ordine antiorario partendo dall'alto. Il pin C collegato al clock, non viene però considerato poiché si presuppone di avere nel circuito un unico segnale di clock uguale per tutti i componenti. La entry

```
9 fdce 7 6 5;
```

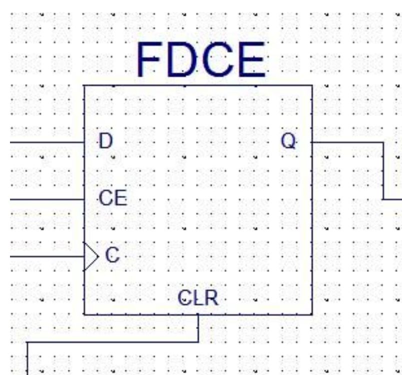

Figura 5.2: Flip flop fdce

rappresenta un flip flop il cui pin D è collegato al componente 7 nel circuito, il pin CE è collegato al componente 6 e il pin CLR è collegato al componente 5.

I multiplexer sono rappresentati in maniera simile alle LUT. La entry *mux_fctn* serve a indicare quanti sono e a quali componenti sono collegati i pin dei selettori, mentre la entry *mux* indica quanti sono i pin dati e a quali componenti sono collegati.

```
13 mux_fctn 1 10;
13 mux 12 11;
```

Anche i *bus di input* ed i *bus di output* sono riconosciuti dal parser, essi non compaiono però esplicitamente sotto forma di entry nel linguaggio poiché sono individuati da una serie di buffer di uscita o una serie di buffer di ingresso.

5.2 Realizzazione del traduttore

Stabilite le regole sintattiche e grammaticali del linguaggio oggetto con cui bisogna descrivere il circuito, si procede alla progettazione del traduttore vero e proprio. Tramite il software di supporto Bison e Flex si è in grado di generare un parser capace di eseguire l’elaborazione di un testo EDIF e di estrarre da esso le informazioni necessarie per la descrizione dettagliata del circuito.

Per prima cosa è stata progettata una struttura dati a tempo di esecuzione per contenere tutte le informazioni relative ai componenti del circuito e le loro interconnessioni. Tale struttura è stata pensata in modo che possa sempre prestarsi a qualunque altro tipo di traduzione si possa desiderare in futuro. Infatti essa è completamente indipendente dal tipo di linguaggio con il quale vogliamo descrivere il circuito. Questo consente che la struttura dati sia riutilizzabile.

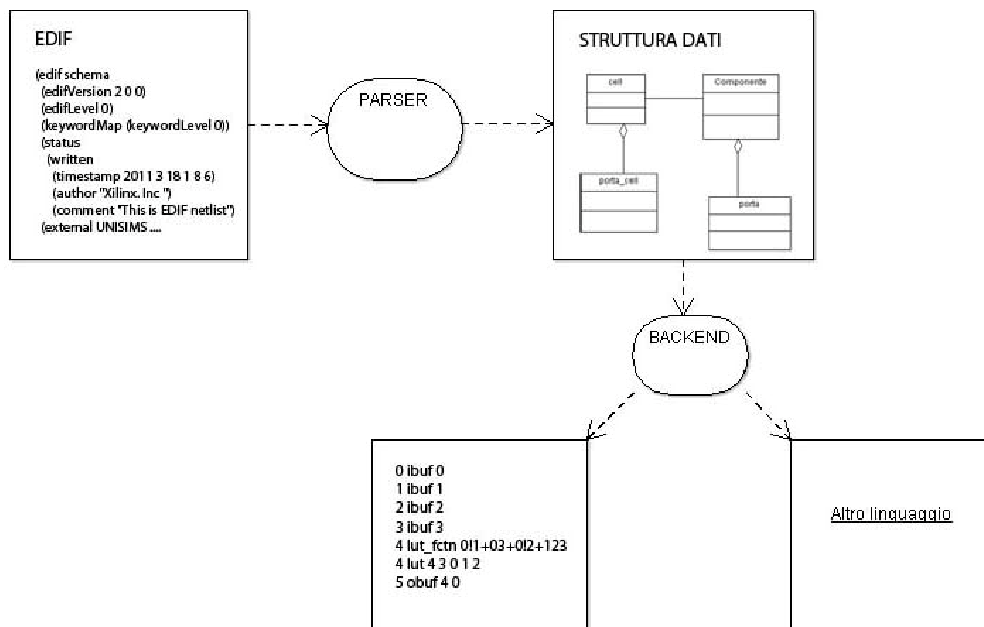

Figura 5.3: Flusso di dati del traduttore

Ciò che in pratica realizza l'indipendenza tra il linguaggio oggetto e la struttura dati del circuito è un processo di *back end* (vedi figura 5.3) realizzato da un insieme di funzioni che concorrono assieme al fine di ottenere la traduzione. I metodi che realizzano il processo di *back end* sono le funzioni che producono la traduzione a partire dalle informazioni contenute nella struttura dati. Dopo che il parser ha eseguito l'analisi sintattica del file di input, il processo di *back end* produce la vera e propria traduzione elaborando le informazioni che sono state memorizzate dal parser stesso nella struttura dati. Si può dire che il processo di *back end* assieme al parser realizza il traduttore.

Basta modificare le funzioni di *back end* per ottenere una diversa traduzione, senza apportare alcuna modifica né al parser né alla struttura dei dati. La struttura dati contiene informazioni acquisite direttamente dal file EDIF e non processate, poiché il compito di elaborare ed ottenere le informazioni relative alla traduzione è lasciato al processo di *back end*.

5.2.1 La struttura dati

Il parser analizza il file EDIF carattere per carattere, catturando le informazioni utili nel testo nell'ordine di come esso esegue la scansione. Si nota dunque che la descrizione completa di ogni componente (nome, tipo delle porte, funzione logica e collegamenti) è

disponibile solo al termine di tre fasi distinte: dichiarazione *cell*, dichiarazione *instance* e dichiarazione dei *collegamenti*.

Al fine di implementare una struttura dati che tenga conto di tutti i componenti e tutti i dati associati ad essi, si deve passare attraverso strutture dati temporanee. In primo luogo si deve memorizzare tutti i tipi dei componenti, le *cell,* e le relative porte che sono utilizzate nella fase successiva per definire le istanze. Nella fase di dichiarazione delle *instance,* devono essere memorizzati tutti i componenti del circuito assieme alle porte memorizzate in precedenza. Infine è necessario inserire le informazioni relative ai collegamenti tra i componenti.

E' possibile descrivere le fasi principali di acquisizione dei dati del circuito elettronico da parte del traduttore tramite il seguente diagramma.

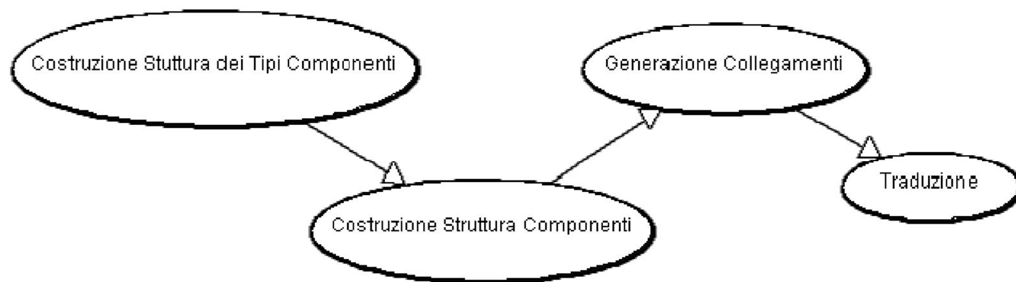

Figura 5.4: Fasi del traduttore a tempo di esecuzione in ordine cronologico

La situazione descritta induce a progettare una struttura dati ad hoc che rispetti i vincoli sopra descritti.

Nella prima fase il parser incontra le dichiarazioni dei tipi dei componenti, le *cell*. Ogni *cell* che il parser incontra è memorizzata in una struttura, chiamata *cell*, che contiene il nome della *cell*, la lista delle sue porte ed il puntatore alla *cell* successiva. Tutte le *cell* vengono concatenate assieme per formare una lista di *cell*. Ogni *cell* possiede un puntatore che permette di accedere alla lista delle sue porte. Ogni porta è memorizzata in una struttura, *porta_cell*, che contiene il nome della porta, il tipo di porta (INPUT o OUTPUT), ed il puntatore alla porta successiva della lista. Il seguente diagramma visualizza la struttura di un'unica *cell*.

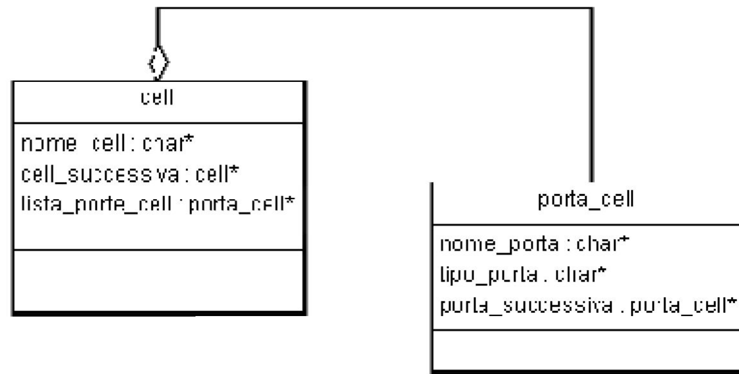

Figura 5.5: Struttura di una cell

I componenti veri e propri sono memorizzati nella struttura dati in una fase successiva alla dichiarazione delle *cell*, la fase in cui il parser incontra nel file EDIF le dichiarazioni delle *instance,* ovvero i componenti presenti nel circuito. Il parser memorizza ogni *instance* che incontra in una struttura chiamata *componente*. Tutti i *componenti* del circuito sono concatenati assieme in una lista di *componenti*. Ogni *componente* contiene il nome del componente che esso rappresenta, il tipo di componente, le informazioni relative alla logica del componente, il numero che identifica il componente nel circuito, il puntatore al *componente* successivo nella lista ed il puntatore alla lista delle porte del componente. Per generare la lista delle porte di ogni componente è necessario usufruire anche delle informazioni contenute all'interno delle *cell* memorizzate nella fase precedente.

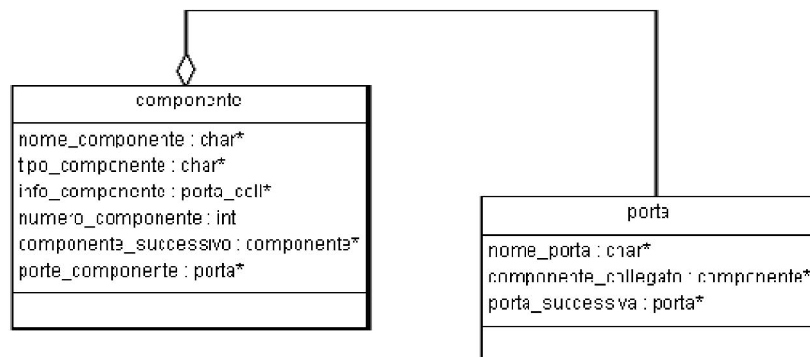

Figura 5.6: Struttura di un componente

L'ultima fase del parser consiste nella generazione dei collegamenti. In questa fase il parser incontra nel file EDIF i vari costrutti *joined* utilizzati per descrivere le interconnessioni dei componenti (cfr. 2.3.1). Ogni volta che il parser incontra un costrutto *joined* inserisce le informazioni relative ai collegamenti all'interno delle porte dei componenti. La struttura dati che rappresenta l'intero circuito è completata solo al termine di questa fase. Si può notare che la struttura *porta* non contiene il campo *tipo_porta* che invece è contenuto in *porta_cell*, ciò è conseguenza della convenzione che tutti i componenti dispongono di una sola porta d'uscita per cui non è necessario distinguere il tipo di porta all'interno dei componenti ma basta tener conto solo delle porte di input. La distinzione del tipo delle porte è invece mantenuta nella struttura *cell* poiché è necessaria per costruire i componenti.

Un rilevante problema, dovuto al fatto che i dati sono disponibili esclusivamente nell'ordine e nei tempi imposti dallo scanner, è costituito dal fatto che le azioni di Bison possono utilizzare i valori semantici relativi ai soli token presenti nella regola associata. Per questo motivo le chiamate a funzioni contenute nelle azioni possono utilizzare come parametri solo i valori semantici dei token contenuti nella regola associata a tali azioni.

```
Collegamento: ATONDA NET IDENTIFICATORE ATONDA JOINED joinseq
CTONDA istanzainfo CTONDA { collega($3);}
           | ...

join:      ATONDA PORTREF IDENTIFICATORE CTONDA
           {ins_collegamento($3,"0");}
           |...

joinseq:   join
           |
           joinseq join
```

Nella grammatica del parser realizzato, il simbolo *collegamento* viene definito come una serie di token tra i quali compare l'espressione *joinseq* che rappresenta una sequenza di *join*. Un *collegamento* viene riconosciuto solo quando un'intera sequenza di token prelevati dal file EDIF corrisponde all'espressione regolare definita come

collegamento. Quando un collegamento viene riconosciuto è chiamata la funzione *collega*() che provvede a memorizzare le informazioni all'interno della struttura dati. Un *collegamento* è però riconosciuto soltanto dopo la lettura di una *joinseq* contenuta tra i token che definiscono il simbolo *collegamento*. Per quanto detto sopra la funzione *collega*() può accedere ai valori semantici dei token presenti nell' espressione *collegamento* ma non ha accesso ai valori semantici presenti nella espressione *joinseq*.

Nasce quindi un problema dal fatto che la funzione *collega*() necessita, oltre che del nome del componente da collegare $3, anche delle informazioni di routing contenute nella espressione *joinseq*. Per risolvere tale problema occorrerebbe modificare la grammatica sostituendo ad ogni singola regola che definisce un *collegamento* tutte le regole che definiscono un *join* dando vita così ad un numero eccessivo di nuove definizioni di *collegamento* più lunghe e complesse.

In realtà una soluzione più semplice, senza stravolgere la grammatica, è quella di memorizzare in una struttura temporanea tutte le informazioni delle interconnessioni del componente contenute nella sequenza di *join* per metterle a disposizione della funzione *collega()* al momento della sua chiamata. Per realizzare questo meccanismo occorre che ogni qualvolta è riconosciuta una espressione regolare *join,* sia chiamata la funzione *ins_collegamento()* che ha il compito di memorizzare i dati relativi al *join* in una struttura dati temporanea accessibile in seguito alla funzione *collega*(). Tale struttura temporanea è chiamata *collegamento*. Con questo piccolo artificio la funzione *collega*() avrà a disposizione tutte le informazioni necessarie per inserire nella struttura dati che descrive l'intero dispositivo tutti i collegamenti di un componente.

Per le medesime motivazioni, è stata utilizzata anche la struttura temporanea *funzione_logica* che memorizza gli attributi relativi alle funzioni logiche di ogni componente prima che siano inserite nella struttura dati del circuito.

Una volta inseriti nella struttura tutti i collegamenti per ogni componente del circuito, il parser ha terminato il suo lavoro avendo ottenuto una struttura dati consistente. Tale struttura è costituita dalla lista completa dei componenti presenti nel dispositivo, ogni componente possiede in sé la lista delle sue porte di ingresso con i relativi puntatori ai componenti collegati.

