AN FPGA-BASED SYNTACTIC PARSER FOR LARGE SIZE REAL-LIFE CONTEXT-FREE GRAMMARS

THÈSE N^{O} 2522 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Cristian Raul CIRESSAN

ingénieur informaticien, Université Polytechnique, Timisoara, Roumanie
et de nationalité roumaine

Lausanne, EPFL
2002

Version abrégée

Le sujet de cette thèse est situé au mi-chemin entre le traitement automatique du langage naturel et (TALN) et la conception de circuits numériques. Le but de cette thèse est la conception d'un coprocesseur pour améliorer le temps de l'analyse syntaxique du langage naturel. Le coprocesseur doit faire l'analyse syntaxique du langage naturel réel et est conçu pour être utile dans plusieurs applications TALN qui ont des contraintes de temps ou qui utilisent beaucoup des données.

Plus claire, les trois buts de cette thèse sont: (1) de proposer un efficient coprocesseur à base d'FPGA pour l'analyse syntaxique du langage naturel qui prend des entrées de la forme des treillis de mots, (2) d'implémenter un coprocesseur dans un outil matériel prêt à être intégré dans un ordinateur et (3) d'offrir une interface (i.e. librairie de composants) entre l'outil matériel et les éventuels logiciels de traitement automatique du langage naturel qui vont s'exécuter sur l'ordinateur.

La technologie FPGA (Field Programmable Gate Array) a été choisie comme support pour l'implémentation du coprocesseur grâce à son habilité d'exploiter de façon efficace tous les niveaux de parallélisme existant dans les algorithmes implémentés, tout en gardant un prix raisonnable. La dernière raison réside dans l'attente de voir les futurs coprocesseur génériques contenir des ressources réconfigurables. Dans un tel contexte, un module (IP core) qui implémente un analyseur syntaxique non contextuel prêt à être configuré dans les ressources réconfigurables du processeur générique serait un support pour toute application basée sur des analyses syntaxiques non contextuelles qui s'exécute dans ce processeur générique.

L'algorithme analyse syntaxique de la grammaire non contextuelle qui a été implémenté est l'algorithme CYK standard aussi bien qu'une de ses versions améliorées. Cette version améliorée de l'algorithme CYK a été développée dans le Laboratoire d'Intelligence Artificielle de l'EPFL. Ces algorithmes ont été sélectionnés (1) grâce à leurs propriétés intrinsèques liées au flux de données et au traitement des données régulières qui font d'eux des bons candidats pour l'implémentation matérielle, (2) pour leur capacité à produire des arbres syntaxiques partiels qui les rend adaptés pour des futures analyses syntaxiques de surface et (3) pour leur habilité à faire l'analyse syntaxique des treillis de mots.

Abstract

This thesis is at the crossroad between Natural Language Processing (NLP) and digital circuit design. It aims at delivering a custom hardware coprocessor for accelerating natural language parsing. The coprocessor has to parse real-life natural language and is targeted to be useful in several NLP applications that are time constrained or need to process large amounts of data.

More precisely, the three goals of this thesis are: (1) to propose an efficient FPGA-based coprocessor for natural language syntactic analysis that can deal with inputs in the form of word lattices, (2) to implement the coprocessor in a hardware tool ready for integration within an ordinary desktop computer and (3) to offer an interface (i.e. software library) between the hardware tool and a potential natural language software application, running on the desktop computer.

The Field Programmable Gate Array (FPGA) technology has been chosen as the core of the coprocessor implementation due to its ability to efficiently exploit all levels of parallelism available in the implemented algorithms in a cost-effective solution. In addition, the FPGA technology makes it possible to efficiently design and test such a hardware coprocessor. A final reason is that the future general-purpose processors are expected to contain reconfigurable resources. In such a context, an IP core implementing an efficient context-free parser ready to be configured within the reconfigurable resources of the general-purpose processor would be a support for any application relying on context-free parsing and running on that general-purpose processor.

The context-free grammar parsing algorithms that have been implemented are the standard CYK algorithm and an enhanced version of the CYK algorithm developed at the EPFL Artificial Intelligence Laboratory. These algorithms were selected (1) due to their intrinsic properties of regular data flow and data processing that make them well suited for a hardware implementation, (2) for their property of producing partial parse trees which makes them adapted for further shallow parsing and (3) for being able to parse word lattices.

Acknowledgements

First of all, I would like to thank my thesis director Dr. Martin Rajman who knew how to guide and encourage me whenever long term decisions were to be made. His ability to guide my work was priceless and the freedom he granted me during this thesis was something I truly appreciated. I would also like to thank the members of my jury, Prof. Roger D. Hersch, Prof. M. Vilares Ferro, Prof. Dominique Lavenier and Prof. Eduardo Sanchez for the interesting discussions I had with them during and after the examination.

Many thanks to my colleagues for the environment they have created in the laboratory. I enjoyed those broodwar evenings (or mornings) in the middle of the week when we struggled for victory. Entaro Adun! I still do not understand how Cedric was able to do it on his laptop – I always needed a large screen – neither how Romaric was always able to gather those hoards of zergs. And a warning for those staying in the lab: do not dear to forget to invite me when WarCraft III is coming out! I only have a concern: Who's the next PC administrator in the lab?

Who's left? Oh, yes! Many thanks to Ionut and Steve who provided me with the two black beasts that run my simulations during the last 3 weeks before I had to hand-out my thesis. I am sorry I had to install Windows 2000 on them! I know Linux is now in place . . .

I would also like to mention my friends here in Lausanne that were a nice company during lunch time or during those long coffy breaks; especially to Mady for being always ready to help and give advice.

Finally, I would like to mention my parents that believed in me and left me full freedom to shape with passion my studies as I choose. Especially my father and my grandmother – still amazed on how she knew to seed the grains of ambition and motivate me – that are no longer here.

Claudia! Many thanks for your patience during the hard time I had writing this thesis. I know I was uneasy. Many thanks for correcting the figures in this thesis in order to make them look as they are now.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis you read is aimed at delivering a custom hardware coprocessor for accelerating natural language parsing. The coprocessor has to parse real-life natural languages and is useful in several Natural Language Processing (NLP) applications that are time constrained or when large amounts of data need to be processed. The coprocessor is built around the Field Programmable Gate Array (FPGA) technology that renders feasible the design and testing of such a hardware coprocessor as well as its integration within a desktop application framework.

This introductory section will first introduce the notions of NLP and parsing and then discuss the types of grammars used for modelling and parsing natural languages. Some reasons for why we chose to implement a parsing algorithm for context-free grammars and why fast parsing is needed are given – the later by means of some typical application examples. The parsing algorithm implemented in the hardware coprocessor and its main features are presented. A short introduction to the FPGA technology is also given. The related work is reviewed and the thesis goals are established. Finally, the thesis outline is provided.

1.1 Natural Language Processing and parsing

NLP is concerned with the study, design and implementation of computational machinery that can communicate with the humans by means of natural language. In general, the processing of natural languages is decomposable into a number of analysis stages coming in this order: lexical and morphological, syntactic, semantic and pragmatics, although this is only an abstraction as there are interactions between any two of these stages. For the sake of completeness we should also mention that some languages, such as Chinese, Japanese, and Thai, require a supplementary stage, called segmentation, before the lexical and morphological stage [10]. The parsing takes place at the syntactic stage and is usually an intermediate step towards further processing, such as the assignment of meaning to a sentence.

In this context, parsing is an important ingredient and is the process of giving syntactic structure to a sentence according to an underlying formal grammar[1]. Some basic notions (i.e. concepts) of parsing are given bellow:

- recognizer : a procedure that decides whether a sentence is syntactically correct or not according to a given grammar;
- parser : a recognizer that can also produce the associated structural analyses (i.e. parse trees) according to the given grammar;

[1]A formal grammar is a human-constructed formalism that is meant to describe languages.

- robust parser : a procedure that attempts to produce partial analyses even in the case when the input sentence is not actually (entirely) syntactically correct according to the given grammar;

As input for the parsing stage we assume to have a sequence of words coming from the lexical and morphological stage, and as output a data-structure (e.g. parse trees) suitable for semantic interpretation.

1.1.1 Why Context-Free Grammars ?

It was in the late 1950s when the linguist Noam Chomsky introduced the formal syntactic description of languages. He defined several classes of formal grammars – and their associated languages – structured in a hierarchy known today as the Chomsky hierarchy [14]. The context-free grammars (CFGs) are also part of this hierarchy[2]. In computer science the grammars introduced by Noam Chomsky are human-constructed formalisms used to describe classes of formal languages that range from regular, context-free, context-sensitive to recursively enumerable. For linguists, the CFGs and context-sensitive grammars, are of particular interest as they allow for describing the complex syntax of natural, human spoken languages.

The ability (i.e. expressiveness power) of context-free languages to deal with certain *subtle* linguistic features of natural languages is sometimes considered to be too weak. Context-sensitive grammars may be needed for dealing with such features of natural languages. On the other hand, in NLP, efficiency (i.e. parsing time) is often a requirement for many practical reasons. The worst-case parsing time of well-known algorithms for CFGs is much more efficient than the parsing time for context-sensitive grammars. Therefore the linguists are often required to find a tradeoff between expressiveness power and efficiency of grammars used to model natural languages. Often, due to the fact that only few features of the natural languages require such "complex" grammars as context-sensitive grammars, CFGs are widely used in the NLP community. Among other reasons for using CFG for NLP we mention:

- any grammar however complex can be rewritten in the form of a CFG. Thus, we only have to eventually rewrite a given grammar to an equivalent CFG in order to benefit of the efficient CFG parsing algorithms;

- the time and space complexity of the state-of-the-art parsing algorithms for CFG are polynomial and thus better adapted for NLP applications that are real-time and/or memory constrained;

1.1.2 Why *fast* parsing ?

NLP applications may be classified in three main fields: data processing, data production and natural human-machine interfaces. For each of these application fields we give some typical application examples that require *fast* parsing.

The first application field, data processing, includes automatic translation, information retrieval and text mining and each of these applications requires parsing for different reasons. Due to the large amounts of data that need to be processed (i.e. parsed), *fast* parsing tools are required.

[2]Interestingly, at about the same time, a committee defining the ALGOL programming language introduced a programming language description formalism called Backus-Naur form (BNF) which turned out to be equivalent to the class of CFGs of the Chomsky hierarchy.

The second application field, data production, includes optical character recognition (OCR) systems and spell checkers. In the case of high quality and reliable OCRs it is not always enough to produce the recognized words since syntactic errors may be produced during this process. For this reason high quality OCR tools should integrate syntactic analysis in order to rule-out the wrong variants of the recognised sentences in the text. On the other hand, the recognition time is relatively small for some commercially available products (e.g. Recognita, Omnipage) and the amount of time we spend in syntactic analysis for ruling-out the wrong variants may be a big pour cent in the overall recognition time. Thus, for not being an overhead in the overall OCR process the parsing has to be *fast*. An automatic spell checker has to look-up a dictionary and find the best replacement for a misspelled word. Again, we can use syntactic analysis for replacing the misspelled word in the sentence with a correct word that is also syntactically correct. For the same reasons as in the case of the OCR systems we need a *fast* parsing tool.

Finally, in the case of human-machine interfaces, state-of-the-art vocal interfaces use standard Hidden Markov Models (HMM) that only integrate very limited syntactic knowledge and a better integration of syntactic processing within speech-recognition systems is an important goal. For instance, in the case of a sequential coupling [22], the output of the speech recognizer (often represented in a compact form called word graph or word lattice) may be further processed with a syntactic parser to filter-out those of the hypotheses that are not syntactically correct. Again, due to the real-time constraints of such an application, fast parsing is required.

1.1.3 Why the CYK algorithm ?

The first designed context-free parsers used backtracking to search exhaustively for the syntactic structures matching the input string. The worst-case running time of these algorithms are exponential in the length of the input and therefore impractical. There are two well-known, practical parsing algorithms for CFG, the Earley's algorithm [1] and the Cocke-Younger-Kasami (CYK) algorithm [1, 32]. Both algorithms have the same polynomial[3] worst-case time complexity (i.e. $O(n^3)$,where n is the length of the input string) and are an instance of dynamic programming, in which the syntactic structure is built in an incremental process with each step relying on previous performed computations.

A disadvantage of the CYK algorithm would be that it requires a CFG written in a restricted normal form (i.e. the Chomsky Normal Form that will be introduced latter). However, this inconvenient was eliminated by the development in our laboratory of an extension of the CYK algorithm [5, 6] – referred henceforth as the enhanced-CYK algorithm – that can deal with almost unrestricted CFGs.

On the other hand, from the hardware point of view both the CYK and the enhanced-CYK algorithms are regular in terms of data movement and data processing, throughout the parsing process. These are important requirements for an algorithm to be implemented in hardware [12, 18]. A required feature of the hardware coprocessor is its ability to parse word lattices in order to be integrated within a speech recognition framework and both the CYK and the enhanced-CYK algorithms can be adapted for parsing word-lattices. An intrinsic feature of the CYK and enhanced-CYK algorithms is that they perform robust parsing. In other words, they produce all the syntactic analyses of all the sequences of words in the input sentence. The

[3]Valiant [27] showed that the computation performed by the CYK algorithm is related to Boolean matrix multiplication and that it can be performed in subcubic time. The method of Valiant runs in $n^{2.81}$, and there are also more recent results for multiplying $n \times n$ matrices in time proportional to $n^{2.55}$ [2]. However, the overhead of these methods renders them impractical for values of n in the range of practical interest.

above feature is a requirement for performing shallow parsing . Finally, both the CYK and the enhanced-CYK algorithms are highly parallel and the FPGA technology can easily exploit the different levels of parallelism, i.e. fine-grained and coarse-grained, available in these algorithms. For the reasons mentioned above we have choose the (enhanced-)CYK algorithm for the hardware coprocessor implementation.

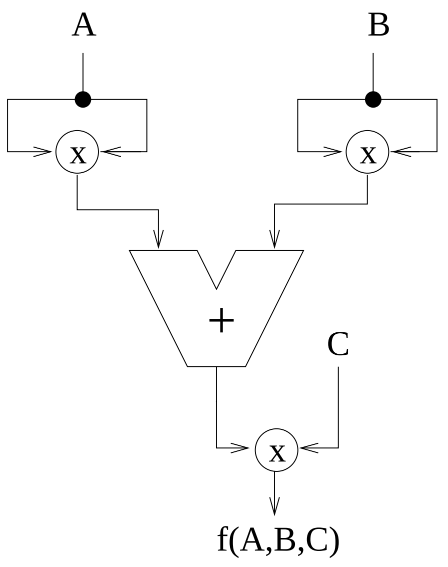
1.2 The FPGA technology

The concept of programmable logic was introduced around the year 1960 but due to technological limitations it took some time, until 1975-1978, when the first programmable logic device (PLD) came to market in the form of a Programmable Array Logic (PAL) circuit. Since then, the technology advanced very fast under various types of devices such as PAL, PLA, GAL, CPLD and Field Programmable Gate Arrays FPGAs [4, 19] . The two most successful technologies today are the complex programmable logic devices (CPLDs) and the FPGAs. The difference between these technologies resides in the kind of hardware primitives they offer for system implementations, although today the distinction between the two becomes more an more blurred. Today state-of-the-art FPGAs offer 4 million gates (e.g. Xilinx's Virtex-E family XCV3200 FPGA) compared to hundreds at the eves of this technology. FPGAs with such a high gate density make feasible the implementation of very complex designs.

The main advantage of the FPGA/CPLD circuits is that they can be configured after manufacturing. Basically, they contain three main types of configurable elements: logic cells, input/output cells and interconnection resources [26]. The logic cells are used to implement combinatorial and sequential logic functions, the input/output cells provide an interface with the external world and the routing resources are used to route signals throughout the circuit. Each of the configurable resources can be assigned a configuration from a set of predefined possible configurations. For instance a logic cell may be assigned a NAND logic function from the set {AND, NAND, OR, NOR} of possible configurations, while an input/output cell may be configured as an output from the set {input, output, input-output}. Complex circuits can be built by interconnecting these basic elements.

The FPGAs may be clasifyied in two broad categories, namely coarse-grained and fine-grained. The logic cells of the fine-grained FPGAs consist of simple functionally complete logical gates (e.g. NAND) or a low-complexity universal function (e.g. 2 or 3 variables controlled multiplexer). The logic cells of the coarse-grained FPGAs usually implement a logic function of several variables (e.g. a 5 variables logic function, often implemented by means of a look-up table) along with several flip-flops (e.g. $2 - 4$). However, state-of-the-art FPGAs often contain some more sophisticated primitives – that were previously features of CPLDs – such as dual-port memories, full-adders or single-cycle multipliers in order to support a broad range of applications. For configuring the FPGAs there are several technologies and we assume throughout this thesis that the static RAM-based technology in used. With the static RAM-based technology, each programmable resource in the FPGA is controlled by means of a set of bits stored in the static RAM. This technology benefits from the fact that the static RAM memory can be reprogrammed an indefinitely number of times.

In digital design, the FPGAs are an alternative to general-purpose processors and Application Specific Integrated Circuits (ASICs) and are often referred to as "the third computational paradigm" [28]. When compared to general-purpose processors and Digital Signal Processors (DSPs) the FPGAs have the important advantage of being able to exploit different levels of parallelism available in the implemented algorithm. When implementing a function, the FP-



```
load R1,A
mul R1,A
load R2,B
mul R2,B
add R1,R2
mul R1,C
store R1,(addr)
```

(a) (b)

Figure 1.1: FPGA spatial (a) vs. general-purpose processor temporal mapping (b) for the function $f(A, B, C) = (A^2 + B^2)C$. R1 and R2 are internal registers of the general-purpose processor and the mnemonics "mul" and "add" correspond to the operation of multiplication, respectively addition.

GAs are doing a spatial mapping while the processors are doing a temporal mapping. This is illustrated in figure 1.1 for the mathematical function $f(A, B, C) = (A^2 + B^2)C$. While the circuit can compute the function in one time step – the latency involved is not relevant now – the software requires seven time steps. However, the FPGAs and the general-purpose processors are two extreme cases of a rich architectural space and both FPGAs and general-purpose processors have advantages and disadvantages when compared to each other. Some applications are better suited for being implemented on a general-purpose processor, while others are better suited for an implementation on an FPGA. However, the distinction is not clear and it is often the case that for a particular application there are parts that are better suited for an FPGA implementation while others are better suited for being implemented within a general-purpose processor. Concretely, when running an application, both the general-purpose processor and the FPGA should execute those parts of the application that they are best suited for. A mixed circuit consisting of both an general-purpose processor and an FPGA would be therefore an elegant and effective solution [28]. On the other hand splitting an application in parts which will run on the general-purpose processor, respectively parts that will run on the FPGA is not easy and by the time this thesis is written still requires human intervention. A "smart" compiler able to do all this process automatically is needed. Eventually, hardware synthesis technology and conventional compiler technology will converge and the compiler will manage both the reconfigurable and fixed resources on the general-purpose processor. At the time being, we can already find on the market processors that contain reconfigurable resources. In the context of this trend we believe that a design of the (enhanced)-CYK algorithm ready to be used (i.e. configured) within the reconfigurable resources of a general-purpose processor running an NLP application that intensively uses parsing, can be powerful tool.

On the other hand, when compared to application specific computers, usually implemented around one or several ASICs, we can say that FPGAs are a cost-effective alternative, while offering almost the same performance. Clearly stated, the FPGAs offer both the flexibility of programmable general-purpose processors and the performance of ASICs in a cost-effective solution. While not all applications benefit from an FPGA implementation, the (enhanced)-CYK algorithm, due to its intrinsic characteristics can dramatically benefit from an FPGA implementation.

1.3 Related work

The various granularity and high degree of parallelism available in the (enhanced-)CYK algorithm allows its implementation on several arrays of processors architectures.

It is known that, the (enhanced-)CYK algorithm has $O(n^3)$ time complexity when executed on a sequential processor, $O(n^2)$ time complexity when executed on a 1D-array of processors, respectively $O(n)$ time complexity when executed on a 2D-array of processors [17], where n is the length of the input sentence.

Several hardware implementations (e.g. Very Large Scale Integration, VLSI) and mappings of the algorithm have been tried (e.g. on different computing platforms) in order to investigate the potential speedup available. A survey is given in this section. The conclusion of this survey is that (1) the proposed VLSI implementations require to many hardware resources when dealing with large-size real-life CFGs and (2) the proposed algorithm mappings always require an expensive parallel machine and are limited by factors such as intensive interprocessor communication and expensive processor allocation methods.

1.3.1 VLSI implementations

Various VLSI designs have been proposed: a syntactic recognizer based on the CYK algorithm on a 2D-array of processors [8] and a robust (error correcting) recognizer and analyzer (with parse tree extraction) based on the Earley algorithm on a 2D-array of processors [7]. Although these designs meet the usual VLSI requirements (constant-time processor operations, regular communication geometry, uniform data movement), the hardware resources they require do not allow them to accommodate large-size real-life context-free grammars used in large-scale NLP applications[4]. An analysis of such a VLSI architecture when mapped on a state-of-the-art FPGA is given in section 3.1.

1.3.2 Hypercube architecture mapping

An implementation of a parallel CYK algorithm for recognition and parsing on a NCUBE/7 machine is given in [16]. This implementation is essentially concerned with the mapping of a parallel CYK algorithm on a hypercube architecture and the problems related with such an implementation (i.e. the cost of interprocessor communication, processor allocation methods). Their benchmarks report a speed-up factor of about 20 in the best case. The disadvantage of such an implementation resides in the fact that it requires an expensive hypercube machine. We want a solution that allows us to integrate our hardware accelerator within a common desktop computer, in a cost effective solution.

[4]For instance the context-free grammar extracted from the SUSANNE corpus [25] contains more than $10,000$ non-terminals and $70,000$ grammar rules when written in Chomsky normal form

1.3.3 Distributed memory systems and shared memory systems

Implementations of parallel CYK algorithms on distributed memory systems were tried in [3, 21]. The implementation in [21] uses a massively parallel computer AP1000+ (with 256 Super Sparc at 50MHz) an the benchmark tests report a speedup factor of about 45 in the best case. Again, the disadvantage resides in the requirement of an expensive parallel computer. The implementation presented in [3] uses as test environment a network of computers in which each computer stores a fragment of the working data and the communication between computers is implemented by passing messages. The conclusions of this work is that inter-computer communication is very expensive and is the cause of poor performance. Basically no significant speed-up is reported.

Finally, an implementation of a shared memory system is given in [3] that uses a multiprocessor computer in which the processors share the same memory for storing the working data. This implementation is again concerned with the mapping of the CYK algorithm on a multiprocessor machine. The problems related with such an implementation are again – as in the case of the NCUBE/7 machine – the overhead introduced by interprocessor communication and processor allocation methods.

1.4 Thesis goals.

The three goals of this thesis are: (1) to propose an efficient coprocessor that can be integrated within a speech-recognition system (i.e. can parse word lattices[5]) and investigate various methods for accelerating natural language syntactic analysis, (2) to integrate the coprocessor in a hardware tool ready for integration within an ordinary desktop computer and (3) to offer an interface between the hardware tool and natural language applications, requiring parsing, and that are running on the desktop computer.

We have choose the FPGA technology as the core of such a coprocessor implementation, due to its ability to efficiently exploit all levels of parallelism available in the implemented algorithms in a cost-effective solution. A reason is also that the future general-purpose processors are expected to contain reconfigurable resources. In such a context an IP core implementing an efficient context-free parser – and ready to be configured within the reconfigurable resources of the general-purpose processor – would be a support for any application relying on context-free parsing and running on that general-purpose processor.

The context-free grammar parsing algorithms implemented are the CYK, respectively an enhanced-CYK algorithm developed in the Artificial Intelligence Laboratory of EPFL. These algorithms were elected (1) due to their intrinsic properties of regular data movement and data processing that render them well suited for a hardware implementation, (2) for their property of producing partial parse trees that makes them adapted for further shallow parsing and (3) for being able to parse word lattices.

The hardware design (i.e. the coprocessor) of a context-free parser can be integrated in several ways within an application framework. Once such integration was already presented and relies on a processor that contains reconfigurable resources. Another typical integration within an application framework would be as an accelerator FPGA board that has a "generic" interface (e.g. a PCI or SCSI interface) to the outside world. An example of such an integration is given for the particular case of a speech-recognition system, namely, a Vocal Information Server in figure 1.2 , in which the coupling between the speech recogniser and the context-free

[5]A compact representation of the output produced by the acoustic modules of speech recognizers.

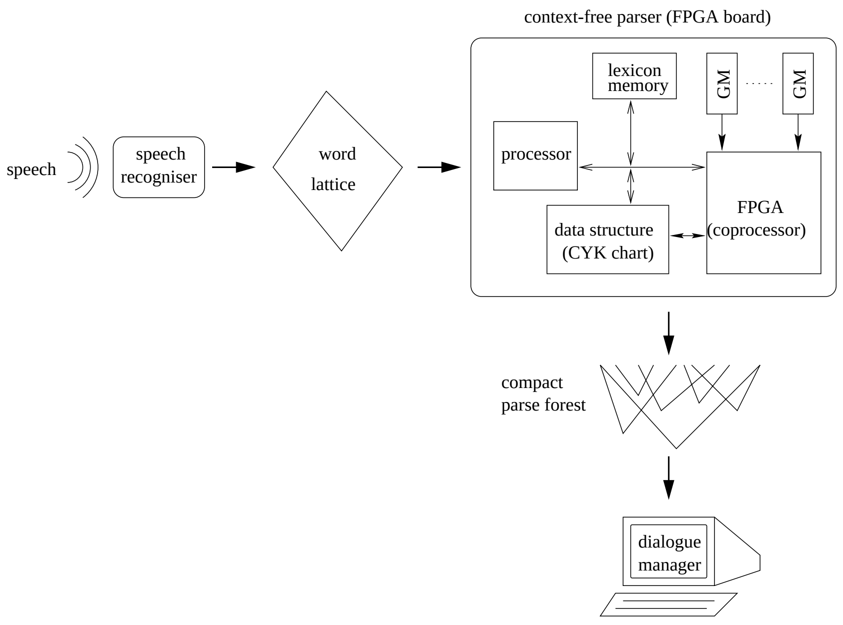


Figure 1.2: An example of integration of the hardware context-free parser accelerator (FPGA-board) in a speech-recognition application framework, namely a Vocal Information Server. The input to the FPGA-board is a word lattice and the output is a compact parse forest. GM stands for grammar memory.

parser is sequential. The output of the speech recogniser (a word lattice) is the input of the FPGA board and the output of the FPGA board (a compact parse forest) is the input of the semantic module that feeds a dialogue manager.

More precisely, the processor on the accelerator FPGA board is the "glue logic" used to convert the word lattice in a format used to initialise the coprocessor (i.e. context-free parser) data structures by making use of a lexicon. The parsing takes place in the FPGA coprocessor and relies on the intensive use of the grammar memories that store the grammar rules of the CFG in a compact format. The resulting parse forest is extracted by the processor from the coprocessor's data structures and forwarded to an external processing unit (e.g. a desktop PC) for further processing.

1.5 Outline

This thesis is on the crossroad between NLP and digital circuit design and for this reason a brief introduction in both the NLP and the FPGA technology has been already given. An introduction to the CYK and enhanced-CYK algorithms and their adaptation for parsing word lattices is also given in chapter 2. The following chapters 3-6 are the milestones on the path we followed along our design methodology.

As the project started we had little or no knowledge about the hardware implementation of the (enhanced-)CYK algorithm. Therefore, the first step (chapter 3) of our design methodology, was to build a working hardware implementation for the CYK algorithm in order to validate and prove its feasibility. The proposed hardware architecture is a linear array of processors that can deal with large-size real-life Chomsky Normal Form CFGs. This hardware architecture was physically implemented, tested and validated on a commercial FPGA board.

The second step (chapter 4), was a throughout analysis of the linear array of processors design supposed to bring the necessary knowledge about several aspects of the CYK algorithm such as: (1) required FPGA resources and size of the employed data-structures, (2) for getting acquainted with the real-life behaviour of the CYK algorithm and (3) for having a first evaluation of the speed-up factor against a software implementation.

The third step (chapter 5) proposes a new hardware architecture called dynamic array of processors that addresses several drawbacks of the linear array of processors design, among which a better processor allocation allocation method. The performance measurements and analysis of the dynamic array of processors adds more knowledge about the real-life behaviour of the CYK algorithm such as (1) unbalanced processor load and (2) required number of processors in the design.

The forth and last step (chapter 6) of our design methodology, proposes a hardware architecture that implements the enhanced-CYK algorithm and a method called tilling that balances the processor load. The proposed hardware architecture reuses the useful features of the previous designs and also addresses their drawbacks.

Chapter 7, presents an FPGA board built around a Xilinx Virtex-E V2000efg1156-6 FPGA that runs the hardware design of the enhanced-CYK algorithm. The FPGA board has a PCI interface and is supposed to work as an accelerator board within a host computer (e.g. PC).

In the conclusions (chapter 8), we describe some possible improvements to the current implementation and propose future directions for the project.

Chapter 2

The (enhanced)-CYK Algorithm Adapted for Word Lattice Parsing

This chapter first presents the CYK algorithm that is restricted to the use of CFGs written in Chomsky Normal Form, and its adaptation for parsing word lattices. Next, an extension of the CYK algorithm, referred henceforth as the enhanced-CYK algorithm is introduced. The enhanced-CYK algorithm is able to deal with a class of CFGs which is larger then the Chomsky Normal Form grammars, called nplCFG ("non partially lexicalized context-free grammars"). The adaptation of the enhanced-CYK to word lattice parsing is then presented.

2.1 Context-free grammars and the Chomsky Normal Form

We are familiar with finite state automata that are equivalent to regular languages [20]. The context-free languages are a superset of the regular languages and are described by CFGs. A simple example illustrating the difference between regular languages and context-free languages is the language $L = \{a^n b^n \mid n \geq 1\}$ that cannot be generated by a finite state automata but can be generated with a CFG (see example 2.1). The language L is commonly used to represent simple syntactic structures in programming languages such as **begin-end** blocks. The reason is that the phenomena of recursion illustrated by the language L requires infinite memory for dealing with. On the other hand, however, all computing machines have only a limited amount of memory and can thus be modelled by a regular language. In this context, CFGs are used more for practical reasons like compactness an simplicity of representation.

Formally, a CFG is a 4-tuple $G = \{N, \Sigma, S, R\}$ where:

- N is a set of non-terminals;
- Σ is a set of terminals;
- $S \in N$ is the top level non-terminal (corresponding to a sentence);
- R is a set of grammar rules, i.e. a subset of $N \times (N \bigcup \Sigma)^*$ written in the form of $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \bigcup \Sigma)^*$. For instance, a grammar rule such as $S \rightarrow NP\ V$, expresses the fact that a sentence (S) consists of a noun-phrase (NP) followed by a verb (V).

In the text that follows, we use capital letters X, Y, Z,... to denote non-terminal symbols and lower case letters a, b, c, ... to denote terminal symbols. An example of CFG is given bellow.

Example 2.1 A CFG that can generate the language $L = \{a^n b^n \mid n \geq 1\}$.

$$N = \{S, X\},$$
$$\Sigma = \{a, b\},$$
$$R = \{S \rightarrow aXb,\ S \rightarrow ab,\ X \rightarrow aXb,\ X \rightarrow ab\} \qquad \diamondsuit$$

There are several normal forms of the CFGs that have some practical and theoretical interest and the Chomsky Normal Form is one of them. In the formal definition of a grammar rule for CFGs there is no restriction on the right-hand side of the grammar rule, and any succession of non-terminals and terminals is allowed. When written in CNF, every grammar rule right-hand side is either a succession of two non-terminals or a single terminal symbol. Formally, every grammar rule is either of the form $X \rightarrow YZ$ or $X \rightarrow a$.

Any general CFG can be rewritten in an equivalent CNF grammar. The rewriting, however, will increase the number of non-terminals and production rules in the CNF grammar and for this reason the CNF rises some practical concerns.

2.2 The CYK algorithm adapted for word lattice parsing

A description of the CYK algorithm can be found in [1, 20]. The CYK algorithm is an instance of dynamic programming and uses a chart (i.e. a lower triangular matrix) of size $n(n+1)/2$, to store the data produced during the parsing process, where n is the size of the input sequence to be parsed. The algorithm starts with an empty chart in which only the bottom cells are initialised according to the input sequence and continues by filling the entries of the chart in a bottom-up fashion. Being a dynamic programming method, it fills incrementally entries of the chart, based on previously computed entries. The CYK algorithm is constrained to use a grammar written in CNF.

The following definitions and terms are used in the NLP community: the term "sentence" for the input sequence to be parsed and the term "word" for the terminals in the input sequence. Also, we will use the term "lexical" rule to denote the rules of the form $X \rightarrow a$. In practice the lexical rules only appear in the lexicon and not in the grammars. This separation of the rules in lexical rules (i.e. $X \rightarrow a$) and grammar rules (i.e. $X \rightarrow YZ$) is convenient for describing the grammars we are usually dealing with in practice.

Assume that we have the CNF context-free grammar $G = \{N, \Sigma, S, R\}$ and an input sentence $w_1 w_2 \ldots w_n,\ n \geq 1$, where $w_i \in \Sigma$ is the i^{th} word in the sentence. Let $w_{ij} = w_i w_{i+1} \ldots w_{i+j-1}$ be the part of the input sentence that starts at w_i and contains the next $j-1$ words and $N_{i,j}$ the subsets of N defined by $N_{i,j} = \{X \in N : X \Rightarrow^\star w_{ij}\}$. The notation $X \Rightarrow^\star w_{ij}$ means that w_{ij} can be derived from X by applying a succession of grammar rules. Every set $N_{i,j}$ can be associated with the entry on column i and row j of the lower triangular matrix, henceforth referred as the chart.