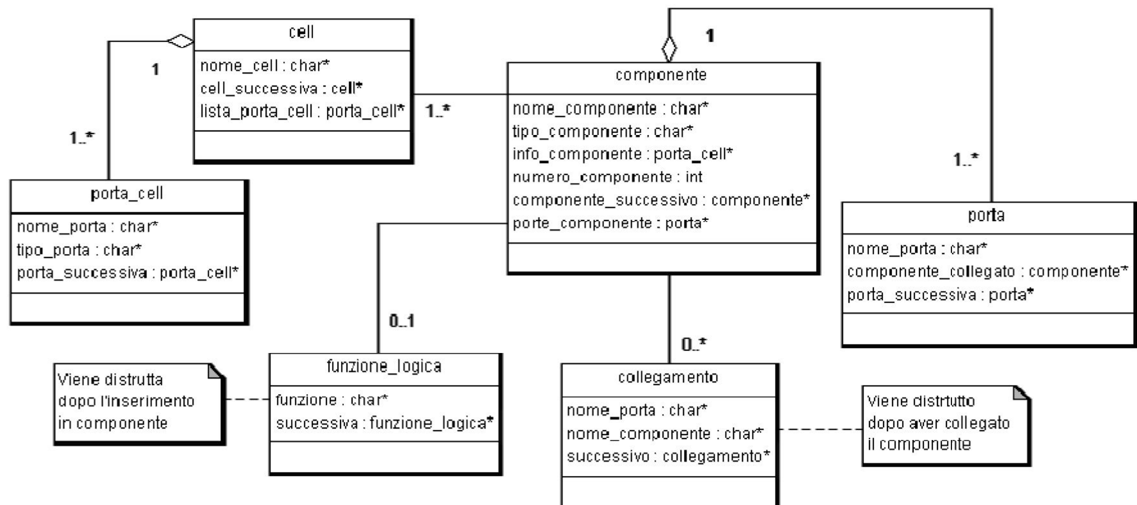

Figura 5.7: Struttura dati completa a tempo di esecuzione

La libertà di scegliere il modello di struttura dati è fortemente limitata per il fatto che le informazioni utili vengono catturate dal parser una alla volta e nella sequenza imposta dal linguaggio EDIF. Si è ritenuto quindi opportuno utilizzare questo tipo di struttura perché si presta bene alla memorizzazione dei dati durante l'analisi del testo EDIF riducendo il numero di strutture dati temporanee.

5.2.2 L'analizzatore lessicale

Di seguito si descrivono i passi più importanti effettuati per la programmazione dell'analizzatore lessicale utilizzato per realizzare il traduttore progettato. Per estrarre le informazioni utili dal file EDIF, occorre ben progettare sia l'analizzatore lessicale che sarà generato da Flex, sia la grammatica del parser per Bison. All'interno di un file EDIF le informazioni da estrarre compaiono in tre differenti formati:

- Identificatori
- Stringhe
- Numeri

Gli *identificatori* sono semplici stringhe separate da spazi, le *stringhe* sono invece delle stringhe delimitate da virgolette come ad esempio "ciao" ed infine i *numeri* sono

semplici numeri interi. L'analizzatore lessicale deve essere programmato per riconoscere questi tre tipi di dato al fine di prelevarli e memorizzarli. Questi tre tipi diventano per Flex tre differenti token chiamati rispettivamente:

IDENTIFICATORE

STRINGA

NUM

Il token identificatore è dichiarato in Flex mediante l'espressione regolare seguente

```
ID [_A-Za-z][_A-Za-z0-9]*
```

ma oltre che riconoscere la presenza di un identificatore nel file è necessario anche associare ad esso un valore semantico che in C corrisponde ad un vettore di caratteri puntato da un char*. Come meglio descritto nel capitolo dedicato a Flex, la stringa catturata e memorizzata nella variabile *yytext* deve essere copiata nella variabile membro *yylval.id* della classe *yylval* per essere utilizzata come valore semantico. L'analizzatore sintattico deve inoltre ritornare il valore numerico associato al token IDENTIFICATORE per far si che il parser riconosca quale token è stato incontrato durante la scansione del file.

```
{ID} {yylval.id=(char*) strdup(yytext); return(IDENTIFICATORE);}
```

Il tipo *stringa* può essere riconosciuto se viene dichiarato come

```
STRING ["][^"]*["]
```

e allo stesso modo di quanto fatto per gli identificatori si ricopia il valore di *yytext* in yylval.st membro di yylval.

```
{STRING} {yylval.st = (char *) strdup(yytext); return(STRINGA);}
```

Si noti che le *stringhe* non possono essere dichiarate come *identificatori* contenuti tra virgolette perché quest'ultime non sono separate dal testo al loro interno tramite spazi, e dunque formano assieme al testo un unico token.

Infine i *numeri* sono dichiarati semplicemente come sequenze di cifre, una cifra è:

`DIGIT [0-9]` mentre una sequenza di cifre è `{DIGIT}+`

Questa volta bisognerà convertire il testo contenuto in *yytext* in un valore intero utilizzando la funzione atoi() e poi copiarlo nella variabile *yylval.intval* di tipo int.

```
{DIGIT}+ { yylval.intval = atoi( yytext ); return(NUM); }
```

Questi tre tipi di dato sono necessari e sufficienti per ottenere tutte le informazioni relative al circuito ma, oltre ad essi, dovranno essere definiti i token necessari per riconoscere le espressioni regolari del linguaggio EDIF. In particolare devono essere associati a token tutti i simboli che permettono di riconoscere le espressioni regolari in cui sono contenute le informazioni da prelevare. Tali simboli sono le keyword del linguaggio e alcuni caratteri come ad esempio le parentesi tonde. Ovviamente a tali simboli non deve corrispondere alcun valore semantico poiché essi sono utilizzati esclusivamente per individuare l'espressioni regolari. La lista dei token da riconoscere è dunque costituita da tutte le keyword usate da EDIF utilizzato dal tool ISE e da alcuni caratteri significativi.

5.2.3 L'analizzatore sintattico

Ultimata la programmazione dell'analizzatore lessicale con Flex, si passa alla definizione della grammatica del linguaggio EDIF utilizzando Bison. Naturalmente la grammatica definita rappresenta solo un sottoinsieme delle regole lessicali e sintattiche che costituiscono interamente la grammatica di EDIF 2.0 ma, tale sottoinsieme è sufficiente per effettuare le traduzioni di file EDIF generati tramite il tool Xilinx ISE.

Per prima cosa si deve dichiarare, nel primo blocco di codice che inizia con il simbolo %%, l'unione destinata a contenere le variabili che contengono i valori semantici dei tipi di dato sopra descritti

```
%union semrec /* Semantica degli Identificatori */
{
int intval; /* Valori Interi */
char *id; /* Identificatori */
char *st; /* Stringhe */
}
```

In questo modo i valori delle informazioni catturate dall'analizzatore lessicale sono assegnate dal parser alle variabili dell'unione *semrec* e possono essere utilizzate come parametri delle funzioni chiamate mediante le azioni associate alle espressioni regolari.

I token che sono definiti nell'analizzatore lessicale sono stati specificati anche nel primo blocco di Bison, ciò è necessario per permettere all'analizzatore sintattico (il parser) di dialogare con l'analizzatore lessicale e riconoscere correttamente le espressioni regolari.

In Bison sono state dichiarate tutte le regole di grammatica che comprendono tutte le espressioni regolari che è possibile trovare nei testi EDIF prodotti da ISE. Per molte regole sono state implementate alcune azioni costituite dalle chiamate di funzioni che realizzano il traduttore.

Il main è stato inserito nella sezione epilogo di Bison (vedi 4.2.4).

Il linguaggio EDIF riferisce i componenti del circuito mediante i loro nomi ovvero con strighe di testo. Spesso però tali nomi possono essere modificati come si vede dall'esempio seguente:

```
(instance (rename a_IBUF_renamed_1 "a_IBUF")
     (viewRef view_1 (cellRef IBUF (libraryRef UNISIMS)))
     (property XSTLIB (boolean (true)) (owner "Xilinx"))
 )
```

In questo esempio una istanza di un buffer d'ingresso viene chiamata *a_IBUF* ma rinominata come *a_IBUF_renamed_1* per riferire lo stesso buffer. Quando EDIF descrive un collegamento, *a_IBUF* è chiamato utilizzando entrambi i nomi come segue.

```
(net a_IBUF
     (joined
     (portRef I0 (instanceRef Mmux_a_a_MUX_3_o11))
     (portRef O (instanceRef a_IBUF_renamed_1))
     (portRef I2 (instanceRef
     faulty_output_rstpot_renamed_10))
     )
 )
```

Questo esempio mostra uno dei vari casi di rinominazione possibile (cfr. 2.1) che rappresenta una difficoltà aggiuntiva al problema della generazione dei collegamenti tra i componenti nella struttura dati a runtime. Infatti, nel momento in cui vengono generati

i collegamenti per un componente, quest'ultimo viene ricercato scorrendo la lista dei componenti effettuando una ricerca per nome. Se il componente a_IBUF è memorizzato nella struttura dati con il nome di `a_IBUF`, la sua ricerca con il nome di `a_IBUF_renamed_1` o `&a_IBUF`, non può avere alcun risultato.

La possibilità di riferire il medesimo componente utilizzando più identificatori differenti è inevitabilmente causa di confusione. La soluzione adottata per risolvere il problema consiste nel ricercare il nome del componente nei vari modi in cui esso può essere rinominato mediante funzioni di ricerca apposite. Nel caso in esame, quando il parser incontra l'identificatore *a_IBUF_renamed_1* il componente è ricercato usando una funzione che restituisce la stringa a sinistra della sequenza di carratteri "*_renamed_*" cioè *a_IBUF*.

5.3 La funzione logica

La funzione logica per una LUT esprime sinteticamente che tipo di corrispondenza esiste tra gli ingressi e l'uscita. In EDIF la funzione logica è implementata mediante la tabella di verità associata al componente, ovvero l'elenco dei valori dell'uscita in funzione degli ingressi.

Per esprimere una tabella di verità in modo compatto viene utilizzato in EDIF l'attributo INIT. Una notazione esadecimale alla quale corrisponde una serie di bit rappresentanti la mappa di Karnaugh della LUT.

Si considera ad esempio una LUT4 avente la seguente tabella di verità:

I3	I2	I1	I0	O	INIT
0	0	0	0	0	8
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	1	B
0	1	0	1	1	
0	1	1	0	0	
0	1	1	1	1	
1	0	0	0	0	E
1	0	0	1	1	
1	0	1	0	1	
1	0	1	1	1	
1	1	0	0	1	B
1	1	0	1	1	
1	1	1	0	0	
1	1	1	1	1	

Figura 5.8: Tabella di verità e corrispondente attributo INIT

l’attributo INIT si ottiene raggruppando gli stati d’uscita in gruppi di quattro, dal basso verso l’alto, e convertendoli nella corrispondente cifra esadecimale. I primi quattro bit sono, in questo caso, 1011 per cui il primo carattere esadecimale dell’attributo INIT è “B”, la seconda quartina è 1110 che corrisponde al carattere “E”, e così via. Infine si ottiene l’attributo INIT completo “BEB8”.

La lunghezza dell’attributo varia ovviamente in base al numero di ingressi, infatti il numero di combinazioni degli ingressi è in ogni caso 2^numero_ingressi: per le LUT2 l’INIT sarà composto da una sola cifra esadecimale, per le LUT3 da due cifre esadecimali, quattro cifre esadecimali per le LUT4, otto per le LUT5 e sedici per le LUT6.

Per ottenere la funzione logica a partire dall’attributo INIT è necessario capire come gli stati di uscita siano rappresentati in una mappa di Karnaugh. Considerando la precedente tabella di verità, la mappa di Karnaugh ad essa corrispondente risulta:

I0 I1 \ I2 I3	00	01	11	10
00	0	0	1	1
01	0	1	0	0
11	1	1	1	1
10	0	1	1	1

Figura 5.9: Mappa di Karnaugh relativa a INIT=BEB8

Gli stati di uscita sono riportati nella mappa leggendo la tabella di verità riga per riga da destra a sinistra. Volendo rappresentare la mappa tramite una matrice 4X4 si deve stabilire una corrispondenza tra le coordinate binarie della mappa di Karnaugh e le coordinate della matrice tenendo conto che le righe e le colonne della mappa di Karnaugh non sono in ordine crescente ma sono posizionate in modo da garantire l'adiacenza degli stati d'ingresso. Nel caso di LUT4 la riga 00 della mappa di Karnaugh corrisponde all'indice di riga 0 della matrice, la riga 01 della mappa corrisponde all'indice di riga 1 della matrice, la riga 10 della mappa corrisponde all'indice 3 della matrice e la riga 11 della mappa corrisponde all'indice 2 della matrice. Questa corrispondenza è identica a quella che sussiste tra le colonne. Gli stati di uscita vengono inseriti nella matrice in base alla convenzione fatta. Scorrendo l'attributo INIT da destra a sinistra, si inseriscono i primi quattro bit indicati dal carattere 8 nella colonna 0 della matrice, rispettivamente alla riga 0, alla riga 3, alla riga 1 e alla riga 2. Si passa poi alla colonna 3 e si inseriscono i successivi quattro bit di B nelle righe secondo lo stesso ordine, poi alla colonna 1 si inseriscono i bit di E ed infine alla colonna 2 i bit di B. Si ottiene quindi una semplice regola per riempire la matrice a partire dalla tabella di verità: si inserisce sia nelle colonne che nelle righe secondo l'ordine 0-3-1-2.

Si può ripetere lo stesso ragionamento con un numero di righe, o di colonne, pari a otto. In questo caso le coordinate della mappa di Karnaugh sono espresse da tre cifre binarie che per rispettare il principio di adiacenza sono sistemate nell'ordine: 000, 001, 011, 010, 110, 111, 101, 100 e sono quindi associate rispettivamente agli indici 0, 1, 2, 3, 4, 5, 6, 7. Ripetendo il ragionamento precedente si trova che l'ordine d'inserimento nel

caso di otto righe, o otto colonne, risulta 0-7-3-4-1-6-2-5. Nel caso più semplice, quando le righe sono due, si trova invece banalmente la sequenza 0-1.

Le mappe di Karnaugh che devono essere prese in esame sono cinque, esattamente quelle che appartengono alle LUT2, LUT3, LUT4, LUT5 e LUT6. Non si prende in considerazione la LUT1 poiché grazie alla sua semplicità è possibile calcolarne la funzione logica senza ricorrere ad algoritmi complessi.

Ad una LUT2 corrisponde una matrice 2x2, ad una LUT3 una matrice 2x4, una LUT4 ha una matrice 4x4, una LUT5 ha una matrice 4x8 e una LUT6 una matrice 8x8. Per gestire tutti i casi bastano dunque le tre sequenze trovate. L'algoritmo per il riempimento della matrice è dunque brevemente il seguente:

1) Conversione dell'attributo INIT in cifre binarie
2) Generazione di una matrice dimensionata in base al tipo di LUT
3) Inserimento nella matrice dei valori binari di INIT a partire dalla cifra meno significativa in poi, rispettando l'ordine delle sequenze di indici:
    - 0-7-3-4-1-6-2-5 quando le righe o le colonne sono otto
    - 0-3-1-2 quando le righe o le colonne sono quattro
    - 1-0 quando ci sono due righe o due colonne

5.3.2 Sintesi della mappa di Karnaugh

L'algoritmo adottato per la sintesi delle mappe di Karnaugh esegue una sintesi, seppure non a costo minino, che ottiene un'espressione della funzione logica di ridotte dimensioni grazie ad un basso numero di implicanti coinvolti. L'algoritmo è il seguente:

1) Scansione riga per riga della mappa alla ricerca di un sottocubo di ordine 1 non contenuto in un sottocubo già esaminato.
2) Ricerca di un sottocubo dello stesso ordine adiacente a quello trovato, se si trova si estende quello di partenza con quello adiacente ottenendo un sottocubo più grande. Si ripete il punto 2 finché non è più possibile trovare un sottocubo adiacente.
3) Memorizzazione del sottocubo trovato

4) Se tutte le caselle della mappa sono state esaminate termina, altrimenti torna al punto 1

La struttura dati utilizzata per memorizzare i sottocubi nasce dall'osservazione che tutti i sottocubi di qualunque ordine sono dei rettangoli, per cui essi sono individuati dalla loro diagonale e per la precisione, dalla coppia di coordinate formate dal vertice in alto a sinistra e dal vertice in basso a destra. Se il sottocubo è di ordine uno le due coordinate coincidono.

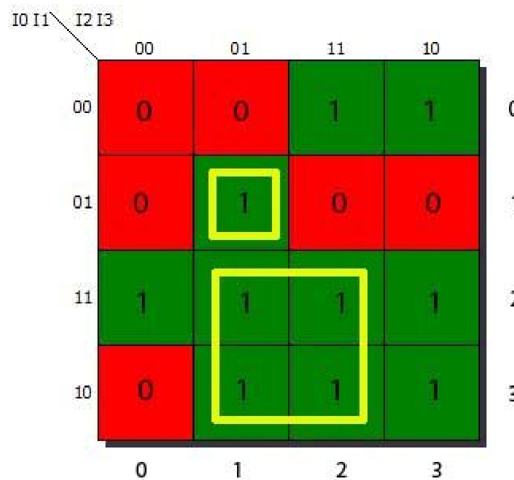

Figura 5.10: Mappa di Karnaugh e la sua corrispondente matrice

Il sottocubo di ordine quattro nella mappa in figura 5.10 si rappresenta con la coppia di coordinate (2,1) (3,2), mentre il sottocubo di ordine uno con le coordinate (1,1) (1,1).

Calcolati i vari sottocubi non resta che trasformarli nella notazione conforme al lessico del linguaggio oggetto e poi collegarli tutti assieme separati dal carattere '+'. Per il sottocubo di ordine quattro in figura si ottiene ad esempio l'implicante 02 mentre per quello di ordine uno !01!23. La sintesi dell'intera mappa mediante l'algoritmo sopra descritto restituisce invece la funzione 02+03+01+1!23+!0!12.

5.3.3 Strutture per il calcolo della funzione logica

Come già anticipato nel paragrafo precedente, le mappe di Karnaugh sono implementate come matrici di numeri interi di dimensioni variabili in base al tipo di LUT.

Tipo di LUT	Dimensioni Matrice	Lunghezza INIT
2	2x2	1
3	2x4	2
4	4x4	3
5	4x8	8
6	8x8	16

Figura 5.11: Dimensioni matrice e INIT in base al tipo di LUT

Per i sottocubi appartenenti alle mappe si usa una semplice struttura contenente quattro numeri interi x1, x2, y1, y2. La prima coppia di interi rappresentano le coordinate della mappa in cui si trova il sottocubo di ordine uno in alto a sinistra, la seconda coppia di interi rappresentano le coordinate del sottocubo di ordine uno in basso a destra. Con questa convenzione quattro numeri interi individuano univocamente un sottocubo in una qualsiasi mappa.

La stringa di caratteri esadecimali INIT è convertita in una sequenza di bit i quali sono inseriti in un vettore di interi per essere facilmente inseriti uno alla volta all'interno della matrice che rappresenta la mappa di Karnaugh secondo l'algoritmo già descritto.

La funzione logica non compare nella struttura dati del circuito ma è calcolata al momento della traduzione a partire dall'attributo INIT che risiede nel campo *info_componente* del *componente*. Questa è una scelta conseguente al fatto che la struttura dati è stata pensata in modo che possa facilmente prestarsi a qualunque altro tipo di traduzione che potremmo implementare in futuro, ovvero essa deve essere completamente indipendente dal tipo di linguaggio con il quale vogliamo descrivere il circuito. In virtù di questo principio tutti i dati che concorrono alla traduzione della stringa INIT in funzione logica, e con essi molti altri dati specifici finalizzati alla produzione della traduzione, sono dichiarati come semplici variabili locali proprie delle loro rispettive funzioni. In questo modo essi sono automaticamente eliminati alla fine del loro utilizzo.

5.4 Fase di traduzione

La fase di traduzione nel linguaggio oggetto ha inizio subito dopo che la struttura dati è stata completata. In questa fase per ogni componente presente nel circuito vengono eseguite le seguenti operazioni:

- Rilevamento del tipo del componente
- Ricerca degli identificatori dei componenti collegati
- Calcolo della funzione logica (per i componenti che la prevedono)
- Scrittura su file della entry relativa al componente

L'assegnamento degli identificatori ai componenti avviene nella fase di costruzione della struttura dati. In fase di traduzione quindi, ogni componente possiede già il suo identificatore che corrisponde ad un numero naturale.

E' necessario rilevare il tipo dei componenti perché in base ad esso si stabilisce il tipo di entry che deve essere scritta e le funzioni che devono essere utilizzate.

La ricerca degli identificatori dei componenti collegati viene effettuata mediante la visita delle porte. Ogni porta di un componente contiene il puntatore al componente che scrive in essa, tramite questo puntatore si accede al componente collegato e quindi da quest'ultimo al suo identificatore.

La funzione logica è ottenuta semplicemente come stringa di testo restituita da una apposita funzione, *converti(),* e scritta di seguito allo statement *lut_fctn* della entry.

Il processo relativo alla traduzione costituisce la parte di *back end* del programma, esso è realizzato dai metodi che traducono ogni componente in una entry del linguaggio oggetto. E' possibile aggiungere nuove entry per nuovi componenti aggiungendo semplicemente i nuovi metodi in questa sezione senza modificare la struttura dati esistente poiché essa è in grado di memorizzare qualsiasi componente del circuito in modo generico. Per ottenere la nuova entry basta implementare il metodo di traduzione specifico per quel componente. Modificando i metodi del processo di *back end* si può anche cambiare completamente il linguaggio oggetto.

Capitolo 6

Casi di studio

In questo capitolo si riportano vari casi d'uso elaborati utilizzando alcune netlist di ITC'99 benchmarks del Politecnico di Torino. ITC'99 benchmarks è una raccolta di netlist scritte nel linguaggio VHDL sviluppate da un gruppo di ricercatori del Politecnico di Torino al fine di testare il software di sintesi dei circuiti elettronici. Le netlist di ITC'99 benchmarks rappresentano circuiti le cui caratteristiche si ritrovano spesso in circuiti elettronici di reale utilizzo.

I file VHDL di ITC'99 benchmarks sono stati tradotti in EDIF utilizzando il tool ngd2edif di ISE Xilinx. In questo modo è stato possibile verificare il corretto funzionamento del traduttore utilizzando le netlist di ITC'99 benchmarks nel linguaggio EDIF.

Per motivi di spazio, per ogni caso d'uso, è riportato solo lo schematico della netlist seguita dal file tradotto da EDIF nel linguaggio oggetto. Sono omessi i testi dei file EDIF poiché risultano lunghi e complessi.

6.1 Circuito b01

La funzione di questo circuito elettronico è quella di una macchina a stati finiti che elabora un segnale seriale in ingresso. b01 è il nome del file che il team di ITC'99 ha attribuito a questa netlist.

Segue lo schematico di b01 ricavato dalla sintesi del file VHDL con ISE.

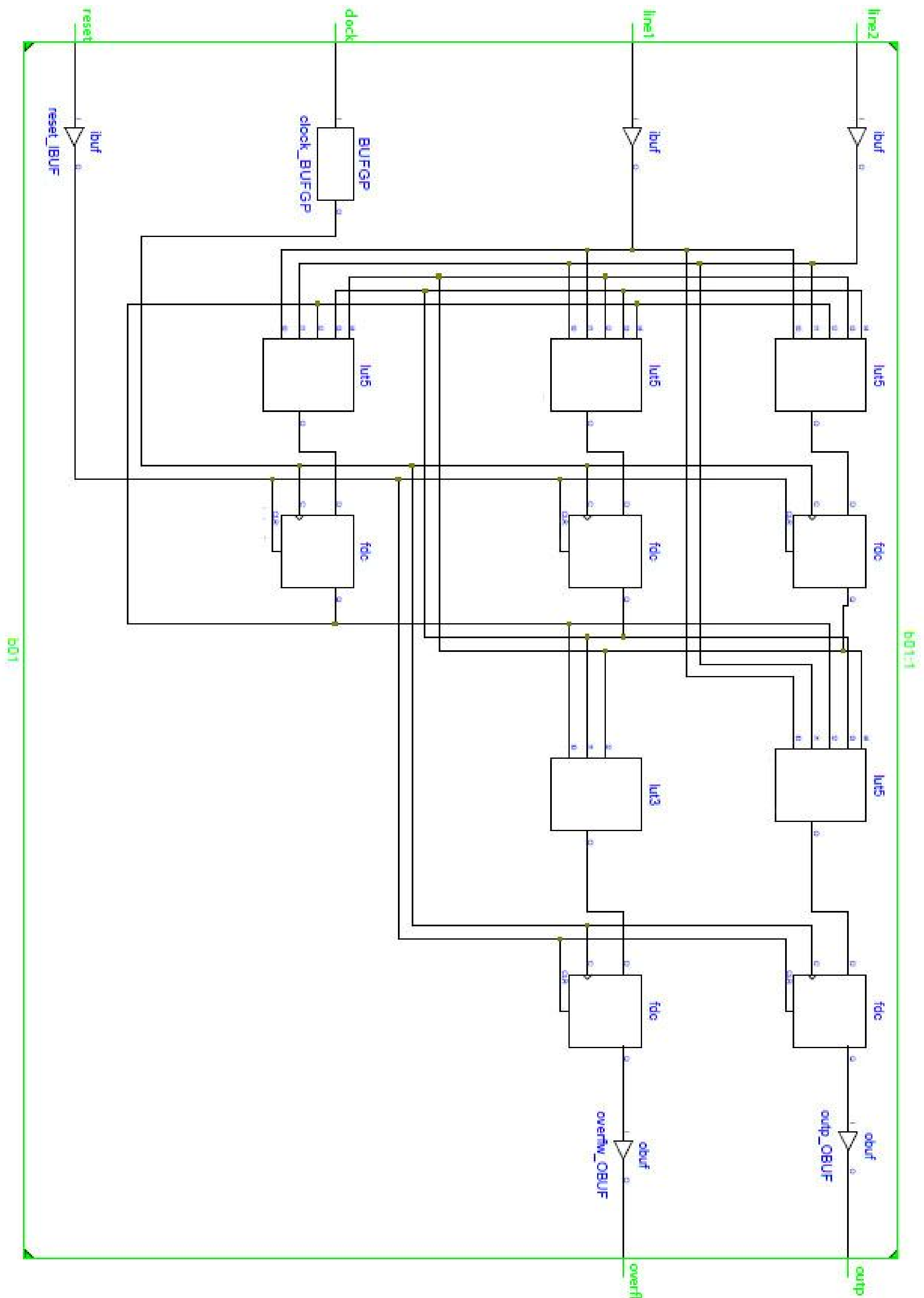

Figura 6.1: netlist b01

Di seguito si riporta la traduzione ottenuta.

```
0 const_inp_prob 0.500000;
1 const_inp_prob 0.500000;
2 const_inp_prob 0.500000;

0 ibuf 0;
1 ibuf 1;
2 ibuf 2;

3 lut_fctn
0!12!3!4+0!13!4+0!1!23+0!1!2!3!4+0124+01!2!34+!012!3!4+!013!4+!0
1!23+!01!2!3!4+!0!124+!0!1!2!34;
3 lut 5 2 1 10 9 8;

4 lut_fctn
024+0!123+0!1!2!3+012!3+01!23+124+1!2!34+!023!4+!0!2!3!4;
4 lut 5 2 1 10 9 8;

5 lut_fctn 03!4+012!3+01!2!3+13!4+23!4+!2!34;
5 lut 5 2 1 10 8 9;

6 lut_fctn 024+0!1!2!34+0123+124+!0!12!3+!23!4+!0!2!34;
6 lut 5 1 2 8 9 10;

7 lut_fctn 01!2;
7 lut 3 10 9 8;

8 fdc 5 0;

9 fdc 6 0;

10 fdc 4 0;

11 fdc 7 0;

12 fdc 3 0;

13 obuf 11 0;
14 obuf 12 1;
```

Oltre alla traduzione, il traduttore produce un file di resoconto del parsing effettuato per controllare il corretto esito della traduzione. In questo file sono mostrati i valori a traduzione ultimata di alcune variabili di controllo. Inizialmente, viene mostrato il numero complessivo di componenti per ogni tipo. `N_LUTS` è ad esempio il numero di LUT contenute nel circuito, `N_FLIP_FLOPS` è il numero complessivo di flip flop e così via. `N_COMPONENTS` indica invece il numero totale di componenti presenti nel circuito indipendentemente dal tipo a cui appartengono. La voce `Devices`, che in questo caso è `Virtex 6`, indica la tecnologia utilizzata.

Infine viene mostrata la lista di tutti i componenti. I numeri naturali che contraddistinguono ogni componente corrispondono agli identificatori dei componenti nel linguaggio oggetto. A destra di tali identificatori viene mostrato invece il corrispondente identificatore utilizzato in EDIF.

```
/*----------------- Parameters of the circuit ------------------

N_LUTS = 5
N_FLIP_FLOPS = 5
N_LATCHS = 0
N_BUFFERS = 5
N_MULTIPLEXERS = 0
N_INPUT_PINS = 3
N_OUTPUT_PINS = 2
N_COMPONENTS = 15

Devices Virtex 6

List of componets:

0 reset_IBUF
1 line2_IBUF
2 line1_IBUF
3 Mmux_stato_2__line1_Mux_9_o11
4 stato_FSM_FFd3_In1
5 stato_FSM_FFd1_In1
6 stato_FSM_FFd2_In1
7 stato_stato_2__GND_2_o_Mux_10_o1
8 stato_FSM_FFd1
9 stato_FSM_FFd2
10 stato_FSM_FFd3
11 overflw
12 outp
13 overflw_OBUF
14 outp_OBUF

---------------------------------------------------------------*/
```

6.2 Circuito b02

Il seguente circuito rappresenta una macchina a stati finiti che riconosce numeri decimali codificati in binary-coded decimal (BCD).

Come per il precedente caso si riporta lo schematico ottenuto con ISE e di seguito la corrispondente traduzione da EDIF nel linguaggio oggetto.

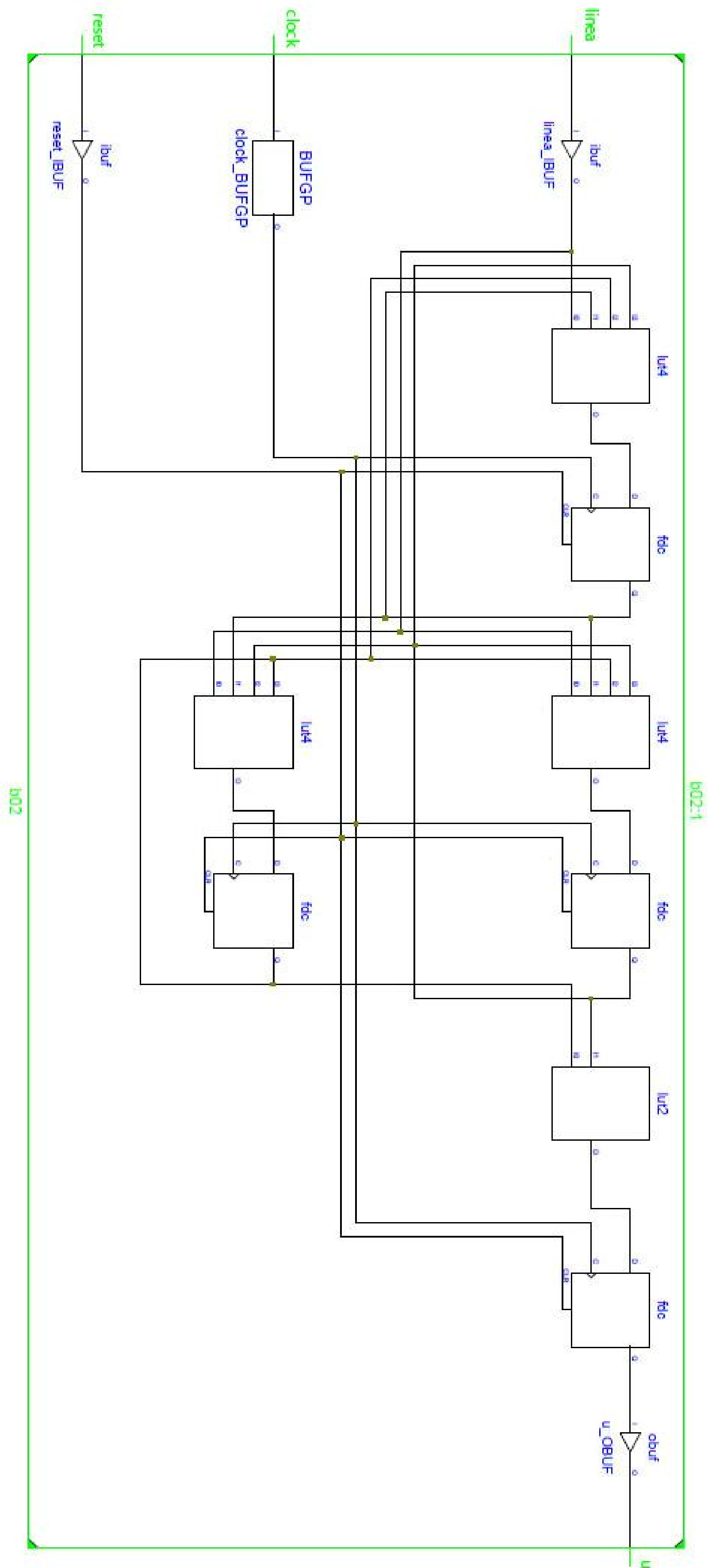

Figura 6.2: netlist b02

```
0 const_inp_prob 0.500000;
1 const_inp_prob 0.500000;

0 ibuf 0;
1 ibuf 1;

2 lut_fctn 0!1!2!3+2!3+1!23+!0!12+!0!1!2!3;
2 lut 4 0 7 8 6;

3 lut_fctn 123+!012+!023+!01!2!3+!0!13;
3 lut 4 0 7 6 8;

4 lut_fctn 0!1+3+!0!2+!0!1;
4 lut 4 0 7 6 8;

5 lut_fctn !01;
5 lut 2 6 8;

6 fdc 2 1;

7 fdc 4 1;

8 fdc 3 1;

9 fdc 5 1;

10 obuf 9 0;


/*---------------- Parameters of the circuit -------------------

N_LUTS = 4
N_FLIP_FLOPS = 4
N_LATCHS = 0
N_BUFFERS = 3
N_MULTIPLEXERS = 0
N_INPUT_PINS = 2
N_OUTPUT_PINS = 1
N_COMPONENTS = 11

Devices Virtex 6

List of componets:

0 linea_IBUF
1 reset_IBUF
2 stato_FSM_FFd2_In1
3 stato_FSM_FFd1_In1
4 stato_FSM_FFd3_In1
5 stato_stato_2__X_2_o_Mux_4_o1
6 stato_FSM_FFd2
7 stato_FSM_FFd3
8 stato_FSM_FFd1
9 u
10 u_OBUF
---------------------------------------------------------------*/
```

6.3 Circuito b06

Il circuito b06 rappresenta un gestore di interrupt (interrupt handler).