The CYK algorithm is defined as follows: In the algorithm given above, the lines $1-6$ correspond to the initialisation step, when the sets $N_{i,1}$ are initialised by using the "lexical" rules of the form $X \rightarrow w_i$. It may be clear now that the advantage of the lexical rules is that they restrict the processing of the "lexicalized" rules to the initialization step. The initialization step consists of filling each set $N_{i,1}$ in the bottom row ($j = 1$) of the chart with all the non-terminals X for which there exists a lexical rule $X \rightarrow w_i$ associated with that word. The time complexity of the initialization step is thus $O(n)$.

The lines $7-13$ correspond to the subsequent filling-up of the chart once the initial sets $N_{i,1}$ were initialized. Finally, the parsing trees can be extracted from the chart if necessary. If

Algorithm 1 The CYK algorithm

```
1:  for $i = 1$ to $n$ do
2:      $N_{i,1} = \{X : (X \rightarrow w_i) \in R\}$
3:      for $j = 2$ to $n - i + 1$ do
4:          $N_{i,j} = \emptyset$
5:      end for
6:  end for
7:  for $j = 2$ to $n$ do
8:      for $i = 1$ to $n - j + 1$ do
9:          for $k = 1$ to $j - 1$ do
10:             $N_{i,j} = N_{i,j} \bigcup \{X : (X \rightarrow YZ) \in P$ with $Y \in N_{i,k}$ and $Z \in N_{i+k,j-k}\}$
11:         end for
12:     end for
13: end for
```

S is among the topmost symbols (root) of such a tree, the sentence is syntactically correct for the grammar G.

The following example illustrates how the CYK algorithm works.

Example 2.2 Let's assume that the CNF grammar is given by:

$$\begin{gathered} N = \{S, X, Y, Z\}, \\ \Sigma = \{a, b, c, d\}, \\ R = \{S \rightarrow XY\ (r1),\ X \rightarrow YX\ (r2),\ Y \rightarrow ZX\ (r3), \\ Z \rightarrow ZY\ (r4),\ X \rightarrow a\ (r5),\ Y \rightarrow b\ (r6), \\ Z \rightarrow a\ (r7),\ X \rightarrow c\ (r8),\ Z \rightarrow d\ (r9)\} \end{gathered}$$

where S is the top level non-terminal and $r1, \ldots, r9$, are used to denote the grammar rules. With this grammar, we want to parse the sentence "`b a d b c`". During the initialisation

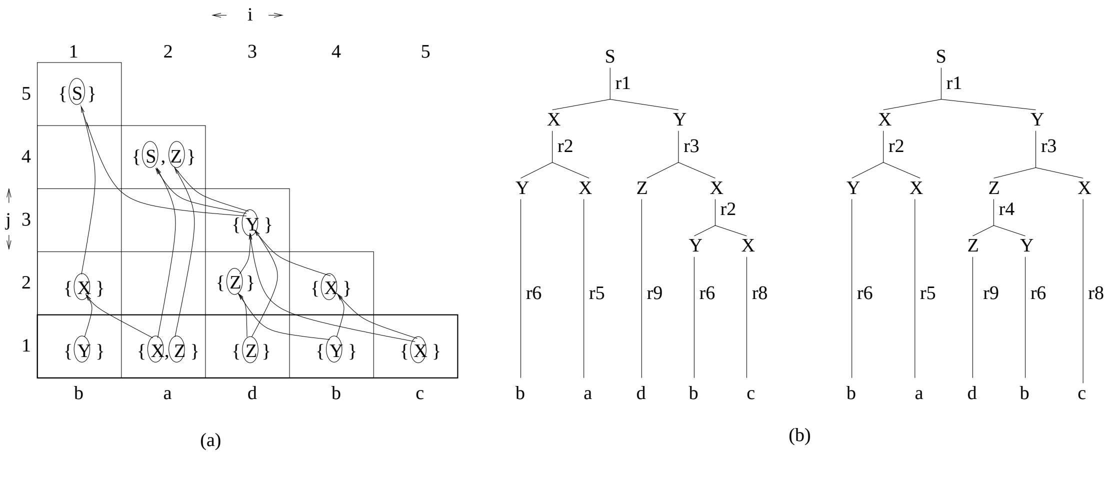


Figure 2.1: CYK (a) chart initialization (j=1) and filling (j=2,3,4,5), (b) two possible parsing trees corresponding to the input sentence "`b a d b c`"

step, only the rules $r5$ to $r9$ are used (see figure 2.1(a)) to fill-in the entries on the bottom line of the chart. During chart filling, the rows $j = 2, \ldots, 5$ are filled successively in this order, making use of the grammar rules $r1$ to $r4$. Finally, two possible parsing trees are extracted from the chart (see figure 2.1(b)). In this figure we represented on the branches of the parse trees the grammar rules that were actually used for building them. $\diamondsuit$

One important aspect of the CYK algorithm is that its initialization step can be easily generalised to accommodate (1) compound words and (2) word lattices, useful within a speech recognition framework for parsing the multiple sentences produced for a given acoustic input. For doing so, the initialization step has to take into account the possibility to initialize any of the sets $N_{i,j}$ and not only the sets $N_{i,1}$ of the bottom row. How this is dealt with in the particular case of compound words and word lattices is explained bellow.

Compound words : An example of compound word is "credit card". If the input sentence contains this compound word at position i in the input sentence, the algorithm should initialize the set $N_{i,1}$ for the word "credit", respectively the set $N_{i+1,1}$ for the word "card" as well as the entry at position $(i, 2)$ for the compound word "credit card" (see figure 2.2). Of course, in this example we supposed that the following lexical rule $X \rightarrow credit\, card$ is present in the lexicon.

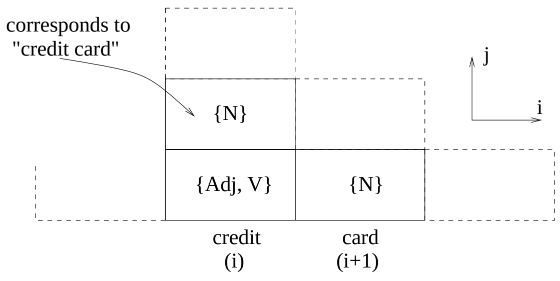


Figure 2.2: An example of initialization for a compound word ("credit card")

Word lattices : An example of word lattice representation is given in figure 2.3. Each path starting in the leftmost node and ending in the rightmost node of the word lattice corresponds to a possible recognised sentence. These sentences are subject to be filtered-out by the syntactic parser whenever they are recognized as incorrect.

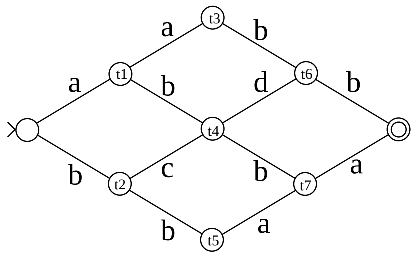
When dealing with word lattices, the difference with the previously presented CYK algorithm is that not only the sets $N_{i,1}$ may be initialised during the initialisation step, but any of the sets $N_{i,j}$ as with word lattice representation words may occur anywhere in the chart (see figure 2.4(b)). Therefore, in order to adapt the CYK algorithm to word lattice parsing, the initialisation step needs to be extended.

To illustrate this point, let us consider the same CNF grammar used in the example 2 and the word lattice given in figure 2.3, containing the six sentences "`a a b b`", "`a b d b`", "`a b b a`", "`b b a a`", "`b c b a`" and "`b c d b`". First the lattice nodes are ordered by increasing depth[1] (see figure 2.4 (a)). Such an ordering is natural in the case of lattices produced by a speech recogniser, in which nodes correspond to (chronologically ordered) time instants $t1, t2, \ldots$

This new representation of the word lattice can be mapped over the chart as follows:

1. the intervals between successive nodes are associated to the column indices of the parsing table;

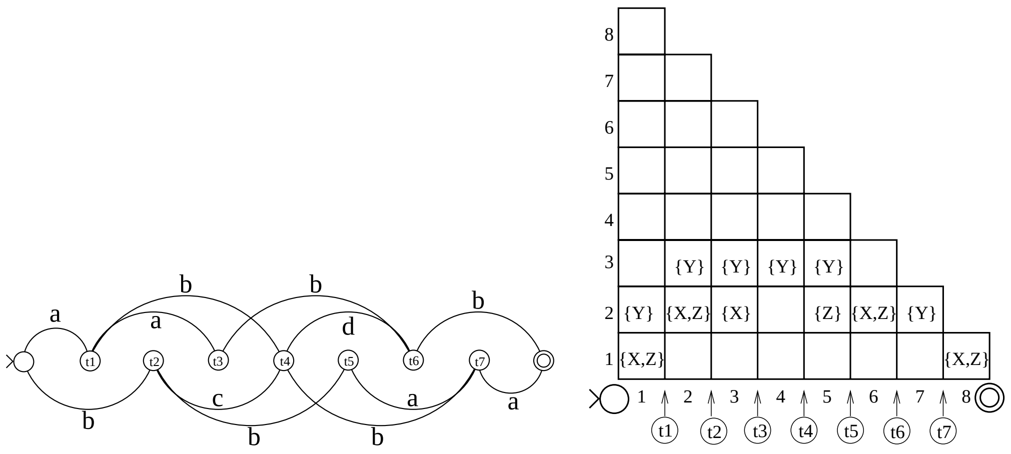
[1]A node depth is the minimal number of arcs from the initial node, with random choice in case of equalities.



a	a	b	b
a	b	d	b
a	b	b	a
b	b	a	a
b	c	b	a
b	c	d	b

Figure 2.3: A toy word lattice containing 6 sentences.

2. if the lattice arc (tm, tn) $(n > m)$ is labelled w, then the set $N_{m+1,n-m}$ corresponding to the chart entry $(m+1, n-m)$ is initialised with $\{X : (X \rightarrow w) \in R\}$ (see figure 2.4 (b)).



(a) (b)

Figure 2.4: (a) Word lattice of fig 2.3 represented as a speech word lattice (nodes are naturally ordered since they represent different time-instants). (b) The representation of the word lattice in the form of a initialized chart.

The time complexity required for the initialization of the chart with a word lattice is $O(n^2)$ while any entry in the chart is subject to initialization.

2.3 The enhanced-CYK algorithm adapted for word lattice parsing

The enhanced-CYK algorithm can be seen as derived either from a bottom-up Earley [11, 29] (with generalizes items and without prediction), or from a CYK [1] (but performing dynamic binarization of the grammar). It could also be seen as an extension of the algorithm presented by Graham et al. [13]. A CYK-like point of view is used subsequently for the description of the enhanced CYK algorithm.

The enhanced-CYK algorithm works with a subclass of CFGs, consisting of "non partially lexicalized rules", hereafter denoted as nplCFG. The restriction of the CYK algorithm to CNF grammars is thus relaxed to the use of nplCFG, a larger subclass of CFGs. A nplCFG is a

CFG in which terminal symbols, i.e. words, only occur in rules of the form $X \rightarrow w_1 w_2 ... w_n$[2], called lexical rules. In practice, such lexical rules do not even appear in the representation of the grammar but are represented in the lexicon.

The restriction to the nplCFGs subclass of CFGs is not however "critical" for the algorithm. It was introduced because it corresponds to the kind of grammars we are actually dealing with in practice, and also because it restricts the processing of "lexicalized rules" to the initialization step. The algorithm could however be easily extended to deal with any CFG.

The enhanced-CYK algorithm, like the CYK algorithm, uses a triangular table (hereafter called chart) with $n(n+1)/2$ cells, where n is the length of the input sentence $w_1 \dots w_n$. However, the elements stored in the chart are different of those used for the CYK algorithm and are a generalization of the Early-like items (the so called "dotted rules"). For the presentation of the enhanced-CYK algorithm we are using the same notations introduced for the CYK algorithm. The cell at row j and column i in the chart will contain two kind of sets of items:

- items of the first set (the $N1$ set) represent the non-terminals X that can derive w_{ij} (i.e. that can produce w_{ij} after the application of a finite number of grammar rules). Formally this set is a subset of N defined by:

$$N1_{i,j} = \{X \in N : X \Rightarrow^{\star} w_{ij}\}$$

- items of the second set (the $N2$ set) representing the partial parsings α of the string w_{ij} (i.e. sequences α of non-terminals such that $\alpha \Rightarrow^{\star} w_{ij}$). Formally this set is defined by:

$$N2_{i,j} = \{\alpha \bullet : \exists \beta \in N^{+}, \exists (X \rightarrow \alpha\beta) \in R, \text{ s.t. } (\alpha \Rightarrow^{\star} w_{ij})\}$$

The items in the $N2$ set represent a generalization of doted rules used in Early-like parsers, since only the beginning (i.e. parsed) part of the rule is represented, independently both of the left-hand side and of the end of the rule. This provides a much more compact representation of dotted rules (allowed by the bottom-up nature of the parsing, during which the left-hand side non-terminals can be ignored as long as they are not actually rewritten). With the notations given above, the enhanced-CYK algorithm is defined as follows: The lines 1-6 of the algorithm correspond to the initialization step. As defined above the initialization step can deal with compound words or/and word lattices. During the initialization step the N1 and N2 sets of each chart cell may be initialized (see lines 2-3 of the enhanced-CYK algorithm). Precisely, for each subsequence of words $w_i w_{i+1} \dots w_{i+j-1}$ in the input sentence, if there is a lexical rule $X \rightarrow w_i w_{i+1} \dots w_{i+j-1}$ in the grammar, then:

- the non-terminal(s) X are added to the $N1_{i,j}$ set;
- if the grammar contains a rule $Y \rightarrow X\beta$, and β is not empty then the partial right-hand side $Y\bullet$ is added to the $N2_{i,j}$ set;

The time complexity of the initialization step is thus $O(n^2)$.

The lines 7-14 of the algorithm correspond to the chart filling step. As defined above the enhanced-CYK algorithm can only deal with CFGs without unitary-rules. This restriction is required in order to render feasible the hardware implementation of the enhanced-CYK algorithm. The explanation is given in the technical note at the end of this chapter. For the enhanced-CYK algorithm defined above the following operations are applied when two source cells (i, k) and

[2]usually $n = 1$; $n > 1$ for compound words.

Algorithm 2 The enhanced-CYK algorithm

```
1:  for $j = 1$ to $n$ do
2:    for $i = 1$ to $n - j + 1$ do
3:      $N1_{i,j} = \{X : X \in N, (X \rightarrow w_{ij}) \in R\}$
4:      $N2_{i,j} = \{X\bullet : X \in N, \exists\beta \in N^{+}, \exists(Z \rightarrow X\beta) \in R,$ s.t. $X \rightarrow w_{ij}\}$
5:    end for
6:  end for
7:  for $j = 2$ to $n$ do
8:    for $i = 1$ to $n - j + 1$ do
9:      for $k = 1$ to $j - 1$ do
10:       $N1_{i,j} = N1_{i,j} \bigcup$
          $\{X : X \in N, \exists(\alpha\bullet \in N2_{i,k}), \exists(Y \in N1_{i+k,j-k})$ s.t. $(X \rightarrow \alpha Y) \in R\}$
11:       $N2_{i,j} = N2_{i,j} \bigcup$
          $\{\alpha X\bullet : \exists(\alpha\bullet \in N2_{i,k}), \exists(X \in N1_{i+k,j-k}), \exists(\beta \in N^{+})$ s.t. $\exists(Z \rightarrow \alpha X\beta) \in R\} \bigcup$
          $\{X\bullet : X \in N1_{i,j}, \exists\beta \in N^{+}$ s.t. $(Z \rightarrow X\beta) \in R\}$
12:     end for
13:   end for
14: end for
```

respectively $(i + k, j - k)$ are combined into the destination cell (i, j). For each pair of items $(\alpha\bullet, X)$, where $\alpha \in N2_{i,k}$ and $X \in N1_{i+k,j-k}$ if there is a rule $Y \rightarrow \alpha X\beta$ in the grammar, then:

- if β is not empty, the partial right-hand side αX is added to the $N2_{i,j}$ set;
- if β is empty, the non-terminal(s) Y are added to the $N1_{i,j}$ set and if there is a grammar rule whose right-hand side starts with Y then the partial right-hand side $Y\bullet$ is also inserted in the $N2_{i,j}$ set;

As there may be a grammar rule with an empty β and another grammar rule with a non empty β simultaneously in the grammar, both operations mentioned above may take place!

Example 2.3 Enhanced CYK parsing. Let us consider the following simple nplCFG:

$$\begin{gathered} N = \{S, X, Y, Z, V, W\}, \\ \Sigma = \{a, b, c, d\}, \\ R = \{S \rightarrow XW\ (r1),\ S \rightarrow XZVY\ (r2),\ W \rightarrow\ ZVY(r3), \\ X \rightarrow YZ\ (r4),\ Y \rightarrow b\ (r5),\ Z \rightarrow a\ (r6), \\ Z \rightarrow d\ (r7),\ V \rightarrow c\ (r8)\} \end{gathered} \qquad \diamondsuit$$

where S is the top level non-terminal and $r1, \ldots, r8$ are used to denote the grammar rules. With this grammar, we want to parse the sentence “`badcb`”. During the initialization step, only the rules $r5$ to $r8$ are used (see figure 2.5(a)) to fill-in the entries on the bottom row ($j = 1$) of the chart. During chart filling, the rows $j = 2, \ldots, 5$ are filled making use of the grammar rules $r1$ to $r4$. Finally, two possible parse trees are extracted from the chart (see figure 2.5(b)).

Figure 2.6 illustrates the initialized chart for the enhanced-CYK algorithm for the toy word lattice in figure 2.3.

Technical Note:

For the enhanced-CYK algorithm as described in [6] a "self-filling" procedure is executed on

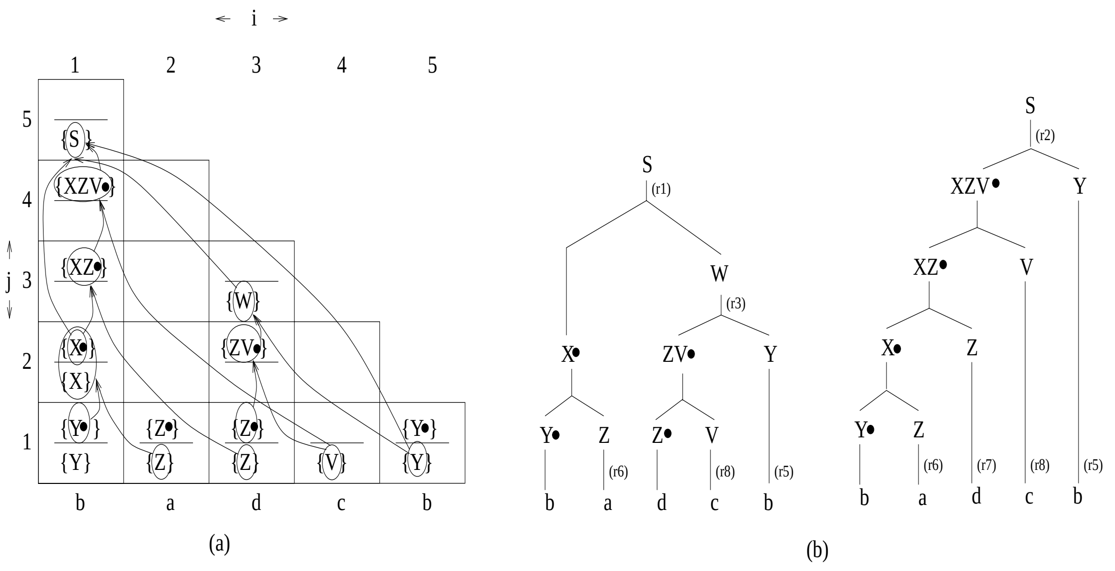


Figure 2.5: Enhanced-CYK (a) chart initialisation (j=1) and filling (j=2,3,4,5), (b) two possible parsing trees corresponding to the input sentence “`b a d c b`”

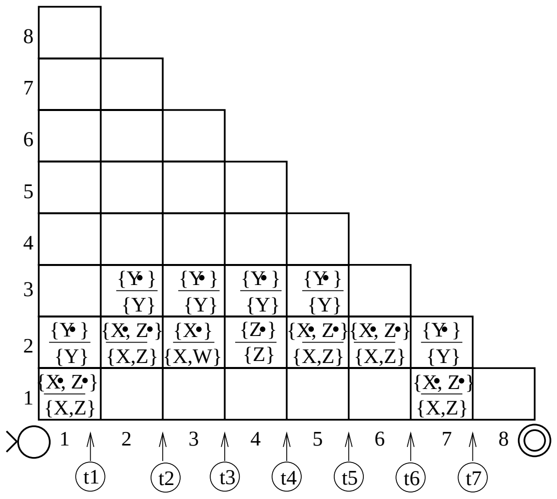


Figure 2.6: Example of word lattice initialisation for the enhanced-CYK algorithm

each chart cell once it is filled. It is required in order to deal with the unitary rules in the grammar. For each chart cell (i, j), it consists in taking each non-terminal X in the $N1_{i,j}$ set and if there exists a rule of the form $Y \rightarrow X\beta$ in the grammar two operations may take place:

- if β is empty (i.e. the unitary rule $Y \rightarrow X$ is in the grammar), the non-terminal(s) Y are added to the $N1_{i,j}$ set
- if β is non empty, the partial right-hand side $X\bullet$ is added to the $N2_{i,j}$ set

As there may be a grammar rule with an empty β and another grammar rule with a non empty β simultaneously in the grammar, both operations mentioned above may take place! The "self-filling" procedure is iterated until no non-terminal X in the $N1_{i,j}$ set will produce new elements according to the procedure described above. It is easy to remark the recursive nature of the "self-filling" procedure, which, if implemented in hardware requires stacks. The reason why the "self-filling" procedure is required are the unitary rules. In order to eliminate the "self-filling" procedure in a hardware implementation the unitary rules are eliminated in a preprocessing step. In the particular case of the SUSANNE grammar the grammar $G1$ is used (see appendix A.3).

2.4 Conclusions

This chapter presents the formalism used to describe CFGs and introduces the Chomsky Normal Form representation of the CFGs.

Next, it presents the CYK algorithm that is restricted to the use of CFGs written in Chomsky Normal Form, and its adaptation for parsing word lattices and compound words. The enhanced-CYK algorithm, which is an extension of the CYK algorithm, that is able to deal with a subclass of CFGs – larger then the Chomsky Normal Form grammars – called nplCFG ("non partially lexicalized context-free grammars") is next introduced. The adaptation of the enhanced-CYK to word lattice parsing is then presented.

The presented algorithms (and their adaptation to word lattice parsing) will be further implemented in the hardware. In a first step we will study the implementation of the CYK algorithm in hardware due to its (relative) simplicity and then based on the knowledge accumulated in the first step we will target a hardware implementation of the enhanced-CYK algorithm which is more complex.

Chapter 3

The Linear Array of Processors Hardware Design of the CYK Algorithm

This chapter starts with an argument for why a 2D-array of processors architecture as presented in [8] is not feasible when using state-of-the-art FPGAs and real-life CFGs. A linear array of processors architecture for the CYK algorithm adapted for word lattice parsing is then proposed.

The linear array of processors architecture fills the chart in a row-by-row manner and was the starting point for exploring the behaviour and characteristics of a hardware implementation for the CYK algorithm when dealing with large-size real-life grammars. The design was useful for several reasons:

1. for acquiring knowledge about the required FPGA resources, the size of the grammar memories (storing the grammar) and the chart memory (storing the chart);
2. for getting acquainted with the real-life behaviour of the CYK algorithm;
3. for having a first evaluation of the speed-up factor over a software implementation;

The first point was the most important at the time the project started, as it was a key requirement for such a design to be feasible. It was useful for defining the data-structures for representing the CNF grammars and the chart in an efficient way both for reducing the size of these data-structures and for reducing the required access time to the stored information. The second point was required to identify the potential bottlenecks in the system in order to pinpoint the critical regions of the design where the efforts for further improvements should be concentrated. The third point was necessary in order to evaluate the advantage of such a hardware implementation over a software implementation. Finally, the design was synthesised and tested for validation on a commercial FPGA-board (RC1000-PP) .

The three points mentioned above and the implementation of the design on the commercial FPGA board (RC1000-PP) was the first step of our design methodology, namely to prove the feasibility, correct functionality and speedup over an equivalent software implementation of the enhanced-CYK algorithm.

The main features of the linear array of processors design are: (1) a speed-up factor of about 16.01 over our best software implementation of the enhanced-CYK algorithm and (2) its ability to parse sentences with up to 26 words (see section 3.6) or time-stamps when dealing with word lattices. These features make the design an interesting solution for integration within real-life NLP applications frameworks that have strong data-size and/or real-time constraints.

3.1 Why not a 2D-array of processors architecture ?

A 2D-array of processors hardware architecture best exploits the parallelism available in the CYK algorithm. Such an architecture is proposed in [8]. The major drawback of this implementation, that makes it uninteresting in practice, resides in the large amount of resources it requires. The amount of memory required for storing the grammars grows with the number of non-terminals in the used CNF grammar, becoming impractical for implementation within a state-of-the-art FPGA even for a small number of non-terminals (e.g. 64 non-terminals). The following example illustrates in a concrete case-study the length of the sentence that can be parsed as a function of the number of non-terminals in the used CNF grammar.

Example 3.1 This example assumes that a Xilinx Virtex-E XCV2000 FPGA is used for implementing the 2D-array of processors architecture proposed in [8].

For a CNF grammar with 16 non-terminals, the FPGA has enough internal memory resources (see Xilinx's Virtex-E family manual [30]) for implementing a 2D-array design that can parse sentences with up to 20 words. For a CNF grammar with 32 non-terminals, the available memory resources are enough to parse sentences with up to 10 words and for 64 non-terminals sentences with up to 4 words! Such a sentence length and number of non-terminals is of no interest in real-life cases. ◇

The memory space required for real-life grammars such as the CNF SUSANNE grammar is large (more than 512 KBytes, see section 3.4.2) and cannot even be accommodated within state-of-the-art FPGA resources. The solution in this case is to use external memories for storing the grammars. The memory space required for the chart is also large (see section 3.4.1) within FPGA resources and an external memory is again required. On the other hand, if external memories are used for storing the chart and the grammar memories, the number of user pins available on a state-of-the-art FPGA becomes the limiting factor. The number of user pins available limits the number of grammar memories that can be connected to the FPGA. The following example illustrates for the 2D-array of processors architecture proposed in [8], the length of the sentence that can be parsed if external grammar and chart memories are used.

Example 3.2 This example assumes that a Xilinx Virtex-E XCV2000 FPGA is used for implementing the 2D-array of processors architecture proposed in [8]. The CNF SUSANNE grammar is used.

The memory space required to store the CNF SUSANNE grammar is more than 512 KBytes and therefore requires 20 address lines. Assuming that the data transfers take place on an 8-bit databus and that 3 command lines are used for interfacing with the memory – SRAM memory is used – a total number of 31 pins are required for the interface between the FPGA and a grammar memory. The XCV2000 FPGA has 804 user pins and therefore about 23 grammar memories can be attached (the remaining pins are used for interfacing the chart memory and for other purposes) which allows the 2D-array design to parse sentences of up to 7 words. If the data transfers take place on a 16-bit databus, then a total number of 39 pins are required for the interface between the FPGA and a grammar memory. A number of 18 grammar memories can be attached, which allows the 2D-array to parse sentences with up to 6 words. ◇

The above examples show that the 2D-array of processors presented in [8] cannot accommodate real-life CFGs if implemented in a state-of-the-art Virtex-E XCV2000 FPGA due to the limited available hardware resources.

3.2 General system description

Given the facts presented in the previous section we decided to design and implement a linear array of processors – henceforth referred as LAP – that requires $n-1$ processors to parse sentences with maximum n words or word lattices with maximum $n+1$ time-stamps. We understand by a processor, a processing element whose basic task is to perform (i.e. process) a cell-combination.

<u>Note</u>: A cell-combination is the operation described by line 10 of the CYK algorithm (see page 13) for a particular value of k. Each cell in the row j of the chart is filled by performing a number of $j-1$ cell-combinations.

The input to the LAP design is an initialized chart – and grammar lookup tables – and the output is a compact parse forest.

During the parsing, each processor requires access to the CNF grammar for performing the cell-combinations it is responsible for. While real-life CNF grammars require a large amount of storage memory the CNF grammar data-structure is currently stored in SRAMs (Static Random Access Memories) that are referred henceforth as grammar memories.

<u>Note</u>: The SRAMs have several advantages: are easy to control and require a minimum of interfacing signals with the FPGA, state-of-the-art SRAM chips have relatively large sizes and fast access times. A disadvantage would be the high power consumption.

Ideally, each processor should have its own grammar memory, such that all processors can process the assigned cell-combinations in parallel. However, as the number of pins available for an FPGA is limited, only a limited number of grammar memories can be connected to the FPGA being available to the processors. Therefore, several processors have to share the same grammar memory and the processors sharing the same grammar memory form a cluster. The way the processors are clustered around a grammar memory is arbitrary and a priori a solution that aims to equilibrate the number of interprocessor collisions to the grammar memories – during runtime – over all the clusters is preferable. All the processors in the system implement a procedure for accessing the CNF grammar data-structure stored in a grammar memory.

The chart data-structure is also stored in a SRAM, referred henceforth as the chart memory and that is shared by all the processors in the system. Again, all the processors in the system implement a procedure for accessing the chart data-structure stored in the chart memory in order to fetch the sets required for performing the cell-combinations they are assigned to compute.

Each processor of the LAP design is (statically) assigned to a column of the chart and therefore only performs cell-combinations for filling the cells on this column. The LAP design executes line 8 of the CYK algorithm (see page 13) in parallel. Concretely, assuming that the sentence length is n, for each row j in the chart the columns $i = \overline{1..n-j+1}$ are filled in parallel and the cell (i,j) on column i is filled by the processor P_i. When the topmost cell of a column is filled the processor associated with that column becomes idle for the rest of the parsing process. Before starting to fill the cells in a new row all the processors should finish to process (i.e. fill) the cell in the current row in order to guaranty that all the data required to process the new row is available. For this reason the processors are synchronized after each filled row (corresponds to line 7 of the CYK algorithm, see page 13).

The block diagram in figure 3.1 depicts a system consisting of 10 processors that can parse any sentence of length less or equal to 11 words. The 10-processor system was synthesised and physically tested (i.e. the results were validated for correctness) on a commercial RC1000-PP FPGA board containing a Xilinx Virtex XCV1000bg560-4 FPGA.

Note: from now on we will use sentence to mean both sentence and word lattice. Depending on the context, when speeching about sentence parsing we also mean word lattice parsing and when speeching about sentence length we also mean number of nodes (i.e. time-stamps) in a word lattice ...

In the figure, the elements inside the dashed line are implemented within the on-board FPGA chip. The other elements (chart and grammar memories) are implemented in SRAM chips available on the board. As the RC1000-PP FPGA board contains 4 SRAM chips, 1 is used for storing the chart while the remaining 3 are used for storing identical copies of the data structure representing the CNF grammar. The 10 processors are assigned to 3 clusters: processors $\{P_1, P_6, P_{10}\}$ to cluster 1, processors $\{P_2, P_5, P_7\}$ to cluster 2 and the processors $\{P_3, P_4, P_8, P_9\}$ to cluster 3.

3.3 Functional Description

A LAP system with n processors can only parse sentences with up to $n + 1$ words but not longer. Therefore, the maximum sentence length to be parsed has to be known a priori, such that the appropriate number of processors are integrated within the LAP design. Based on the length of the sentence to be parsed, the module (e.g. on-board processor) that initializes the chart memory decides whether the parsing will be done in the hardware – if the sentence length has no more than $n + 1$ words – or by the software.

If a LAP design with n processors is available and the sentence we want to parse has no more than $n + 1$ words, the following initializations are necessary before the parsing can start:

- grammar memories : before any sentence can be parsed, each of the grammar memories have to be configured with the binary image of the data structure representing the CNF grammar (for details see section 3.4.2). The initialization of the grammar memories is done by some software running on the on-board processor or on the host system. As the grammar data structure does not change during the parsing, the grammar memories only have to be configured once for multiple parsings with the same grammar. Only if a different grammar has to be used the grammar memories have to be reconfigured;

- the chart memory : the initialization of the chart memory is done by some software running on the on-board processor or on the host system. It consists of initializing certain cells (i, j) of the chart with sets of non-terminals $N_{i,j}$ along with their corresponding guard-vectors (see section3.4.1).