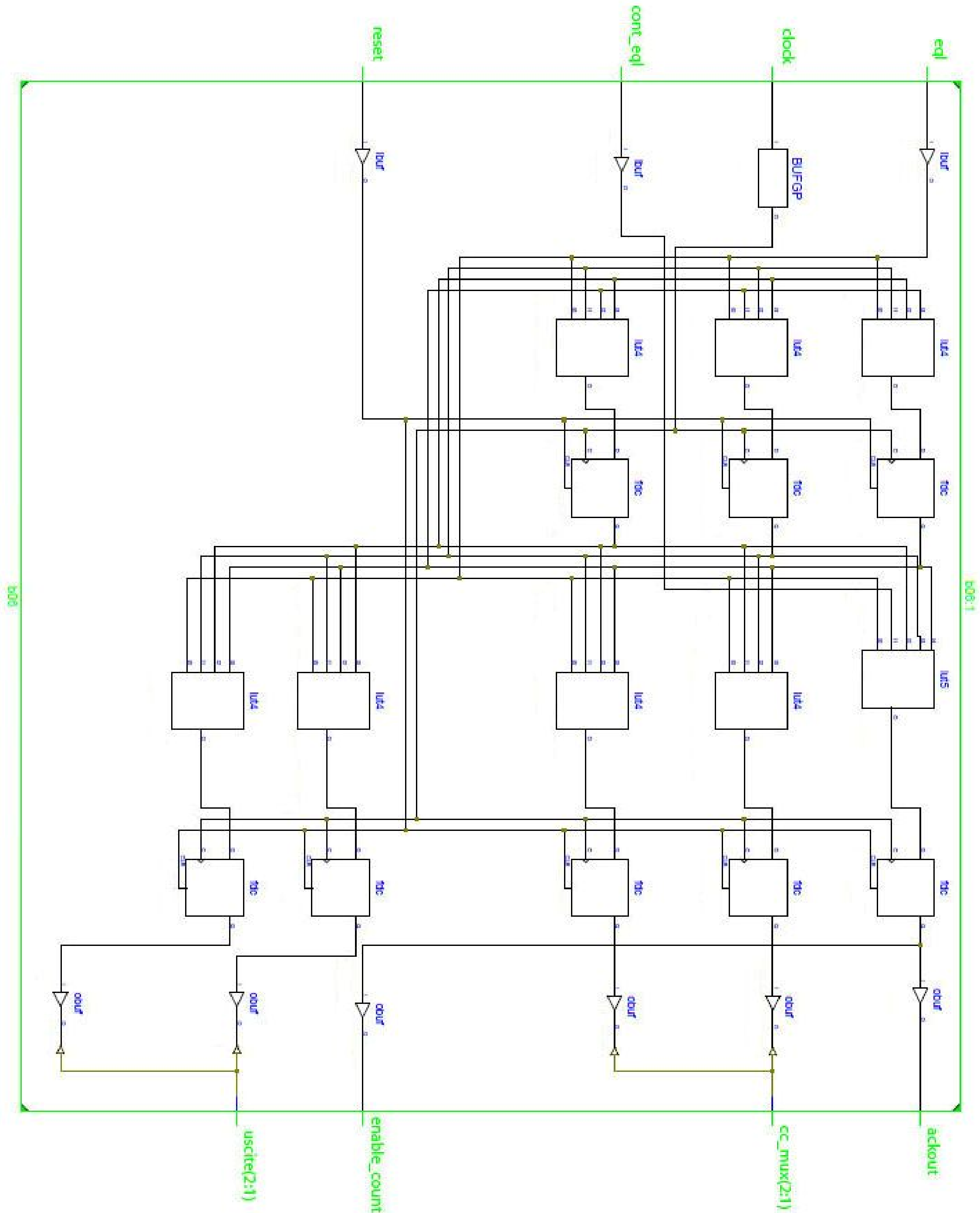

Figura 6.3: netlist b06

```
0 const_inp_prob 0.500000;
1 const_inp_prob 0.500000;
2 const_inp_prob 0.500000;

0 ibuf 0;
1 ibuf 1;
2 ibuf 2;

3 lut_fctn 23+0!1!2!3+123+!02+!03+!0!1;
3 lut 4 2 11 13 12;

4 lut_fctn 03+!01!2!3+!0!1!23;
4 lut 4 2 13 12 11;

5 lut_fctn 0!12+03+1!2!3+!0!1!23;
5 lut 4 2 13 12 11;

6 lut_fctn 23+0!1!2!3+13+!0;
6 lut 4 2 13 11 12;

7 lut_fctn 02!3+0!13+12!3+!013+!023+!2!3;
7 lut 4 2 12 13 11;

8 lut_fctn 03+01!2+!0!12!3+!0!1!23;
8 lut 4 2 13 11 12;

9 lut_fctn 023+!0!1!23;
9 lut 4 2 13 12 11;

10 lut_fctn 0!1+!02!3!4+!0!1;
10 lut 5 2 0 12 13 11;

11 fdc 4 1;

12 fdc 8 1;

13 fdc 3 1;

14 fdc 9 1;

15 fdc 6 1;

16 fdc 5 1;

17 fdc 7 1;

18 fdc 10 1;

19 obuf 18 0;
20 obuf 18 1;
21 obuf 15 2;
22 obuf 14 3;
23 obuf 17 4;
24 obuf 16 5;
```

```
/*---------------- Parameters of the circuit -------------------

N_LUTS = 8
N_FLIP_FLOPS = 8
N_LATCHS = 0
N_BUFFERS = 9
N_MULTIPLEXERS = 0
N_INPUT_PINS = 3
N_OUTPUT_PINS = 6
N_COMPONENTS = 25

Devices Virtex 6

List of componets:

0 cont_eql_IBUF
1 reset_IBUF
2 eql_IBUF
3 state_FSM_FFd3_In1
4 state_FSM_FFd1_In1
5 Mmux_state_2__X_2_o_wide_mux_16_OUT21
6 Mmux_state_2__X_2_o_wide_mux_17_OUT11
7 Mmux_state_2__X_2_o_wide_mux_16_OUT11
8 state_FSM_FFd2_In1
9 Mmux_state_2__X_2_o_wide_mux_17_OUT21
10 Mmux_state_2__X_2_o_Mux_20_o11
11 state_FSM_FFd1
12 state_FSM_FFd2
13 state_FSM_FFd3
14 uscite_2
15 uscite_1
16 cc_mux_2
17 cc_mux_1
18 enable_count
19 ackout_OBUF
20 enable_count_OBUF
21 uscite_1_OBUF
22 uscite_2_OBUF
23 cc_mux_1_OBUF
24 cc_mux_2_OBUF

--------------------------------------------------------------*/
```

Capitolo 7
Conclusioni

Gli strumenti Flex e Bison con cui il traduttore è stato realizzato si sono rivelati utilissimi al fine di generare il parser con cui è stato realizzato il traduttore. Il lavoro di sviluppo del parser è stato prevalentemente quello della dichiarazione del lessico e della grammatica di EDIF utilizzato da ISE. Inizialmente è stata utilizzata una grammatica in grado di riconoscere solo i più elementari costrutti del linguaggio EDIF e quindi solo alcuni circuiti elettronici non complessi. In seguito tale grammatica è stata estesa aggiungendo tutte le regole che hanno consentito l'elaborazione di circuiti anche complessi.

Per motivi di spazio non è stato possibile mostrare un caso d'uso particolarmente complesso nel capitolo 6, perciò i casi d'uso presentati sono solo alcuni semplici esempi che non rivelano del tutto le potenzialità del software realizzato. In realtà il traduttore è in grado di tradurre qualunque netlist EDIF prodotta dal tool ISE anche molto complessa e con un gran numero di componenti. Infatti il parser riconosce un numero di componenti sufficienti a descrivere completamente netlist che definiscono dispositivi FPGA.

Il traduttore è stato inoltre progettato in maniera che esso sia estendibile. Se infatti in futuro si presentasse l'esigenza di estendere l'insieme di componenti che esso è in grado di riconoscere, basterà implementare solamente una nuova funzione di scrittura su file delle entry che rappresentano il nuovo componente. Basterà quindi modificare le funzioni di back end senza apportare alcuna modifica al parser che è già in grado di memorizzare nella struttura dati tutti i componenti contemplati da EDIF.

Un importante test è stato svolto con l'elaborazione del file b15 di ITC'99 benchmarks. Esso rappresenta una netlist contenente ben 3613 componenti. La traduzione di questo file è stata eseguita completamente in soli 4.427 secondi su un PC equipaggiato di un modesto processore AMD Turion TL60 a 2000 Mhz. Ciò dimostra una elevata efficienza del software oltre che la sua efficacia.

Gli sviluppi futuri del software possono essere molteplici. Quello più imminente può essere considerato l'estensione del traduttore al riconoscimento dei componenti delle netlist post place&route al fine di ottenere traduzioni a partire da netlist fisiche oltre che da netlist logiche.

Appendice A

Implementazione del Parser

In questa appendice è riportata l'implementazione di Flex e Bison per definire rispettivamente il lessico e la sintassi di EDIF di ISE.

A.1 Implementazione lessico (Flex)

In questa sezione si riporta integralmente l'implementazione dell'analizzatore lessicale utilizzando Flex ovvero il file di input per Flex. Questo file descrive il lessico del linguaggio EDIF usato da ISE di Xilinx specificando i token utilizzati e le corrispondenti azioni.

```
/**************************************************************
                     Scanner per EDIF
**************************************************************/

%{
/*=============================================================
                          Librerie C
=============================================================*/

#include <string.h> /* per strdup */
#include <stdlib.h>  /* per atoi */
#include "eparse.tab.h" /* per le definizioni dei token e yylval
*/
%}
/*=============================================================
                     Espressioni Regolari
=============================================================*/
DIGIT [-0-9]
ID [&_A-Za-z][&_A-Za-z0-9]*
STRING ["][^"]*["]
/*=============================================================
                      Definizioni TOKEN
=============================================================*/
%%
{DIGIT}+ { yylval.intval = atoi( yytext ); return(NUM); }
cell { return(CELL); }
status { return(STATUS); }
```

```
comment { return(COMMENT); }
keywordMap|keywordmap {  return(KEYWORDMAP); }
accounting { ;return(ACCOUNTING); }
version { return(VERSION); }
author {  return(AUTHOR); }
keywordLevel|keywordlevel { return(KEYWORDLEVEL); }
external { return(EXTERNAL); }
edif {  return(EDIF); }
edifVersion|edifversion { return(EDIFVERSION); }
technology { return(TECHNOLOGY); }
view { return(VIEW); }
viewType {  return(VIEWTYPE); }
edifLevel|ediflevel { return(EDIFLEVEL); }
direction {  return(DIRECTION); }
port { return(PORT); }
"(" { return(ATONDA); }
")" { return(CTONDA);}
celltype { return(CELLTYPE); }
cellType {return(CELLTYPE); }
location {return(LOCATION); }
program { return(PROGRAM); }
timestamp|timeStamp {return(TIMESTAMP); }
library { return(LIBRARY); }
written { return(WRITTEN); }
interface { return(INTERFACE); }
technologydefinition { return(TECHNOLOGYDEFINITION); }
viewtype { return(VIEWTYPE); }
design {  return(DESIGN); }
qualify { return(QUALIFY); }
instance { return(INSTANCE); }
contents { return(CONTENTS);}
cellRef {  return(CELLREF);}
libraryRef { return(LIBRARYREF); }
viewRef { return(VIEWREF); }
designator { return(DESIGNATOR);}
property { return(PROPERTY);}
instanceRef { return(INSTANCEREF); }
joined { return(JOINED); }
portRef { return(PORTREF); }
rename { return(RENAME); }
array { return(ARRAY); }
net { return(NET); }
{STRING} { yylval.st = (char *) strdup(yytext); return(STRINGA);
}
{ID} { yylval.id = (char *) strdup(yytext);
return(IDENTIFICATORE); }
[ \t\n]+ /* ignora spazi vuoti e ritorno carrello */
. { return(yytext[0]);}
%%
int yywrap(void)
     {
     // Quì è possibile cambiare il valore della variabile yyin
     // per continuare lo scanning su un altro file input.
     return 1;
     }
/****************** Fine Scanner File ***********************/
```

A.2 Implementazione sintassi (Bison)

In questa sezione si riporta l'intero file input per Bison che definisce tutti i simboli della grammatica utilizzata. Ad alcune regole sono associate le azioni che chiamano le funzioni che realizzano la struttura dati a runtime. Le funzioni chiamate dal main costituiscono invece alcuni metodi che realizzano il processo di traduzione.

```
%{/**********************************************************
     PARSER PER EDIF (Electronic Design Interchange Format)
**********************************************************/

/*==========================================================
              Librerie C incluse e Prototipi delle funzioni
==========================================================*/

struct porta * genera_lista_porte(char *);
struct componente * cerca_componente(char *);
struct componente * cerca2_componente(char *);
struct porta * cerca_porta(struct componente*, char *);

int yylex(void);
int yyerror (char *);
int rinomina(char *, char *);
int hex2bin(char *,int[]);
int coord_s(int, int );
int in_sottocubo(int, int);
int coordinate(int[], int[], int tipo_lut);
int converti(char *, char *);
int converti_num(char *, int);

#include <stdio.h>
#include <stdlib.h> /* Per usare malloc */
#include <string.h> /* Per manipolare le stringhe */
#include "ST.h" /* Contiene funzioni per struttura dati */
#include "outfctn.c" /* Funzioni per produrre l'output della
traduzione */
#include "karnaugh.c" /* Funzioni per la sintesi delle LUT */

int errors; /* Tiene il conteggio degli errori */


/*==========================================================
                         Semantica dei records
==========================================================*/
%}
%expect 7
%union semrec /* La Semantica degli Identificatori */
{
```

```
int intval; /* Valori Interi */
char *id; /* Identificatori */
char *st; /* Stringhe */
}
/*===============================================================
                                TOKENS
===============================================================*/

%start dispositivo
%token <intval> NUM /* Numero intero */
%token <id> IDENTIFICATORE /* Identificatore */
%token <st> STRINGA /* Stringa */
%token CELL STATUS COMMENT KEYWORDMAP VERSION AUTHOR
KEYWORDLEVEL
%token EXTERNAL EDIF EDIFVERSION TECHNOLOGY VIEW EDIFLEVEL
PROGRAM
%token DIRECTION PORT ATONDA CTONDA CELLTYPE TIMESTAMP WRITTEN
LIBRARY
%token INTERFACE TECHNOLOGYDEFINITION VIEWTYPE ACCOUNTING
%token INSTANCE DESIGN LOCATION QUALIFY PROPERTY DESIGNATOR
%token CONTENTS VIEWREF CELLREF LIBRARYREF NET PORTREF JOINED
INSTANCEREF
%token RENAME ARRAY

/*===============================================================
                    DICHIARAZIONE DELLA GRAMMATICA
===============================================================*/
%%
dispositivo :   ATONDA EDIF IDENTIFICATORE infoseq esterna
dichiarazioneseq info CTONDA
;

info :           ATONDA EDIFVERSION NUM NUM NUM CTONDA
                 |
                 ATONDA EDIFLEVEL NUM CTONDA
                 |
                 ATONDA   KEYWORDMAP   ATONDA   KEYWORDLEVEL   NUM
CTONDA CTONDA
                 |
                  ATONDA       TECHNOLOGYDEFINITION       ATONDA
                  IDENTIFICATORE CTONDA CTONDA
                 |
                ATONDA CELLTYPE IDENTIFICATORE CTONDA
                |
                ATONDA WRITTEN infoseq CTONDA
                |
                ATONDA TIMESTAMP NUM NUM NUM NUM NUM NUM CTONDA
                |
                ATONDA ACCOUNTING PROGRAM STRINGA CTONDA
                |
                ATONDA QUALIFY idseq CTONDA
                |
                ATONDA ACCOUNTING LOCATION STRINGA CTONDA
                |
                ATONDA COMMENT STRINGA CTONDA
                |
```

```
                ATONDA VERSION STRINGA CTONDA
                |
                ATONDA ACCOUNTING info CTONDA
                |
                ATONDA AUTHOR STRINGA CTONDA
                |
                ATONDA PROGRAM STRINGA info CTONDA
                |
                ATONDA TECHNOLOGY dichiarazione CTONDA
                |
                ATONDA TECHNOLOGY ATONDA idseq CTONDA CTONDA
                |
                ATONDA STATUS infoseq CTONDA
                |
                ATONDA DESIGN IDENTIFICATORE infoseq propseq
                CTONDA
                |
                ATONDA CELLREF IDENTIFICATORE ATONDA LIBRARYREF
                IDENTIFICATORE CTONDA CTONDA
;

infoseq :    info
            |
            infoseq info
;

esterna:              ATONDA EXTERNAL IDENTIFICATORE CTONDA
                      |
                      ATONDA EXTERNAL IDENTIFICATORE infoseq
                      componenteseq CTONDA
;

dichiarazione :       ATONDA LIBRARY IDENTIFICATORE infoseq
                      componenteseq CTONDA
                      |
                      componente
;
dichiarazioneseq :         dichiarazione
                           |
                           dichiarazioneseq dichiarazione
;

componente :    ATONDA CELL IDENTIFICATORE descrcomponente
                CTONDA  { ins_tipo($3);}
                |
                ATONDA CELL ATONDA RENAME IDENTIFICATORE STRINGA
                CTONDA descrcomponente CTONDA {ins_tipo($5);}
;

componenteseq   :          componente
                           |
                           componenteseq componente
;

descrcomponente :     info vista
```

```
;

vista : ATONDA VIEW nome tipovista interfaccia contenuti CTONDA
;

tipovista  : ATONDA VIEWTYPE IDENTIFICATORE CTONDA
;

interfaccia : ATONDA INTERFACE portaseq propseq CTONDA
;

contenuti  :    /* nessuno */
                |
                ATONDA CONTENTS istanzeseq collegamentiseq
CTONDA
;

istanza :  ATONDA INSTANCE IDENTIFICATORE ATONDA VIEWREF
           IDENTIFICATORE ATONDA CELLREF IDENTIFICATORE ATONDA
           LIBRARYREF IDENTIFICATORE CTONDA CTONDA CTONDA
           istanzainfoseq CTONDA {genera_componente($3,$9,0);}
           |
           ATONDA INSTANCE ATONDA RENAME IDENTIFICATORE STRINGA
           CTONDA ATONDA VIEWREF IDENTIFICATORE ATONDA CELLREF
           IDENTIFICATORE ATONDA LIBRARYREF IDENTIFICATORE
           CTONDA CTONDA CTONDA istanzainfoseq CTONDA
           {genera_componente($5,$13,1);}
;

istanzeseq :         istanza
                     |
                     istanzeseq istanza
;

istanzainfo:    ATONDA PROPERTY IDENTIFICATORE ATONDA
                IDENTIFICATORE ATONDA IDENTIFICATORE CTONDA
                CTONDA ATONDA idseq STRINGA CTONDA CTONDA
                { imposta_info($12);}
                |
                ATONDA PROPERTY IDENTIFICATORE ATONDA
                IDENTIFICATORE NUM CTONDA ATONDA  idseq STRINGA
                CTONDA CTONDA {imposta_info($10);}
                |
                ATONDA PROPERTY IDENTIFICATORE ATONDA
                IDENTIFICATORE STRINGA CTONDA ATONDA
                IDENTIFICATORE STRINGA CTONDA CTONDA
                { imposta_info($6);}
                |
                ATONDA IDENTIFICATORE IDENTIFICATORE STRINGA
                CTONDA {imposta_info($4);}
;

istanzainfoseq  :    /* nessuna */
                     |
                     istanzainfo
                     |
```

```
                      istanzainfoseq istanzainfo
;

collegamento :  ATONDA NET ATONDA IDENTIFICATORE STRINGA CTONDA
                ATONDA JOINED joinseq CTONDA CTONDA
                { collega($4);}
                |
                ATONDA NET ATONDA RENAME IDENTIFICATORE STRINGA
                CTONDA  ATONDA JOINED joinseq CTONDA
                istanzainfoseq CTONDA { collega($5);}
                |
                ATONDA NET IDENTIFICATORE ATONDA JOINED joinseq
                CTONDA istanzainfoseq CTONDA { collega($3);}
;

collegamentiseq :     collegamento
                      |
                      collegamentiseq collegamento
;

join  :     ATONDA PORTREF IDENTIFICATORE CTONDA
            {ins_collegamento($3,"0");}
            |
            ATONDA PORTREF IDENTIFICATORE ATONDA INSTANCEREF
            IDENTIFICATORE CTONDA CTONDA
            { ins_collegamento($3,$6);}
            |
            ATONDA PORTREF ATONDA IDENTIFICATORE IDENTIFICATORE
            NUM CTONDA CTONDA {ins_collegamento($5,"0");}
;

joinseq:            join
                    |
                    joinseq join
;

Propseq :       /* nessuna */
                |
                propriet
                |
                propseq propriet
;

propriet : ATONDA DESIGNATOR STRINGA CTONDA
           {info_tecnologia($3);}
           |
           ATONDA PROPERTY IDENTIFICATORE propinfoseq CTONDA

;

propinfo   :    ATONDA IDENTIFICATORE STRINGA CTONDA
                |
                ATONDA IDENTIFICATORE NUM CTONDA
;

propinfoseq    :         propinfo
```

```
                          |
                          propinfoseq propinfo
;

porta :     ATONDA PORT IDENTIFICATORE ATONDA DIRECTION
            IDENTIFICATORE CTONDA propseq CTONDA {
            insport($3,$6);}
            |
           ATONDA PORT ATONDA ARRAY ATONDA RENAME IDENTIFICATORE
           STRINGA CTONDA NUM CTONDA ATONDA DIRECTION
           IDENTIFICATORE CTONDA CTONDA {insport($7,$14);}
;

portaseq   :         porta
                     |
                     portaseq porta
;

idseq :    IDENTIFICATORE
           |
           idseq IDENTIFICATORE
;

nome :            IDENTIFICATORE
                  |
                  keyword IDENTIFICATORE
;

keyword :  VIEW

%%
/*===============================================================
                                   MAIN
===============================================================*/

int main( int argc, char *argv[] )
    {
    if(argc>4 || argc<2)
      {
      printf("\nSintassi: <sorgente> <destinazione> <parametro(-
      s)> \nSolo il primo parametro e' obligatorio.\n\n");
       exit(-1);
      }
    FILE *outStream;
    FILE *logStream;
    extern FILE *yyin;
    ++argv; --argc;
    yyin = fopen( argv[0], "r" );
    //errors = 0;
    if(yyin!=NULL) yyparse ();
    else
      {
       printf("\nFile sorgente non trovato.\n\n");
       exit(-1);
      }
    collega_obuf();
```

```
    elimina_funzioni();
    ordina();
    int s=0;
    char destinazione[32];
    char resoconto[32];
      if(argc==1) { strcpy(destinazione,"output.net");
      strcpy(resoconto,"output.chr");
      }
      if(argc==2)
            {
            if(strcmp(argv[1],"-s")==0)
                  {strcpy(destinazione,"output.net");
                   strcpy(resoconto,"output.chr");
                   s=1;
                  }
            else
                  {
                  strcpy(destinazione,argv[1]);
                  strcat(destinazione,".net");
                  strcpy(resoconto,argv[1]);
                  strcat(resoconto,".chr");
                  }
            }
      if(argc==3)
            {
            if(strcmp(argv[2],"-s")!=0)
            {
             printf("\nSintassi: <sorgente> <destinazione>
<parametro(-s)> \nSolo il primo parametro e' obligatorio.\n\n");
       exit(-1);
            }
            else
                  {
                  strcpy(destinazione,argv[1]);
                  strcat(destinazione,".net");
                  strcpy(resoconto,argv[1]);
                  strcat(resoconto,".chr");
                  s=1;
                  }
            }
      if(s==1) controlla_componenti();
    if(!(outStream = fopen(destinazione,"w+"))) // Apre il file
destinazione in scrittura.
            {
        fprintf(stderr,"Errore nell'apertura del file di
output.\n");
        exit(-1);
        }
    if(!(logStream = fopen(resoconto,"w+"))) // Apre in
scrittura il file per il resoconto .
            {
        fprintf(stderr,"Errore nell'apertura del file
resoconto.\n");
        }
    produci(outStream, logStream);
    fclose(outStream);
```

```
 fclose(logStream); // chiude il file di resoconto
 printf ( "\nTraduzione terminata.\n");
 return 1;
 }

/********************** Fine Bison File ********************/
```

Appendice B

Implementazione Struttura Dati e Azioni

In questa appendice è riportata l'implementazione della struttura dati e delle azioni chiamate dal parser. Nel file *ST.h* sono contenute tutte le funzioni che generano e gestiscono la struttura dati a runtime. Nel file *Karnaugh.c* sono contenute invece le funzioni che rendono possibile la sintesi delle LUT ed il calcolo delle funzioni logiche. Infine nel file *outfctn*.c risiedono le funzioni che realizzano la traduzione e gestiscono le operazioni di scrittura su file delle entry del linguaggio oggetto.