- sentence length : the global controller (IO-CTRL) is initialised before every parsing with the length of the sentence to be parsed. Based on the sentence length the IO-CTRL gen-

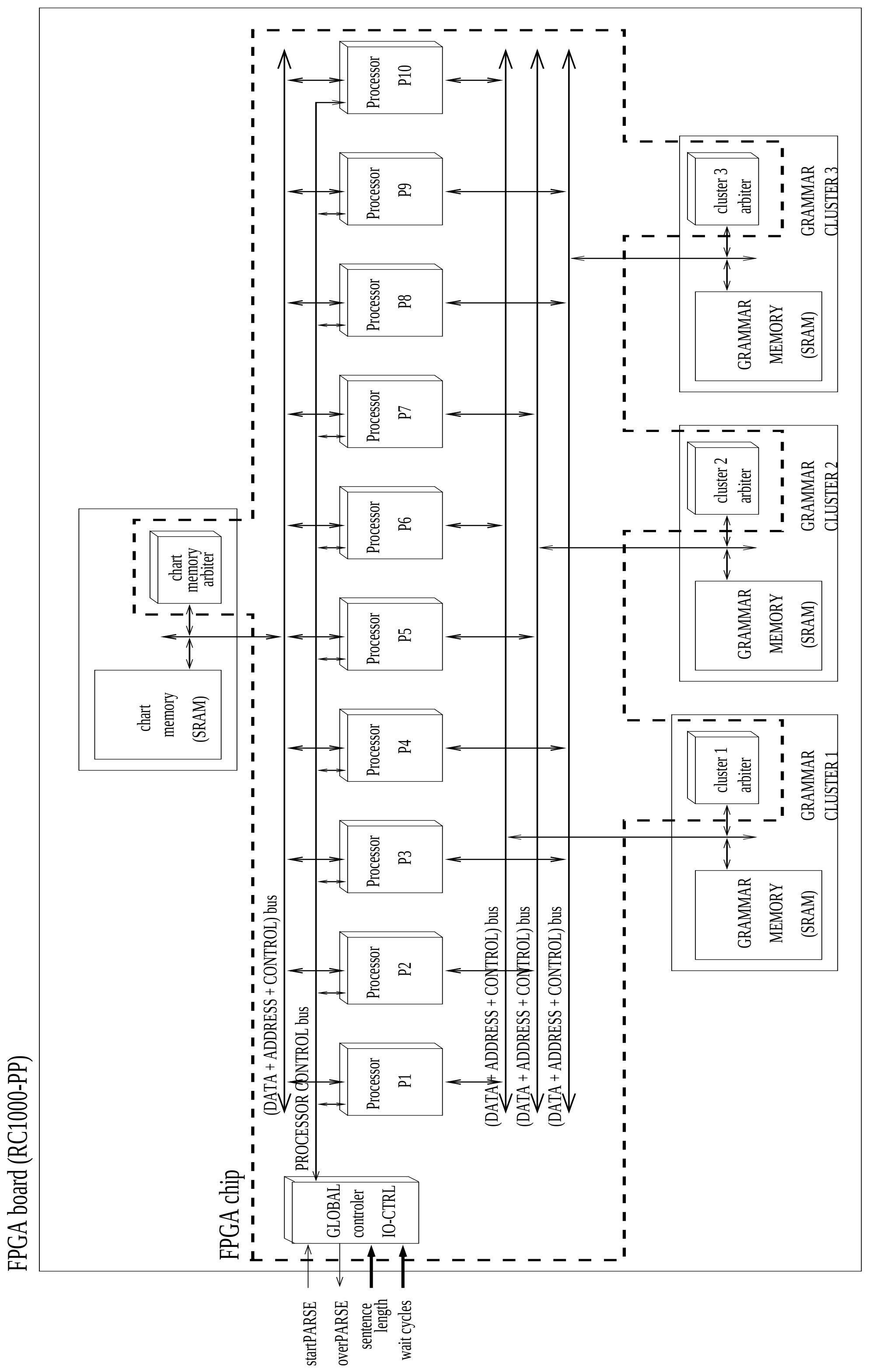


Figure 3.1: A 10-processor LAP design (implemented and tested on the RC1000-PP FPGA board) for parsing any sentence of length up to 11 words.

erates the signals that activate, respectively deactivate the processors during the parsing process. A register is used for configuring the sentence length.

- wait cycles : due to the fact that the chart memory is accessed by a relatively large number of processors the logic required to access this memory may have a large propagation time (i.e. delay) and in consequence the chart memory access cycles may be long. In order to keep the rest of the system frequency high, wait cycles are introduced in the chart memory access cycles. As the number of necessary wait cycles cannot be foreseen from the beginning (they actually depend on the way the signals are routed, the number of processors and other factors), we use a register for configuring the number of wait cycles. A first estimate for the number of wait cycles required to access the chart memory is known after the design is synthesized. Within the current design a number of 1 to 8 wait cycles can be pre-configured (default value) in the VHDL code with the generic `CYK_WAIT_CYCLES`.

Among these initializations, the initialization of the chart memory is the most time consuming an may represent and important amount in the overall parsing time.

Once the system was initialized – according to the procedure described above – the parsing starts as soon as the signal `startPARSE` is activated. At this point the processors start to fill the cells in row 2 of the chart. As soon as all the processors finish to process the cells in row 2 (i.e. synchronize) they start to fill the cells in row 3 and so on. The processors are synchronized under the control of the global controller IO-CTRL unit. Note, that each time a row is filled the processor assigned to the rightmost column will become idle for the rest of the parsing process. As soon as the topmost cell of the chart is filled the signal `overPARSE` is activated to mark the end of the parsing process.

The parsing results are available at some output `outPARSE` (not represented in figure 3.1) and can be collected during runtime for building the compact parsing forest.

3.4 The CYK algorithm data-structures

3.4.1 The chart data-structure memory representation

This section discusses the chart data-structure used for the hardware implementation of the CYK algorithm. The factors that influence the chart data-structure organization are also discussed.

There are several data-structures that can be used for the chart memory representation and when choosing one data-structure or another, a good compromise has to be found between: (1) the required memory space, (2) data access time and (3) data access circuit complexity. It turns out that, these parameters are highly dependent. For instance, the hardware for data access depends on the data-structure chosen for the chart memory representation. The more complex the data-structure the more complex becomes the hardware required to access it, which also results in increased access times. On the other hand a complex (i.e. compact) data-structure can significantly reduce the memory space used for the chart representation. Before trying to define the chart data-structure reprezentation let's see which are the functionalities it has to support. In order to discuss these functionalities, let's consider that the source cells (i, k) and $(i + k, j - k)$ have to be combined and stored in the destination cell (i, j) (see the CYK algorithm at page 13). The required functionalities are the following:

- $F1$: go through all elements of a set. This is required in order to pair each non-terminal of the set $N_{i,k}$, corresponding to cell (i, k) with each non-terminal of set $N_{i+k,j-k}$ corresponding to cell $(i + k, j - k)$.
- $F2$: is an element in a set ? This functionality is required for testing the presence of a non-terminal X in the set $N_{i,j}$. Required, in order to store only once a non-terminal X in the set $N_{i,j}$ of the destination cell (i, j).
- $F3$: insert an element in a set. Required in order to store a non-terminal X in the set $N_{i,j}$ of the destination cell (i, j).

Let's take a look at a case-study of chart data-structure memory representation .

Case-study: We assume that the CNF SUSANNE grammar is used. The chart data-structure considered in this case-study was used in early versions of the current design but proved inefficient[1]. However, it is a good starting point in discussing the issues related to the chart data-structure memory representation.

We know that every cell (i, j) in the chart contains a set $N_{i,j}$ of non-terminals. The set $N_{i,j}$ is a subset of N of the considered grammar (see the CFG definition in section 2.1). Therefore, we can represent $N_{i,j}$ as a bit-vector of length $|N|$, where $|N|$ stands for the cardinal of N. Each bit of this bit-vector corresponds to a distinct non-terminal and has value '1' if the non-terminal is present in the set $N_{i,j}$, or '0' otherwise. Therefore, we need $\lceil |N|/8 \rceil$ bytes to store a cell in the chart memory with each byte storing 8 non-terminals.

If we consider the CNF SUSANNE grammar that has $|N| = 10,129$ non-terminals (see appendix A.3), we require a number of $1'267$ bytes to represent as a bit-vector the non-terminals in a cell. If we allocate a power of 2 memory size for a cell with the above data-structure, the cell address can be easily computed and requires simple access circuitry. More precisely, a memory size of $2^{\lceil log2(|N|/8) \rceil}$ (e.g. $2,048$ bytes when using CNF SUSANNE grammar) allows to retrieve the location of a cell (i, j) in memory by using inexpensive shift operations. With such a data-structure a chart memory size of 1 Mbyte can parse any sentence with up to 32 words and a memory size of 16 Mbyte any sentence with up to 128 words. However, with such a data-structure the functionality $F1$ is not well supported. In order to combine two cells a processor needs to fetch from memory $(\lceil |N|/8 \rceil)^2$ bytes (e.g $1'267 * 1'267$ bytes in the particular case of the CNF SUSANNE grammar) in order to form all pairs of non-terminals for grammar look-up. This takes significant time and causes many collisions between the processors accessing the chart due to the large number of accesses.

The previous case-study shows that the reprezentation of the set of non-terminals in a cell as a bit-vector is fast, requires a reasonable amount of memory and simple access hardware but does not well support the $F1$ functionality. A list representation of the non-terminal set in a cell is an alternative solution. However, when using lists, the search for a particular non-terminal in order to only store distinct non-terminals (in the destination cell), the $F2$ functionality, is not well supported. A solution is to use both a list and a bit-vector for representing a cell in the chart. With such a data-structure, the lists will be used to sequentially access the elements in the source cells, to support the $F1$ functionality, while the bit-vectors will be used to check for the

[1]It only works for CNF grammars with no more than several hundreds of non-terminals.

presence of non-terminals in the list, to support the $F2$ functionality. Such a data-structure has a large redundancy, as both the bit-vector and the list in a cell represent the same information, namely the non-terminals in that cell.

Like the bit-vectors, the lists represent subsets of the set N. Each list requires an amount of memory proportional to $|N|$, more precisely $2|N|$ byte if we assume that each non-terminal is represented on 16 bit. In the particular case of the CNF SUSANNE grammar the memory required is $2 * 10,129 = 20,258$ byte.

On the other hand, we know that to allocate for each set $N_{i,j}$ an amount of memory proportional to $|N|$ would represent an important memory waste (see appendix B.3), as in practice $|N_{i,j}| \ll |N|$. Therefore, in order to reduce the size of the chart memory, we assume that any set $N_{i,j}$ contains a maximum of K non-terminals. During run-time, if a cell receives more than K distinct non-terminals, the hardware generates a fault signal and the parsing stops. This would be, however, a very unlikely event for a well chosen value of K and we can thus use tables of size proportional to K to store the non-terminals of the sets $N_{i,j}$. The value of K depends on the considered grammar and can be found by investigating the characteristics of the grammar. For the CNF SUSANNE grammar (see section B.3) we have a value of $K = 256$ (much smaller when compared to $|N| = 10,129$), which requires a chart memory size of 1 Mbyte for parsing any sentence with up to 32 words and a memory size of 15 Mbyte for a 128 words sentence.

For understanding how the $F1$, $F2$ and $F3$ functionalities are implemented, the chart data-structure organization is given in figure 3.2. The chart memory storing the chart data-structure is addressed on 4 byte words. Also, in order to efficiently access the data stored in the chart memory, the content of the data-structure is aligned to a 4-byte boundary. The chart data-structure is

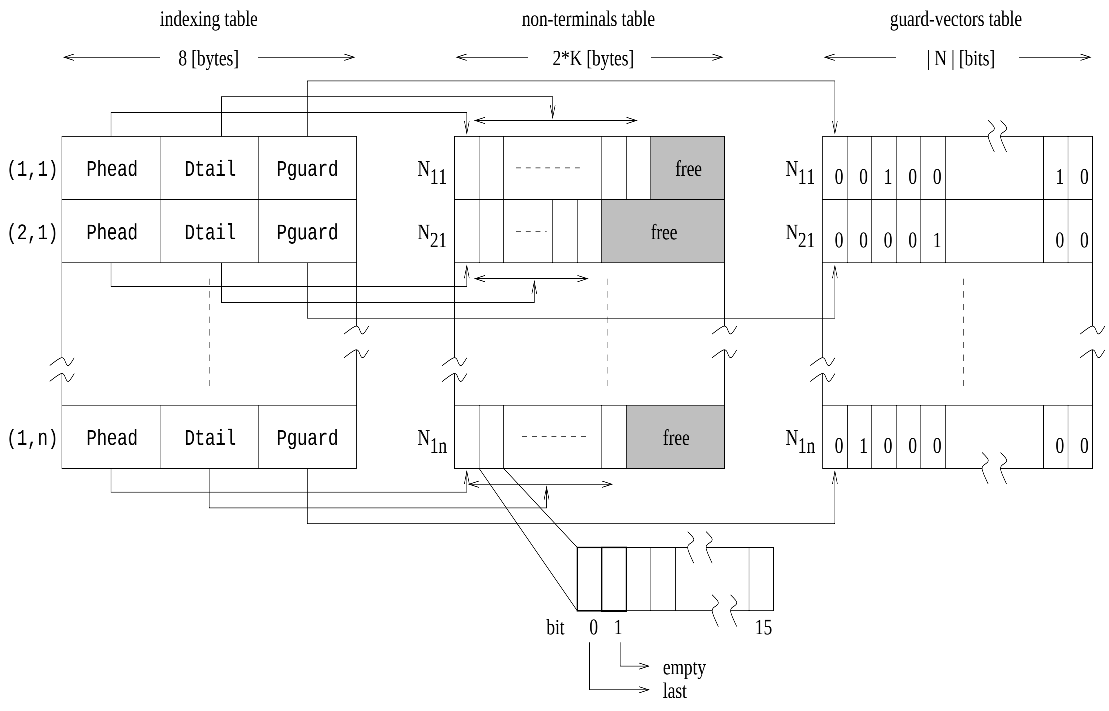


Figure 3.2: CYK chart data-structure organization

organized in memory as three distinct tables and is defined by some parameters whose values depend on the used CNF grammar. These parameters, their maximal allowed size and their

parameter	maximal allowed size	actual size	meaning
`NT`	2^{14}	$10,129$	total # of non-terminals
`NT_SIZE`	14 [bits]	14 [bits]	size of a non-terminal
`CYK_ADR_SIZE`	32-DTAIL_SIZE [bits]	19 [bits]	size of a chart memory pointer
`DTAIL_SIZE`	32 [bits]	7 [bits]	bits used to represent the constant K

Table 3.1: The maximal allowed size for the parameters that define the CYK chart data-structure. Parameter sizes in the particular case of the CNF SUSANNE grammar.

actual size in the particular case of the CNF SUSANNE grammar are tabulated in table 3.1. A CNF grammar for which any of these parameters requires a size larger than its corresponding maximal allowed size cannot be accommodated within the chart data-structure. These parameters are used in order to correctly access the chart data-structure by (1) the software used to initialise the chart data-structure and (2) for configuring the VHDL code (i.e. using generics) used to synthesize the design.

The first table in figure 3.2, called indexing table, contains for each chart cell (i, j) an entry that allows to retrieve: (1) a pointer `Phead` to the memory location where the list representation for the set $N_{i,j}$ is stored, (2) a pointer `Pguard` to the memory location where the guard-vector for the set $N_{i,j}$ is stored and (3) a value `Dtail`, taking values from 0 to K, representing the number of elements in the set $N_{i,j}$. Each entry in the indexing table is stored on 8 bytes and is organized as illustrated in figure 3.3. The physical address of an entry in the indexing

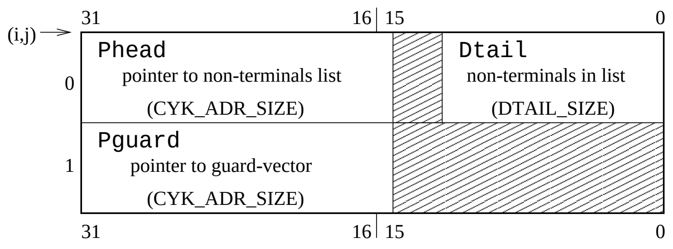


Figure 3.3: The organization of an entry in the indexing table of the CYK chart data-structure.

table where the information about the cell (i, j) resides is built by concatenating the binary representation of i and j and performing a left-shift with 3 positions (8 byte aligned). The indexing table is 8 KBytes for parsing sentences with up to 32 words or 128 KBytes for parsing sentences with up to 128 words.

The second table called the non-terminal table contains for each chart cell (i, j) an entry storing the list representations for the set $N_{i,j}$. While a non-terminal is stored on 2 bytes, each entry in the non-terminal table requires $2 * K$ bytes. In the particular case of the CNF SUSANNE grammar the size of an entry in the non-terminal table is 512 bytes and the size of the non-terminal table is 256 KBytes for parsing sentences with up to 32 words or 4 MBytes for parsing sentences with up to 128 words.

The third table called the guard-vectors table contains for each chart cell (i, j) an entry storing the guard-vector associated to set $N_{i,j}$. An entry in the guard-vectors table requires $\lceil |N|/8 \rceil$ bytes. In the particular case of the CNF SUSANNE grammar the size of an entry

last	empty	meaning
0	0	the list is not empty and the non-terminal is not the last
0	1	the list is empty
1	0	the list is not empty and the non-terminal is the last
1	1	not used

Table 3.2: last/empty bits meaning

in the guard-vectors table is $1,267$ bytes and the size of the guard-vectors table is 654 KByte for parsing sentences with up to 32 words and 9.97 MBytes for parsing sentences with up to 128 words.

Let's now look at how the functionalities $F1$, $F2$ and $F3$ are implemented. For the implementation of $F1$, the `Phead` pointer is used as a base address that points to the first non-terminal in the non-terminals table. A displacement, i.e. index, is used for going through all the non-terminals in the non-terminals table. The addition of the base memory address and the displacement gives the physical memory location of the addressed item. The unit implementing $F1$, needs to know whether the table is empty or not, and, in the later case, to know which is the last non-terminal in the table. This is implemented by means of two special bits attached to each non-terminal in the table. One bit is set when the table is empty, the other when the non-terminal is the last in the table (see figure 3.2 and table 3.2) and each time a non-terminal is fetched the last and empty bits are checked. While 2 bits out of 16 representing the non-terminal are used for special purposes, only 14 are used to code the non-terminal and for this reason only grammars having less than $2^{14} = 16,384$ non-terminals can be considered.

The function $F2$ would be highly time consuming if we search X over the set $N_{i,j}$ every time. To have an efficient implementation for this function, a guard-vector (i.e. bit-vector) is used. If the bit associated with X is set in the guard-vector then X is already in $N_{i,j}$. This gives constant time for answering $F2$ while only few chart memory accesses are enough to check the bit associated to X in the cell (i,j). For the implementation of $F2$ the `Pguard` pointer is used as a base address that points to the beginning of the guard-vector (organized in a 4-byte words). The displacement for addressing the bit associated to a non-terminal X is given by the binary representation of X. The physical memory address where the bit associated to a non-terminal X is located is obtained by adding the base address and the displacement.

Finally, for the implementation of $F3$, both `Phead` and `Dtail` are used to point the beginning of the non-terminals table, and, respectively, to index the last non-terminal. The physical memory address where the next non-terminal is to be stored is obtained by adding the base address `Phead` to the displacement `Dtail`. Each time a non-terminal is stored in memory the displacement `Dtail` is incremented to reflect the new set size. If during the parsing `Dtail` $> K$, a fault signal is raised to signal that the $N_{i,j}$ has too many non-terminals. In this case the parsing will stop and a signal should tell the host computer (or local microprocessor) that the current parse could not finish and that the software has to redo it.

When parsing sentences, `Dtail` is always 0, and is not used since every set $N_{i,j}$, with $i \geq 2$, is empty when the parsing starts. However, for word lattice parsing the sets $N_{i,j}$ of the destination cells are not necessarily empty when the parsing starts and the `Dtail` is necessary, while the insertion of new non-terminals has to be made at the end of the non-terminals table where the pointer `Phead` $+$ `Dtail` points .

3.4.2 The CNF grammar data-structure memory representation

This section discusses the CNF grammar data-structure used in the hardware implementation of the CYK algorithm. The factors that influence the selection of the grammar data-structure are also discussed.

The grammar memories are used for storing the CNF grammar data-structure and are accessed by the processors during the parsing for grammar rules lookup. Given that real-life CNF grammars data-structures require a large amount of storage memory, SRAM chips are used for this purpose. The SRAMs have several advantages: (1) are easy to control, (2) state-of-the-art chips have relatively large memory sizes, (3) have fast access times and (4) require a minimum of interfacing signals with the FPGAs. The disadvantage is the high power (i.e. current) consumption. Before being used the grammar memories have to be loaded with the CNF grammar data-structure. The content of the grammar memories can be changed whenever a new CNF grammar is to be used. If there are more processors in the design than available grammar memories, it is possible to cluster several processors around each grammar memory. When clusters are used, the processors have to be arbitered when accessing the cluster's grammar memory.

Precisely, the purpose of a grammar memory is to allow the processor, to retrieve for a given rule right-hand side YZ :

1. all non-terminals X_i for which there is a rule $X_i \to YZ$ in the CNF grammar;
2. a code `RHScode` that uniquely identifies the given rule right-hand side;

As the processors heavily rely on CNF grammar look-up during parsing, an important attribute of the grammar lookup unit is to allow a fast access to the CNF grammar data-structure. However, a simple (i.e. fast) hardware for accessing the grammar memory may require a large memory space. On the other hand, a compact data-structure, requiring a small memory space, also requires complex (i.e. slow) hardware for accessing this data-structure. Therefore, a compromise has to be found between (1) the memory space required for the CNF grammar representation, (2) access circuit complexity and (3) data access time.

Let's start looking for such a compromise in a case-study on the CNF SUSANNE grammar. We consider a data-structure that allows a very fast access to the stored information.

Case-study: In CNF every grammar rule[2] is of the form $X_i \to YZ$. The rule right-hand side YZ can be any pair of non-terminals and for each such rule right-hand side we can have any set of non-terminals X_i in the left-hand side.

A data-structure that allows a fast access (i.e. retrieval) of all the left-hand side non-terminals X_i corresponding to a given rule right-hand side YZ would be a table indexed by the rule right-hand side and storing on each table entry (of size proportional to $|N|$) sets of non-terminals (i.e. the left-hand sides). The advantage of such a data-structure is that the physical memory address for retrieving the set of left-hand side non-terminals can be constructed from the binary representations of the two non-terminals (i.e. Y and Z) in the rule right-hand side.

Concretely, for the SUSANNE CNF grammar we have $|N| = 10,129$ non-terminals and each non-terminal is stored on 2 bytes. For having an easy way of computing the physical memory address to the set of left-hand side non-terminals the amount of memory required for a set is $2^{\lceil log_2 2*|N|\rceil}$. The physical memory address to a set is then computed by concatenating the binary representations of Y and Z. There are $|N|^2$ such sets stored

[2]The lexical rules $X \to a$ are stored in a lexicon and are only used for initializing the chart memory.

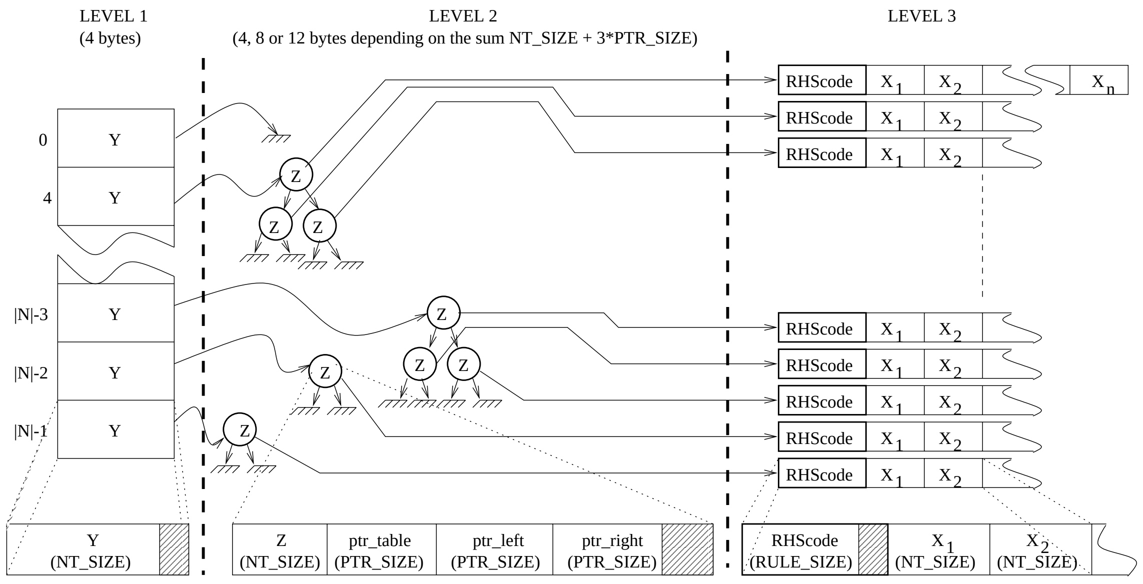


Figure 3.4: The CNF grammar data-structure. Levels of representation.

in the table and by making a simple calculus we see that the amount of memory required for storing such a data-structure – in the particular case of the SUSANNE CNF grammar – is 2^{42} bytes which is too large for being practical.

An improvement to the above data-structure would be to have in the table instead of sets of non-terminals, lists of non-terminals and a mechanism for computing the physical memory address pointing to these lists. Such a mechanism can be a table indexed by the binary representations of Y and Z (like above) for retrieving a pointer to the associated list. If such a pointer is represented on 4 bytes the amount of memory required to store the data-structure is about 2^{30} bytes. The improvement comes from the fact that the sets of left-hand sides represented as lists of variable length required – in the particular case of the SUSANNE CNF grammar – only the insignificant amount of memory, namely $211,374$ bytes. Unfortunately, even the 2^{30} bytes amount of memory is impractical.

The first data-structure in the above case-study is characterised by a very fast access time to the stored information. However, the memory space required is to large for being practical. The second data-structure, which is a refinement of the first studied data-structure, is a better solution, but however still unacceptable due to the required memory space.

The CNF grammar data-structure we propose is depicted in figure 3.4, and is organized on 3 levels. The grammar memories storing the CNF grammar data-structure are addressed on 4 byte words and therefore for efficiently accessing the stored data the proposed data-structure is aligned to a 4 byte boundary. The CNF grammar data-structure is defined by some parameters whose values depend on the used CNF grammar. These parameters, their maximal allowed size and their actual size in the particular case of the CNF SUSANNE grammar are tabulated in table 3.3 (for more details regarding these values see appendix A.4). A CNF grammar for which any of these parameters requires a size larger than its maximal allowed size cannot be accommodated within the proposed CNF grammar data-structure. Level 1 is a table with an entry for each distinct non-terminal present in the grammar. Such an entry contains either a NULL pointer if there is no right-hand side starting with the corresponding non-terminal Y or

parameter	maximal size [bit]	SUSANNE size	meaning
`NT_SIZE`	14	14	bits for a non-terminal
`PTR_SIZE`	26	19	bits for a pointer (byte address)
`RULE_SIZE`	32	15	bits for a rule right-hand side

Table 3.3: The maximal size of the parameters that define the CNF grammar data-structure. Parameter sizes in the particular case of the SUSANNE grammar.

a non-NULL pointer pointing to the root of a sorted binary-tree at level 2. The number of bits required to represent a pointer is given by the parameter `PTR_SIZE` (see figure 3.4 and table 3.3). Each entry in this table has a fixed size of 4 bytes even if less are enough to represent a pointer given the final size of the required grammar memory. This allows to construct the index for indexing the table from the binary representation of the Y. Precisely, the binary representations of the non-terminal Y is shift left with 2 positions for obtaining the index in the table. In the particular case of the CNF SUSANNE grammar, there are $10,129$ non-terminals and therefore the level 1 table has a size of $40,516$ bytes.

Level 2 is a collection of sorted binary-trees. Each sorted binary-tree corresponds to a certain Y and contains in its nodes all the non-terminals Z for which there is a rule having the right-hand side of form YZ. The fields of a node in a sorted binary tree are:

- ▷ `value` : a non-terminal Z;
- ▷ `ptr_table` : pointer to a table that contains a list of all left hand-sides for the rules having as right hand-side YZ;
- ▷ `ptr_left` : pointer to the left subtree;
- ▷ `ptr_right` : pointer to the right subtree;

The number of bits required to represent a non-terminal Z is given by the parameter `NT_SIZE`, and for representing the pointers `ptr_table`, `ptr_left` and `ptr_right` by the parameter `PTR_SIZE`. Each node in a sorted binary tree is represented on 4, 8 or 12 bytes, depending on the characteristics of the used CNF grammar. For instance, in the particular case of the CNF SUSANNE grammar the size of a node is $\texttt{NT\_SIZE} + 3 * \texttt{PTR\_SIZE} = 71$ [bits] and therefore 12 bytes are used. The amount of memory required for level 2 is in this case $111,820$ bytes.

Searching for a certain Z in the sorted binary-tree is done in the traditional way. We take the current node – we start with the root – look whether the `value` in the node is equal to the Z and if this is the case we found the node. If the `value` in the node is greater we continue searching with the left subtree pointed by `ptr_left` , otherwise with the right subtree pointed by `ptr_right`. If we have to search the left subtree and the `ptr_left` is NULL or we have to search the right subtree and `ptr_right` is NULL, we stop because the right hand-side we are looking-up does not exist in the grammar.

If we found a node for which `value` equals Z we go to level 3 of the data-structure to a table storing the required information. This table is pointed by `ptr_table` and starts with a header (always stored on 4 bytes) containing a code `RHScode` that uniquely identifies the rule right hand-side and the non-terminals X_1, X_2, $\ldots$, corresponding to this rule right hand-side in no special order. The last non-terminal in the list is marked by a flag. Recall that a non-terminal is represented in the chart on 14 bits (stored on 2 bytes) and therefore one of the remaining two

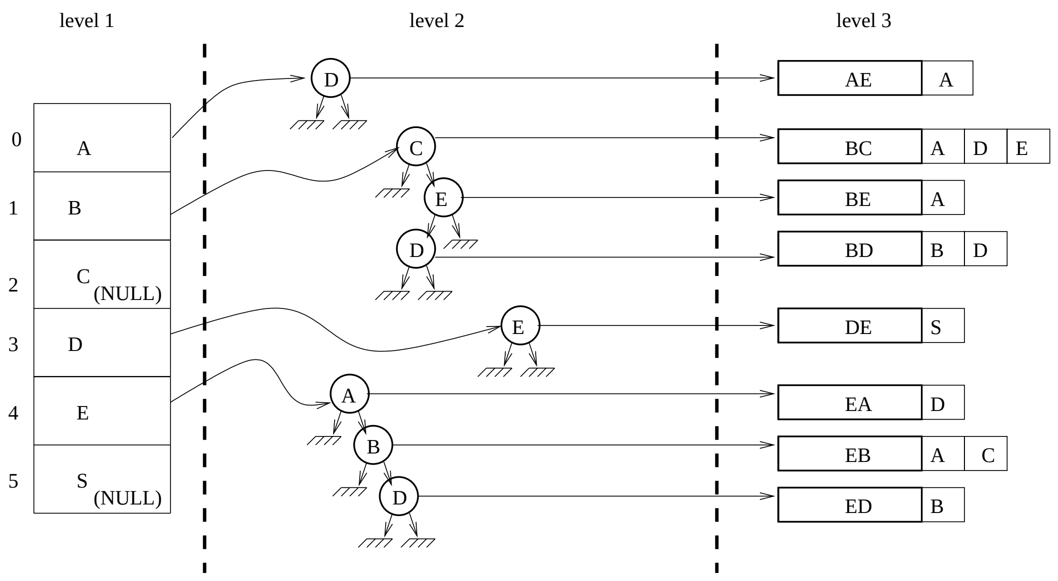


Figure 3.5: The CNF grammar data-structure representing the toy CNF grammar in example 3.3.

bits can be used for the flag that marks the last non-terminal in the list. Together with a non-terminal the `RHScode` uniquely identifies a grammar rule. The size in bits of `RHScode` is given by the parameter `RULE_SIZE` and for the non-terminals $X_1, X_2, \ldots$, by the parameter `NT_SIZE` (see figure 3.4 and table 3.3). In the particular case of the CNF SUSANNE grammar, the level 3 table has a size of $291,292$ bytes.

For better understanding the CNF grammar data-structure an example is given bellow for a toy CNF grammar.

Example 3.3 The considered CNF grammar is given by:

$$\begin{gathered} N = \{A, B, C, D, E, S\}, \\ \Sigma = \{\ldots undefined \ldots\}, \\ R = \{A \to BC \ , \ D \to BC \ , \ E \to BC \ , \\ B \to BD \ , \ C \to BD \ , \ A \to BE \ , \\ A \to EB \ , \ C \to EB \ , \ A \to AD \ , \\ D \to EA \ , \ B \to ED \ , \ S \to DE \, \} \end{gathered}$$

The data-structure representing the above CNF grammar is illustrated in figure 3.5. Each node in level 2 is represented on 4 bytes and the amount of memory required to store this data-structure is 124 bytes. $\diamondsuit$

Some FPGAs of the Xilinx Virtex family (e.g. XCV1000, XCV2000) have enough memory resources for storing the data-structure representing the toy CNF grammar considered in the example above. In this particular case SRAMs are not required and can be replaced by such internal memory resources.

For the CNF SUSANNE grammar the memory size required for this representation is of $558,576$ bytes. For details regarding the creation of the binary memory image of the CNF grammar refer to appendix A.4.

3.5 Design units

3.5.1 The processor

The datapath of a processor used in the linear array of processors architecture is depicted in figure 3.6. Within the linear array of processors architecture, the processors are synchronously working on the same clock signal and are supervised by the global controller IO-CTRL (see figure 3.1). Their main task is to compute (i.e. fill) new cells in the chart based on previously computed cells. The new cells are obtained by performing a series of cell-combinations. For describing how a cell-combination is performed, let's assume that we have two source cells $S1$ and $S2$ to be combined and a destination cell D. A cell-combination is performed as follows: the processor pairs each non-terminal $RHS1$ in the source cell $S1$ with each non-terminal $RHS2$ in the source cell $S2$ (the order is important) and checks whether a grammar rule having the right-hand side $RHS1$ $RHS2$ exists in the CNF grammar. If such a right hand-side exists, all the non-terminals X_i for which there is a rule $X_i \rightarrow RHS1\ RHS2$ in the grammar are added to the destination cell D in the chart such that every distinct non-terminal in the set D occurs once. This is particularly important as a processor performs several cell-combinations in order to fill a destination cell D and a certain non-terminal may occur several times but only stored once in the destination cell D. The procedure continues until all the non-terminals in the input sets $S1$ and $S2$ have been paired.

Note: The fact that each distinct non-terminal is stored once in a cell is strictly related to the set representation of a chart cell. Obviously, for building the compact parse forest all the occurences of a non-terminal in a cell are relevant. However, as shown in [15] it is enough to store the unique occurences of the non-terminals in the chart cells while shipping on-line (i.e. during the parsing) the information necessary to rebuild the compact forest.

Precisely, the procedure described above corresponds to line 10 in the CYK algorithm (see page 13) where the cell $S1$ corresponds to set $N_{i,k}$, the cell $S2$ to set $N_{i+k,j-k}$ and the cell D to set $N_{i,j}$. Also, during iteration j ($2 \leq j \leq l$), where l ($2 \leq l \leq n+1$) is the length of the input sentence, all processors P_i ($1 \leq i \leq l-j+1$) work in parallel, and the processor P_i is constructing the set $N_{i,j}$. At the end of iteration j the processors wait for a synchronisation signal raised by the IO-CTRL before beginning the next iteration.

In order to achieve synchronization the processors are using the synchronization&validation unit (see figure 3.1). How the synchronisation&validation unit is used to achieve processor synchronization is described in detail in section 3.5.1.4.

For retrieving the $RHS1$, and respectively $RHS2$ non-terminals from the source cells, respectively write back the non-terminals X_i in the destination cell, the processors need to access the chart data-structure stored in the chart memory. The access to the cells in the chart is implemented with the chart memory addressing unit (see figure 3.1) which is described in section 3.5.1.1. Moreover, the information stored in the current destination cell is updated with the update unit which is described in section 3.5.1.3. For checking the existence of a grammar rule, the processors have to access the data-structure representing the CNF grammar, stored in the cluster's local grammar memory. This functionality is implemented within the grammar look-up unit and is described in detail in section 3.5.1.2.

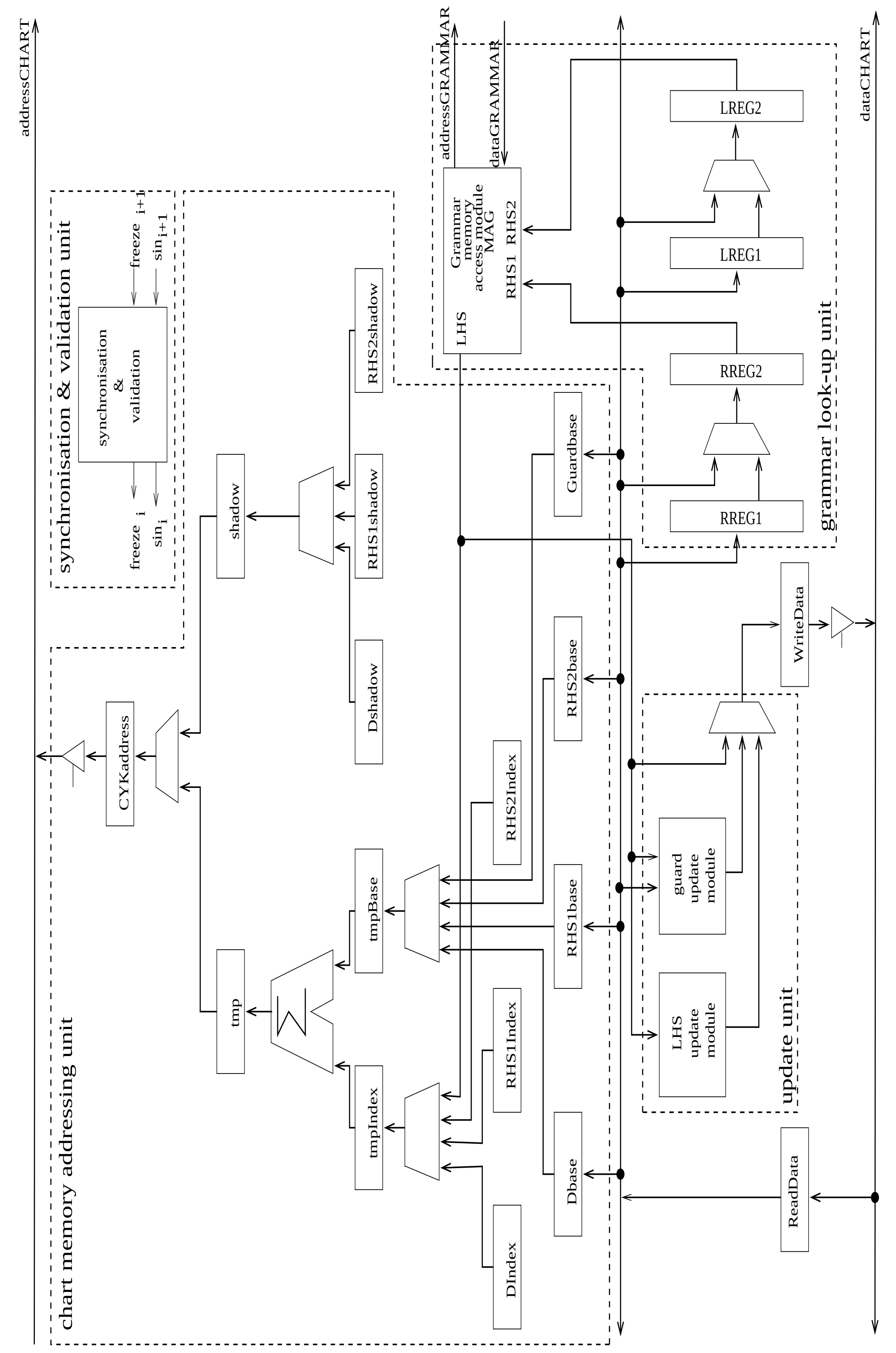


Figure 3.6: The datapath of the processor used in the linear array of processors (LAP) architecture.

3.5.1.1 The processor's interface to the chart memory

The chart memory stores the chart and is shared for read and write by all processors in the system. Concurrent accesses to this memory are solved with the chart memory arbiter. The task of this interface is to allow the processors to access, for read or write, the elements in a cell. The chart memory is a SRAM and as depicted in figure 3.6 the interface uses an address bus `addressCHART`, a data bus `dataCHART` and three command lines (not illustrated).

Before each parsing the chart memory is initialized. The initialization of the chart memory includes: the initialization of the indexing table, non-terminals table and guard-vectors table (see section 3.4.1). It is done by some software running on the on-board processor or on the host system.

In the case of the indexing table, the `Phead` and `Pguard` pointers are initialized only once before any parsing starts and will not change their values afterwards as the non-terminal and guard-vector tables do not move in memory. When parsing sentences, all `Dtail` values in the chart are set on 0. When parsing word lattices the indexes `Dtail` are initialized in order to reflect how many non-terminals are in a particular chart cell.

The most time-consuming initialization is the initialization of the guard-vectors. When parsing sentences, the guard-vector are initialized to all '0'. When parsing word lattices certain bits should also be set to reflect the presence of some non-terminals in the initial sets $N_{i,j}$.

Finally, the non-terminals table also has to be initialized and the last/empty flags should be set accordingly.

This chart memory addressing unit (see figure 3.6) is used to access the chart data-structure. Bellow we explain how the three distinct tables of the chart data-structure are accessed and the hardware resources involved:

<u>index table access</u>: the index table is accessed only for read and used to retrieve the pointers `Phead` and `Pguard` and the displacement `Dtail` of a cell with known coordinates (i, j) in the chart. For a given chart cell (i, j), the physical address of its associated entry in the index table is build from the (i, j) coordinates.

Precisely, if the chart data-structure can accommodate parsings for a maximal sentence length L, for a given cell (i, j) the physical memory address of its associated entry in the index table is built as $8(L * i + j)$. Note that the index table is stored in the chart memory starting at address 0.

In hardware, the physical memory addresses are constructed in the *shadow* register(s) as bellow:

- in `RHS1shadow` as $8 * (L * i + k)$ for cell $N_{i,k}$ (source cell 1)
- in `RHS2shadow` as $8 * (L * (i + k) + j - k)$ for cell $N_{i+k,j-k}$ (source cell 2)
- in `Dshadow` as $8 * (L * i + j)$ for cell $N_{i,j}$ (destination cell)

If the sentence length L is a power of 2 (which is actually a requirement) the memory addresses for the source and destination cells can be simply computed by performing binary shift operations on the cell coordinates.

<u>non-terminals table access</u>: accessed both for read and write. Read accesses are from $N_{i,k}$ (source cell 1) and $N_{i+k,j-k}$ (source cell 2) and write accesses to $N_{i,j}$ (destination cell).

In the hardware, the physical memory address for accessing the non-terminals in the non-terminals table is constructed as bellow:

- (`RHS1base` + `RHS1Index`) for $N_{i,k}$ (source cell 1)
- (`RHS2base` + `RHS2Index`) for $N_{i+k,j-k}$ (source cell 2)
- (`Dbase` + `DIndex`) for $N_{i,j}$ (destination cell)

As the non-terminals in a source cells are read sequentially after each read the index stored in `RHS1Index` or `RHS2Index` is incremented to address the next non-terminal in the list. The registers `RHS1base` and `RHS2base` are initialized from the `Phead` field of the corresponding cell and the counter registers `RHS1Index` and `RHS2Index` are initialized always on 0, even when parsing word lattices.

The result of the cell-combination between the source cells is written in the destination cell (i,j). The processors write the non-terminals in the destination cell sequentially at the physical address (`Dbase` + `DIndex`) and after each write the index stored in `DIndex` is incremented to point at the next non-terminal. The register `Dbase` is initialized from the `Phead` field of the corresponding cell and the counter register `DIndex` from the `Dtail` field. The displacement `Dtail` is always 0 in case of sentence parsing. However, in the case of word lattice parsing it may be different than 0.

guard-vectors table access: the table is accessed both for read and write. In hardware the physical memory address for accessing a particular guard-vector in the guard-vectors table is constructed from the registers `Guardbase` and `LHS` as (`Guardbase` + `LHS`), where `LHS` is actually the binary representation of the non-terminal we want to check (read) or set (write). The `Guardbase` register is initialized from the `Pguard` field associated to the destination cell and `LHS` is the output of the grammar lookup unit.

3.5.1.2 The processor's interface to the grammar memory.

Each grammar memory contains a copy of the CNF grammar data-structure (see section 3.4.2) and is used for grammar rules look-up during the parsing process. A grammar memory is only accessed for read and can be shared by several processors – clustered around the grammar memory – in which case an arbiter is used to arbiter concurrent accesses.

The unit that interfaces the processor to the grammar memory is the MAG unit[3] (see figure 3.6). The MAG unit uses an address bus `addressGRAMMAR`, a data bus `dataGRAMMAR` and three command lines (not illustrated) to connect to the grammar memories. The tasks of the MAG unit are (1) given the non-terminals $RHS1$ and $RHS2$, to allow the processor to retrieve all the non-terminals X_i for which there exists a grammar rule $X_i \rightarrow RHS1\ RHS2$ and (2) a code `RHScode` that uniquely identifies the right-hand side $RHS1\ RHS2$.

Before being used the grammar memories have to be configured. The CNF grammar data-structure is configured (i.e. loaded) once before any parsing can start, and remains unchanged for successive parsing. Only when a different CNF grammar needs to be used the grammar memories have to be reconfigured (i.e. reloaded) with the new CNF grammar data-structure. The initialization of the grammar memories is done by some software running on the on-board processor or on the host system.

The MAG unit allows the processor to retrieve the `RHScode` and all the non-terminals X_i that are left hand-sides for a given right hand-side $RHS1\ RHS2$. For doing so, the MAG unit implements the mechanism described in section 3.4.2 for accessing the three levels of the CNF grammar data-structure.

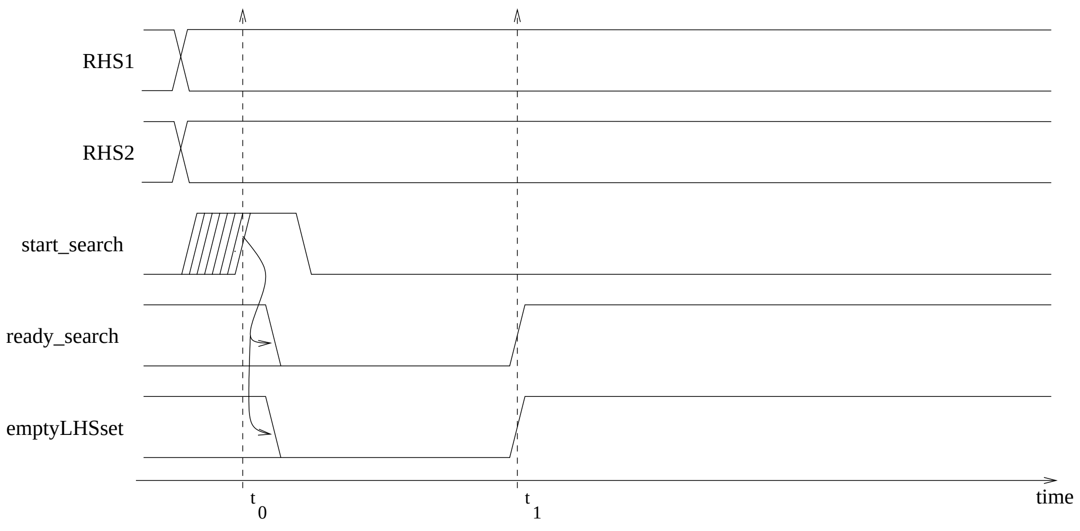
[3]MAG is an anagram of the initials of Grammar Memory Access unit

Signal	Direction	Size (bits)	Details
`RHS1`	I	NT_SIZE	the first non-terminal in the right hand-side of the searched grammar rule
`RHS2`	I	NT_SIZE	the second non-terminal in the right hand-side of the searched grammar rule
`start_search`	I	1	initiates the search (activated when $RHS1$ and $RHS2$ are stable)
`ready_search`	O	1	marks the end of the search
`emptyLHSset`	O	1	if active when `ready_search` is active, means that a valid non-terminal is available at the output `oneLHS`
`oneLHS`	O	NT_SIZE	the retrieved non-terminal(s)
`RHScode`	O	RULE_SIZE	a code that uniquely identifies a right hand-side. Used together with `oneLHS` it uniquely identifies a grammar rule
`nextLHS`	I	1	requests the following left hand-side in the list
`takeJETON`	O	1	requests the cluster's grammar memory (to the cluster's arbiter)
`releaseJETON`	O	1	releases the cluster's grammar memory (to the cluster's arbiter)
`jeton`	I	1	if active, the grammar memory belongs to this MAG unit
`dataGRAMMAR`	I	32	connected to the grammar memory data bus
`addressGRAMMAR`	O	PTR_SIZE-1	connected to the grammar memory address bus

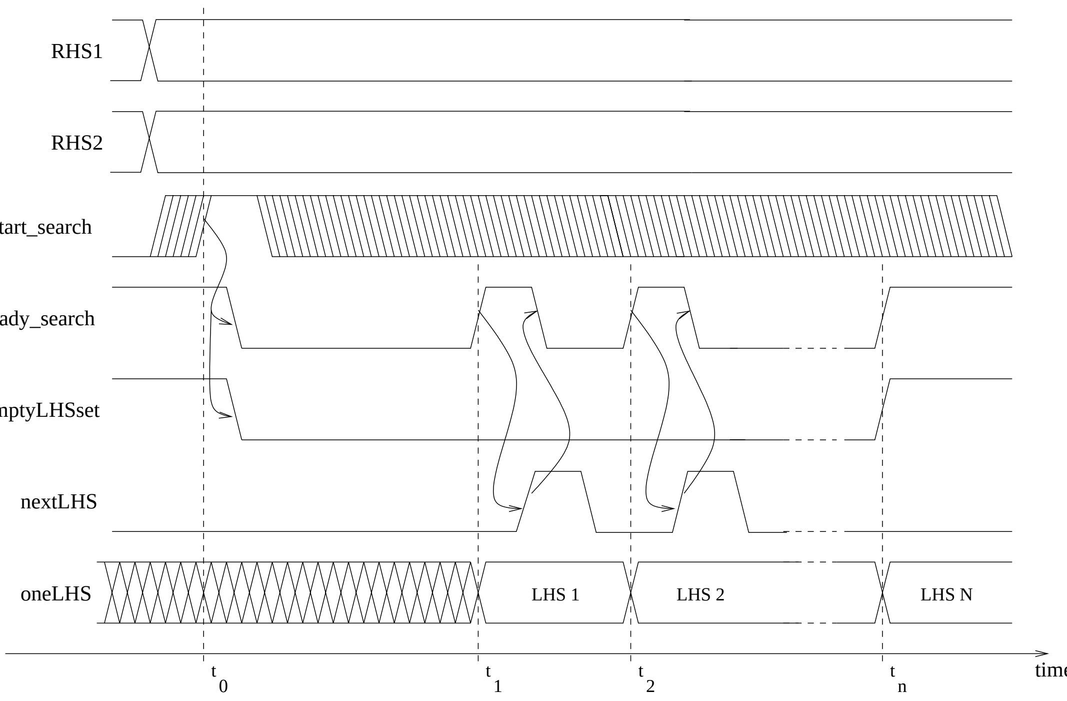
Table 3.4: MAG I/O signals used to interface with the processor, arbiter and cluster's grammar memory.

The signals used for interfacing the MAG unit to the processor along with their functional description are given in table 3.4. All these signals are active high and operate according to the switching waveforms (i.e. protocol) given in figure 3.7. We can distinguish two possible situations when the MAG unit accesses the CNF grammar data-structure, (1) there is no rule having the given right hand-side and (2) at least one such rule exists. The switching waveform for the first case is given in figure 3.7(a). At the moment t_0, the `RHS1` and `RHS1` are stable and the processor activates the signal `start_search` asking the MAG unit to start the search. After a non predictable duration, at time t_1 the MAG unit will answer by activating `ready_search` to mark the end of the searching and also `emptyLHSset` to signal that there is no rule in the CNF grammar with the given right hand-side $RHS1$ $RHS2$.

The switching waveform for the second case is given in figure 3.7(b). The searching is



(a)



(b)

Figure 3.7: Grammar memory access (MAG) unit interface protocol.

initiated as described above. However, in this case, at time t_1 when the processor activates `ready_search`, the signal `emptyLHSset` stays inactive (i.e. low). This means that the MAG unit has a valid non-terminal on the output `oneLHS` that is available to the processor. Whenever the processor wants the next left hand-side it activates `nextLHS`. The sequence described above is repeated for t_2, t_3, until we read all the left hand-sides for the given right hand-side. This moment t_n is marked by the signal `emptyLHSset` that is activated.

At this moment the $RHS1$ and $RHS2$ can be changed and a new search can begin.

3.5.1.3 The update unit

The tasks of the update unit are:

- set the value of flags (i.e. the last/empty special bits) of each non-terminal before writing them in the chart memory, in the non-terminals table. This is done in the LHS update module inside the update module (see figure 3.6);
- update the guard-vectors. Every time a new non-terminal is inserted in the non-terminals table, its corresponding bit in the guard-vector has to be set. This is done in the guard update module inside the update module (see figure 3.6). In order to update the guard-vector, a read operation is performed on the chart memory, for fetching the 4-byte word containing the bit. Once the corresponding bit is set the 4-byte word is written back to the chart memory.

3.5.1.4 The synchronization/validation unit

The synchronization and validation unit has two tasks (1) to allow the processors achieve synchronization after each filled row of the chart and (2) to generate the stop condition for the processors. The processors are synchronized at the end of each iteration j (line 7 of the CYK algorithm, see page 13), before starting iteration $j + 1$ and is necessary for insuring that the data dependency among the sets $N_{i,j}$ is satisfied. A processor will also check the stop condition at the beginning of each iteration j. If the stop condition is false the processor is validated (i.e. working) during the next iteration. By the contrary, if the stop condition is true, the processor is idle during the next iterations until the current parsing ends. Let's first see how the validation is implemented.

validation : The state (i.e. stopped or working) of processor P_i depends on the length of the parsed sentence and the state of the processor P_{i+1}, next to it at the right. For illustrating how the length of the parsed sentence conditions the initial state of the processors, let's consider the following example:

Example 3.4 For this example we consider the $n = 10$ processors system in figure 3.1. If we want to parse a sentence of $L = 4$ words, the initial configuration is: $\{P_1, P_2, P_3\}$ working and $\{P_4, P_5, \ldots, P_{10}\}$ stopped. If the sentence to parse has $L = 7$ words, the initial configuration is: $\{P_1, P_2, \ldots, P_6\}$ working and $\{P_7, P_8, P_9, P_{10}\}$ stopped. ◊

Also during run-time (i.e. parsing), the state of processor P_i depends on the processor P_{i+1} at his right. Precisely, if the processor P_{i+1} at his right is stopped, the processor P_i will be stopped during the next iteration.

Example 3.5 For the same $n = 10$ processors system in figure 3.1 we consider a sentence of length $L = 6$ words. The initial configuration is: $\{P_1, P_2, \ldots, P_5\}$ working and $\{P_6, P_7, \ldots, P_{10}\}$

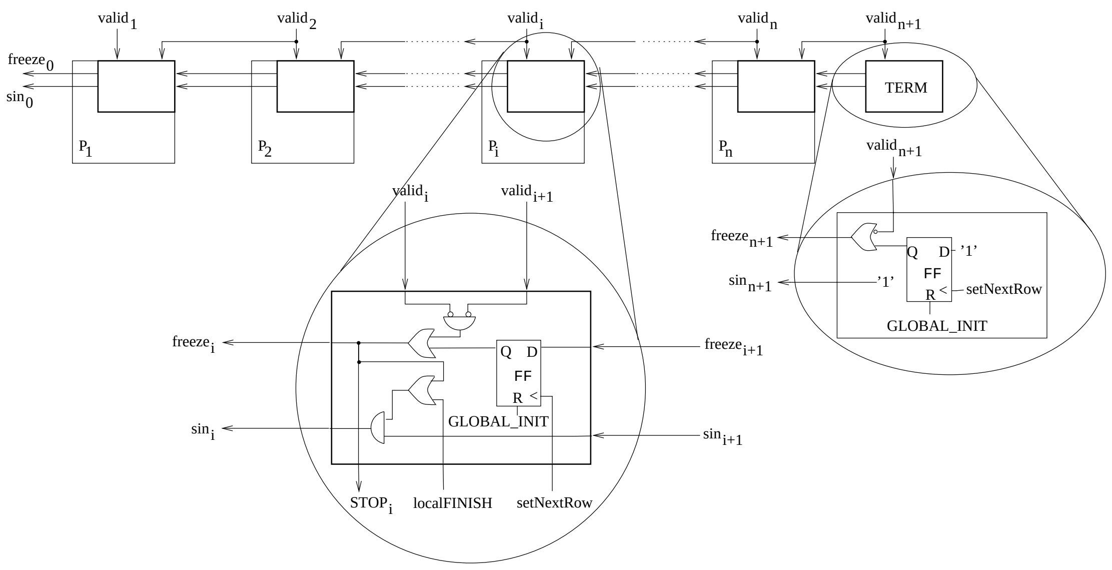


Figure 3.8: The hardware used to achieve processor synchronization and for generating the stop condition.

stopped. During run-time, after the first row of the chart is filled, the processor P_5 will stop. After the second row of the chart is filled, the processor P_4 will stop and so on until the fifth row is filled and processor P_1 stops and the parsing is finished. $\diamondsuit$

In order to explain how the validation mechanism is implemented in hardware, let's consider the figure 3.8. In this figure we have the signals involved in the generation of the stop condition as well as those used for achieving processor synchronization. For a processor P_i, the input signals $\texttt{freeze}_{\texttt{i+1}}$, $\texttt{sin}_{\texttt{i+1}}$ are cascaded from the processor P_{i+1}. The signals $\texttt{valid}_{\texttt{i}}$ are generated from the length L of the parsed sentence and the processor identifier i, where $1 \leq i \leq n$, as $\texttt{valid}_{\texttt{i}} = 1$ if $L \geq i$, 0 otherwise. The signal `localFINISH` is activated by the processor when it has finished his job. The signals `GLOBAL_INIT` and `setNextRow` are activated by the global controller IO-CTRL. The signal `GLOBAL_INIT` is used for initializing the processors each time a new parsing starts. The flip-flop `FF` used to propagate the stop condition is reset by this signal. The `setNextRow` signal is activated each time the processors have achieved synchronization (after a filled row) in order to update the processors state.

The output signal $\texttt{STOP}_{\texttt{i}}$ is the stop condition for the local processor and the signals $\texttt{freeze}_{\texttt{i}}$ and $\texttt{sin}_{\texttt{i}}$ are cascaded towards the processor P_{i-1}. Their boolean equations are given bellow:

$$\begin{aligned}
\texttt{STOP}_{\texttt{i}} &= \texttt{freeze}_{\texttt{i+1}} + \overline{\texttt{valid}_{\texttt{i}}} \cdot \overline{\texttt{valid}_{\texttt{i+1}}} \\
\texttt{freeze}_{\texttt{i}} &= \texttt{STOP}_{\texttt{i}} \\
\texttt{sin}_{\texttt{i}} &= \texttt{sin}_{\texttt{i+1}} \cdot (\texttt{STOP}_{\texttt{i}} + \texttt{localFINISH})
\end{aligned}$$

As we can see from the figure 3.8, all the output signals are updated when the `setNextRow` signal is activated.

synchronization : Let's now look at how the synchronization is implemented. We can interpret the boolean expression for the signal $\texttt{sin}_{\texttt{i}}$ given above, as follows: the output signal $\texttt{sin}_{\texttt{i}}$ propagates the input signal $\texttt{sin}_{\texttt{i+1}}$ if either the local processor has finished working (i.e. $\texttt{localFINISH} = '1'$) or the state of the local processor is stopped (i.e. $\texttt{STOP}_{\texttt{i}} = '1'$). As we can see in figure 3.8 the $\mathtt{sin}_{n+1}$ is connected directly to a $'1'$-logic. This $'1'$ propagates through the chain of processors, from right to left up to $\mathtt{sin}_0$ if all the processors are either in a stopped state or have finished working on the current row.

3.5.2 The I/O controller

The tasks of the I/O controller IO-CTRL are:

- system interface: all the configuration registers (see section 3.3) are accessed through the IO-CTRL. The host system is interfaced with the design through the IO-CTRL which monitors the `startPARSE` signal and generates the `overPARSE` signal to mark the end of the parsing. The end of the parsing is marked by the activation of the $\mathtt{freeze}_1$ signal (see section 3.5.1.4);

- system initialization: at power-up the hardware is reset. However, the state of the processors in the system requires initialization between successive parsings. This initialization is performed by the IO-CTRL with the `GLOBAL_INIT` signal;

- processor synchronization: although the hardware for achieving synchronization is located in the synchronization/validation unit of each processor, the IO-CTRL monitors the signal $\mathtt{sin}_1$. If this signal is activated it concludes that the processors have achieved synchronization and will activate the `setNextRow` signal for setting-up the processors for the processing of the next row.

3.5.3 The arbiters

All arbiters are token-passing units, adapted to the current design. Being synchronous with the system clock, their response does not depend on the size of the system (i.e. the number of arbitered processors) as is the case with the asynchronous arbiters. The interface between a processor P_i requesting access to a shared resource and the arbiter consists of three signals: $\mathtt{takeJETON}_\mathtt{i}$, $\mathtt{releaseJETON}_\mathtt{i}$ and $\mathtt{JETON}_\mathtt{i}$. The signal $\mathtt{takeJETON}_\mathtt{i}$ is used to signal the processor's intention to obtain the shared resource and the signal $\mathtt{releaseJETON}_\mathtt{i}$ is used to release the shared resource. The signal $\mathtt{JETON}_\mathtt{i}$ is the arbiter's answer and is activated when the shared resource is granted to processor P_i.

The arbiter works as follows: when the arbiter sees the token passed to processor P_i who has previously activated the signal $\mathtt{takeJETON}_\mathtt{i}$, it answers the processor by activating the $\mathtt{JETON}_\mathtt{i}$ signal. The processor sees the $\mathtt{JETON}_\mathtt{i}$ signal activated and knows that the requested resource has been granted to it. The arbiter will also block the token until the processor P_i will release the resource by activating the $\mathtt{releaseJETON}_\mathtt{i}$ signal.

The shared resources in the system are the chart memory (shared by all the processors in the system) and the cluster grammar memory (shared by the processors in the same cluster).

3.6 Performance measurements

All tests and performance measurements presented in this section were performed on the CNF SUSANNE grammar that contains $10,129$ non-terminals and $74,350$ rules. The size of the memory required to store the data-structure representing the CNF SUSANNE grammar is of $558,576$ bytes. The chart memory size depends on the number of processors in the system, or

Parameter		Size[units]	Details
system	CLOCK	50 [MHz]	the system clock
	SLEN_SIZE	5 [bits]	can parse sentences with up to $2^5 = 32$ words
	PROCESSORS	10	the number of processors in the system
chart memory	CYKLATENCY	17 [ns]	the access-time of the SRAM memory used to store the chart memory
	DTAIL_SIZE	9 [bits]	see section 3.4.1
	CYK_ADR_SIZE	19 [bits]	chart data-structure pointer size, see section 3.4.1
	CYK_WAIT_CYCLES	4	delay until chart memory data lines are stable
grammar memory	GMEMSIZE	558,576 [bytes]	the size of each grammar memory
	GLATENCY	17 [ns]	the access-time of the SRAM memory used to store the grammar memory
	WAIT_CYCLES	2	delay until grammar memory data lines are stable
	NT	10,129	the number of non-terminals in the grammar
	NT_SIZE	14 [bits]	bits used to represent a non-terminal, see section 3.4.2
	RULE_SIZE	15 [bits]	bits used to represent a distinct right-hand side 3.4.2
	PTR_SIZE	19 [bits]	grammar data-structure pointer size 3.4.2

Table 3.5: Parameter values used to configure the synthesized 10-processor LAP design.

the maximum length of the sentence we want to parse (e.g. $446,496$ bytes for the 10 processors system or $1,733,696$ bytes for a 32 processors system).

In order to determine the real maximal clock frequency at which the system is able to work, the 10 processor system shown in figure 3.1 was synthesized[4] and placed&routed[5] in a Xilinx FPGA, Virtex XCV1000bg560-4. The synthesis was performed on a design instance characterised by the parameters given in table 3.5. A summary of the FPGA resources used by the design units is given in table 3.6. The used FPGA resources are expressed in terms of D Flip-Flops and/or Latches (DFFs/Latches), function generators (FGs) and configurable logic blocks (CLBs). For detailed explanations of these terms you can refer to the Virtex manual [31]. Here, we only mention that for the Virtex XCV1000 FPGA a number of maximum $24,576$ DFF/Latches, $24,576$ FGs and $12,288$ CLBs are available.

As we can see in the table, the synthesized 10-processor system uses less then 37% of the FPGA resources and therefore a system with 25-processors may eventually fit in the same

[4]With LeonardoSpectrum v1999.1f

[5]With Design Manager (Xilinx Alliance Series 2.1i)

unit	DFFs/Latches	FGs	CLBs	XCV1000bg560-4 area utilization (%)	XCV2000bg1156-6 area utilization (%)
chart memory arbiter	36	91	46	0.37	0.24
3-GM cluster arbiter	8	18	9	0.07	0.05
4-GM cluster arbiter	12	27	14	0.11	0.07
IO-CTRL	28	27	14	0.11	0.07
MAG unit	168	296	148	1.20	0.76
processor	658	887	444	3.61	2.22
10-processor system	6’507	8’934	4’467	36.35	21.81

Table 3.6: Virtex XCV1000 FPGA resource utilization per LAP design unit in terms of Flip-Flops/Latches, function generators (FGs) and configurable logic blocks (CLBs).

FPGA that will allow us to parse sentences of real-life length of up to 26 words.

The system was then tested, and checked for correctness, on a RC1000-PP board with a clock frequency of 50 MHz. However, due to the fact that the RC1000-PP board can accommodate only 3 grammar clusters, the hardware run-times we present were obtained by simulating[6] the VHDL model of a system with 14 processors and 14 grammar clusters (one processor per grammar cluster) and respectively a system with 14 processors and 7 grammar clusters. In the last case the processors were clustered such that a priori the number of concurrent accesses to the grammar memories are as reduced as possible. The clusters are $\{P_1, P_{14}\}$, $\{P_2, P_{13}\}$, $\{P_3, P_{12}\}$,… $\{P_7, P_8\}$.

The software used for comparison is an implementation of the enhanced CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The hardware performance (i.e. run-time) of the 14 processors and 14 grammar clusters system (hereafter denoted as `hard_14P_14C`) and the 14 processors and 7 grammar clusters system (hereafter denoted as `hard_14P_7C`) was compared against two software run-times. The first (`soft_CNFG`) uses the SUSANNE grammar in CNF, as it is also the case for the hardware. The second (`soft_CFG`) uses the SUSANNE grammar in its original context-free form. The software was run on a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* `C` library function and not by profiling the code.

For the purpose of the comparison, $2,043$ sentences were parsed and validated[7]. The sentences have a length ranging from 3 to 15 and were all taken from the SUSANNE corpus. Figure 3.9 shows the average run-times `soft_CFG`, `hard_14P_14C` and `hard_14P_7C` as functions of the sentence length (vertical axe). Figure 3.10(a) shows the hardware speedup, for both `hard_14P_14C` and `hard_14P_7C` in comparison with `soft_CNFG` and figure 3.10(b) shows the hardware speedup, for both `hard_14P_14C` and `hard_14P_7C` in comparison with `soft_CFG` as a function of the sentence length. From these figures we can see that there is no significant difference for the speedups factors when comparing the `hard_14P_14C` and

[6]With ModelSim EE/Plus 5.2e

[7]The hardware output was compared to the software output for detecting mismatches. However, as the validation process consists of many technical details, it is not described here. A detailed description can be found in [9].