B.1 file ST.h

```
/*===================================================================
                      Dichiarazioni Strutture Dati
===================================================================*/

struct cell
      {
      char *nome_cell; /* nome della cell */
      struct cell *cell_successiva; /* puntatore alla cell
successiva */
      struct porta_per_cell *lista_porte_cell;/* puntatore alla
prima porta */
      };

struct porta_per_cell
      {
      char *nome_porta;
      char *tipo_porta;
      struct porta_per_cell *porta_successiva; /* puntatore alla
porta successiva */
      };

struct componente
      {
      int numero; /* ordina i componenti per facilitare la
traduzione */
      char *nome_componente; /* nome del componente */
      char *tipo_componente; /* tipo dell'elemento: ibuf obuf lut
etc... */
      char *info_componente; /* informazioni sul componente,
fuzione logica per le lut */
      struct componente *componente_successivo; /* puntatore al
componente successivo */
```

```
      struct porta *porte_componente;/* puntatore alla prima porta
del componente */
      };

struct porta
      {
      char *nome_porta; /* nome della porta di tipo INPUT */
      struct componente *componente_collegato; /* puntatore al
componente collegato */
      struct porta *porta_successiva;
      };

struct collegamento /* lista collegamenti temporanea, viene
distrutta appena creati i collegamenti dell'elemento */
      {
      char *nome_porta;
      char *nome_componente;
      struct collegamento *successivo;
      };

struct collegamento_obuf /* lista collegamenti OBUF che possibile
inserire solo alla fine. */
      {
      struct componente *obuf;
      struct porta *porta_collegata;
      struct collegamento_obuf *successivo;
      };

struct funzione_logica /* lista funzioni logiche temporanea, viene
distrutta dopo la generazione della struttura del dispositivo */
      {
      char *funzione;
      int usata;// Se è 1 la funzione è stata già inserita nella
struttura del dispositivo
      struct funzione_logica *successiva;
      };

struct symrec // Per gestire gli errori (meccanismo non ancora
attivo)
      {
      char *name; /* nome del componente */
      struct symrec *next;
      };

typedef struct symrec symrec;
typedef struct cell cell;
typedef struct porta_per_cell porta_per_cell;
typedef struct componente componente;
typedef struct funzione_logica funzione_logica;
typedef struct collegamento collegamento;
typedef struct collegamento_obuf collegamento_obuf;
typedef struct porta porta;

symrec *identifier; // Per gestire gli errori (meccanismo non
ancora attivo)

/*-----------------------------------------------------------------
                      PUNTATORI A STRUTTURE
```

```
----------------------------------------------------------------*/
symrec *sym_table = (symrec *)0;
cell *cell_corrente = (cell *)0; /* Puntatore al primo tipo-cell */
porta_per_cell *porta_corrente = (porta_per_cell *)0; /* Puntatore
al primo elemento della lista porte dei tipi*/
componente *componente_corrente = (componente *)0; /* Puntatore al
primo componente */
collegamento *collegamento_corrente = (collegamento *)0; /*
Puntatore al primo dei collegamenti da effettuare */
collegamento_obuf *collegamento_obuf_corrente = (collegamento_obuf
*)0;
funzione_logica *funzione_corrente = (funzione_logica *)0;/* Punta
alla funzione logica da inserire nel componente */
/*=================================================================
                      Funzioni per la Struttura Dati
================================================================*/

// Funzione che stampa a video la struttura dati dell'intero
circuito
void controlla_componenti()
      {
      printf("\n\nLISTA COMPONENTI \n");
      componente *ptr=componente_corrente;
      componente *cmp;
      while(ptr!=0)
            {

      printf("\n++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++");
            printf("\n\nComponente: %s ",ptr->nome_componente);
            printf(" Tipo: %s ",ptr->tipo_componente);
            printf(" Info: %s ",ptr->info_componente);
            printf(" No: %d\n",ptr->numero);
            //printf(" Num: %d \n",ptr->numero);
            porta *porte_ptr=ptr->porte_componente;
                  while(porte_ptr!=0)
                  {
                  printf("\n Porta: %s  ",porte_ptr->nome_porta);
                  if(porte_ptr->componente_collegato!=0)
                        {
                        cmp=porte_ptr->componente_collegato;
                        printf(" Componente collegato: %s  ",cmp-
>nome_componente);
                        }
                  porte_ptr=porte_ptr->porta_successiva;
                  }
            printf("\n");
            ptr=ptr->componente_successivo;
            }
      }

// Aggiunge un tipo-cell usato successivamente per definire i
//componenti
cell * ins_tipo (char *tipo_nome)
      {
      cell *ptr;
      ptr = (cell *) malloc (sizeof(cell));
      ptr->nome_cell = (char *) malloc (strlen(tipo_nome)+1);
```

```
      strcpy (ptr->nome_cell,tipo_nome);
      ptr->cell_successiva = cell_corrente;
      cell_corrente = ptr;
      ptr->lista_porte_cell=porta_corrente; /* Punta al primo
elemento della lista delle porte dell'elemento corrente */
      porta_corrente = (porta_per_cell *)0;
      return ptr;
      }

// Inserisce una porta nella lista porta del tipo-cell corrente
porta_per_cell * insport (char *nome_p, char *tipo)
      {
      porta_per_cell * ptr;
      ptr = (porta_per_cell *) malloc (sizeof(porta_per_cell));
      ptr->nome_porta = (char *) malloc (strlen(nome_p)+1);
      strcpy (ptr->nome_porta, nome_p);
      ptr->tipo_porta = (char *) malloc (strlen(tipo)+1);
      strcpy (ptr->tipo_porta, tipo);
      ptr->porta_successiva = porta_corrente;
      porta_corrente= ptr;
      return ptr;
      }

// Genera un componente e inserisce i dati. Se il parametro rename
// è 1 allora il nome deve essere rinominato
componente * genera_componente (char *nome, char *tipo, int rename
)
      {
      int caratteri=strlen(nome);
      if (rename==1 && caratteri>9) rinomina(nome,"_renamed_");
      componente *ptr;
      ptr = (componente *) malloc (sizeof(componente));
      ptr->nome_componente = (char *) malloc (strlen(nome)+1);
      strcpy (ptr->nome_componente,nome);
      ptr->tipo_componente = (char *) malloc (strlen(tipo)+1);
      strcpy (ptr->tipo_componente,tipo);
      ptr->numero=0;
      if(funzione_corrente!=0 && funzione_corrente->usata==0)
                  {
                  ptr->info_componente = (char *) malloc
(strlen(funzione_corrente->funzione)+1);
                  strcpy (ptr->info_componente,funzione_corrente-
>funzione);
                  funzione_corrente->usata=1;
                  }
      ptr->porte_componente=genera_lista_porte(tipo);
      ptr->componente_successivo = componente_corrente;
      componente_corrente = ptr;
      return ptr;
      }

// Genera la lista porta in base al tipo del componente
porta * genera_lista_porte(char *tipo)
      {
      int porte_trovate=0;
      porta *porta_temp_ptr = (porta *)0;
      cell *cell_ptr=cell_corrente;
      while (cell_ptr!=0)
```

```
		{
		if(strcmp (tipo,cell_ptr->nome_cell)==0) break;
		cell_ptr=cell_ptr->cell_successiva;
		}
	if(cell_ptr==0) return 0;
	// A questo punto cell_ptr contiene il puntatore al tipo da
	// cui devo copiare le porte
	porta_per_cell *ppc_ptr=cell_ptr->lista_porte_cell;
	porta *porta_ptr;
	while(ppc_ptr!=0)
		{
		if(strcmp (ppc_ptr->tipo_porta,"INPUT")==0)
			{
			porta_ptr = (porta *) malloc (sizeof(porta));
			porta_ptr->nome_porta = (char *) malloc
(strlen(ppc_ptr->nome_porta)+1);
			strcpy (porta_ptr->nome_porta, ppc_ptr-
>nome_porta);
			porta_ptr->componente_collegato=0;
			porta_ptr->porta_successiva=porta_temp_ptr;
			porta_temp_ptr=porta_ptr;
			porte_trovate++;
			}
		ppc_ptr=ppc_ptr->porta_successiva;
		}
	if(porte_trovate==0) return 0; else return porta_ptr;
	}


// Aggiunge un collegamento nella lista temporanea dei collegamenti
// del componente. Se nome (del componente) è "0" allora il
// collegamento è con un pin del circuito
collegamento * ins_collegamento (char *porta, char *nome)
	{
	if(strcmp(nome,"0")!=0)
		{
		componente *verifica=cerca_componente(nome);
		if (verifica==0) rinomina(nome,"_renamed_");
		}
	collegamento *ptr;
	ptr = (collegamento *) malloc (sizeof(collegamento));
	ptr->nome_porta = (char *) malloc (strlen(porta)+1);
	strcpy (ptr->nome_porta,porta);
	ptr->nome_componente = (char *) malloc (strlen(nome)+1);
	strcpy (ptr->nome_componente,nome);
	ptr->successivo = collegamento_corrente;
	collegamento_corrente = ptr;
	return ptr;
	}

// considera l'untimo blocco net letto in edif e inserisce le
// informazioni di routing contenute in esso nella struttura dati
int collega (char *nome)
	{
	porta* porta_da_collegare;
	componente *cmp;
	componente *componente_da_collegare; //componente che scrive
	collegamento *ptr=collegamento_corrente;
```

```
      componente_da_collegare=cerca2_componente(nome);
      collegamento_obuf * obuf_ptr;
            if(componente_da_collegare==0)
                  {
                  // se si vuole aggiungere istruzioni per
                  // collegamenti a pin
                  }
                  else
                  {
                  while(ptr!=0)
                        {
                        // il collegamento è interno.Vedo che tipo
                        // di porta è (INPUT o OUTPUT),
                        // si deve considerare solo quelle di input
                        cmp=cerca2_componente(ptr-
>nome_componente);// componente in cui si deve inserire il nome
                   // della porta
                        if (cmp!=0)
                              {
                              porta_da_collegare=cerca_porta(cmp,ptr-
>nome_porta); // porta del componente in cui scrive in componente
              // da collegare
                              if(porta_da_collegare!=0)
                                    {
                                    if(strstr(nome,"_OBUF")!=NULL)
                                          {
                                          /* la porta è di input e il
                                          componente da collegare è un
                                          buffer di uscita memorizza
                                          collegamento OBUF che non è
                                          possibile inserire ora */
obuf_ptr = (collegamento_obuf *) malloc(sizeof(collegamento_obuf));
obuf_ptr->obuf=componente_da_collegare;
obuf_ptr->porta_collegata=porta_da_collegare;
obuf_ptr->successivo = collegamento_obuf_corrente;
collegamento_obuf_corrente = obuf_ptr;
                                          }
                                    else
                                          {
                                    /* la porta e di input e il
                                    componente da collegare non è un
                                    buffer out */
porta_da_collegare->componente_collegato=componente_da_collegare;
                                          }
                                    }
                              else
                                    {
                                    // la porta è di output
            if(porta_da_collegare==0 && strstr(nome,"_OBUF")!=NULL)
                                          {
                        // collegamento porta di un buffer d'uscita

      porta_da_collegare=componente_da_collegare->porte_componente;
      porta_da_collegare->componente_collegato=cmp;
                                          }
                                    }
                              }
      ptr=ptr->successivo;
```

```
                    }
                }
        // Elimina il collegamento temporaneo obsoleto per un
        // corretto uso della memoria
        free(collegamento_corrente);
        collegamento_corrente = (collegamento *)0;
        return 1;
        }

// Cerca un componente e restituisce il suo indirizzo, se il
// componente non viene trovato restituisce 0
componente * cerca_componente(char *nome)
        {
        componente *ptr_componente=componente_corrente;
        while(ptr_componente != 0)
                {
                 if(strcmp (nome,ptr_componente->nome_componente) == 0)
return ptr_componente;
               ptr_componente=ptr_componente->componente_successivo;
                }
        return 0;
        }

// Cerca un componente e restituisce il suo indirizzo. Cerca2
// effettua prima la ricerca standard, se il componente non si
// trova a causa delle anomalie degli identificatori viene
// ricercato usando i collegamenti temporanei
componente * cerca2_componente(char *nome)
{
        componente *nome_standard=cerca_componente(nome);
        if(nome_standard!=0) return nome_standard;
        collegamento *ptr_collegamento=collegamento_corrente;
        componente *cmp;
        porta *porta_ptr;
        while(ptr_collegamento!=0)
        {
        if(strcmp(ptr_collegamento->nome_componente,"0")!=0)
                {
                cmp=cerca_componente(ptr_collegamento->nome_componente);
                porta_ptr=cerca_porta(cmp,ptr_collegamento->nome_porta);
// se tra i collegamenti temporanei la porta non esiste allora il
// componente cercato è quello giusto
                if (porta_ptr==0) return cmp;
                }
        ptr_collegamento=ptr_collegamento->successivo;
        }
//componente non trovato (collegamento a pin)
return 0;
}


// cerca la porta di un componente e restituisce il suo indirizzo
porta * cerca_porta(componente *ptr_componente, char *nome_porta)
        {
        if(ptr_componente->porte_componente==0) return 0;;
        porta* cerca_porta=ptr_componente->porte_componente;
        if(cerca_porta->nome_porta==0) return 0;
        while(cerca_porta!=0)
```

```
            {
            if(strcmp(cerca_porta->nome_porta,nome_porta)==0) break;
            cerca_porta=cerca_porta->porta_successiva;
            }
      if(cerca_porta!=0) return cerca_porta;
      else return 0;// la porta è di output
      }

// Funzione che rinomina i nomi dei componenti per facilitarne la
// ricerca
int rinomina(char *nome, char *taglio)
            {
            char* sottostringa = strstr(nome,taglio);
            if(sottostringa!=NULL)
                  {
                  int caratteri=strlen(nome)-strlen(sottostringa);
                  char copia_nome[caratteri];
                  strncpy(copia_nome, nome, caratteri);
                  copia_nome[caratteri]=nome[strlen(nome)];
                  strcpy (nome, copia_nome);
                  return 0;
                  }
                  return 1;
            }

// memorizza le funzioni logiche dei componenti
void imposta_info(char *stringa)
            {
            int i=0;
            int l=strlen(stringa);
            char s[l-2];
            while(i<l-2)
                  {
                  s[i]=stringa[i+1];
                  i++;
                  }
            s[i]='\0'; /* carattere che termina la stringa */
            funzione_logica *ptr;
            ptr = (funzione_logica *) malloc
(sizeof(funzione_logica));
            ptr->funzione = (char *) malloc (strlen(s)+1);
            strcpy (ptr->funzione,s);
            ptr->usata=0;
            ptr->successiva = funzione_corrente;
            funzione_corrente = ptr;
            }

// elimina la lista delle funzioni logiche temporanea per un
// corretto uso della memoria
void elimina_funzioni()
      {
      funzione_logica *ptr=funzione_corrente;
      funzione_logica *ultimo;
      while(ptr!=0)
            {
            ultimo=ptr->successiva;
            free(ptr);
            ptr=ultimo;
```

```
		}
	}

// Ordina i componenti inserendone il numero d'ordine
void ordina()
	{
	int num=0;
	componente *ptr=componente_corrente;
	// i componenti di tipo clock, massa e alimentazione non
hanno identificatore nella
	// traduzione e vengono numerati con interi negativi
	while(ptr!=0)
	{
	if(strcmp(ptr->tipo_componente,"BUFGP")==0) ptr->numero=-3;
	if(strcmp(ptr->tipo_componente,"VCC")==0) ptr->numero=-2;
	if(strcmp(ptr->tipo_componente,"GND")==0) ptr->numero=-1;
	ptr=ptr->componente_successivo;
	}
	ptr=componente_corrente;
	// comincia ad ordinare i componenti di tipo IBUF
	while(ptr!=0)
		{
		if(strcmp(ptr->tipo_componente,"IBUF")==0)
		   {
		      ptr->numero=num;printf("\n%s - %d - %s",ptr-
>nome_componente,num,ptr->tipo_componente);
		      num++;
		   }
		ptr=ptr->componente_successivo;
		}
	ptr=componente_corrente;
	// ora ordina i componenti che non sono di tipo OBUF nè IBUF
	while(ptr!=0)
		{
		if(strcmp(ptr->tipo_componente,"IBUF")!=0 && strcmp(ptr-
>tipo_componente,"OBUF")!=0 && ptr->numero==0)
		   {
		      ptr->numero=num;printf("\n%s - %d - %s",ptr-
>nome_componente,num,ptr->tipo_componente);
		      num++;
		   }
		ptr=ptr->componente_successivo;
		}
	ptr=componente_corrente;
	// infine numera i componenti di tipo OBUF
	while(ptr!=0)
		{
		if(strcmp(ptr->tipo_componente,"OBUF")==0)
		   {
		      ptr->numero=num;printf("\n%s - %d - %s",ptr-
>nome_componente,num,ptr->tipo_componente);
		      num++;
		   }
		ptr=ptr->componente_successivo;
		}
	}
```

```
// effettua i collegamenti a buffer out che non è stato possibile
// inserire prima per mancanza di dati nella struttura del
// dispositivo
void collega_obuf()
      {
      collegamento_obuf *da_cancellare;
      porta *porta_da_collegare;
      componente *componente_da_collegare;
      collegamento_obuf *ptr=collegamento_obuf_corrente;
      while(ptr!=0)
            {
            porta_da_collegare=ptr->obuf->porte_componente;
            componente_da_collegare=porta_da_collegare-
>componente_collegato;
            ptr->porta_collegata-
>componente_collegato=componente_da_collegare;
            da_cancellare=ptr;
            ptr=ptr->successivo;
            // Elimina il collegamento obsoleto per un corretto uso
della memoria
            free(da_cancellare);
            }
      }

// memorizza il seriale del chip passato come stringa, se la
stringa in ingresso
// è "query" allora non memorizza nulla ma calcola e restituisce
famiglia e modello
// corrispondente al seriale memorizzato in precedenza
char * info_tecnologia(char *info)
      {
      static char *modello_chip;
      if(strcmp(info,"query")!=0)
            {
            modello_chip = (char *) strdup(info);
            return NULL;
            }
      else
      {
      int m=0;
      char modello[12]; strcpy(modello,"");
      char marca[12]; strcpy(marca,"");
      char marca_g[12]; strcpy(marca_g,"");
      char energia[12]; strcpy(energia,"");
      // modello
      if(modello_chip[2]=='q') sprintf(modello,"QPro ");
      if(modello_chip[2]=='a') sprintf(modello,"Automotive ");
      // marca
      if(modello_chip[4]=='v') {sprintf(marca,"Virtex"); m=1;}
      if(modello_chip[4]=='s') {sprintf(marca,"Spartan"); m=1;}
      // energia
      if(modello_chip[8]=='l') sprintf(energia,"Low Power");
      if(m==1)
            {
            sprintf(marca_g,"%s %c ",marca,modello_chip[3]);
            sprintf(info,"%s",modello);
            strcat(info,marca_g);
            strcat(info,energia);
```

```
            }
        else sprintf(info,"%s","unknown");
        return info;
        }
    }
```