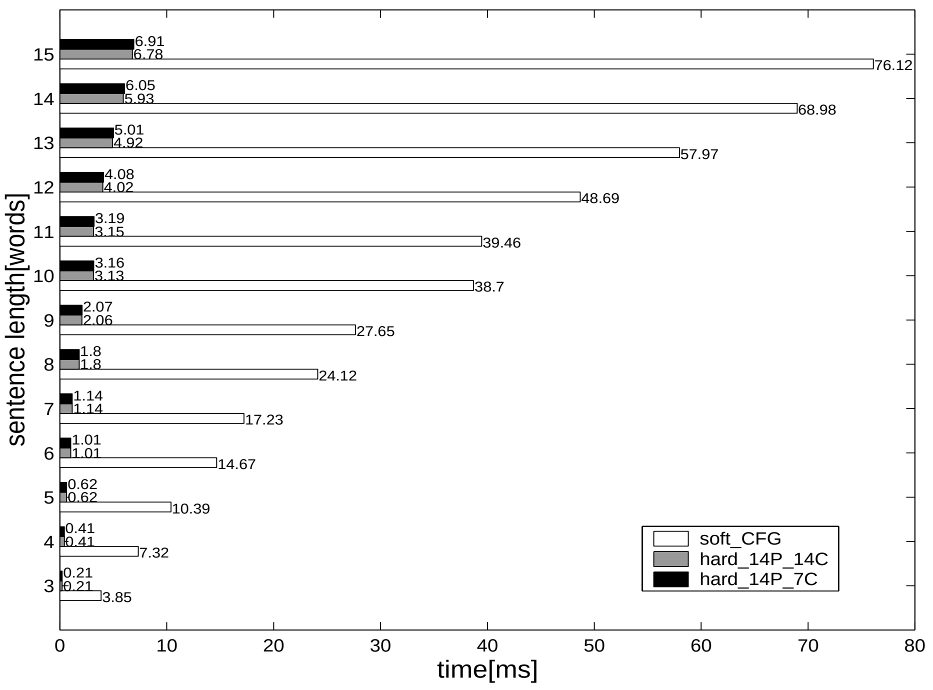


Figure 3.9: Average run-times for `soft_CFG` and `hard_14P_14C`, `hard_14P_7C` LAP systems as a function of sentence length. For each sentence length more than 100 sentences were parsed.

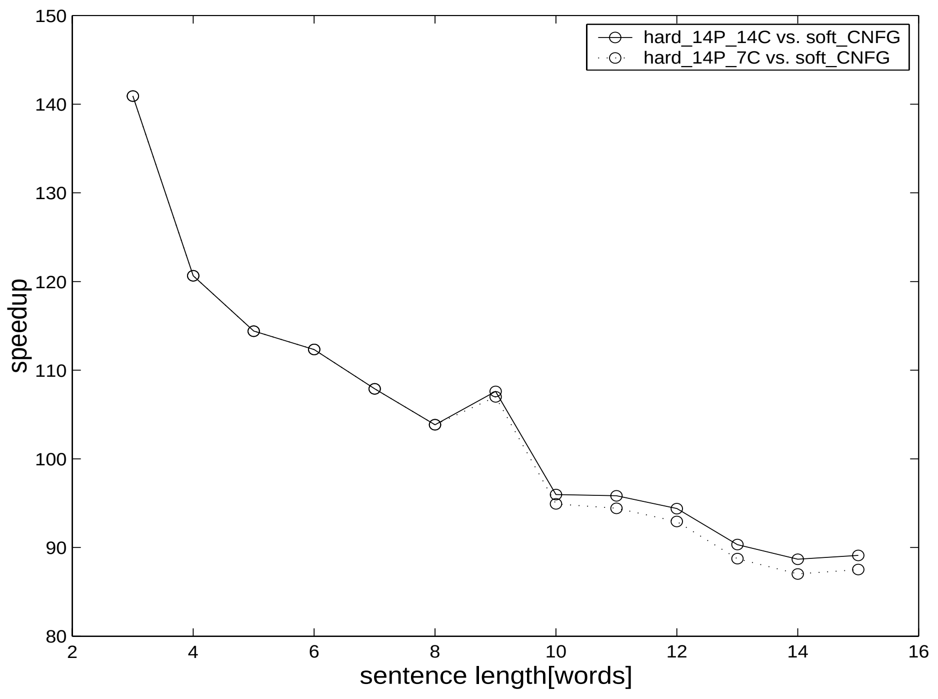
`hard_14P_7C` systems. For sentences with up to 8 words there is no difference between the two systems while only the processors $P_1, P_2, \ldots, P_7$ are working – which means that there is only one active processor in each grammar memory cluster – and therefore there are no collisions between the processors when accessing the grammar memories. For longer sentences, however, the explanation for the fact that there is no significant difference between the two systems is that the processors seldom work in parallel which results in sporadic grammar memory colissions. We also note that the system performance decreases with the sentence length which is a bad behaviour when dealing with real-life sentences.

The average speedup factor $E(S)$ has been computed for `hard_14P_14C` hardware implementation against both the `soft_CNFG` and `soft_CFG`. For `soft_CNFG`, $E_{\texttt{soft\_CNFG}}(S) = 102.89$ and for `soft_CFG`, $E_{\texttt{soft\_CFG}}(S) = 16.01$.

3.7 Design testing on the RC1000-PP FPGA board.

For obtaining the real clock frequency and validating the presented design, the 10-processor LAP system (see figure 3.1) was physically tested on a commercial FPGA-board.

We used Celoxica's RC1000-PP FPGA-board whose block diagram is given in figure 3.11. This board contains a Xilinx Virtex XCV1000 FPGA and 4 SRAM memories of 2 MBytes (organized as 512K x 32) each. The RC1000-PP board is shipped with host-side development software that support the initialization and communication with the hardware. The software provides key features such as handling the FPGA configuration files, initialization of the on-



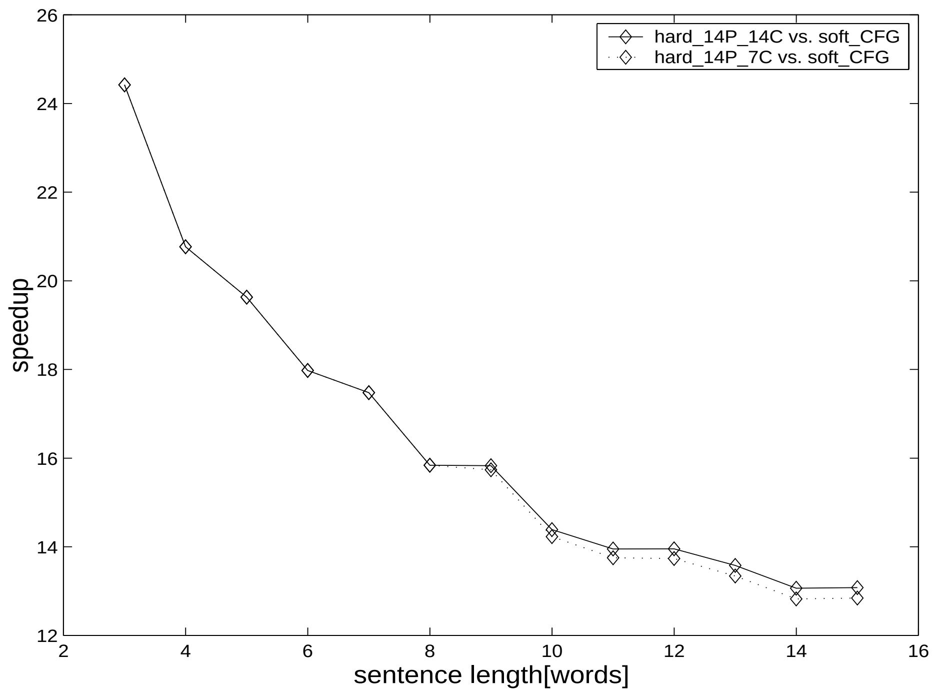


Figure 3.10: Hardware speedup for the `hard_14P_14C` and `hard_14P_7C` LAP systems against (a) `soft_CNFG` and (b) `soft_CFG` software as a function of sentence length. For each sentence length more than 100 sentences were parsed.

board programmable clock and data transfer support (i.e. handshake and DMA[8]) for transferring data between the host system an the RC1000-PP board. The software comes in the form of a static `C` library that can be linked to host-side programs. As the tested system contains 3

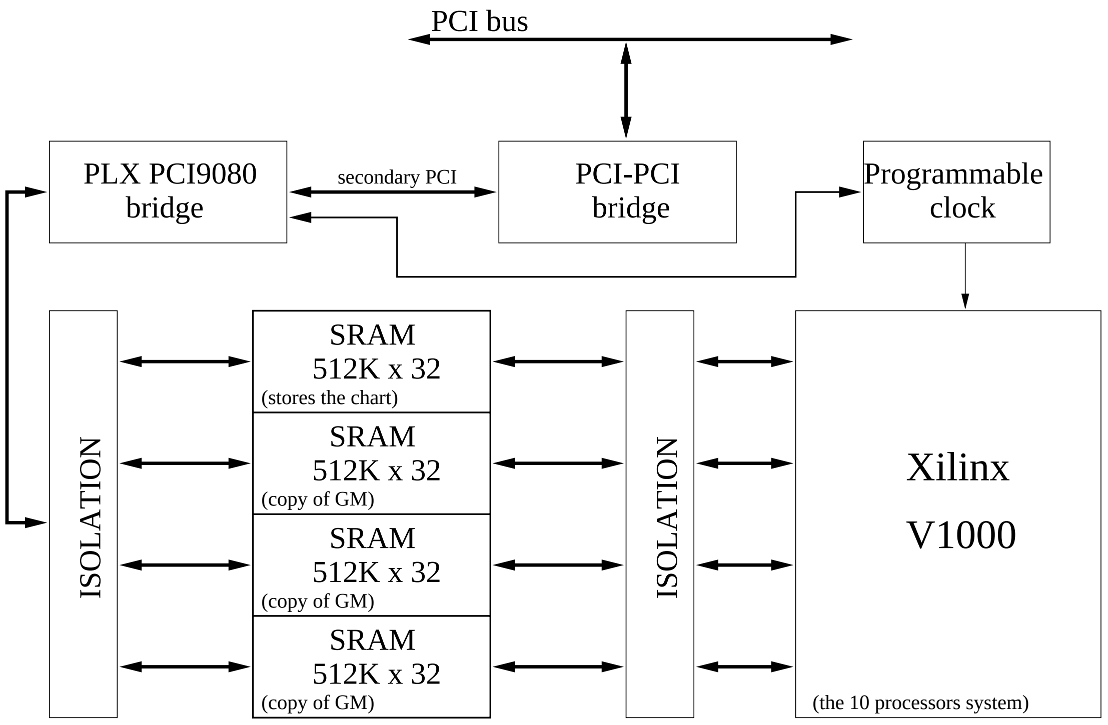


Figure 3.11: RC1000-PP FPGA-board block diagram illustrating the mapping of the 10-processor LAP system.

grammar clusters ($3 + 3 + 4$ processors) and the chart memory it can be fit on the 4 memory SRAM banks available on the RC1000-PP board. Also, the SRAM banks on the RC1000-PP board have a 32-bit databus that match the databus width used on the 10 processors system. Given these remarks, the synthesized 10 processors system can be placed&routed in the Virtex XCV1000 FPGA available on the RC1000-PP board without any further modification. The place&route tool only requires a pin constraint file `*.ucf` that will force signal routing inside the FPGA to the correct pins that are hardwired from fabrication to the external resources (i.e. memory banks, data communication units). The output of the place&route tool is a `*.bit` file that can be downloaded (using the development software library) to the FPGA without any further modification. Previous to the FPGA design configuration the 3 grammar memories are initialized with the binary image of the CNF grammar (i.e. SUSANNE CNF grammar) used for parsing and the programmable clock is set on the required frequency. The initialization of the grammar memories is performed through DMA transfers between the host system and the RC1000-PP board. Precisely, the CNF grammar image file is read by the host software and further transfered to each of the 3 grammar memories on the RC1000-PP board through 3 successive DMA transfers. The chart memory is initialized before each parsing by first creating an image of the chart memory in the host system. The image is then transfered through a DMA transfer in the SRAM memory associated with the chart memory. The `startPARSE`,

[8]Direct Memory Access.

respectively `overPARSE` signals are implemented with the 2 single-bit user signals available for host-FPGA communication. Finally, the sentence length and wait cycles (see section 3.3) are configured using one of the two 8-bit ports available for host-FPGA communication. In our case 5-bits are used to set the sentence length and the remaining 3-bits (a maximum of 8) represent the wait cycles.

The host system pools the `overPARSE` signal for detecting the end of the parsing. Once the parsing is finished the chart memory can be read (through a DMA transfer for instance) for validating the results .

3.8 Conclusions

The chapter shows by means of some examples why a 2D-array of processors architecture as presented in [8] is not feasible when using state-of-the-art FPGAs and real-life CFGs. Next it proposes a linear array of processors (LAP) architecture as an alternative solution. The LAP design was the first step of our design methodology during which we proved the feasibility and correct functionality of a FPGA-based hardware implementation of the CYK algorithm. The feasibility was proved in terms of required hardware resources and size of data-structures for storing the chart and the CNF grammars. For proving the correct functionality the design was synthesized and tested on a commercial FPGA board (RC1000-PP). We also showed that a significant speedup factor is obtained when compared against a software implementation of a similar algorithm.

The main features of the LAP design are: (1) a speedup factor of about 16.01 over our best software implementation of the enhanced-CYK algorithm and (2) its ability to parse sentences with up to 26 words or time-stamps when dealing with word lattices. These features make the design an interesting solution for integration within real-life NLP applications frameworks that have strong real-time constraints.

Chapter 4

Linear Array of Processors Design Analysis

In this section we discuss some important features of the LAP design . First, in order to get a general idea about the design we look at the processor activity and measure the processor utilization for some arbitrary sentences extracted from the SUSANNE corpus. More precisely, as the processor activity mainly consists of two tasks (1) data processing and (2) accessing the chart memory data-structure, we investigate the fraction of time spent by the processors in performing each of the two tasks. Next, the effect of increased number of collisions when accessing the chart memory as the sentence length increases on the overall design performance is investigated. We conclude with some remarks on the LAP architecture that pinpoint several critical changes required for building an improved design.

4.1 Processors utilization

In order to get a general idea about the processor activity during the parsing process we illustrate in figure 4.1(a), (b) and (c) the processor activity for three sentences (given in table 4.1) of length 4, 10 and respectively 15 words. The figures 4.1(a)-(c) illustrate for each processor in the design (vertical axis) its activity time during the parsing. More precisely, with black is depicted the amount of time spent by the processor for processing data and with gray the time spent for chart memory read/write accesses. In these figures we can see the moments when the

length	sentence
4	*“One wing stood open”*
10	*“In fact our whole defensive unit did a good job”*
15	*“In societies like ours , however , its place is less clear and more complex”*

Table 4.1: Three sentences parsed with the LAP design for which the processor activity is represented in figure 4.1.

processors synchronise and start the filling of a new row. An important thing to note about the processor activity is that there is a large discrepancy between the load of different processors when processing a row. In fact, when processing a row the most loaded processor, i.e. the one that finishes last, triggers the filling of the next row while all the other processors are idle. The above observations suggest that a better processor allocation mechanism is needed in order to better exploit the available processing power.

Another feature of the LAP design that can be observed in the figures 4.1(a)-(c) is the fact that as the length of the sentence increases the time spent by a processor for chart read/write accesses (i.e. gray area) also increases becoming a significant fraction in the overall processor activity time. This behaviour is explained by an increased number of collisions – that are arbitered – when accessing the chart memory.

The design efficiency can be described by means of the average processor utilization which is a measure for the efficiency with which the processors are used during the parsing process. In order to investigate the average processor utilization in the current design we will parse one sentence for each length between 3 and 15 words – chosen from the SUSANNE corpus – and compute for each of these sentences the average processor utilization. The average processor utilisation is computed as follows.

We define the utilisation U_{P_i} of processor P_i as the ratio between the time it spends in data-processing (i.e. black area without the gray area) during the entire parsing process and the time elapsed until the parsing finishes. Then, for a system with N processors the average processor utilization U is computed as follows:

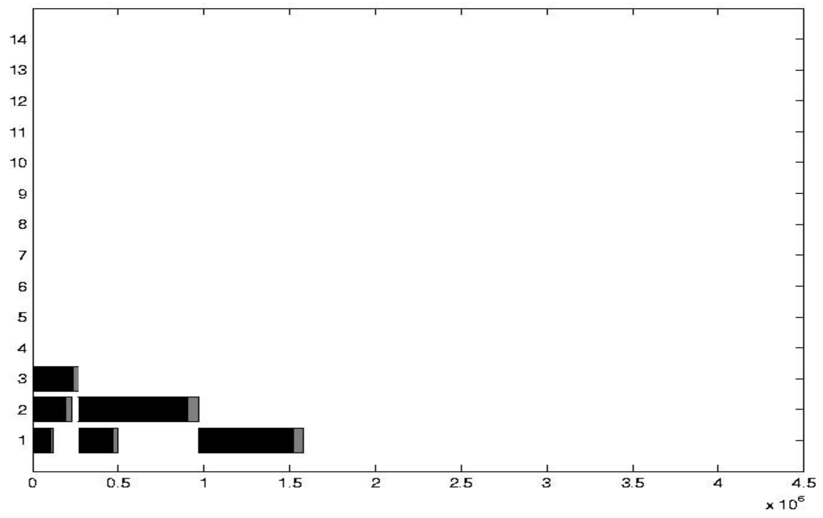
$$U = \frac{\sum_{i=1}^{N} U_{P_i}}{N} \tag{4.1}$$

The table 4.1 tabulates for each of the sentences chosen from the SUSANNE corpus the average processor utilization using the formula given above. Note that the average processor utilization

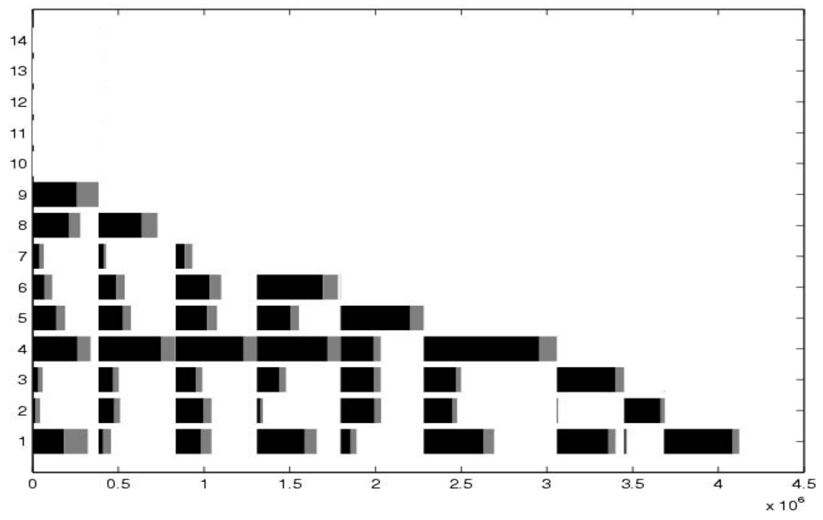
length	sentence	average processor utilisation U[%]
3	*"One pass only"*	9.26
4	*"There was no moon"*	8.27
5	*"She too began to weep"*	9.35
6	*"She must not think about time"*	10.99
7	*"The form and the chaos remain separate"*	12.72
8	*"The games were over , this was life"*	10.58
9	*"Like Napoleon , he was the worst of losers"*	8.90
10	*"In fact our whole defensive unit did a good job"*	15.01
11	*"I told him who I was and he was quite cold"*	8.36
12	*"Nerves tight as a bowstring , he paused to gather his wits"*	12.86
13	*"I told him no , that I had had a very happy childhood"*	10.00
14	*"As he had longed to be , he became the echo of a saga"*	10.48
15	*"It is all around us and our only chance now is to let it in"*	13.42

Table 4.2: The LAP design average processor utilization for a set of sentences extracted from the SUSANNE corpus with lengths between 3 and 15 words.

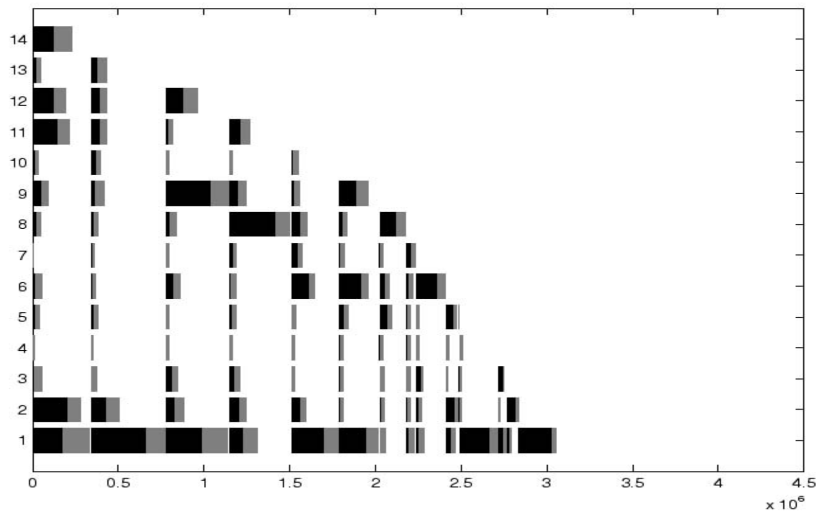
neither decreases nor increases as the sentence length increases. The average processor utilization actually oscillates around 10%.



(a)



(b)



(c)

Figure 4.1: LAP design processor activity in BLACK+GRAY when parsing a sentence of length (a) 4 words, (b) 10 words and (c) 15 words. GRAY: represents the time spent for chart read/write accesses. BLACK: represents the time spend for data processing.

4.2 Expected performance depreciation

As the length of the parsed sentence increases and more processors contribute to the parsing process, the system's performance (i.e. parsing time) does not scale at the expected rate. This depreciation in the expected system's performance is the consequence of an increased number of collisions (i.e. concurrent accesses) between the processors when accessing the chart memory as the sentence length increases. The increased number of collisions between processors can be observed on the figures 4.1(a)-(c) depicting the processor activity for three sentences randomly chosen from the SUSANNE corpus. In these figures, the gray area – corresponding to the time spent by the processors accessing the chart memory – increases as the sentence length increases, becoming a significant amount of the overall processor activity. However, for better illustrating this phenomena we will build our own grammar that has the particularity that any cell-combination in the chart requires the same amount of processing. In other words, such a grammar guarantees that the processing is uniformly distributed among the processors during the parsing.

The following example proposes such a grammar and uses it in an experiment in order to illustrate this expected performance depreciation phenomena.

Example 4.1 Let's consider the CNF grammar given by:

$$N = \{X_1, X_2, \ldots, X_{63}, X_{64}\},$$
$$\Sigma = \{a\},$$

$$\begin{aligned} R \;=\; & \{X_i \to X_i X_i \;\; and \;\; X_i \to a, \;\; for \;\; i \in \overline{1,8}\} \bigcup \\ & \{X_{64} \to X_i X_j \;\;\; for \;\; i \in \overline{1,8}\,, j \in \overline{9,63}\} \bigcup \\ & \{X_{64} \to X_i X_j \;\;\; for \;\; i \in \overline{9,63}\,, j \in \overline{1,63}\} \end{aligned}$$

The symbol S does not have any meaning for this grammar an is not defined. Although, of no practical use, this grammar helps us illustrate the expected performance depreciation phenomena, and for this purpose we will use it to parse the sentences: "a a", "a a a", "a a a a", ... up to a similar sentence of length 15.

After initialization each cell in the bottom row of the chart will contain the set of non-terminals $\{X_1, X_2, \ldots, X_8\}$. During parsing the grammar generates the non-terminals X_1, X_2, ..., X_8 for each cell-combination performed. This means that at the end of the parsing, each cell in the chart will contain the same set of non-terminals X_1, X_2, ..., X_8 and that during the parsing the processors work on the same data each time two cells are combined. In conclusion, if we know the time T spent by a processor for performing a cell-combination (i.e. the parsing time for the sentence "a a") we can compute as $L * T * (L-1)/2$ the expected parsing time for a sentence of length L. Note, that when parsing the sentence "a a" only one processor is working and therefore no collisions occur when accessing the chart.

In reality the parsing time is greater than the computed expected parsing time. This is due as we already said, to the fact that as the sentence length increases, the number of collisions when accessing the chart memory also increases.

Figure 4.2 illustrates the computed expected parsing time vs. the real parsing time when parsing the sentences "a a", "a a a", "a a a a", ... up to a similar sentence of length 15. The figure 4.3 depicts the processor activity when parsing a sentence of length 4 and respectively 7 "a"s, with the grammar proposed above. Note that the time required for filling the first row of the chart has increased due to the gray area (chart read/write accesses) while the black area

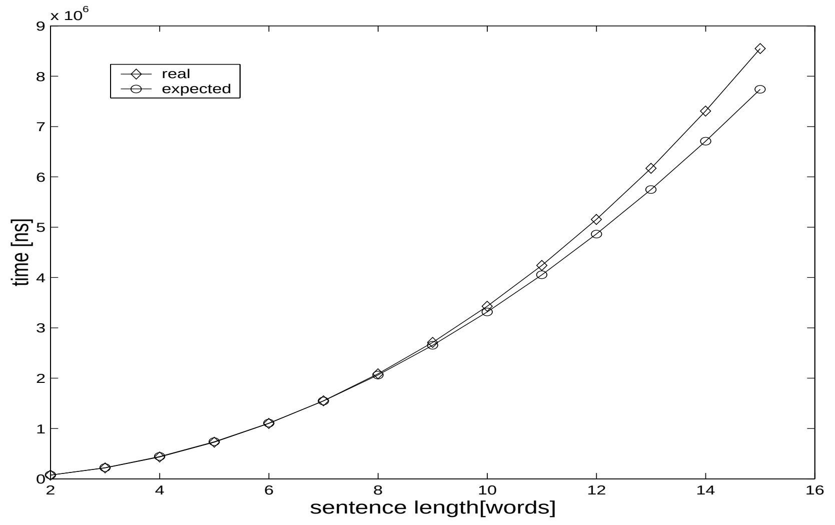


Figure 4.2: LAP design real vs. expected parsing time when parsing the sentences "a a", "a a a", "a a a a", . . . up to a similar sentence of length 15. Real parsing time is greater than the expected parsing time which illustrates the expected performance depreciation phenomenon.

(processing time) stays unchanged. This can be observed also for the second and the third row.

$\diamondsuit$

4.3 Conclusions

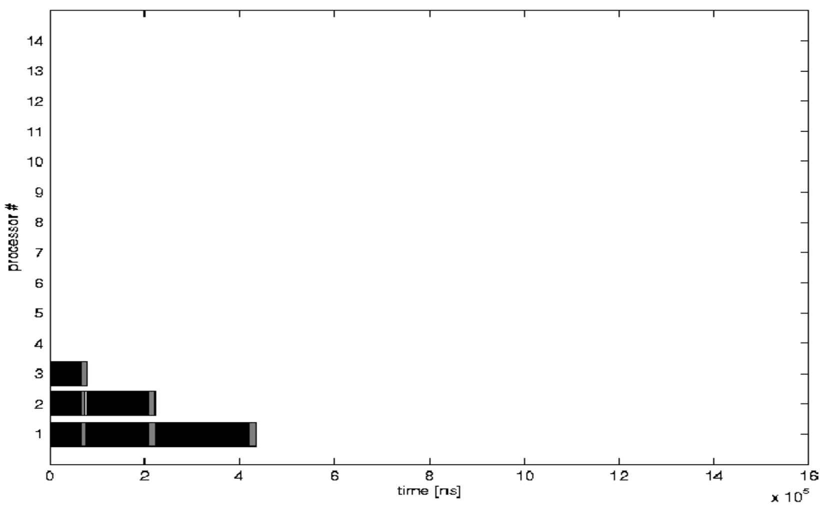
This chapter proposes a linear array of processors (LAP) FPGA-based hardware implementation of the CYK algorithm adapted for word lattice parsing that can deal with large-size real-life Chomsky Normal Form CFGs.

The LAP design fills the chart is a row-by-row fashion – all the cells in a row in parallel – and can parse any sentence of length $n + 1$ words or word lattice with $n + 1$ time stamps if n processors are available. The parsing results are available at the design's output as a compact parse forest that can be used for further processing (e.g. a semantic module). The design can be configured in FPGA chips such as the Xilinx Virtex/Virtex-E FPGA family available on the market by the time this thesis is written. Concretely, in the particular case when the SUSANNE grammar is used, the resources available in a Xilinx Virtex XCV1000-bg560 FPGA can fit a LAP design with up to $n = 25$ processors that allows us to parse any sentence with up to 26 words.

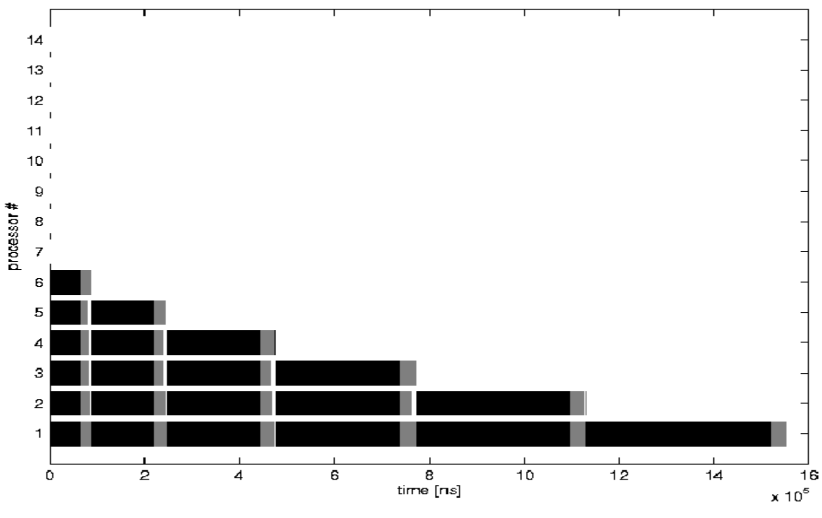
For the implemented LAP design, the performance measurements show an average speedup of 16.01 for the hardware when compared with our software implementation of the enhanced-CYK algorithm using a general context-free grammar and a speedup of 102.89 when compared with the same software using a CNF version of the same grammar. These preliminary experiments are encouraging results for the application of the reconfigurable computing paradigm for the NLP applications requiring efficient parsing with large-size real-life context-free grammars.

A 10-processors LAP design was synthesized and tested for validation on the commercial RC1000-PP FPGA board.

The initialisation and interface with the LAP design were conceived as simple as possible in order to allow an easy integration within a larger system. The FPGA – configured with



(a)



(b)

Figure 4.3: LAP design processor activity for a sentence of length (a) 4 and respectively (b) 7 words. The larger gray area is the reason for the expected performance depreciation phenomenon.

the LAP design – can be further integrated within a larger system, such as an FPGA-board for building a hardware tool that can work as an accelerator in an application framework (e.g. NLP) that requires efficient parsing of context-free languages. An example of such an application framework, namely a Vocal Information Server, is described in section 1.4 and depicted in figure 1.2.

The LAP design was the starting point in exploring the behaviour and characteristics of a hardware implementation of the CYK algorithm when dealing with large-size real-life grammars. The LAP design was the first step of our design methodology and was useful for:

- proofing the feasibility of an FPGA-based hardware implementation of the CYK algorithm both in terms of hardware resources and data structures used for storing the chart and the grammar memories;
- getting acquainted with the real-life behaviour of the CYK algorithm;

- having a first evaluation of the speed-up factor over a software implementation.

During design implementation and testing we acquired knowledge about the required FPGA resources, the size of the grammar memories (storing the grammar) and the chart memory (storing the chart), (2) we identified the critical regions and features of the design that are subject for further improvements and (3) we shown that a hardware implementation of the CYK algorithm offers a substantial advantage over a software implementation.

From the performed experiments and performance measurements we remark a number of drawbacks of the presented design:

- for a given sentence length, an increase in the number of processors in the system does not contribute to an increase in performance when parsing the same sentence. An increased number of processors only allows the system to parse longer sentences;
- as the sentence length grows, the number of working processors grows, and there is an increased number of interprocessor collisions when accessing the chart memory. This results in a performance depreciation as the sentence length grows. The accesses to the guard-vectors is the main reason for this phenomena and in the future implementations an alternative to the guard-vectors should be found;
- the analysis performed on some sentences extracted from the SUSANNE corpus shows that the average processor utilization is around 10%. A better processor control is required in order to better exploit the parallelism available in the CYK algorithm and therefore to increase the average processor utilization;
- the initialization procedure is relatively complex and time consuming. Again, the source of this problem are the guard-vectors that require a ’0’-filling before each new parsing;
- the design is only able to deal with CNF grammars. A design that could cope with general context-free grammars is sought;
- the design does not have yet the ability to recover from fatal errors such as that encountered when exceeding the maximum number K of non-terminals in a cell (see section 3.4.1) nor does it integrate the unit that extracts on-line the compact parse forest. Although the above mentioned features are not key for a design aiming at proofing the feasibility of the FPGA technology for implementing the CYK algorithm, they are requirements for a future design;

Chapter 5

The Dynamic Array of Processors Hardware Design

This chapter presents an improved design architecture implementing the CYK algorithm adapted for word lattice parsing. The proposed architecture is the second step of our design methodology during which:

- we investigate a better processor allocation method while also increasing the maximum sentence length that can be parsed;

- we refine and extend our background knowledge about the real-life behaviour and critical features of the CYK algorithm before attempting attempting to design an architecture for the enhanced-CYK algorithm;

- we improve the speed-up factor;

In order to better exploit the parallelism available in the CYK algorithm – when compared to the row-by-row method used in the LAP design – a better processor allocation method is implemented within the current design. The implemented processor allocation method tries to process a cell-combination as soon as the cells it relies upon become available. In other words, the implemented processor allocation method tries to "follow" the dataflow in the chart. Due to the fact that the processors are assigned dynamically to process cell-combinations we will call the current design a Dynamic Array of Processors – henceforth referred as DAP – and the processor allocation mechanism, the dynamic processor allocation method . The dynamic processor allocation method is studied in appendix B.2 in the particular case of the SUSANNE grammar.

Another improvement is that the maximal sentence length that can be parsed with the DAP design is independent of the number of processors in the system. The maximal sentence length only depends on the size of the available chart memory and does not depend anymore on the resources available in the used FPGA. In other words, if the syntactic analysis of a sentence fits in the chart memory, then any number of processors can parse the sentence and the number of processors we can fit in the FPGA will only influence the parsing performance.