B.2 file Karnaugh.c

```
/*-------------------------------------------------------------------
      Funzioni per la sintesi delle LUT tramite mappa di Karnaugh
-------------------------------------------------------------------*/

struct sottocubo /* sottocubi generati dalla ispezione delle mappe
di karnaugh */
      {
      int x1, y1, x2, y2; // coordinate della diagonale del
sottocubo nella mappa
      struct sottocubo *successivo;
      };

struct sottocubo * genera_sottocubo(int, int);
struct sottocubo * cerca_sottocubo(struct sottocubo *, int, int,
int [][*]);

typedef struct sottocubo sottocubo;
sottocubo *sottocubo_corrente = (sottocubo *)0;

int traduci_sottocubo(sottocubo *s, int tipo_lut, char *expr);

// converte INIT di edif in funzione logica intuitiva
int converti(char *funz_edif, char *traduzione)
      {
      sottocubo *scp;
      int i=0;
      int tipo_lut=0;
      int righe, colonne;
      int l=strlen(funz_edif);
      if(l==16) // lut6
            {
            righe=8;
            colonne=8;
            tipo_lut=6;
            }
      if(l==8) // lut5
            {
            righe=4;
            colonne=8;
            tipo_lut=5;
            }
      if(l==4) // lut4
            {
            righe=4;
            colonne=4;
            tipo_lut=4;
```

```
		}
	if(l==2) // lut3
		{
		righe=2;
		colonne=4;
		tipo_lut=3;
		}
	if(l==1) // lut2
		{
		righe=2;
		colonne=2;
		tipo_lut=2;
		}
	int mappa[righe][colonne];
	int lut_out[righe*colonne];
	hex2bin(funz_edif,lut_out);
	int k[2];
	k[0]=0; k[1]=0;
	int k_prec[2];
	for(i=0; i<righe*colonne; i++)
		{
		mappa[k[0]][k[1]]=lut_out[i];
		k_prec[0]=k[0];
		k_prec[1]=k[1];
		coordinate(k_prec,k,tipo_lut);
		}
	// ispezione della mappa
	sottocubo *ptr;
	int x=0; int y=0;
	for(i=0; i<righe*colonne; i++)
		{
		if(mappa[x][y]==1 && in_sottocubo(x,y)==0)
			{
			ptr=genera_sottocubo(x,y);
			while(ptr!=0) ptr=cerca_sottocubo(ptr, righe,
colonne, mappa);
			}
		y++;
		if (y==colonne){x++; y=0;}
		}
	// a questo punto si produce la traduzione della funzione
	char sottocubo_tradotto[20];
	sottocubo *sc=sottocubo_corrente;
	while(sc!=0)
		{
		traduci_sottocubo(sc,tipo_lut,sottocubo_tradotto);
		sc=sc->successivo;
		strcat(traduzione,sottocubo_tradotto);
		if(sc!=0) strcat(traduzione,"+");
		}
	// cancella la lista dei sottocubi che non serve più
	scp=sottocubo_corrente;
	sottocubo *ultimo;
	while(scp!=0)
		{
		ultimo=scp->successivo;
		free(scp);
		scp=ultimo;
```

```
            }
        sottocubo_corrente = 0;
        return 1;
        }

// converte una stringa che esprime un numero esadecimale in un
vettore di bit ordinato
// al contrario rispetto l'ordine delle cifre che rappresenta la
stringa.
int hex2bin(char *funz_edif,int lut_out[])
        {
        int l=strlen(funz_edif);
        int i=(l*4)-1;
        int j=0;
        char c;
        while (i>=0)
                {
                c=funz_edif[j];
                switch (c)
                {
                case '0':
                        lut_out[i]=0; lut_out[i-1]=0; lut_out[i-2]=0;
lut_out[i-3]=0; i=i-4;
                        break;
                case '1':
                        lut_out[i]=0; lut_out[i-1]=0; lut_out[i-2]=0;
lut_out[i-3]=1; i=i-4;
                        break;
                        case '2':
                        lut_out[i]=0; lut_out[i-1]=0; lut_out[i-2]=1;
lut_out[i-3]=0; i=i-4;
                        break;
                case '3':
                        lut_out[i]=0; lut_out[i-1]=0; lut_out[i-2]=1;
lut_out[i-3]=1; i=i-4;
                        break;
                        case '4':
                        lut_out[i]=0; lut_out[i-1]=1; lut_out[i-2]=0;
lut_out[i-3]=0; i=i-4;
                        break;
                case '5':
                        lut_out[i]=0; lut_out[i-1]=1; lut_out[i-2]=0;
lut_out[i-3]=1; i=i-4;
                        break;
                        case '6':
                        lut_out[i]=0; lut_out[i-1]=1; lut_out[i-2]=1;
lut_out[i-3]=0; i=i-4;
                        break;
                        case '7':
                        lut_out[i]=0; lut_out[i-1]=1; lut_out[i-2]=1;
                        lut_out[i-3]=1; i=i-4;
                        break;
                case '8':
                        lut_out[i]=1; lut_out[i-1]=0; lut_out[i-2]=0;
lut_out[i-3]=0; i=i-4;
                        break;
                case '9':
```

```
                lut_out[i]=1; lut_out[i-1]=0; lut_out[i-2]=0;
lut_out[i-3]=1; i=i-4;
                break;
                case 'A':
                lut_out[i]=1; lut_out[i-1]=0; lut_out[i-2]=1;
lut_out[i-3]=0; i=i-4;
                break;
           case 'B':
                lut_out[i]=1; lut_out[i-1]=0; lut_out[i-2]=1;
lut_out[i-3]=1; i=i-4;;
                break;
                case 'C':
                lut_out[i]=1; lut_out[i-1]=1; lut_out[i-2]=0;
lut_out[i-3]=0; i=i-4;
                break;
           case 'D':
                lut_out[i]=1; lut_out[i-1]=1; lut_out[i-2]=0;
lut_out[i-3]=1; i=i-4;
                break;
                case 'E':
                lut_out[i]=1; lut_out[i-1]=1; lut_out[i-2]=1;
lut_out[i-3]=0; i=i-4;
                break;
           case 'F':
                lut_out[i]=1; lut_out[i-1]=1; lut_out[i-2]=1;
lut_out[i-3]=1; i=i-4;
                break;
           }
           j++;
           }
      return 1;
      }


// riceve le ultime coordinate della mappa di karnaugh in cui si è
// inserito un valore e produce le coordinate successive in cui
// inserire
int coordinate(int precedenti[], int successive[], int tipo_lut)
{
if(tipo_lut==6)
      {
      if(precedenti[0]==5) successive[1]=coord_s(precedenti[1],8);
      successive[0]=coord_s(precedenti[0],8);
      }
if(tipo_lut==5)
      {
      if(precedenti[0]==2) successive[1]=coord_s(precedenti[1],8);
      successive[0]=coord_s(precedenti[0],4);
      }
if(tipo_lut==4)
      {
      if(precedenti[0]==2) successive[1]=coord_s(precedenti[1],4);
      successive[0]=coord_s(precedenti[0],4);
      }
if(tipo_lut==3)
      {
      if(precedenti[0]==1) successive[1]=coord_s(precedenti[1],4);
      successive[0]=coord_s(precedenti[0],2);
```

```
      }
if(tipo_lut==2)
      {
      if(precedenti[0]==1) successive[1]=coord_s(precedenti[1],2);
      successive[0]=coord_s(precedenti[0],2);
      }
return 1;
}


// restituisce la coordinata successiva della mappa di karnauh in
// base alla coordinata precedente e al numero di righe o colonne
// (num_rc)
int coord_s(int c, int num_rc)
      {
      if (num_rc==8)
            {
            switch (c)
                  {
            case 0:    return 7;
                  case 7:    return 3;
                  case 3:    return 4;
                  case 4:    return 1;
                  case 1:    return 6;
                  case 6:    return 2;
                  case 2:    return 5;
                  case 5:    return 0;
                  }
            }
      if (num_rc==4)
            {
            switch (c)
                  {
            case 0:    return 3;
                  case 3:    return 1;
                  case 1:    return 2;
                  case 2:    return 0;
                  }
            }
      if (num_rc==2)
            {
            if (c==0) return 1; else return 0;
            }
      return -1; // ritorna -1 se num_rc è un valore non permesso
      }

// genera un sottocubo di ordine 1
sottocubo * genera_sottocubo(x,y)
            {
            sottocubo *ptr;
            ptr = (sottocubo *) malloc (sizeof(sottocubo));
            ptr->x1=x;
            ptr->y1=y;
            ptr->x2=x;
            ptr->y2=y;
            ptr->successivo = sottocubo_corrente;
            sottocubo_corrente = ptr;
            return ptr;
```

```
            }

// se possibile espande il sottocubo con uno adiacente, altrimenti
// restituisce 0
sottocubo * cerca_sottocubo(struct sottocubo *s, int righe, int
colonne, int mappa[righe][colonne])
      {
      int lut;
      char expr[20];
      if(righe==2 && colonne==2) lut=2;
      if(righe==2 && colonne==4) lut=3;
      if(righe==4 && colonne==4) lut=4;
      if(righe==4 && colonne==8) lut=5;
      if(righe==8 && colonne==8) lut=6;
      sottocubo *s_test;
      s_test = (sottocubo *) malloc (sizeof(sottocubo));
      s_test->x1=s->x1; s_test->x2=s->x2; s_test->y1=s->y1; s_test-
>y2=s->y2;
      int x; int y; int xp; int yp; int salta=0;
      // cerca sopra
      int altezza=s->x2-s->x1+1;
      int base=s->y2-s->y1+1;
      x=s->x1-altezza; xp=x;
      y=s->y1; yp=y;
      if(x<0) salta=1;
      while(x<s->x1)
            {
            while(y<=s->y2)
                  {
                  if(mappa[x][y]!=1 || (y<0 || y>=colonne) )
salta=1;
                  if (salta==1) break;
                  y++;
                  }
            x++;
            if (salta==1 || (x<0 || x>=righe)) break;
            y=s->y1;
            }
      if(salta==0) // è stato trovato un sottocubo
            {
            s_test->x1=xp; s_test->y1=yp;
            if(traduci_sottocubo(s_test, lut, expr)==1)
                  {
            s->x1=xp;
            s->y1=yp;
            return s;
                  }
            return 0;
            }
      // cerca a destra
      salta=0;
      altezza=s->x2-s->x1+1;
      base=s->y2-s->y1+1;
      x=s->x1; xp=x;
      y=s->y2+1; yp=y;
      if(y>=colonne) salta=1;
      while(x<=s->x2)
            {
```

```
		while(y<=s->y2+base)
			{
			if(mappa[x][y]!=1 || (y<0 || y>=colonne)) salta=1;
			if (salta==1) break;
			y++;
			}
		if (salta==1 || (x<0 || x>=righe)) break;
		x++;
		y=s->y1;
		}
	if(salta==0) // è stato trovato un sottocubo
		{
		s_test->y2=s->y2+base;
		if(traduci_sottocubo(s_test, lut, expr)==1)
			{
			s->y2=s->y2+base;
			return s;
			}
printf("CERCA_SOTTOCUBO: trovato un sottocubo non compatibile");
		return 0;
		}
	// cerca sotto
	salta=0;
	altezza=s->x2-s->x1+1;
	base=s->y2-s->y1+1;
	x=s->x2+1; xp=x;
	y=s->y1; yp=y;
	if(x>=righe) salta=1;
	while(x<=s->x2+altezza)
		{
		while(y<=s->y2)
			{
			if(mappa[x][y]!=1 || (y<0 || y>=colonne)) salta=1;
			if(salta==1) break;
			y++;
			}
		if (salta==1 || (x<0 || x>=righe)) break;
		x++;
		y=s->y1;
		}
	if(salta==0) // è stato trovato un sottocubo
		{
		s_test->x2=s->x2+altezza;
		if(traduci_sottocubo(s_test, lut, expr)==1)
			{
		s->x2=s->x2+altezza;
		return s;
			}
			printf("CERCA_SOTTOCUBO: trovato un sottocubo non
compatibile");
		return 0;
		}
	// cerca a sinistra
	salta=0;
	altezza=s->x2-s->x1+1;
	base=s->y2-s->y1+1;
	x=s->x1; xp=x;
	y=s->y1-base; yp=y;
```

```
      if(y<0) salta=1;
      while(x<=s->x2)
            {
            while(y<s->y1)
                  {
                  if(mappa[x][y]!=1 || (y<0 || y>=colonne)) salta=1;
                  if (salta==1) break;
                  y++;
                  }
            if (salta==1 || (x<0 || x>=righe)) break;
            x++;
            y=s->y1-base;
            }
      if(salta==0) // è stato trovato un sottocubo
            {
            s_test->y1=yp;
            if(traduci_sottocubo(s_test, lut, expr)==1)
                  {
            s->y1=yp;
            return s;
                  }
                  printf("CERCA_SOTTOCUBO: trovato un sottocubo non
compatibile");
            return 0;
            }
// non esiste alcun sottocubo adiacente ad s del suo stesso ordine
      return 0;
      }

// calcola la traduzione di un sottocubo secondo la convenzione
// adottata
int traduci_sottocubo(sottocubo *s, int tipo_lut, char *expr)
      {
      int i=s->x1;
      int j=s->y2;
      int r=0;
      int righe=(s->x2-s->x1+1)*(s->y2-s->y1+1
      int matrice[righe][tipo_lut];
      if(tipo_lut==6)
            {
            for (i=s->x1; i<=s->x2; i++)
            {
            for(j=s->y1; j<=s->y2; j++)
                  {
      if(j==0) {matrice[r][3]=0; matrice[r][4]=0; matrice[r][5]=0;}
      if(j==1) {matrice[r][3]=0; matrice[r][4]=0; matrice[r][5]=1;}
      if(j==2) {matrice[r][3]=0; matrice[r][4]=1; matrice[r][5]=1;}
      if(j==3) {matrice[r][3]=0; matrice[r][4]=1; matrice[r][5]=0;}
      if(j==4) {matrice[r][3]=1; matrice[r][4]=1; matrice[r][5]=0;}
      if(j==5) {matrice[r][3]=1; matrice[r][4]=1; matrice[r][5]=1;}
      if(j==6) {matrice[r][3]=1; matrice[r][4]=0; matrice[r][5]=1;}
      if(j==7) {matrice[r][3]=1; matrice[r][4]=0; matrice[r][5]=0;}

      if(i==0) {matrice[r][0]=0; matrice[r][1]=0; matrice[r][2]=0;}
      if(i==1) {matrice[r][0]=0; matrice[r][1]=0; matrice[r][2]=1;}
      if(i==2) {matrice[r][0]=0; matrice[r][1]=1; matrice[r][2]=1;}
      if(i==3) {matrice[r][0]=0; matrice[r][1]=1; matrice[r][2]=0;}
      if(i==4) {matrice[r][0]=1; matrice[r][1]=1; matrice[r][2]=0;}
```

```
if(i==5) {matrice[r][0]=1; matrice[r][1]=1; matrice[r][2]=1;}
if(i==6) {matrice[r][0]=1; matrice[r][1]=0; matrice[r][2]=1;}
if(i==7) {matrice[r][0]=1; matrice[r][1]=0; matrice[r][2]=0;}
r++;
            }
      }
}
if(tipo_lut==5)
      {
      for (i=s->x1; i<=s->x2; i++)
            {
            for(j=s->y1; j<=s->y2; j++)
                  {
if(j==0) {matrice[r][2]=0; matrice[r][3]=0; matrice[r][4]=0;}
if(j==1) {matrice[r][2]=0; matrice[r][3]=0; matrice[r][4]=1;}
if(j==2) {matrice[r][2]=0; matrice[r][3]=1; matrice[r][4]=1;}
if(j==3) {matrice[r][2]=0; matrice[r][3]=1; matrice[r][4]=0;}
if(j==4) {matrice[r][2]=1; matrice[r][3]=1; matrice[r][4]=0;}
if(j==5) {matrice[r][2]=1; matrice[r][3]=1; matrice[r][4]=1;}
if(j==6) {matrice[r][2]=1; matrice[r][3]=0; matrice[r][4]=1;}
if(j==7) {matrice[r][2]=1; matrice[r][3]=0; matrice[r][4]=0;}

                  if(i==0) {matrice[r][0]=0; matrice[r][1]=0;}
                  if(i==1) {matrice[r][0]=0; matrice[r][1]=1;}
                  if(i==2) {matrice[r][0]=1; matrice[r][1]=1;}
                  if(i==3) {matrice[r][0]=1; matrice[r][1]=0;}
                  r++;
                  }
            }
      }
if(tipo_lut==4)
      {
      for (i=s->x1; i<=s->x2; i++)
            {
            for(j=s->y1; j<=s->y2; j++)
                  {
                  if(j==0) {matrice[r][2]=0; matrice[r][3]=0;}
                  if(j==1) {matrice[r][2]=0; matrice[r][3]=1;}
                  if(j==2) {matrice[r][2]=1; matrice[r][3]=1;}
                  if(j==3) {matrice[r][2]=1; matrice[r][3]=0;}

                  if(i==0) {matrice[r][0]=0; matrice[r][1]=0;}
                  if(i==1) {matrice[r][0]=0; matrice[r][1]=1;}
                  if(i==2) {matrice[r][0]=1; matrice[r][1]=1;}
                  if(i==3) {matrice[r][0]=1; matrice[r][1]=0;}
                  r++;
                  }
            }
      }
if(tipo_lut==3)
      {
      for (i=s->x1; i<=s->x2; i++)
            {
            for(j=s->y1; j<=s->y2; j++)
                  {
                  if(j==0) {matrice[r][1]=0; matrice[r][2]=0;}
                  if(j==1) {matrice[r][1]=0; matrice[r][2]=1;}
                  if(j==2) {matrice[r][1]=1; matrice[r][2]=1;}
```

```
                    if(j==3) {matrice[r][1]=1; matrice[r][2]=0;}

                    if(i==0) matrice[r][0]=0;
                    if(i==1) matrice[r][0]=1;
                    r++;
                    }
                }
            }
      if(tipo_lut==2)
            {
            for (i=s->x1; i<=s->x2; i++)
                {
                for(j=s->y1; j<=s->y2; j++)
                    {
                    if(j==0) matrice[r][1]=0;
                    if(j==1) matrice[r][1]=1;

                    if(i==0) matrice[r][0]=0;
                    if(i==1) matrice[r][0]=1;
                    r++;
                    }
                }
            }
      // calcolo espressione logica
      int ok=1;
      strcpy(expr,"");
      char p[3];
      for(j=0; j<tipo_lut; j++)
            {
            for(i=1; i<righe; i++)
                {
                if(matrice[i][j]!=matrice[0][j]) {ok=0; break;}
                }
            if(ok==1)
                {
                if(matrice[0][j]==1) sprintf(p,"%d",j);
                if(matrice[0][j]==0) sprintf(p,"!%d",j);
                strcat(expr,p);
                }
            ok=1;
            }
      if(strcmp(expr,"")==0) {printf("\nTRADUCI_SOTTOCUBO: return 0
da expr %s",expr);return 0; }
      return 1;
      }

// controlla se la casella (x,y) appartiene ad un sottocubo già
// preso
int in_sottocubo(int x, int y)
      {
      sottocubo *s=sottocubo_corrente;
      while(s!=0)
            {
      if((x>=s->x1 && x<=s->x2)&&(y>=s->y1 && y<=s->y2)) return 1;
      s=s->successivo;
            }
      return 0;
      }
```