While essentially the same architecture will be used for implementing the enhanced-CYK algorithm the analysis and performance measurements of the proposed design architecture will add more background knowledge and also help to identify the critical spots of such an architecture. The last point is an argument for the dynamic processor allocation method.

This chapter starts with a general and functional system description. It continues by presenting the system units in detail. The design performance measurements and an analysis discussing some key features of the proposed design are given before concluding this chapter.

5.1 General system description

The design proposed in this chapter can parse sentences of any length, regardless of the number of processors in the system, given that the chart memory is large enough to store the chart data-structure. As for the LAP design, the input to the DAP design is an initialized chart and grammar lookup tables and the output is a compact parse forest. The DAP design uses SRAMs for storing the data-structures representing the chart (stored in the chart memory) and the CNF grammar (stored in grammar memories). These data-structures are the same as those employed by the LAP design (see section 3.4.1 for the chart data-structure and section 3.4.2 for the grammar memory data-structure).

As in the current design the maximal length of the sentence that can be parsed does not depend on the number of processors, there is no need to cluster the processors around grammar memories. This was only necessary with the LAP design when the length of the parsed sentence – and therefore the number of processors – was exceeding the number of available grammar memories. In the current design there are as many grammar memories as processors in the system. However, as the number of pins available for an FPGA is limited, only a limited number of grammar memories can be connected to the FPGA which will also limit the number of processors. Due to the fact that most of a processor's processing time is spent in accessing the grammar memory, having only one processor per grammar memory will result in a more efficient grammar memory utilization (no time is lost for grammar memory cluster arbitration).

The chart memory is shared by all the processors in the system as it was the case for the LAP design. All the processors in the system implement a procedure for accessing the chart data-structure stored in the chart memory in order to fetch the sets required for performing the cell-combinations they are assigned to compute.

Each processor in the DAP design is dynamically assigned to perform cell-combinations on the chart. The difficulty with such a method comes from the fact that a cell-combination cannot be issued before the cells it relies upon are not available (i.e. processed). In other words, it has to deal with the dynamic data-dependency during runtime. The data-dependency issue was simply solved in the LAP design by (re)synchronizing the processors after each filled row. While the (re)synchronization mechanism has the advantage of being easy to implement it does not yield good results. For this reason the DAP design implements a more sophisticated mechanism that starts to process the cell-combinations as soon as the cells (i.e. sets) they rely upon become available.

The general architecture of an n-processor system implementing the dynamic allocation method is depicted in the block diagram in figure 5.1. The elements inside the dashed line are implemented within the FPGA chip. The other elements (chart and grammar memories GMi) are implemented in SRAM chips present on the system board. The system's initialization procedure as well as its interface with the external world did not change, being identical with the one described for the LAP design. A system with $n = 3$ processors was synthesized and physically tested (i.e. the results were validated for correctness) on the commercial RC1000-PP FPGA board containing a Xilinx Virtex XCV1000bg560-4 FPGA. As the RC1000-PP FPGA board contains 4 SRAM chips, one is used for storing the chart and 3 for storing identical copies of the CNF grammar data-structure – one for each processor.

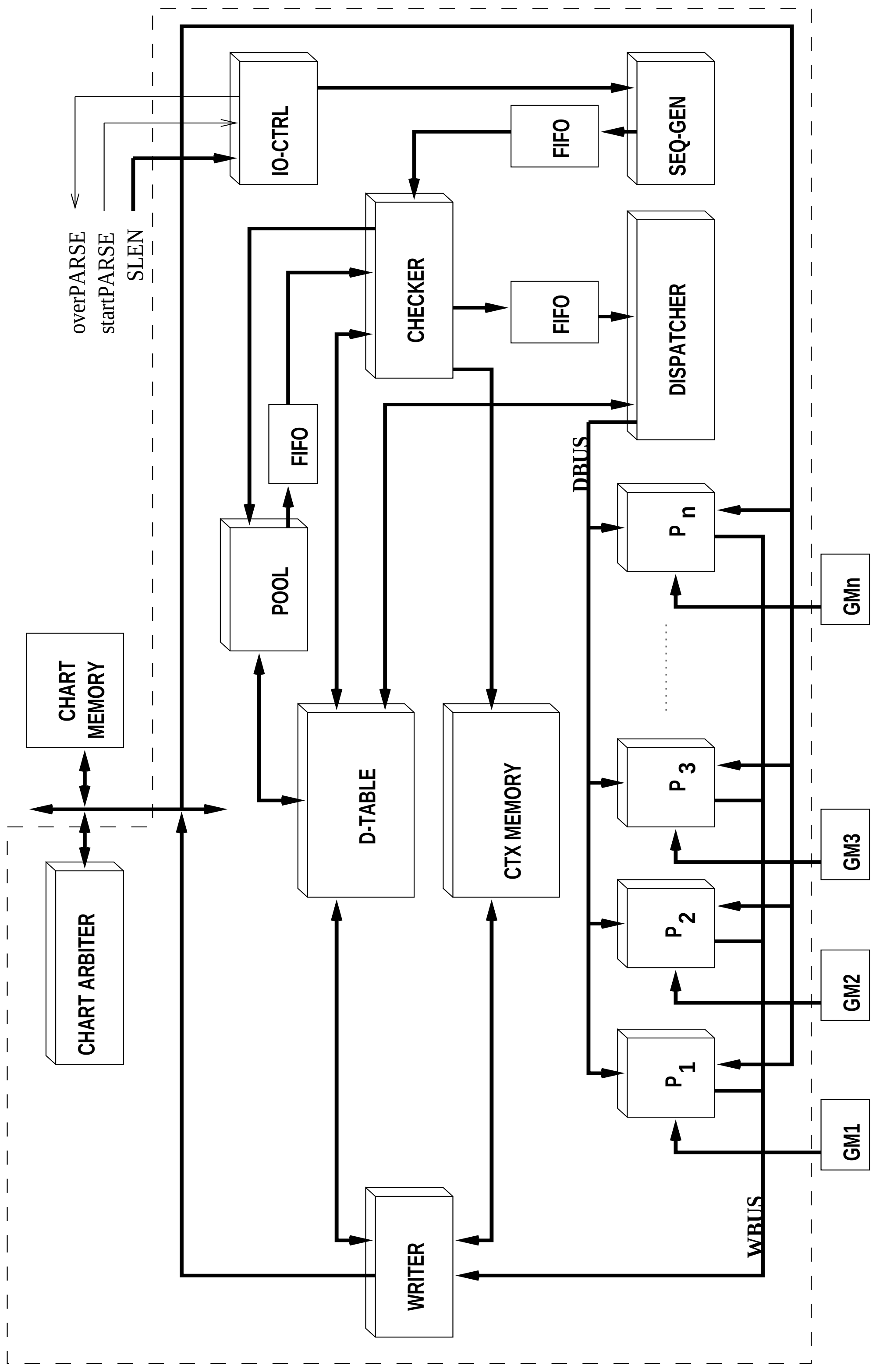


Figure 5.1: The block diagram of an n-processor DAP design.

5.2 Functional description

The following initialisations are necessary before the parsing can start:

- grammar memories : before any sentence can be parsed, each of the grammar memories have to be configured with the binary image of the data-structure representing the CNF grammar (for details see section 3.4.2). The initialization of the grammar memories is done by some software running on the on-board processor or on the host system. As the grammar data-structure does not change during the parsing, the grammar memories only have to be configured once for multiple parsings with the same grammar. Only if a different grammar has to be used the grammar memories have to be reconfigured;

- the chart memory : the initialization of the chart memory is done by some software running on the on-board processor or on the host system. It consists of initializing certain cells (i, j) of the chart with sets of non-terminals $N_{i,j}$ along with their corresponding guard-vectors (see section3.4.1);

- sentence length : the global controller (IO-CTRL) is initialised before every parsing with the length of the sentence to be parsed. Based on the sentence length the SEQ_GEN will generate the sequence of cell-combinations required for parsing the sentence. The sentence length is configured in a register;

- wait cycles : due to the fact that the chart memory is accessed by a relatively large number of processors the logic required to access this memory may have a large propagation time (i.e. delay) and in consequence the chart memory access cycles may be long. In order to keep the rest of the system frequency high, wait cycles are introduced in the chart memory access cycles. As the number of necessary wait cycles cannot be foreseen from the beginning (they actually depend on the way the signals are routed, the number of processors and other factors), we use a register for configuring the number of wait cycles. A first estimate for the number of wait cycles required to access the chart memory is known after the design is synthesized. Within the current design a number of 1 to 8 wait cycles can be pre-configured (default value) in the VHDL code with the generic `CYK_WAIT_CYCLES`;

Among these initializations, the initialization of the chart memory is the most time consuming an may represent and important amount in the overall processing time.

Once the system was initialized – according to the procedure described above – the parsing can start when the signal `startPARSE` is activated. At this moment the sequence generator unit SEQ_GEN (see section 5.3.1 for details) will start to generate triples $(S1,S2,D)$ of two chart source cells $S1$ and $S2$, that need to be combined (see lines 7-9 of the CYK algorithm on page 13) along with their corresponding destination chart cell D, where the combination result will be stored.

<u>Note</u>: A source cell or the destination cell in a triplet is in fact a pair of coordinates representing the row and the column of that cell. For instance, $S1$ is written as (r_{s1}, c_{s1}), where r_{s1} is the row and c_{s1} is the column of the first source cell. The number of bits used to represent a row/column is configured in the VHDL code with the generic `COORD_GEN`. The value allocated to this parameter limits the length of the sentence that can be parsed. For instance, if $\texttt{COORD\_GEN} = 5$, only sentences with up to 32 words can be parsed.

These triples depend on the length of the parsed sentence and the same series of triples is generated for any two sentences of the same length. The triples are then passed to the CHECKER (see section 5.3.2 for details) unit who's task is to check whether the source chart cells are ready, i.e. are not destinations of unfinished previous cell-combinations. In other words, the CHECKER verifies that the data-dependency among the sets $N_{i,j}$ in the chart, is satisfied. For doing this, the CHECKER uses a table D-TABLE (see section 5.3.6 for details) that stores all destination cells D, currently under processing. The CHECKER inserts a new destination cell D in the D-TABLE when it is encountered for the first time in a triplet and deletes it when all the triples containing the destination cell D have been treated. Each time the CHECKER inserts a new destination cell D in the D-TABLE it does two things: (1) sets some "context information" for the destination cell D and stores it in the CTX-memory and (2) allocates an unique identifier ID for the destination cell D. The CHECKER tags all subsequent triplets containing the destination cell D with the identifier ID before sending them to the DISPATCHER.

A triplet that passes the CHECKER's test for data-dependency is forwarded to the task dispatching unit DISPATCHER, while a triplet that does not pass the CHECKER's test for data-dependency is stored in the POOL (see section 5.3.3 for details) where it waits until the data-dependency is satisfied. In the POOL all triplets are continuously checked for data-dependency against the D-TABLE and as soon as a triplet passes the data-dependency test it is forwarded to the CHECKER (that will update the D-TABLE) and immediately further to the DISPATCHER. The CHECKER has therefore two inputs, one from the POOL and the other from the SEQ_GEN – both passing through FIFO memories (see figure 5.1). The triplets coming from the POOL have higher priority and are handled first. The reason for giving higher priority to the triplets coming from the POOL is that, for a SEQ_GEN generating triplets in a natural chronological order, the POOL will store old triplets that require to be treated first.

The DISPATCHER (see section 5.3.4 for details) unit has the task of distributing the tasks (i.e. triplets and their associated unique ID) to the available processors that will further perform the cell-combinations. The result of the cell-combinations (i.e. the processors output) is fetched by the WRITER and finally stored in the chart memory if necessary. The WRITER uses the ID provided by the flushed processor to retrieve the "context information" of the associated destination cell D in which the processing results will be stored. After fetching a processor's output the WRITER also updates the data-dependency information in the D-TABLE.

The `overPARSE` signal indicates the end of the parsing. It is activated as soon as the SEQ_GEN unit has generated all the triplets according to the length of the parsed sentence and there are no more triplets under processing. The parse result is available at some output `outPARSE` (not represented in figure 5.1) of each processor and can be collected for building the compact parsing forest.

5.3 Design units

5.3.1 The sequence generator (SEQ_GEN) unit

The sequence generator is presented in the block diagram (see figure 5.1) as the SEQ_GEN unit. Its task is to generate all triples $(S1,S2,D)$ of two chart source cells $S1$ and $S2$ that need to be combined, along with their corresponding destination chart cell D, where the cell-combination result will be stored. A source/destination cell in a triplet is in fact a pair of coordinates representing the row and column of that cell. For the CYK algorithm (see section 2.2) and respectively the enhanced CYK algorithm (see section 2.3) the triplets are generated in the lines $7-9$ of the algorithms as (i,j) for D, (i,k) for $S1$ and respectively $S2$ for $(i+k, j-k)$. The following example illustrates the sequence of triples generated by the SEQ_GEN unit each time a sentence of length $L=5$ is parsed.

Example 5.1 In figure 5.2 we have a possible sequence of all the generated triplets. The

	$t_i \ : \ (S1 \ + \ S2 \ \to \ D)$	
2-nd row	$t_0 : (1,1)+(1,2) \to (2,1)$	$t_1 : (1,2)+(1,3) \to (2,2)$
	$t_2 : (1,3)+(1,4) \to (2,3)$	$t_3 : (1,4)+(1,5) \to (2,4)$
3-rd row	$t_4 : (1,1)+(2,2) \to (3,1)$	$t_5 : (2,1)+(1,3) \to (3,1)$
	$t_6 : (1,2)+(2,3) \to (3,2)$	$t_7 : (2,2)+(1,4) \to (3,2)$
	$t_8 : (1,3)+(2,4) \to (3,3)$	$t_9 : (2,3)+(1,5) \to (3,3)$
4-th row	$t_{10} : (1,1)+(3,2) \to (4,1)$	$t_{11} : (2,1)+(2,3) \to (4,1)$
	$t_{12} : (3,1)+(1,4) \to (4,1)$	$t_{13} : (1,2)+(3,3) \to (4,2)$
	$t_{14} : (2,2)+(2,4) \to (4,2)$	$t_{15} : (3,2)+(1,5) \to (4,2)$
5-th row	$t_{16} : (1,1)+(4,2) \to (5,1)$	$t_{17} : (2,1)+(3,3) \to (5,1)$
	$t_{18} : (3,1)+(2,4) \to (5,1)$	$t_{19} : (4,1)+(1,5) \to (5,1)$

Figure 5.2: A (possible) sequence of triples $(t_0, t_1, \ldots)$ generated when a sentence of length $L=5$ is parsed.

triplets are ordered according to the time instant t_i at which they are generated. The notation $t_i \ : \ (S1 \ + \ S2 \ \to \ D)$ stands for the triplet $(S1, S2, D)$ generated at time t_i, and represents the source cells to be combined into the destination cell D. $\diamondsuit$

The sequence of triplets generated by the SEQ_GEN unit is always the same for a given sentence length L. In the above example and in general we assume that the destination cells are visited in a row-by-row manner. This order of visiting (i.e. processing) the destination cells is natural as every cell in a row depends on cells in the rows bellow it.

However, there are several possible orderings for the generated sequence of triplets, corresponding to different orders of processing the cells in a given row. The best order in which the triplets can be generated is a sequence with chronological ordered source cells. Let's illustrate this with the following example.

Example 5.2 In the previous example, consider the sequence of triplets (t_{10}, t_{11}, t_{12}) generated for the destination cell $(4,1)$. In this sequence, $t_{11} \ : \ (2,1) + (2,3) \ \to \ (4,1)$ comes after $t_{10} \ : \ (1,1) + (3,2) \ \to \ (4,1)$, although the source cell $(3,2)$ of the triplet generated at t_{10} has less chances of being available before any of the source cells $(2,1)$ or $(2,3)$ in the triplet generated at t_{11}. A better ordering would be therefore (t_{11}, t_{10}, t_{12}). $\diamondsuit$

In other words, a triplet $t_i = (S_{i_1}, S_{i_2}, D_i)$ will be generated before a triplet $t_j = (S_{j_1}, S_{j_2}, D_j)$ if both of the source cells S_{i_1}, S_{i_2} were produced (as destinations) before the later produced among the source cells S_{j_1}, S_{j_2}. Such a triplet ordering guarantee a priori the processing of the cell-combinations in chronological order and gives the best chances to pass the data-dependency test performed by the CHECKER.

In figure 5.3 we have the example of a sequence of triplets generated for the same sentence length $L=5$ but with chronological ordered source cells. Note that successive triplets do not necessarily belong to the same destination cell. The sequence generation method illustrated in

	$t_i\ :\ (S1\ +\ S2\ \to\ D)$	
2-nd row	$t_0 : (1,1) + (1,2) \to (2,1)$	$t_1 : (1,2) + (1,3) \to (2,2)$
	$t_2 : (1,3) + (1,4) \to (2,3)$	$t_3 : (1,4) + (1,5) \to (2,4)$
3-rd row	$t_7 : (2,1) + (1,3) \to (3,1)$	$t_4 : (1,1) + (2,2) \to (3,1)$
	$t_8 : (2,2) + (1,4) \to (3,2)$	$t_5 : (1,2) + (2,3) \to (3,2)$
	$t_9 : (2,3) + (1,5) \to (3,3)$	$t_6 : (1,3) + (2,4) \to (3,3)$
4-th row	$t_{14} : (2,1) + (2,3) \to (4,1)$	$t_{10} : (3,1) + (1,4) \to (4,1)$
	$t_{12} : (1,1) + (3,2) \to (4,1)$	$t_{15} : (2,2) + (2,4) \to (4,2)$
	$t_{11} : (3,2) + (1,5) \to (4,2)$	$t_{13} : (1,2) + (3,3) \to (4,2)$
5-th row	$t_{19} : (3,1) + (2,4) \to (5,1)$	$t_{17} : (2,1) + (3,3) \to (5,1)$
	$t_{16} : (4,1) + (1,5) \to (5,1)$	$t_{18} : (1,1) + (4,2) \to (5,1)$

Figure 5.3: Another (possible) sequence of triplets, but with chronological ordered $(t_0, t_1, \ldots)$ source cells, generated for the same sentence length $L = 5$.

this example is actually implemented in the SEQ_GEN unit and consists of a set of counters and adder/subtracters units, easy to implement.

The output of the SEQ_GEN unit is forwarded to the CHECKER through a FIFO memory for decoupling the two units. The size of this FIFO memory is configured in the VHDL code with the generic `OUT_GEN_DEPTH`. A small FIFO memory is actually required (i.e. 2-4) as the SEQ_GEN unit is fast and will always keep the FIFO memory full.

5.3.2 The data-dependency checking (CHECKER) unit

The task of the CHECKER unit is to verify that the data-dependency among the triplets coming from the sequence generator is satisfied. For doing so, the CHECKER makes use of the destination table D-TABLE that stores all destination cells currently under processing. Whenever a triplet $(S1, S2, D)$ coming from the sequence generator contains a source cell – either $S1$ or $S2$ – that is in the D-TABLE (i.e. that source cell is a destination under processing) the CHECKER concludes that the triplet fails the data-dependency test and therefore the task associated with it cannot be yet dispatched to the processors. At this point, the CHECKER has two possibilities: (1) either it waits for the triplet to satisfy the data-dependency constraint or (2) stores the problematic triplet in a buffer and continues to process the following triplets coming from the sequence generator. The second solution is obviously better and was implemented within the current design. The triplets that do not pass the data-dependency test are actually stored in a buffer, represented in the block diagram (see figure 5.1) as the POOL unit, where they are continuously checked for data-dependency against the D-TABLE. As soon as a triplet stored in the POOL passes the data-dependency test, it is forwarded back to the CHECKER and treated by the later with higher priority over the triplets coming from the sequence generator. This is reasonable in order to process the cell-combinations (i.e. triplets) in the chronological order in which they were generated. On the other hand, if the buffer in the POOL unit is full the only possibility is to wait until either a triplet in the POOL or the triplet to be stored in the POOL will pass the data-dependency test.

A destination cell that is encountered for the first time in a triplet that passed the data-dependency test will trigger the insertion of a new entry in the D-TABLE. The index in the D-TABLE where the destination cell is inserted will be used as an identifier, henceforth referred to as ID, for the destination cell under discussion. Every destination cell under processing has therefore a unique identifier that will be sent along all the triples containing the same destination cell. Also, when a destination cell is inserted in the D-TABLE a certain amount of "context information" is set up for it. This "context information" is stored in an internal memory, referred to as the context memory and represented in the block diagram as the CTX-memory. In the targeted Virtex family FPGAs the CTX-memory is implemented with `RAMB4_S16_S16` primitives. The CTX-memory stores for each unique destination cell information about: the physical memory address of the non-terminals list in the chart table, the physical memory address of the guard-vectors in the chart table, the number of non-terminals and so on. A processor receives the ID with the triplet to process and when finishes the processing it forwards the ID along with the processing results to the WRITER. The later uses this ID for retrieving the "context information" from the CTX-memory in order to store the processing results in the associated destination cell.

The first empty entry in the D-TABLE is allocated to the CHECKER whenever a new destination cell is inserted. If there are no free entries in the the D-TABLE the CHECKER will wait until an entry is freed. This will happen as soon as a destination cell was processed.

The output of the CHECKER unit is forwarded to the DISPATCHER unit through a FIFO memory for decoupling the two units. The size of this FIFO memory is configured in the VHDL code with the generic `OUT_CHK_DEPTH`. Ideally this FIFO memory should be never empty – nor full – such that the DISPATCHER will always have tasks to distribute to the idle processors.

5.3.3 The triplets buffer (POOL) unit

A triplet coming from the sequence generator and containing a source cell that is in the D-TABLE (i.e. means that the source cell is a destination under processing) will not pass the data-dependency test performed by the CHECKER. In this case the task associated with the triplet cannot be dispatched to the processors. As already mentioned, in this case, the CHECKER will store the problematic triplet in a buffer, represented in the figure 5.1 as the POOL unit. This unit is detailed in figure 5.4 and has two tasks: (1) to store the triplets that did not pass the data-dependency test and (2) to further periodically check all the stored triplets for data-dependency against the D-TABLE. In the later case, as soon as a triplet in the POOL satisfies the data-dependency test, it will be forwarded back to the CHECKER and treated by the later with higher priority over the triplets coming from the sequence generator. The POOL unit is implemented as a pile of registers (represented in figure 5.4 as $\mathtt{R}_0$, $\mathtt{R}_1$,..., $\mathtt{R}_{\mathtt{n}-1}$) whose size is configured in the VHDL code with the generic `POOL_ENTRIES`. Each such register can store a triplet. During run-time it is possible that the register pile is full in which case no more triplets can be stored and eventually the CHECKER will have to wait, until an entry in the register pile is freed and can be reused.

The POOL unit implements the following functionalities (the signals related to each functionality are grouped in the bottom of the figure 5.4) in parallel:

- write a triplet: requested by the CHECKER for storing a triplet. A write takes always place in the top of the register pile pointed by the write pointer `writePTR` which is incremented after each write operation. Initially (i.e. after reset and before each parsing), the top of the register pile is the register $\mathtt{R}_0$ and after the first write the top is the register $\mathtt{R}_1$ and so on. The flag `fullPOOL` is used to indicate that there is no more place left in the register pile. A write should not take place if the `fullPOOL` flag is active and it is the CHECKER's task to verify the `fullPOOL` flag before writing a triplet in the POOL.

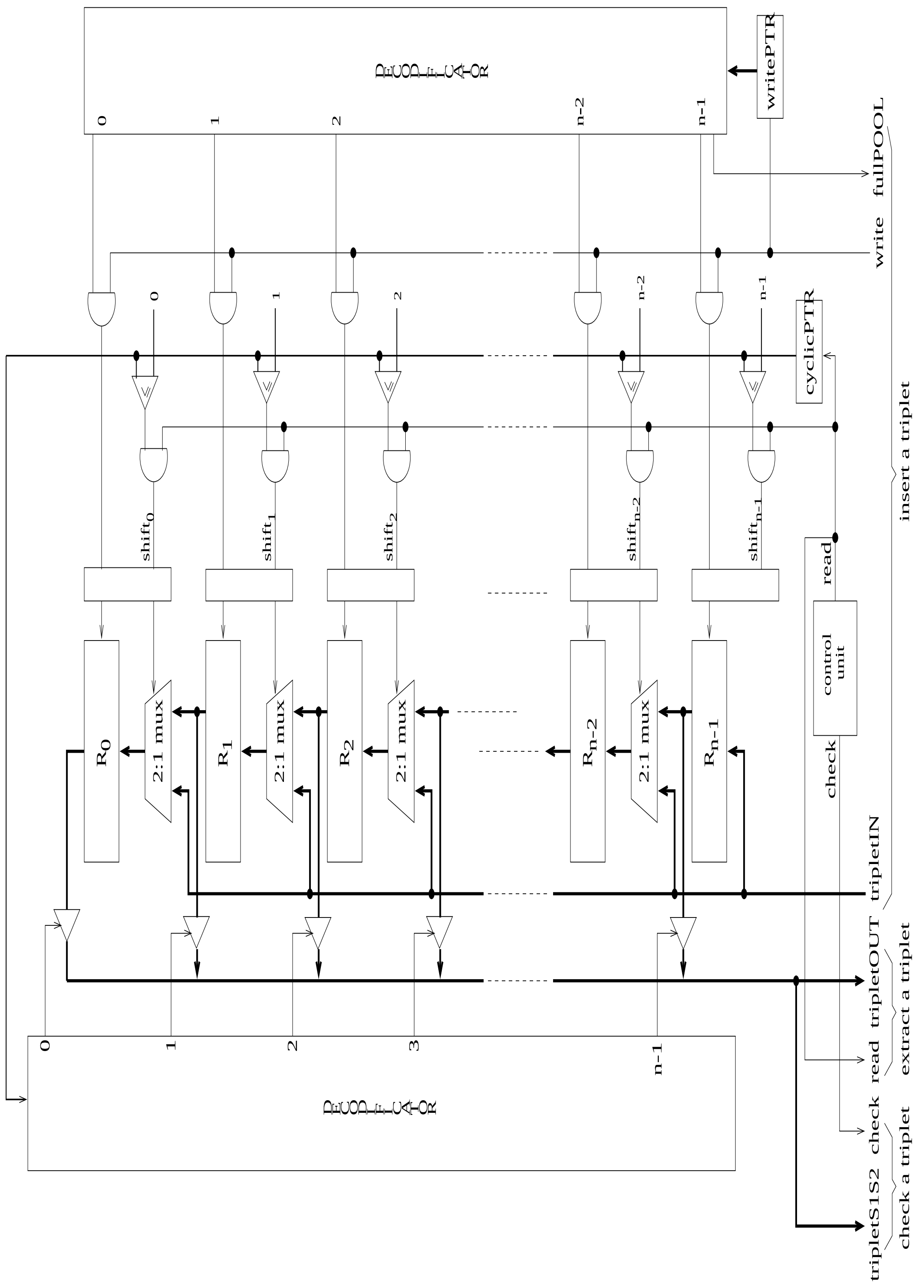


Figure 5.4: The POOL unit datapath

The triplet to be stored in the POOL is placed on the input `tripletIN` and is written in the register pointed by `writePTR` when the signal `write` is activated.

- cyclic triplets check: for a fast retrieval (i.e. efficient) of the triplets in the register pile during the cyclical check against the D-TABLE, the triplets are kept contiguous in the registers $\mathtt{R}_0$, $\mathtt{R}_1$,..., up to the temporary top register of the register pile – pointed by `writePTR`. With this assumption – requiring a careful implementation of the read operation (see the extract operation bellow)– it is possible to access the triplets in the register pile by using a counter `cyclicPTR` that counts cyclically from 0 (when addressing register $\mathtt{R}_0$) to the value of the temporary `writePTR` corresponding to the top of the register pile.

  A triplet is checked against the D-TABLE under the control of the local control unit. The signals used for interfacing with the D-TABLE are the two source cells in a triplet available on the output signal `tripletS1S2` and the `check` signal that will initiate the search in the D-TABLE.

- extract a triplet: a read on the register pile is performed each time a triplet passes the data-dependency test and is returned to the CHECKER. A read can take place anywhere in the register pile as there is no rule on the order in which the triplets pass the data-dependency test. Therefore, read operations may cause gaps in the register pile that are eliminated by performing shifts in order to keep the stored triplets contiguous (required for an efficient hardware implementation of the cyclic check).

  A read is immediately performed on a register that passed the data-dependency check against the D-TABLE and which is therefore pointed by `cyclicPTR`. The `cyclicPTR` pointer is used to activate the shift signals $\mathtt{shift}_\mathtt{i}$, for all values of $i \in \{\mathtt{cyclicPTR}, \ldots, n-1\}$. If the signal $\mathtt{shift}_\mathtt{i}$ is activated the content of register $\mathtt{R}_{\mathtt{i}+1}$ is moved to register $\mathtt{R}_\mathtt{i}$. The `read` signal is generated by the local control unit and means that the register pointer by `cyclicPTR` is available and stable at the output `tripletOUT`.

Note that, during run-time it is possible to have concurrent write and read operations on the register pile. The current hardware implementation can deal with this situation without having to arbiter the concurrent accesses. In order to illustrate the working of the POOL let's consider the following example:

Example 5.3 For a register pile of size $n = 8$, with the initial content illustrated in figure 5.5(a) we illustrate the content of the registers after: the read of the triplet `t2` in figure 5.5(b), the write of the triplet `t5` in figure 5.5(c), and a concurrent read and write of the triplets `t1` and respectively `t6` in figure 5.5(d).

In the current design a FIFO memory is used for decoupling the POOL and CHECKER units. The size of this FIFO memory is configured in the VHDL code with the generic `OUT_POOL_DEPTH`. A small size (e.g. 2-4) should be sufficient.

5.3.4 The task dispatching (DISPATCHER) unit

A triplet that passed the CHECKER's data-dependency test will be allocated to an idle processor for processing. The allocation of triplets (i.e. cell-combinations) to idle processors is the task of the DISPATCHER unit (see figure 5.1). The state of a processor in the system is either idle

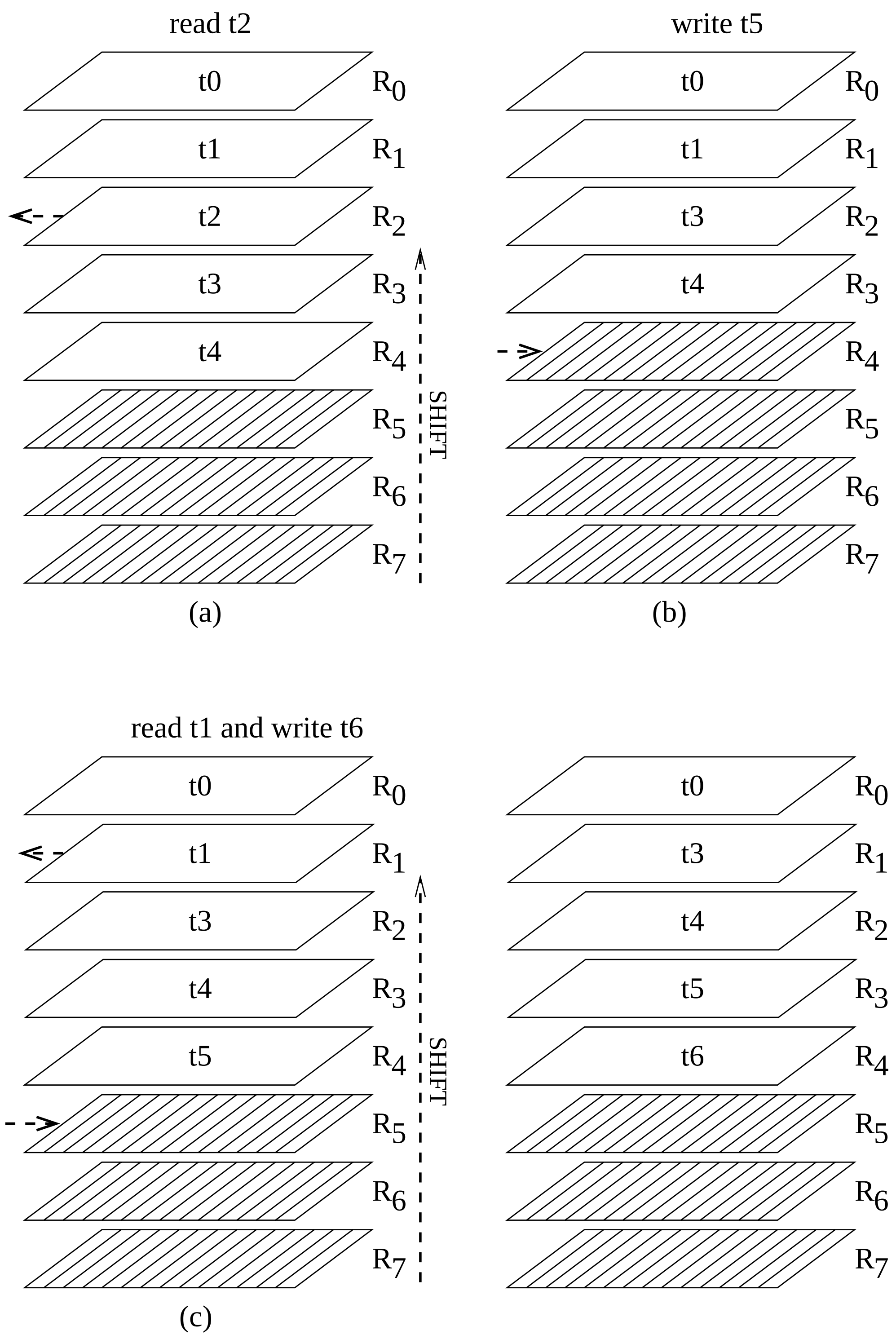


Figure 5.5: Typical POOL operations on the register pile: read (read *t2*), write (write *t5*) and concurrent read/write (read *t1*, write *t6*)

or busy, and the precise task of the DISPATCHER unit is to find an idle processor to which the triplet can be dispatched. It may be that all the processors in the system are busy in which case the DISPATCHER unit will have to wait until a processor becomes idle. The mechanism implemented in the current design for finding an idle processor is a continuous polling over all the processors in the system. Even though the pooling implementation has in general a bigger latency when compared to an equivalent asynchronous implementation it does not depreciates the global system clock when the number of processors in the system grows. It is precisely for this reason that the pooling was adopted.