B.3 file outfnct.c

```
/*------------------------------------------------------------------
          Funzioni per per la produzione del file di output
------------------------------------------------------------------*/

int emetti_codice(FILE *, componente *, int *, int *);

#define DEFAULT_PROB 0.5

// Produce il file traduzione
void produci(FILE *outStream, FILE *logStream)
      {
      int codice=0; // per controllare se la scrittura su file è
eseguita correttamente
      int pin_out=0; // indice pin di uscita del dispositivo
      int pin_in=0; // indice pin di ingresso del dispositivo
      int totale_componenti=0;
      int numero_componente=0;
      int n_lut=0;
      int n_mux=0;
      int n_flip_flop=0;
      int n_latch=0;
      int n_buffer=0;
      int n_input_pin=0;
      int n_output_pin=0;
      int n_components=0;
      componente *ptr=componente_corrente;
      while(ptr!=0)
            {
            totale_componenti++;
            if(strcmp(ptr->tipo_componente,"BUFGP")==0)
totale_componenti--;
            if(strcmp(ptr->tipo_componente,"GND")==0)
totale_componenti--;
            if(strcmp(ptr->tipo_componente,"VCC")==0)
totale_componenti--;
            if(strncmp(ptr->tipo_componente,"LUT",3)==0 ||
strcmp(ptr->tipo_componente,"INV")==0) n_lut++;
            if(strncmp(ptr->tipo_componente,"FD",2)==0)
n_flip_flop++;
            if(strncmp(ptr->tipo_componente,"LD",2)==0) n_latch++;
            if(strstr(ptr->tipo_componente,"BUF")!=NULL) n_buffer++;
            if(strcmp(ptr->tipo_componente,"IBUF")==0)
                  {
                  n_input_pin++;
                  fprintf(outStream,"%d const_inp_prob %f;\n",ptr-
>numero,DEFAULT_PROB);
                  }
            if(strcmp(ptr->tipo_componente,"OBUF")==0)
n_output_pin++;
            if(strstr(ptr->tipo_componente,"MUX")!=NULL) n_mux++;
            if(strstr(ptr->tipo_componente,"BUFG")!=NULL) n_buffer--
; // per non confondere il clock con un buffer
            ptr=ptr->componente_successivo;
            }
```

```
	n_components=totale_componenti-n_input_pin-n_output_pin;
	fprintf(outStream,"\n");
	char fpga[20]; strcpy(fpga,"query");
	fprintf(logStream,"/*------------------- Parameters of the
circuit -----------------------\n");
	fprintf(logStream,"N_LUTS = %d\n",n_lut);
	fprintf(logStream,"N_FLIP_FLOPS = %d\n",n_flip_flop);
	fprintf(logStream,"N_LATCHS = %d\n",n_latch);
	fprintf(logStream,"N_BUFFERS = %d\n",n_buffer);
	fprintf(logStream,"N_MULTIPLEXERS = %d\n",n_mux);
	fprintf(logStream,"N_INPUT_PINS = %d\n",n_input_pin);
	fprintf(logStream,"N_OUTPUT_PINS = %d\n",n_output_pin);
	fprintf(logStream,"N_COMPONENTS = %d\n",n_components);
	fprintf(logStream,"\nDevices %s\n",info_tecnologia(fpga));
	fprintf(logStream,"\nList of componets:\n\n");
	while(totale_componenti>0)
		{
		ptr=componente_corrente;
		while(ptr!=0)
			{
			if(ptr->numero==numero_componente) break;
			ptr=ptr->componente_successivo;
			}
		// a questo punto effettua la traduzione
		codice=emetti_codice(outStream, ptr, &pin_out, &pin_in);
		if(strcmp(ptr->tipo_componente,"OBUF")==0) pin_out++;
		if(strcmp(ptr->tipo_componente,"IBUF")==0) pin_in++;
		numero_componente++;
		totale_componenti--;
		fprintf(logStream,"%d %s\n",ptr->numero,ptr-
>nome_componente);
		}
	fprintf(logStream,"-----------------------------------------
---------------------------*/\n\n");
	}

// Scrive su file la entry per ogni componente. Ritorna 1 se la
entry è stata scritta, 0 altrimenti.
int emetti_codice(FILE *outStream, componente *ptr, int *pin_o, int
*pin_i)
	{
	porta *porta_ptr;
	componente *collegato_ptr;
	char str_d[9];
	char str_pre[9];
	char str_clr[9];
	char str_ce[9];
	char str_c[9];
	char str_funzione[200]; strcpy(str_funzione,"");
	int num_d, num_pre, num_clr, num_ce, num_c;
	static int accapo=0;
	if(strcmp(ptr->tipo_componente,"LUT1")==0) printf("\n LUT1
numero %d info %s",ptr->numero, ptr->info_componente);
	// porta xor viene convertita in lut2
	if(strcmp(ptr->tipo_componente,"XORCY")==0)
		{
		strcpy(ptr->tipo_componente,"LUT2");
		strcpy(ptr->info_componente,"6");
```

```
            }
      // buffer d'ingresso
      if(strcmp(ptr->tipo_componente,"IBUF")==0)
fprintf(outStream,"%d ibuf %d;\n",ptr->numero,*pin_i);
      if(accapo==0 && strcmp(ptr->tipo_componente,"IBUF")!=0)
            {
            fprintf(outStream,"\n");
            accapo=1;
            }
      if(strcmp(ptr->tipo_componente,"IBUF")==0) return 1;
      // inverter
      if(strcmp(ptr->tipo_componente,"INV")==0)
            {
            fprintf(outStream,"%d lut_fctn !0;\n",ptr->numero);
            porta_ptr=ptr->porte_componente;
            fprintf(outStream,"%d lut 1",ptr->numero);
            while(porta_ptr!=0)
                  {
                  collegato_ptr=porta_ptr->componente_collegato;
                  converti_num(str_d,collegato_ptr->numero);
                  fprintf(outStream," %s",str_d);
                  porta_ptr=porta_ptr-> porta_successiva;
                  }
            fprintf(outStream,";\n\n");
            return 1;
            }
      // flip flop FD
      if(strcmp(ptr->tipo_componente,"FD")==0)
            {
            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            if(num_d>=0) fprintf(outStream,"%d fd %d;\n\n",ptr-
>numero,num_d);
            else
                  {
                  if(num_d==-1) fprintf(outStream,"%d fd
GND;\n\n",ptr->numero);
                  if(num_d==-2) fprintf(outStream,"%d fd
VCC;\n\n",ptr->numero);
                  }
            return 1;
            }
      // flip flop FDP
      if(strcmp(ptr->tipo_componente,"FDP")==0)
            {
            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"PRE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_pre=collegato_ptr->numero;
            converti_num(str_pre,num_pre);

            fprintf(outStream,"%d fdp %s %s;\n\n",ptr-
>numero,str_pre,str_d);
```

```
            return 1;
            }
      // flip flop FDC
      if(strcmp(ptr->tipo_componente,"FDC")==0)
            {
            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"CLR");
            collegato_ptr=porta_ptr->componente_collegato;
            num_clr=collegato_ptr->numero;
            converti_num(str_clr,num_clr);

            fprintf(outStream,"%d fdc %s %s;\n\n",ptr-
>numero,str_d,str_clr);
            return 1;
            }
      // flip flop FDCE
      if(strcmp(ptr->tipo_componente,"FDCE")==0)
            {
            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"CE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_ce=collegato_ptr->numero;
            converti_num(str_ce,num_ce);

            porta_ptr=cerca_porta(ptr,"CLR");
            collegato_ptr=porta_ptr->componente_collegato;
            num_clr=collegato_ptr->numero;
            converti_num(str_clr,num_clr);

            fprintf(outStream,"%d fdce %s %s %s;\n\n",ptr-
>numero,str_d,str_ce,str_clr);
            return 1;
            }
      // flip flop FDCP
      if(strcmp(ptr->tipo_componente,"FDCP")==0)
            {
            porta_ptr=cerca_porta(ptr,"PRE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_pre=collegato_ptr->numero;
            converti_num(str_pre,num_pre);

            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"CLR");
            collegato_ptr=porta_ptr->componente_collegato;
            num_clr=collegato_ptr->numero;
            converti_num(str_clr,num_clr);
```

```
            fprintf(outStream,"%d fdcp %s %s %s;\n\n",ptr-
>numero,str_pre,str_d,str_clr);
            return 1;
            }
      // flip flop FDCPE
      if(strcmp(ptr->tipo_componente,"FDCPE")==0)
            {
            porta_ptr=cerca_porta(ptr,"PRE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_pre=collegato_ptr->numero;
            converti_num(str_pre,num_pre);

            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"CE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_ce=collegato_ptr->numero;
            converti_num(str_ce,num_ce);

            porta_ptr=cerca_porta(ptr,"CLR");
            collegato_ptr=porta_ptr->componente_collegato;
            num_clr=collegato_ptr->numero;
            converti_num(str_clr,num_clr);

            fprintf(outStream,"%d fdcpe %s %s %s %s;\n\n",ptr-
>numero,str_pre,str_d,str_ce,str_clr);
            return 1;
            }
      // latch LDPE
      if(strcmp(ptr->tipo_componente,"LDPE")==0)
            {
            porta_ptr=cerca_porta(ptr,"PRE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_pre=collegato_ptr->numero;
            converti_num(str_pre,num_pre);

            porta_ptr=cerca_porta(ptr,"D");
            collegato_ptr=porta_ptr->componente_collegato;
            num_d=collegato_ptr->numero;
            converti_num(str_d,num_d);

            porta_ptr=cerca_porta(ptr,"GE");
            collegato_ptr=porta_ptr->componente_collegato;
            num_ce=collegato_ptr->numero;
            converti_num(str_ce,num_ce);

            porta_ptr=cerca_porta(ptr,"G");
            collegato_ptr=porta_ptr->componente_collegato;
            num_c=collegato_ptr->numero;
            converti_num(str_c,num_c);

            fprintf(outStream,"%d ldpe %s %s %s %s;\n\n",ptr-
>numero,str_pre,str_d,str_ce,str_c);
            return 1;
```

```
		}
	// look up table LUT1
	if(strcmp(ptr->tipo_componente,"LUT1")==0)
		{
		if(strcmp(ptr->info_componente,"0")==0)
strcpy(str_funzione,"GND");
		if(strcmp(ptr->info_componente,"1")==0)
strcpy(str_funzione,"!0");
		if(strcmp(ptr->info_componente,"2")==0)
strcpy(str_funzione,"0");
		if(strcmp(ptr->info_componente,"3")==0)
strcpy(str_funzione,"VCC");
		fprintf(outStream,"%d lut_fctn %s;\n",ptr-
>numero,str_funzione);
		porta_ptr=ptr->porte_componente;
		fprintf(outStream,"%d lut %d",ptr->numero,1);
		collegato_ptr=porta_ptr->componente_collegato;
		converti_num(str_d,collegato_ptr->numero);
		fprintf(outStream," %s",str_d);
		fprintf(outStream,";\n\n");
		return 1;
		}
	// look up table LUT2 ... LUT6
	if(strncmp(ptr->tipo_componente,"LUT",3)==0)
		{
		converti(ptr->info_componente,str_funzione);
		fprintf(outStream,"%d lut_fctn %s;\n",ptr-
>numero,str_funzione);
		porta_ptr=ptr->porte_componente;
		fprintf(outStream,"%d lut %c",ptr->numero,ptr-
>tipo_componente[3]);
		while(porta_ptr!=0)
			{
			collegato_ptr=porta_ptr->componente_collegato;
			converti_num(str_d,collegato_ptr->numero);
			fprintf(outStream," %s",str_d);
			porta_ptr=porta_ptr-> porta_successiva;
			}
		fprintf(outStream,";\n\n");
		return 1;
		}
	// multiplexer
	if(strncmp(ptr->tipo_componente,"MUX",3)==0)
		{
		int numero_selettori=0;
		porta_ptr=ptr->porte_componente;
	while(porta_ptr!=0)
		{
		if(strstr(porta_ptr->nome_porta,"S")!=0)
numero_selettori++;
		porta_ptr=porta_ptr-> porta_successiva;
		}
	fprintf(outStream,"%d mux_fctn %d ",ptr->numero,
numero_selettori);
	porta_ptr=ptr->porte_componente;
	while(porta_ptr!=0)
		{
		if(strncmp(porta_ptr->nome_porta,"S",1)==0)
```

```
                {
                collegato_ptr=porta_ptr->componente_collegato;
                converti_num(str_d,collegato_ptr->numero);
                fprintf(outStream,"%s ",str_d);
                }
            porta_ptr=porta_ptr-> porta_successiva;
            }
        converti_num(str_d,ptr->numero);
        fprintf(outStream,";\n%s mux ",str_d);
        porta_ptr=ptr->porte_componente;
        while(porta_ptr!=0)
            {
            if(strncmp(porta_ptr->nome_porta,"S",1)!=0)
                {
                collegato_ptr=porta_ptr->componente_collegato;
                converti_num(str_d,collegato_ptr->numero);
                fprintf(outStream,"%s ",str_d);
                }
            porta_ptr=porta_ptr-> porta_successiva;
            }
        fprintf(outStream,";\n\n");
        return 1;
        }
        // buffer d'uscita
        if(strcmp(ptr->tipo_componente,"OBUF")==0)
            {
            porta_ptr=ptr->porte_componente;
            collegato_ptr=porta_ptr->componente_collegato;
            converti_num(str_d,collegato_ptr->numero);
            fprintf(outStream,"%d obuf %s %d;\n",ptr-
>numero,str_d,*pin_o);
            return 1;
            }
        printf("\nAttenzione: il componente %s di tipo %s non è
riconosciuto.",ptr->nome_componente,ptr->tipo_componente);
        return 0; // la entry del componente non è stata scritta
        }

// Funzione che converte interi naturali in stringhe, in "GND" o
"VCC" se l'intero è -1 o -2
int converti_num(char *stringa, int valore)
            {
            if(valore>=0) {sprintf(stringa,"%d",valore);}
            else
                {
                if(valore==-1) sprintf(stringa,"GND");
                if(valore==-2) sprintf(stringa,"VCC");
                }
            return 1;
            }
```

Appendice C
Utilizzo del traduttore

In quest'ultima appendice viene infine descritto come si utilizza il traduttore. Il traduttore è stato chiamato *eparse* e funziona con interfaccia a linea di comando. La sintassi del programma è la seguente:

```
eparse <nome_file_sorg> <nome_file_traduzione> <parametro (-s)>
```

`Nome_file_sorg` è il nome del file sorgente in EDIF che si deve tradurre nel linguaggio oggetto. Il file generato da *eparse* in cui è salvata la traduzione è chiamato con il nome specificato da `nome_file_traduzione` con estensione `net`. Il file di resoconto viene chiamato con lo stesso nome di quello che contiene la traduzione ma con estensione `chr`.

L'ultimo campo della riga di comando può essere infine il parametro `–s` (structure). Quando quest'ultimo viene specificato, *eparse* stampa a video tutte le informazioni memorizzate nella struttura dati. Questa funzionalità è stata utile per verificare il corretto funzionamento del traduttore e la consistenza della struttura dati.

I campi `nome_file_traduzione` e `parametro (-s)` sono entrambi opzionali. Quando `nome_file_traduzione` non viene specificato *eparse* produce per default i file d'uscita `output.net` e `output.chr`.

Bibliografia

Aho Alfred V., Lam Monica S., Sethi Ravi, Ullman D. Jeffrey

2006, Compilers Principles, Techniques, and Tools 2nd Edition, Addison Wesley

Bison - GNU parser generator

2010, Bison The Yacc-compatible Parser Generator, Charles Donnelly, Richard Stallman

< http://www.gnu.org/software/bison/manual >, ultima consultazione Mar. 2011

EDIF - Electronic Design Interchange Format

2000, International Electrotechnical Commission (IEC). IEC 61690-2 Electronic Design Interchange Format (EDIF) - Part 2: Version 4 0 0, Hilary Kahn, Robin La Fontaine, Rachel Lau

Flex – Fast Lexical Analyzer

2007, Lexical Analysis With Flex, Vern Paxson

< http://flex.sourceforge.net/manual/index.html#Top >, ultima consultazione Mar. 2011

Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John

1998, Elements of Reusable Object-Oriented Software, Addison Wesley

Hillawi J.I. and Bennett K.R.

(1986) *EDIF - An Overview,* “Computer-Aided Engineering J.” Giugno 1986, 102-107.

ISE - Design Suite Software

2010, ISE Design Suite Software Manuals and Help

< http://www.xilinx.com/support/documentation > , ultima consultazione Mar. 2011

Paxson Vern

1995, Flex a fast scanner generator Edition 2.5

Rubin Steven M.

1994, Computer Aids for VLSI Design, Addison-Wesley

Smith Michael John Sebastian

1997, Application-Specific Integrated Circuits, Addison Wesley

Ringraziamenti

Ringrazio il prof. Andrea Domenici e la prof.ssa Cinzia Bernardeschi per aver sostenuto questa tesi.

Ringrazio l’ing. Luca Cassano per la sua disponibilità e competenza.

Ringrazio i miei genitori per aver sostenuto il mio percorso universitario.

Ringrazio Laura per essere stata di grande sostegno morale durante questi anni.