The triplet dispatching solution implemented within the current design is illustrated in figure 5.6. The state of processor $\mathtt{P_i}$ is available on its output signal $\mathtt{stateP_i}$ ('1' if busy, '0'

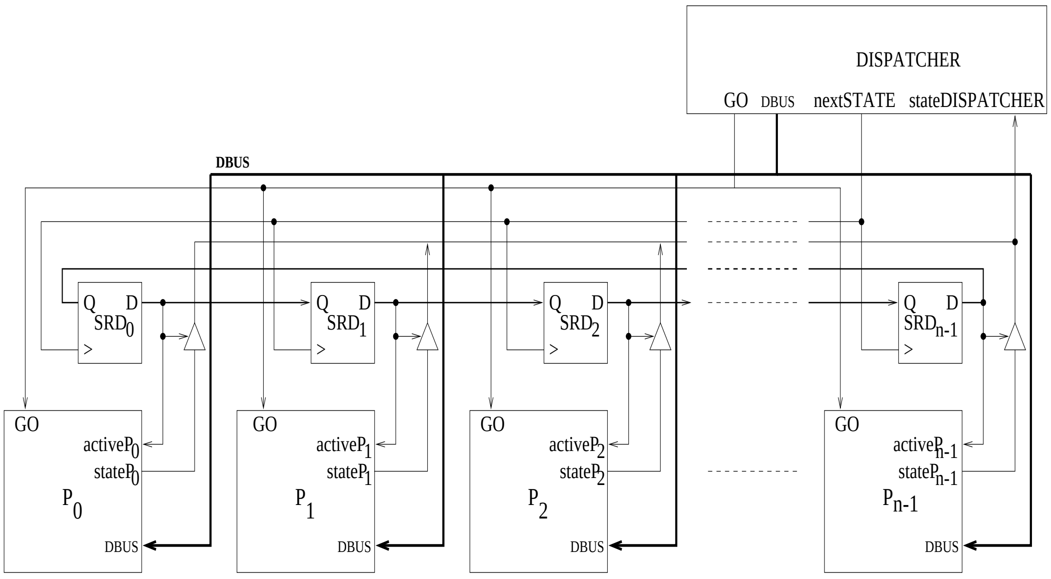


Figure 5.6: The interface between the DISPATCHER and the processors. Triplet dispatching solution implemented in the DAP design.

when idle) and further to the DISPATCHER on the `stateDISPATCHER` input. Only one processor state is available to the DISPATCHER at a given moment on the `stateDISPATCHER` input and corresponds to the processors whose output tri-state gate is driven open (only one tri-state is open at a time). The tri-state gate on the output of processor $\mathtt{P_i}$ is driven by the flip-flop $\mathtt{SRD_i}$ of the rotating shift-register $\mathtt{SRD_{0:n-1}}$. After system reset, only the flip-flop $\mathtt{SRD_0}$ of the rotating shift-register is set on '1', all the others on '0'. The content of the shift-register is rotated one position each time the signal `nextSTATE` is activated by the DISPATCHER and the bit set on '1' will drive open one after the other the tri-state gates of the processors, making available their state information to the DISPATCHER. The DISPATCHER keeps activated the signal `nextSTATE` until it finds an idle processor.

When the DISPATCHER finds an idle processor $\mathtt{P_i}$, it sends the triplet along with its associated ID to the processor using the bus `DBUS` and activates the signal `GO` that will start the processing for the processor. A processor $\mathtt{P_i}$ knows that the information on the `DBUS` refers to it from the signal $\mathtt{SRD_i}$ (available on the input $\mathtt{activeP_i}$) that is used to identify the processor that communicates with the DISPATCHER.

When dispatching a triplet, the DISPATCHER will also update the information in the D-

TABLE, associated with the destination cell corresponding to the ID that tags the triplet (see section 5.3.6 for details).

5.3.5 The WRITER unit

Within the current design the processors do not access the chart memory directly in order to store the the processing results of the cell-combination they perform. Instead, they use the WRITER as an interface for storing the processing results in the chart memory. Such a mechanism is supposed to reduce the number of collisions when accessing the chart memory. For fetching the processing results from the processors, the WRITER is using a pooling mechanism similar to the one implemented for the DISPATCHER and continuously checks the processors for two conditions: (1) whether a processor's processing results, require to be flushed and (2) the processor has finished the processing.

In the first case the WRITER checks whether a processor requires to flush its output buffer (i.e. a FIFO memory, see section 5.3.7) containing the (partial) cell-combination result. A processor requests flushing each time the output buffer is filled, or almost filled with processing results. Note that a processor requesting flushing did not necessarily finish the cell-combination it is working on. A processor with a filled output buffer stays idle until is flushed and therefore triggering the flushing before the processor's output buffer becomes full would be better. If a processor's flush request is triggered when the output buffer is $3/4$ full for instance, this will hopefully overlap the processor's working with its flushing.

For flushing a processor's output buffer, the WRITER needs the "context information" of the destination cell the flushed processor is working on. Recall that the context information for each distinct destination cell under processing is stored in the CTX-memory and that this information can be retrieved by using the unique identifier ID that the DISPATCHER sends along a triplet to a processor. Each processor stores the ID of the destination cell it is working on in a local register and the WRITER retrieves this ID each time it communicates (i.e. flushes the results) with a processor. Using this ID the WRITER reads the CTX-memory and sets up the environment (i.e. physical addresses uses for accessing the chart memory,...) for the destination cell in which the flushed data will be stored. The environment setup will be referred henceforth to as a context switch. Note that when flushing a new processor it may be the case that the context does not need to be switched. This is the case when the new processor to be flushed has the same ID, and therefore works on the same destination cell as the previously flushed processor. Once the processor's output data is fetched and stored it in the chart memory the WRITER updates the destination cell context information and rewrites it back in the CTX-memory. At this point the WRITER will continue to pool the processors.

The second check the WRITER performs on a processor is to see whether the processor has finished to process the assigned cell-combination or not. The WRITER requires to know if a processor has finished the processing in order to update the D-TABLE and keep the data-dependency information up-to-date. It may be the case that a processor has finished the processing and also has some processing results to be flushed in which case a flush step will take place as explained above. If the processor has finished the processing and there are no processing results to be flushed – or the processing results have been already flushed – the WRITER checks whether the processed cell-combination was the last for the destination cell identified by the ID. If this is the case the WRITER will also release the D-TABLE entry and the CTX-memory information associated with the ID.

The system described above is illustrated in figure 5.7. A processor's state is available on its output signal $\mathtt{stateP_i}$ and further to the WRITER on the input `stateWRITER`. Only one

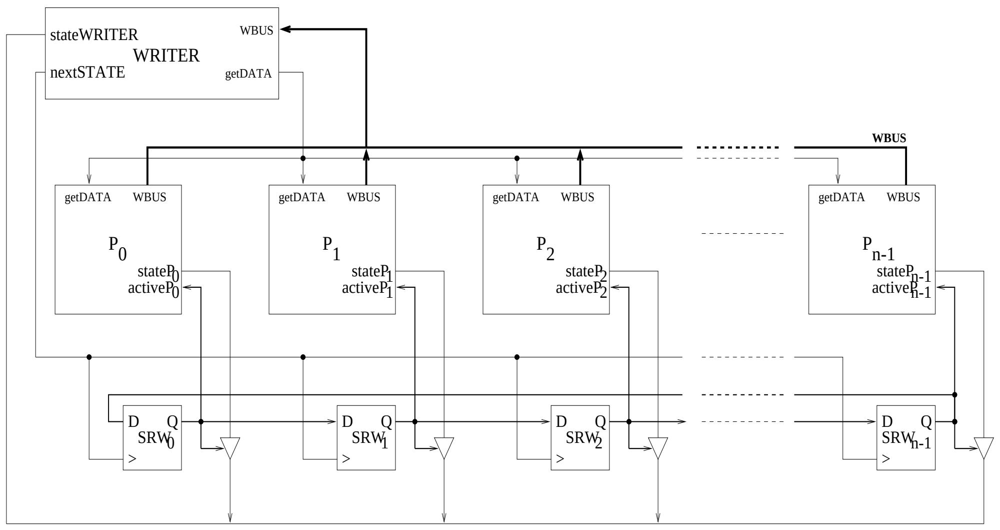


Figure 5.7: The interface between the WRITER and the processors. Parsing results flushing solution implemented in the DAP design.

processor state is available to the WRITER and corresponds to the processor whose output tri-state gate is driven open at that moment. The tri-state gate on the state output of processor $\mathtt{P_i}$ is driven by the flip-flop $\mathtt{SRW_i}$ of the rotating shift-register $\mathtt{SRW_{0:n-1}}$. After system reset, only the flip-flop $\mathtt{SRW_0}$ of the rotating shift-register is set on '1', all the others on '0'. The content of the shift-register is rotated one position each time the signal `nextSTATE` is activated by the WRITER and the bit set on '1' will drive open one after the other the tri-state gates of the processors, making available their state information to the WRITER. The WRITER keeps activated the signal `nextSTATE` until it finds a processor that requests flushing and/or has finished the processing.

When the WRITER finds a processor requiring to be flushed it will use the signal `getDATA` and the bus `WBUS` for fetching the processor's output data. The processor $\mathtt{P_i}$ knows that it communicates with the WRITER – that it is flushed in particular – from the signal $\mathtt{SRW_i}$ available on its input $\mathtt{activeP_i}$.

The WRITER unit implements the same $F2$ and $F3$ functionalities that were presented in section 3.4.1. The datapath diagram of the WRITER given in figure 5.8 illustrates in greater detail the internals of the WRITER unit. Each word fetched from a processor consists of an identifier ID, a left-hand side (`LHS`) and two right-hand sides (`RHS1` and `RHS2`) producing the left-hand side[1]. The fetched ID is used by the context setup module for switching the context (if necessary). The context switch corresponds to the initialization of the registers `Dindex`, `Dbase` and `Guardbase` with new values `Dtail`, `Phead` and respectively `Pguard`. For details about these values see section 3.4.1). All these values are fetched from the CTX memory – where they were stored by the CHECKER.

The WRITER writes sequentially the left-hand side non-terminals `LHS` fetched from the processor's output buffer in the destination cell at the physical address ($\mathtt{Dbase} + \mathtt{Dindex}$) and after each write the index `Dindex` is incremented to point at the next non-terminal. The displacement `Dtail` (stored in `Dindex`) is always 0 in case of sentence parsing. However, in

[1]Within the current design the right-hand sides are further ignored. However, they are available for extracting on-line the parsing forest. Not implemented within this design version.

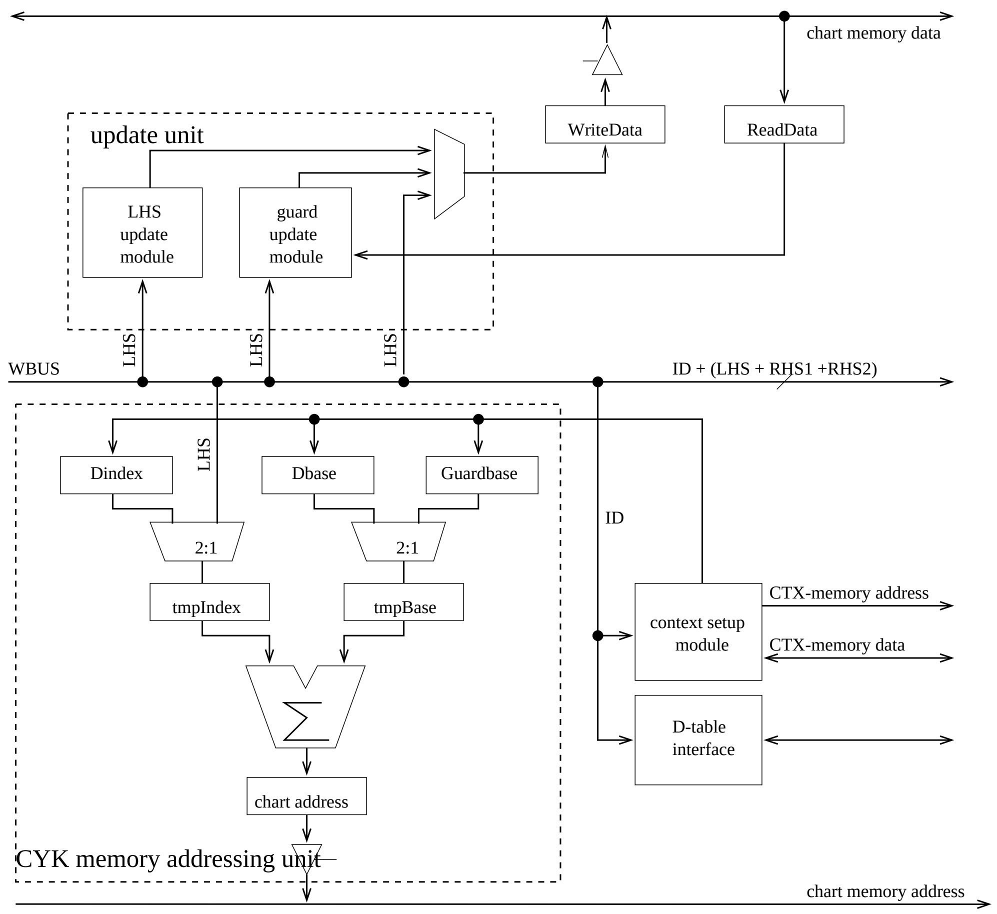


Figure 5.8: The WRITER unit datapath.

the case of word lattice parsing it may be different than 0.

The WRITER accesses the guard-vectors table both for read and write. In hardware, the physical memory address for accessing a particular guard-bit in the guard-vectors table is constructed from the registers `Guardbase` and `LHS` as (`Guardbase` + `LHS`), where `LHS` is actually the binary representation of the non-terminal we want to check (read) or set (write).

Each time the WRITER finishes with a processor it restores the context information (i.e. the updated content of the `Dindex`, `Dbase` and `Guardbase` registers) in the CTX-memory. If the flushed processor has finished the processing, the WRITER will also update the information in the D-TABLE associated with the destination cell corresponding to the processor's ID (see section 5.3.6 for details).

5.3.6 The destination cells table (D-TABLE) unit

The LAP design uses a row-by-row method for filling the chart which insures – by default – run-time cell data-dependency satisfaction. The row-by-row method however yields a low processor utilization and for this reason and others the dynamic allocation method presented in appendix B.2 is implemented within the current design. As with the dynamic allocation method the processors run freely for filling the chart without being synchronized during run-time, a mechanism that can keep track of the chart's cells data-dependency is required. Essentially, the chart's cells data-dependency is guaranteed to be satisfied if only triplets containing source cells that are not destination cells under processing are sent (i.e. dispatched) to processors for processing. Whenever a triplet containing a source cell that is a destination cell under processing is encountered it will be temporarily put aside (i.e. stored in a buffer) until the source cells it relies upon become available. The triplets stored in the buffer (i.e. the POOL unit, see section 5.3.3) will be sent to processors for processing as soon as they satisfy the data-dependency they failed to pass previously. Such a mechanism is implemented within the current design with the aid of the D-TABLE unit (see figure 5.1).

The D-TABLE comes from *d*estination (cells) *table* and stores a list of all destination cells under processing at a given moment. A destination cells is inserted in this table when encountered for the first time in a triplet that satisfies the data-dependency check and deleted when all the triplets issued for its processing have been processed and their processing results have been flushed and written in the chart. As there are two units that perform data-dependency checks on the D-TABLE – the POOL and the CHECKER units – the D-TABLE should handle concurrent checks. Given these considerents the D-TABLE should implement and support the following functionalities:

- data-dependency check: the D-TABLE is organized as an associative memory and therefore a look-up for a source cell takes place in parallel over all the entries of the D-TABLE. As we saw in section 5.3.2 and 5.3.3 both the CHECKER and the POOL units may perform data-dependency checks on the D-TABLE at any moment in time and therefore in order to achieve maximum efficiency the D-TABLE should handle concurrent accesses without having to arbiter them.

  The circuitry used by the CHECKER and POOL units to perform data-dependency checks on the D-TABLE is illustrated in figure 5.9 – for a D-TABLE that can store n destination cells at once. In the figure there are two groups of signals: one used by the CHECKER (on the left) and the other one by the POOL (on the right) for communicating with the D-TABLE during a data-dependency check. In the figure certain signals have the same denomination as they have the same functionality both for the CHECKER and the POOL unit and this should not be confusing in the text that follows.

  On the right are the signals used by the POOL to perform a data-dependency check for a triplet $(S1, S2, D)$. In a POOL's check query only the source cells $(S1, S2)$ are important, the destination cell being irrelevant as the POOL only has to verify that the source cells are not destination cells under processing. The POOL starts the data-dependency check by activating the `startPOOL` signal. The result of the performed data-dependency check is available on the outputs `resultS1` and `resultS2` that are valid when the signal `resultREADY` is activated. When the `startPOOL` signal is activated the POOL control unit (bottom right) will first activate the `samplePOOL1` signal to latch the state of each D-TABLE entry – that can be either used or free. Next, the POOL control unit will activate the `samplePOOL2` signal for latching, for each D-TABLE entry, the results of the lookup performed for $S1$ and respectively $S2$. The result of a lookup for a D-TABLE entry is validated if that entry is not free. Finally two wide-OR gates will output the lookup result over the entire D-TABLE for $S1$ and respectively $S2$. The signal `resultREADY` is activated to mark the availability of the result, three clock ticks later after the `startPOOL` signal was activated.

  On the left are the signals used by the CHECKER to perform a data-dependency check for a triplet $(S1, S2, D)$. In a CHECHER's query both the source cells and the destination

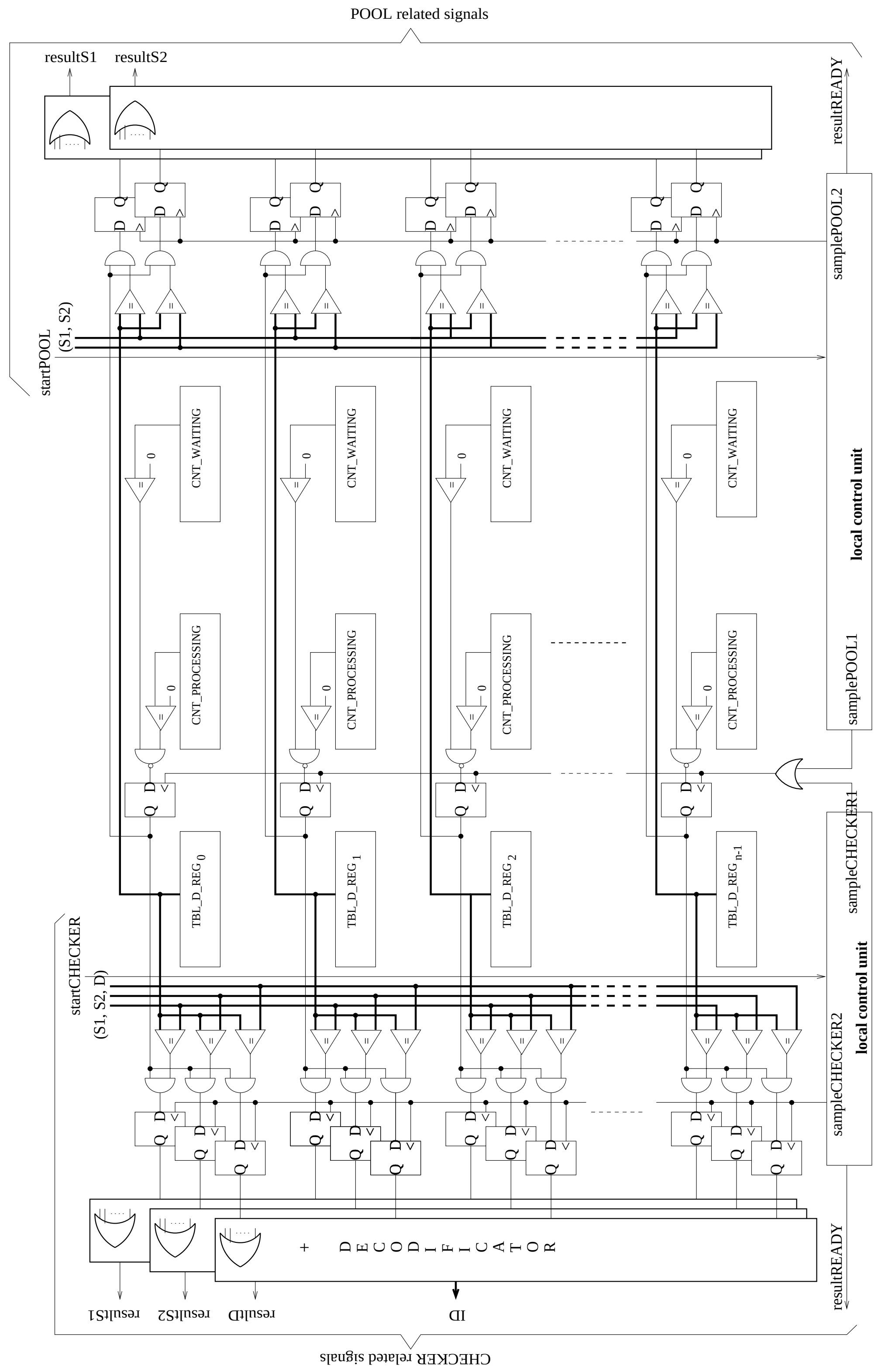


Figure 5.9: The D-TABLE unit circuitry used for performing a data-dependency check.

cell are looked-up in the D-TABLE. The source cells are looked-up for performing the data-dependency check while the destination cell is looked-up for knowing whether it is encountered for the first time in which case it has to be inserted in the D-TABLE. On the other hand, if the destination cell is not encountered for the first time the CHECKER has to retrieve the ID (i.e. the index where it is stored in the D-TABLE) associated with it. This ID will be sent along the triplet to the processors. The CHECKER starts the data-dependency check by activating the `startCHECKER` signal. The result of the performed data-dependency check is available on the outputs `resultS1`, `resultS2` and respectively `resultD` that are valid when the signal `resultREADY` is activated. A supplementary signal `ID_CHECKER` is available and outputs the ID if the `resultD` signal is valid. When the `startCHECKER` signal is activated the CHECKER control unit (bottom left) will first activate the `sampleCHECKER1` signal to latch the state of each D-TABLE entry that can be either used or free. Next, the CHECKER control unit will activate the `sampleCHECKER2` signal for latching, for each D-TABLE entry, the results of the lookup performed for $S1$, $S2$ and respectively D. The result of a lookup on a D-TABLE entry is validated if that entry is not free. Finally the wide-OR gates will output the lookup result over the entire D-TABLE for $S1$, $S2$ and respectively D and the decoder will output the `ID_CHECKER`. The signal `resultREADY` is activated to mark the availability of these results, three clock ticks later after the `startCHECKER` signal was activated.

- insertion: only the CHECKER inserts destination cells in the D-TABLE and always in a

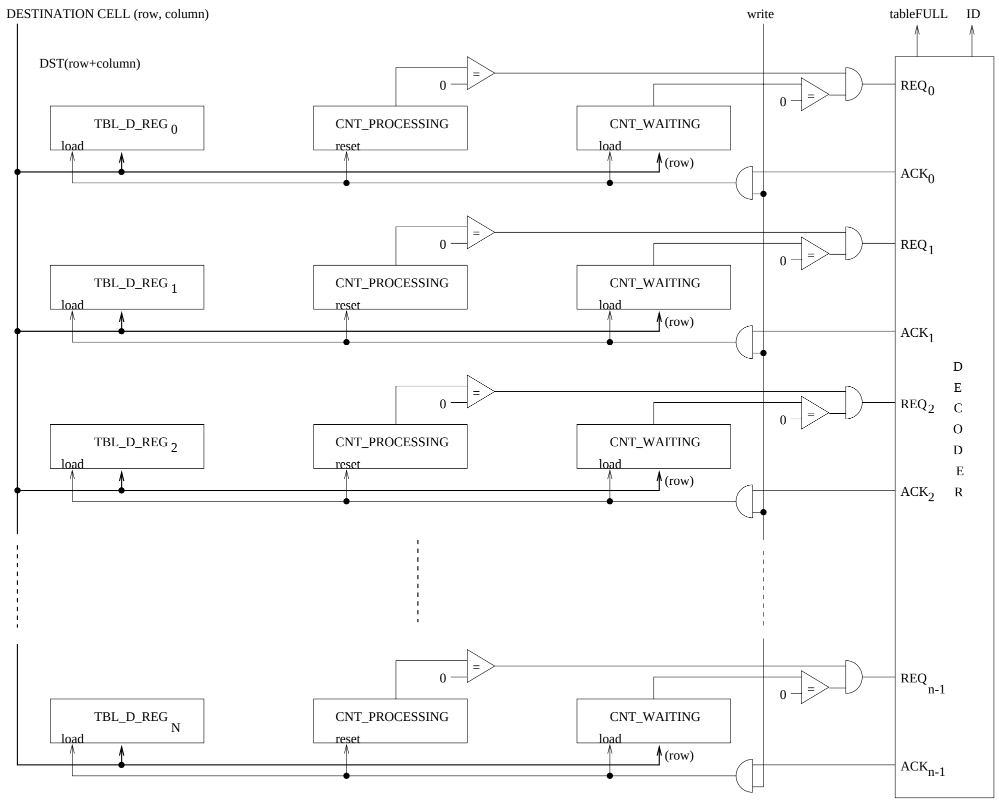


Figure 5.10: The D-TABLE unit, circuitry used for inserting a destination cell.

free entry of the D-TABLE. An entry in the D-TABLE is free if both `CNT_PROCESSING` and CNT_WAITING counters are 0. The index of the D-TABLE entry where a destination cell is inserted will be the unique identifier associated with that destination cell until the destination cell is deleted from the D-TABLE. The circuitry used by the CHECKER to insert a destination cell in the D-TABLE is illustrated in figure 5.10 – for a D-TABLE that can store n destination cells at once.

In this figure the signal used to store a new destination cell is write and the index of the entry where the destination cell was written is returned on output ID. A destination cell can be inserted in any free entry of the D-TABLE. In other words, any priority scheme may be used to allocate an entry when inserting a destination cell. For instance within the current implementation the $\mathtt{REQ}_0$ entry has the highest priority and $\mathtt{REQ}_{n-1}$ the lowest. If during run-time the D-TABLE is filled the signal tableFULL will be activated and in this case the CHECKER will have to wait for an entry to be released. It is precisely the task of the CHECKER to verify that the signal tableFULL is not activated before attempting to insert a new destination cell in the D-TABLE.

When a destination cell is inserted in the D-TABLE, two counters are initialized that entry. These two counters are necessary for the WRITER to detect when the system has finished the processing of the associated destination cell. The counters CNT_WAITING represents the number of cell-combinations left for processing and the counter CNT_PROCESSING, represents how many cell-combinations are currently under processing. When a destination cell is inserted in the D-TABLE the CNT_WAITING counter is initialized with the row of the same destination cell as there are exactly as many triplets to process for filling that destination cell. The counter CNT_PROCESSING is initialized on 0. A destination cell is processed (i.e. filled) as soon as both counters are zero.

- deletion: the deletion of an entry in the D-TABLE is done by the WRITER, but the delete operation does not take place explicitly on the D-TABLE. Each time the WRITER finds a processor that has finished the processing it decrements the counter CNT_PROCESSING. If both the CNT_PROCESSING and CNT_WAITING are 0 for an entry, that entry will be automatically released – which corresponds to a delete.

- update: an update is operated on a D-TABLE entry each time the DISPATCHER sends a triplet to a processor or when the WRITER finds a processor that has finished the processing. Both the DISPATCHER and the WRITER are using the identifier ID sent along the triplets for updating an entry in the D-TABLE. Precisely, each time the DISPATCHER sends a triplet for processing, the counter CNT_WAITING is decremented and the counter CNT_PROCESSING is incremented. Once the processor that processed the same triplet has finished the WRITER will decrement the counter CNT_PROCESSING. And as we already said when both the counters become 0 (i.e. when the signal finishedCELL is activated) the associated entry in the D-TABLE is released. The circuitry used by the DISPATCHER and the WRITER to update a D-TABLE entry is illustrated in figure 5.11 – for a D-TABLE that can store n destination cells at once. The WRITER uses the ID_WRITER to select the D-TABLE entry it wants to update and the DISPATCHER uses the ID_DISPATCHER for the same purpose. Both the WRITER and the DISPATCHER may concurrently update the same entry (or different entries) of the D-TABLE.

  The signal incdecDISPATCHER is used by the DISPATCHER and the decWRITER signal is used by the WRITER to update the counters as described above. The signal finishedCELL is used by the WRITER to detect a destination cell that was processed (i.e. filled) and whose associated entry in the D-TABLE should be released.

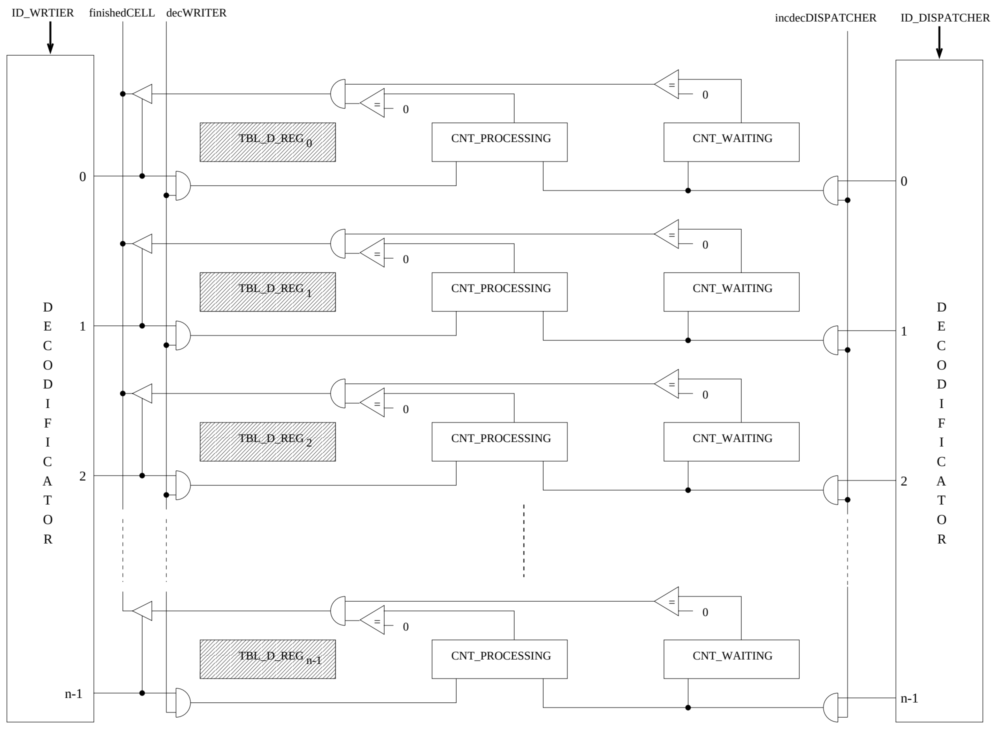


Figure 5.11: The D-TABLE unit, circuitry used for updating a D-TABLE entry.

The size of the D-TABLE can be configured in the VHDL code with the generic `ENTRIES`. Note, that in the current system the number of distinct destination cells under processing at the same time cannot exceed the number of processors in the system. Therefore, there is no point in having a D-TABLE with more entries than the number of processors in the system – which is a waste of hardware resources.

5.3.7 The processor

The figure 5.12 illustrates the processor datapath. A processor receives the processing tasks (i.e. triplets) from the DISPATCHER which also activates the signal `GO` (see section 5.3.4) that initiates the processing. Based on the triplet received from the DISPATCHER on the bus `DBUS`, the processor will access the chart memory (see section 3.5.1.1) in order to fetch the non-terminals in the source cells. The interface with the grammar memory is the same as for the processors used in the LAP design (see section 3.5.1.2). Each time two non-terminals `RHS1` and `RHS2` in the source cells are producing the left-hand side `LHS`, the item (`RHS1`, `RHS2`, `LHS`) is stored in the output FIFO buffer. When the output FIFO is full (or almost full) the flush request module asserts the $\mathtt{stateP_i}$ signal to request the output FIFO flush. The WRITER will flush the processor's output FIFO – using the `WBUS` – as soon as it pools the processor and detects the request. The $\mathtt{stateP_i}$ signal is also used to signal that a processor has finished the processing of the cell-combination.

A processor whose output buffer is full cannot store its processing results and stays idle until the output buffer is flushed. Therefore, requesting the flush of the output FIFO when it is almost full will eventually eliminate the time lost by a processor waiting for the results to be flushed. A processor that has finished the processing and is flushed, becomes immediately

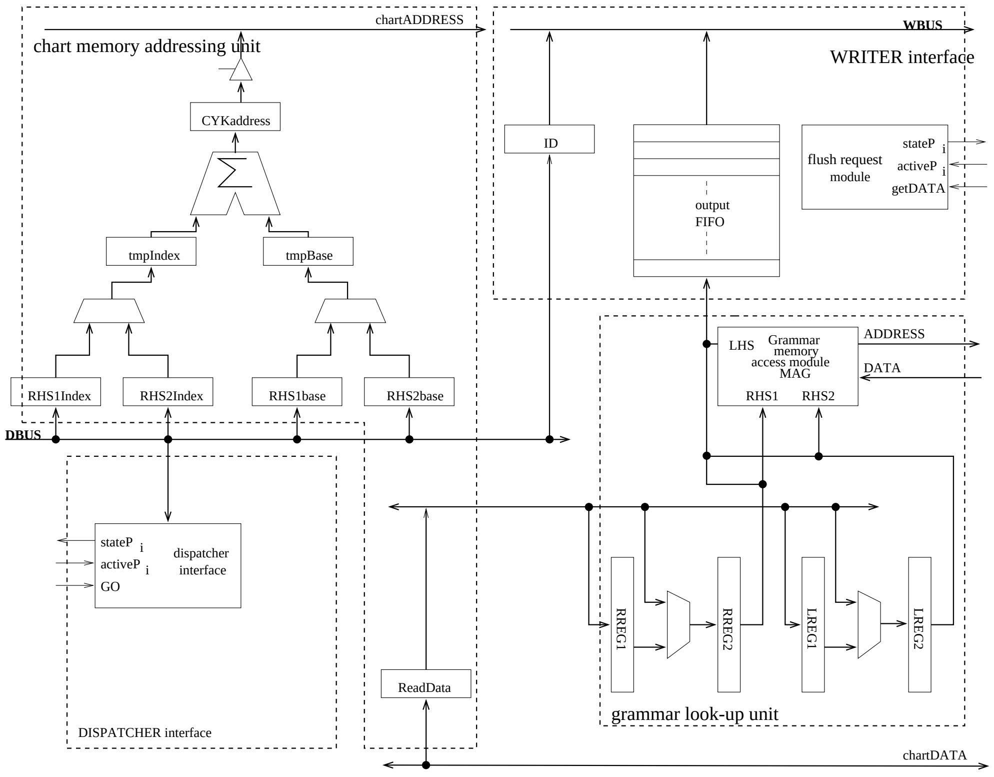


Figure 5.12: The datapath of the processor used in the dynamic array of processors (DAP) architecture.

available for processing other triplets.

5.4 Performance measurements

The tests and performance measurements presented in this section were performed on the same CNF SUSANNE grammar – containing $10,129$ non-terminals and $74,350$ rules – used to benchmark the previous design (see section 3). The data-structures representing the CNF SUSANNE grammar and the chart in the current design are the same as those used for the LAP design. The size of the memory required to store the CNF SUSANNE grammar data-structure is $558,576$ bytes and the size of the chart memory depends on the length of the sentence we want to parse (e.g. $446,496$ bytes for parsing sentences with up to 10 words or $1,733,696$ bytes for parsing sentences with up to 32 words).

In contrast to the previous design, the maximal length of the sentences that can be parsed is independent of the number of processors in the system. In other words, any number of processors can be used for parsing sentences of any length in the condition that the amount of memory available for storing the chart is enough (e.g. for 32 words requires about 2 [MBytes] and for 128 words requires 26 [MBytes]).

In order to determine the size and the clock frequency at which the system is able to work, a 3-processors system configuration was synthesized[2] and placed&routed[3] in a Xilinx FPGA,

[2]With LeonardoSpectrum v2000.1a2
[3]With Design Manager (Xilinx Alliance Series 2.1i)

Virtex XCV1000. The synthesis of the 3-processors system was made on a design instance characterised by the parameters given in table 5.1 and was physically tested and checked for correctness on a RC1000-PP FPGA board with a clock frequency of 50 MHz. The tested system was restricted to 3 processors as there are only 4 memory banks on the RC1000-PP FPGA board (1 bank allocated for the chart memory and the rest of 3 banks allocated for the grammar memories).

For details regarding the grammar memory parameters see section 3.4.2 and appendix A.4, for the chart memory parameters see section 3.4.1 and for the other parameters in the table see the DAP design block diagram (figure 5.1). The table 5.4 gives for a Xilinx Virtex XCV1000 FPGA a summary of the resources used by each unit in the synthesized 3-processors system. The overall amount of resources required by a 3-processors system and for a 10-processors system are also given.

The 3-processors system synthesized with the "extract RAM" feature enabled, uses less than 22% of the available FPGA resources and allows us to parse sentences of any length if there is enough room in the chart memory.

Due to the drastic restriction on the number of processors imposed by the RC1000-PP board the hardware run-times we present were obtained by simulating[4] the VHDL model of a system with 4, 7, 10 and respectively 14 processors. The POOL size used in these simulations gave the best time results on the $2,043$ sentences of length 3 to 15 from the SUSANNE corpus, on a 10-processors system (see section 5.6.3).

The software used for comparison is an implementation of the enhanced CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The hardware performance (i.e. run-times) of a system containing 4 processors (denoted as `hard_P4`), 7 processors (denoted as `hard_P7`), 10 processors (denoted as `hard_P10`) and respectively 14 processors (denoted as `hard_P14`) was compared against two software run-times. The first (`soft_CNFG`) uses the SUSANNE grammar in CNF, as it is also the case for the hardware. The second (`soft_CFG`) uses the SUSANNE grammar in its original context-free form. The software was run on a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* `C` library function and not by profiling the code.

For the purpose of the comparison, $2,043$ sentences were parsed and validated[5]. The sentences have a length ranging from 3 to 15 and were all taken from the SUSANNE corpus. Figure 5.13 shows the average run-times `soft_CFG` and `hard_P10` as functions of the sentence length (vertical axe). The run-time `soft_CNFG` was not presented as it is slower than the run-time of `soft_CFG` and therefore not relevant. The average speedup factor $E(S)$ has been computed for the 10-processors system against both the `soft_CNFG` and `soft_CFG`. For `soft_CNFG`, $E_{\texttt{soft\_CNFG}}(S) = 206.45$ and for `soft_CFG`, $E_{\texttt{soft\_CFG}}(S) = 31.90$. Figure 5.14(a) shows the hardware speedup, for the `hard_P4`, `hard_P7`, `hard_P10` and `hard_P14` in comparison with `soft_CNFG` and the figure 5.14(b) in comparison with `soft_CNFG` as a function of the sentence length. This figure shows that the best performance when parsing sentences above 10 words is obtained with a `hard_P10` system. The figure illustrates that the overhead required for managing more processors than needed results in a performance deterioration. This is the case for short sentences (e.g. 3 words) when the `hard_P4` system gives best

[4]With ModelSim EE/Plus 5.2e

[5]The hardware output was compared to the software output for detecting mismatches. However, as the validation process consists of many technical details, it is not described here. A detailed description can be found in [9].

Parameter		Size[units]	Details
system	CLOCK	50 [MHz]	the system clock
	COORD_BITS	5 [bits]	number of bits representing a row/column in the chart, see section 5.2
	PROCESSORS	3	the number of processors in the system
chart memory	CYKLATENCY	17 [ns]	the access-time of the SRAM memory used to store the chart
	DTAIL_SIZE	9 [bits]	see section 3.4.1
	CYK_ADR_SIZE	19 [bits]	chart data-structure pointer size, see section 3.4.1
	CYK_WAIT_CYCLES	2	delay until chart memory data lines are stable
grammar memory	GMEMSIZE	558,576 [bytes]	the size of each grammar memory
	GLATENCY	17 [ns]	the access-time of the SRAM memory used to store the grammar memory
	WAIT_CYCLES	2	delay until grammar memory data lines are stable
	NT	10,129	number of non-terminals in the grammar, see section 3.4.2
	NT_SIZE	14 [bits]	bits used to represent a non-terminal, see section 3.4.2
	RULE_SIZE	15 [bits]	bits used to represent a distinct right-hand side, see section 3.4.2
	PTR_SIZE	19 [bits]	grammar data-structure pointer size, see section 3.4.2
other	ENTRIES	31	size of D-TABLE, see section 5.3.6
	OUT_GEN_DEPTH	16	size of FIFO at output of SEQ_GEN, see section 5.3.1
	OUT_CHK_DEPTH	16	size of FIFO at output of CHECKER, see section 5.3.2
	OUT_POOL_DEPTH	16	size of FIFO from POOL to CHECKER, see section 5.3.3
	POOL_ENTRIES	16	size of the POOL, see section 5.3.3

Table 5.1: The parameter values used to configure (i.e. instantiate) the synthesized 3-processors DAP system.

component	DFFs/Latches	FGs	CLBs	XCV1000 area utilization (%)
IOctrl	23	19	12	0.10
SEQ_GEN	57	80	40	0.33
SEQ_GEN out FIFO	422	284	211	1.72
	15	26	13	0.11 ("extract RAM"/see note bellow)
CHECKER	242	210	121	0.98
CHECKER out FIFO	1122	686	561	4.57
	15	33	17	0.14 ("extract RAM")
DISPATCHER	2	4	2	0.02
POOL	416	880	440	3.58
POOL out FIFO	422	284	211	1.72
	15	26	13	0.11 ("extract RAM")
D-TABLE	1129	2205	1103	8.98
CTX	(uses 6 Block SelectRAMs/see Xilinx's XCV1000 Manual [31])			
CHART ARBITER	24	48	24	0.26
WRITER	257	303	152	1.24
MAG	168	296	148	1.20
processor	463	511	256	2.08
3-processors system	3511	5313	2657	21.62
10-processors system	5674	7360	3680	29.95

Table 5.2: Virtex XCV1000 FPGA resource utilization per DAP design unit in terms of Flip-Flops/Latches (DFFs/Latches), function generators (FGs) and configurable logic blocks (CLBs).

Note: The Virtex XCV1000 FPGA has several types of RAM structures available (see XCV1000 manual [31]). The "extract RAM" is a synthesizer feature that allows to infer RAM structures whenever possible.

performance. For sentence lengths between 5 and 10 words, the `hard_P7` system gives best performance. In this case the `hard_P4` system has not enough processors, while the `hard_P10` and `hard_P14` systems have too many processors that are not all used. For sentences with more than 10 words it is the `hard_P10` system that performs best for the same reasons. The figure does not show what happens for sentence lengths above 15 words but is very likely that for a certain sentence length the `hard_P14` system will give best performance .

5.5 Design testing on the RC1000-PP board.

For obtaining the real clock frequency and validating the presented design, a 3-processor system was physically tested on a commercial FPGA board. For this purpose we used Celoxica's RC1000-PP FPGA board whose block diagram is given in section 3.7 in figure 3.11. As the tested system contains 3 processors (and therefore 3 grammar memories) and the chart memory it can be fit in the 4 memory SRAM banks available on the RC1000-PP FPGA board. Each SRAM bank on the RC1000-PP FPGA board has a 32-bit databus that match the current design's databus widths for the grammars and respectively the chart memories. Under these condi-

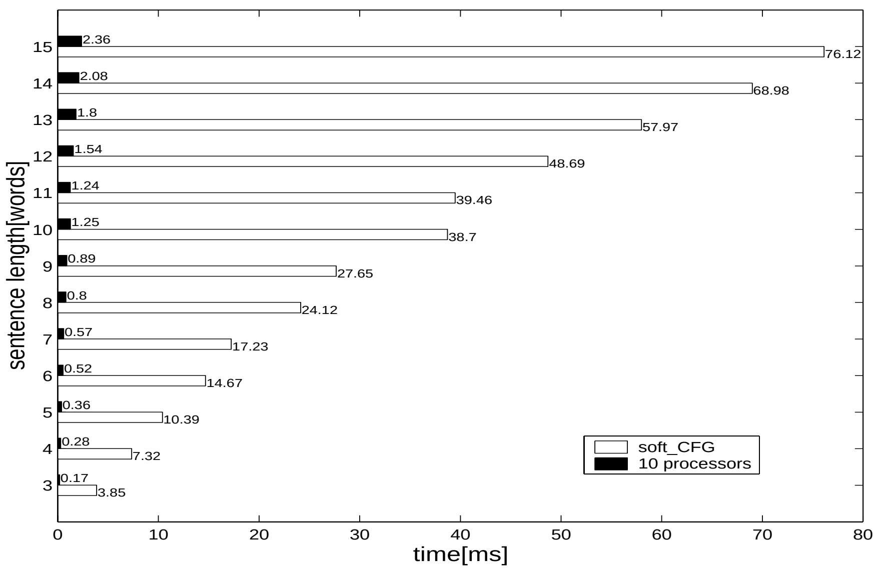


Figure 5.13: Average run-times for `soft_CFG` and `hard_P10` DAP system as a function of sentence length. For each sentence length more than 100 sentences were parsed.

tions, the synthesized 3-processor system can be placed&routed in the Xilinx XCV1000bg560-4 FPGA available on the RC1000-PP board without any further modification.

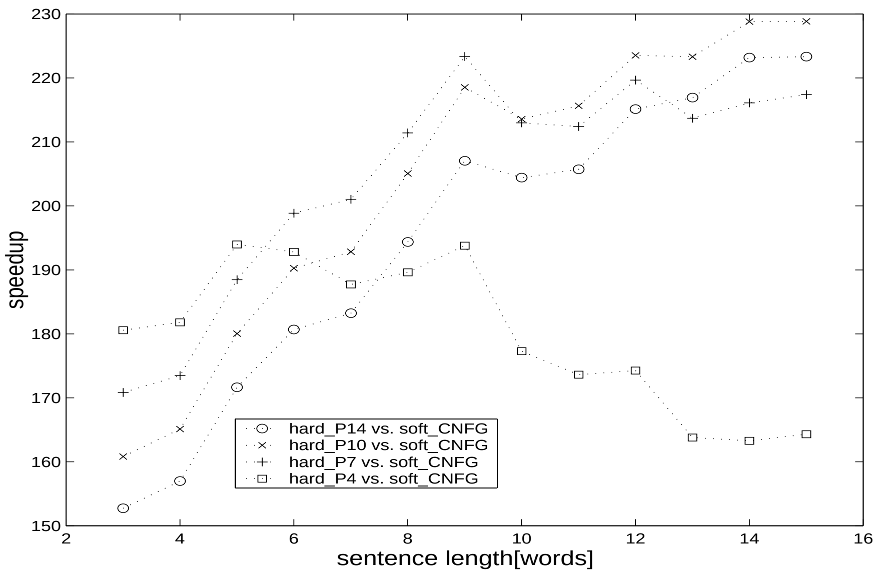
The initialization, running the design and the result readback is done exactly in the same way as for the previous design (see section 3.7). The interfacing signals are the same.

5.6 Dynamic Array of Processors Design Analysis

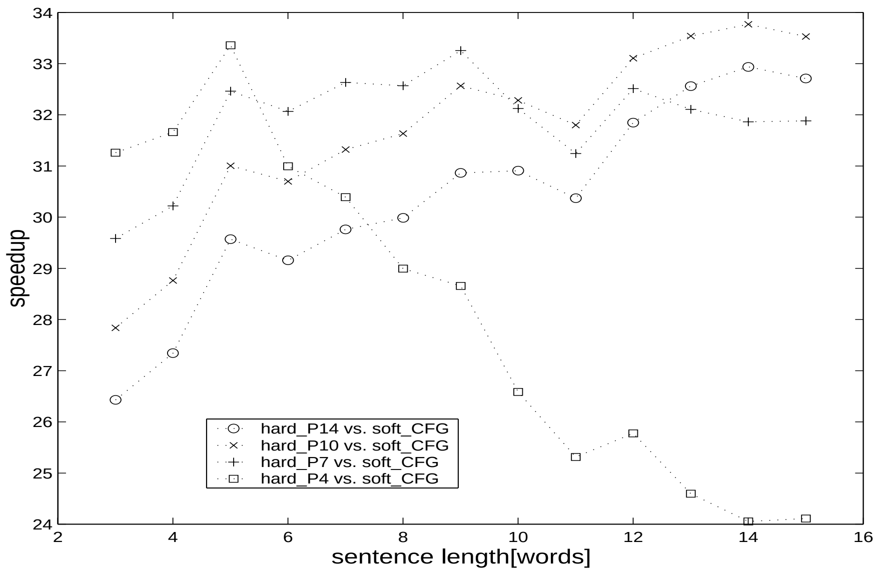
In this section some important aspects of the DAP design are analysed and discussed . First, in order to get a general idea about the design, we look at the processor activity and measure the processor utilization for some arbitrary sentences extracted from the SUSANNE corpus. In order to compare the DAP design with the LAP design we actually use the same sentences that we used for studying the LAP design. More precisely, as the processor activity mainly consists of two tasks (1) data processing and (2) accessing the chart memory data-structure, we investigate – as in the case of the LAP design – the fraction of time spent by the processors in performing each of the two tasks. Second, the effect of an increased number of collisions when accessing the chart memory as the sentence length increases on the overall design performance is investigated. Third, the influence of the POOL unit size on the design performance is investigated.

5.6.1 Average processor utilization

In order to get a general idea about the processor activity during the parsing process we illustrate in figure 5.15(a), (b) and (c) the processor activity for three sentences (given in table 5.3) of length 4, 10 and respectively 15 words. In the figures 5.15(a)-(c) with black is depicted the amount of time spent by the processor for processing data and with gray the time spent for chart memory read accesses or waiting for data to be flushed. In order to have a comparison



(a)



(b)

Figure 5.14: Hardware speedup for the `hard_P14`, `hard_P10`, `hard_P7` and `hard_P4` DAP systems against (a) `soft_CNFG` and (b) `soft_CFG` software as a function of sentence length. For each sentence length more than 100 sentences were parsed.

between the processor activity in the DAP an respectively LAP designs, we use for this purpose the same three sentences illustrated for the LAP design in section 4.1). The speedup factor for both the LAP and DAP designs when parsing these sentences (against the `soft_CFG` software implementation) is also given for comparison in table 5.3. It results from these figures that the

len	sentence	speedup LAP	speedup DAP
4	*“One wing stood open”*	11.98	22.41
8	*“In fact our whole defensive unit did a good job”*	15.38	34.18
15	*“In societies like ours , however , its place is less clear and more complex”*	13.56	36.31

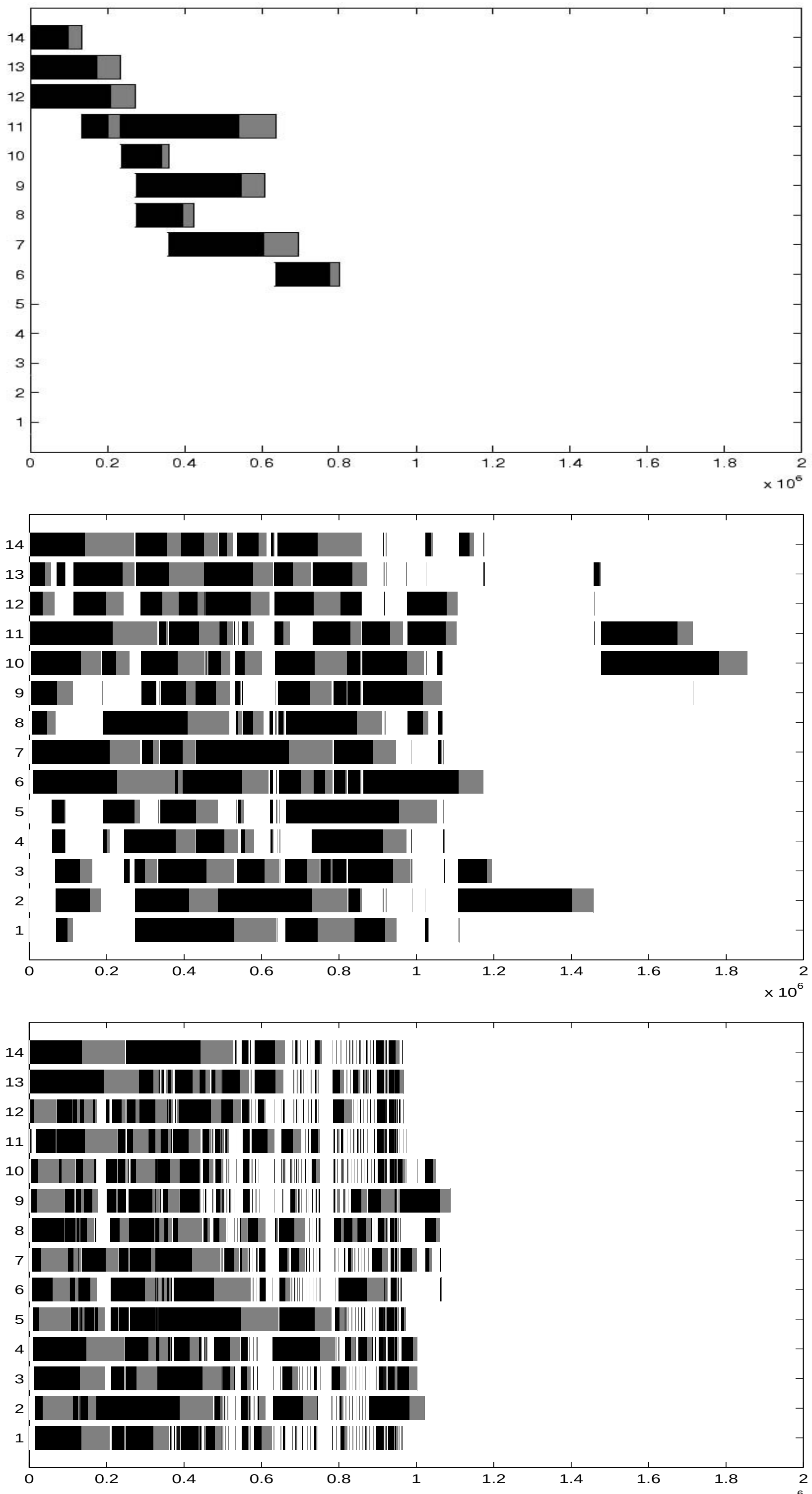
Table 5.3: Three sentences parsed with the DAP design for which the processor activity is given in figure 5.15. Speedup factor is given against the `soft_CFG` software implementation.

DAP design is using the processors more efficiently than the LAP design in particular for longer sentences (see the figures depicting the LAP design processor activity for the same sentences in figure 4.1). The reason is the ability of the DAP design to use during the parsing all the processors if necessary.

As the parsing time decreases with the DAP design (2 to 3 times in comparison with the LAP design) and the processor activity becomes more "intense", the time spent by the processors for reading chart memory data and for flushing the processing results (i.e. gray area) becomes a significant fraction in the overall processor activity time. Moreover, as for the DAP design there are more processors working in parallel, the number of collisions when accessing the chart memory also increases. This is particularly important for longer sentences and can also be observed in figure 5.15(c). In this case the 32-bit databus with the chart memory used in the DAP design becomes a bottleneck. For comparing the average processor utilization of the DAP and respectively LAP designs, we use some sentences that were arbitrarily extracted from the SUSANNE corpus. The average processor utilisation is computed in the same way as it was computed for the LAP design (see section 4.1). These sentences are tabulated in table 5.4. The average processor utilization of the DAP design increases significantly in comparison to the average processor utilization of the LAP design especially for some long sentences. An important thing to note about the processor activity is that during the parsing the amount of processing is not evenly distributed among the processors. There are processors carrying on an important amount of processing while the others are waiting – due to data dependency restrictions – for the hard working processor to finish. This suggests a design with the ability to have several processors working in parallel on the same cell-combination. Such a system will distribute better the processing power among the processors.

5.6.2 Expected performance depreciation

As already discussed in section 4.2 the expected performance depreciation phenomena means that the system performance (i.e. parsing time) does not scale at the expected rate when the number of working processors grows. This depreciation in the expected system’s performance is the consequence of an increased number of collisions (i.e. concurrent accesses) between the processors when accessing the chart memory as the sentence length increases. The increased number of collisions between processors can be observed on the figures 5.15(a)-(c) depicting the processor activity for three sentences randomly chosen from the SUSANNE corpus. In these figures, the gray area – corresponding to the time spent by the processors accessing the



(a)

(b)

(c)

Figure 5.15: DAP design processor activity in BLACK+GRAY when parsing a sentence of length (a) 4 words, (b) 10 words and (c) 15 words. GRAY: represents the time spent for chart read accesses and for flushing processing results. BLACK: represents the time spend for processing data.

len	sentence	U[%]	
		DAP	LAP
3	*“One pass only”*	10.8	9.26
4	*“There was no moon”*	8.27	8.27
5	*“She too began to weep”*	16.78	9.35
6	*“She must not think about time”*	15.78	10.99
7	*“The form and the chaos remain separate”*	23.65	12.72
8	*“The games were over , this was life”*	21.34	10.58
9	*“Like Napoleon , he was the worst of losers”*	18.41	8.90
10	*“In fact our whole defensive unit did a good job”*	33.68	15.01
11	*“I told him who I was and he was quite cold”*	20.01	8.36
12	*“Nerves tight as a bowstring , he paused to gather his wits”*	24.86	12.86
13	*“I told him no , that I had had a very happy childhood”*	28.36	10.00
14	*“As he had longed to be , he became the echo of a saga”*	47.40	10.48
15	*“It is all around us and our only chance now is to let it in”*	48.07	13.42

Table 5.4: The average processor utilization for the DAP (respectively LAP) design, for a set of sentences extracted from the SUSANNE corpus with lengths between 3 and 15 words.

chart memory – increases as the sentence length increases, becoming a significant amount of the overall processor activity.

However, in order to better illustrate this phenomena we will use an own built grammar – the same that we used in example 4.1 – that has the particularity that any cell-combination in the chart requires the same amount of processing. The following example uses this grammar in an experiment that illustrate the expected performance depreciation phenomena.

Example 5.4 We are using the same grammar as in example 4.1 to parse the sentences: "a a", "a a a", "a a a a", ..., up to a similar sentence of length 15.

After initialization each cell in the bottom row of the chart will contain the set of non-terminals $\{X_1, X_2, \ldots, X_8\}$. During parsing the grammar generates the set of non-terminals $\{X_1, X_2, \ldots, X_8\}$ for each cell-combination performed. This means that at the end of the parsing, each cell in the chart will contain this set of non-terminals and that during the parsing the processors work on the same data each time two cells are combined. In conclusion, if we know the time T spent by a processor for performing a cell-combination (i.e. the parsing time for the sentence "a a") we can compute the expected parsing time for a sentence of length L. The method used to compute the expected parsing time is not analytic and we used a program for this purpose. The program computes the minimum number of steps S required for filling the chart when parsing a sentence of length L, using a given number of processors (e.g. 14 in our case). With S and T we compute the expected parsing time as $S * T$.

Note, that when parsing the sentence "a a" only one processor is working and therefore no collisions occur when accessing the chart.

For each of these sentences the figure 5.16 illustrates the (computed) expected parsing time vs. the real parsing time. Within the DAP design the processors are more intensively (i.e. efficiently) used and the number of collisions between the processors when accessing the chart memory increases. This results in a significant difference between the expected and the real DAP design performance (i.e. parsing time) as illustrated in figure 5.16.

Figure 5.17 illustrates the processor activity when parsing a sentence of length 4 and re-

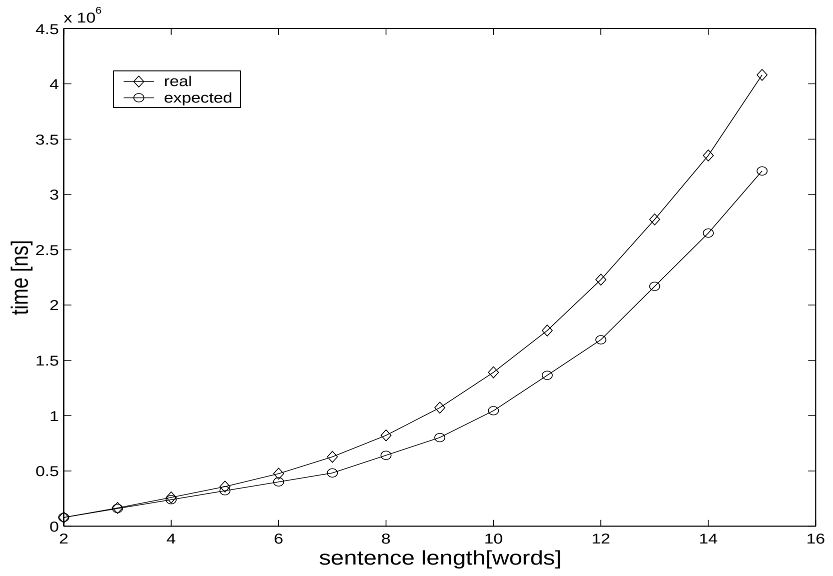


Figure 5.16: DAP design real vs. expected parsing time when parsing the sentences "a a", "a a a", "a a a a", . . . up to a similar sentence of length 15. Real parsing time is greater than the expected parsing time illustrating the expected performance depreciation phenomenon.

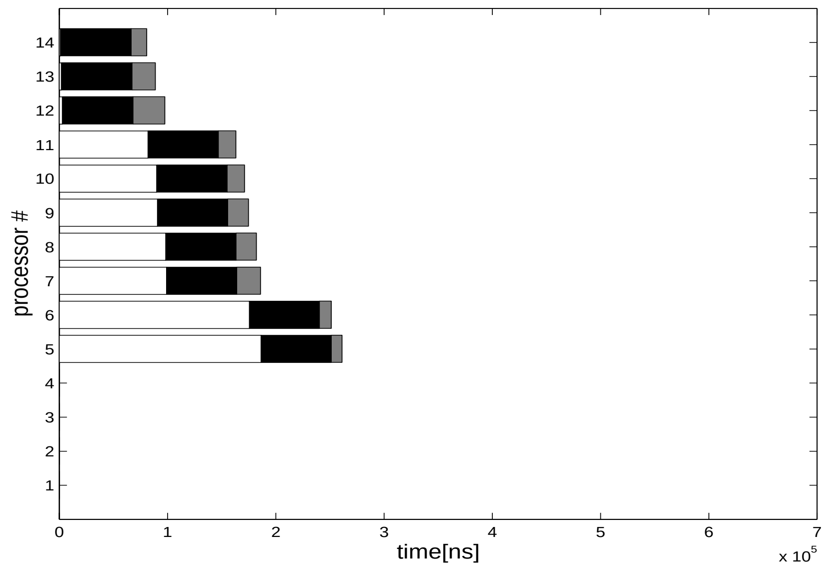
spectively 7 "a"s. Note that the gray area for each cell combination increased for the sentence of length 7 when compared to the sentence of length 4 while the sizes of the black areas did not change. ♢

5.6.3 POOL influence on DAP design performance

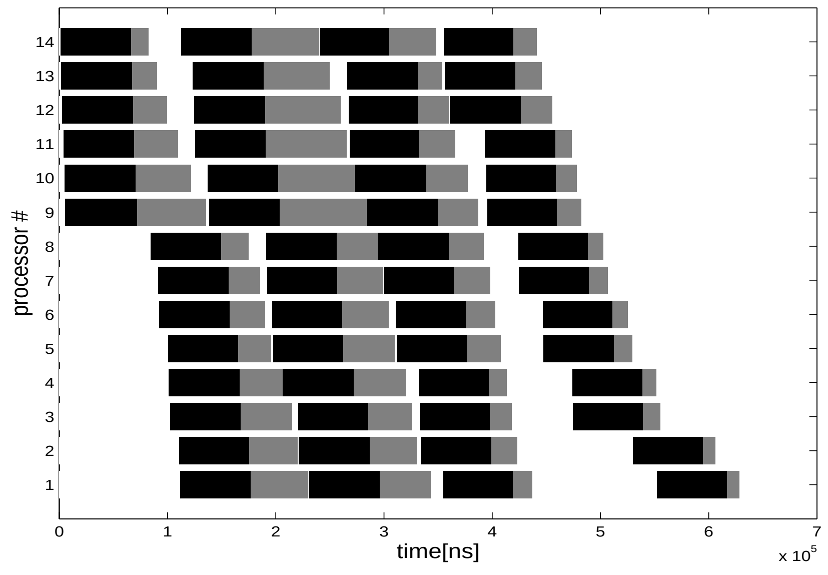
The POOL unit is key to the design ability to dynamically allocate the processors to cell-combinations. Without the POOL unit the design will hang on the first cell-combination that does not satisfy a data-dependency constraint and keep all the other cell-combinations – even if they satisfy their data-dependency constraints – unissued. On the other hand, a POOL unit allows the design to pass over (i.e. to store and temporarily ignore) the cell-combinations that do not satisfy the data-dependency constraints while still try to issue other cell-combinations that satisfy their data-dependency constraints. This results in a better average processor utilization due to a better exploitation of the parallelism available in the parsed sentence. One can interpret the POOL unit functionality as a window sliding over the unprocessed cells of the chart that allows the design to issue the cell-combinations that satisfy the data-dependency constraints while keeping under observation those that do not satisfy the data-dependency constraints. In this context, a larger POOL size (i.e. a larger sliding window) is supposed to be better as it allows the design to search over a larger number of cell-combination that can be potentially issued.

In order to investigate the influence of the POOL size on the DAP design performance a number of benchmarks have been performed on $2,043$ sentences of length 3 to 15 from the SUSANNE corpus. The benchmarks were performed by simulating[6] the VHDL model of a DAP design (configured for the CNF SUSANNE grammar) with 10 processors for several

[6]With ModelSim EE/Plus 5.2e



(a)



(b)

Figure 5.17: DAP design processor activity for a sentence of length (a) 4 "a"s and respectively (b) 7 "a"s. The larger gray area is the reason for the expected performance depreciation phenomenon.

POOL sizes. A POOL size of 16, 8, 4, 2 and respectively 1 was used for the purpose of these simulations.

The software used for comparison is an implementation of the enhanced-CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The hardware performance (i.e. run-time) of the 10-processor DAP design with POOL size 1 (denoted as `hard_pool1`), POOL size 2 (denoted as `hard_pool2`), POOL size 4 (denoted as `hard_pool4`), POOL size 8 (denoted as `hard_pool8`), POOL size 16 (denoted as `hard_pool16`) was compared against two software run-times. The first (`soft_CNFG`) uses the SUSANNE grammar in CNF, as it is also the case for the hardware. The second (`soft_CFG`) uses the SUSANNE grammar in its original context-free form. The software was run on a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* `C` library function and not by profiling the code.

Figure 5.18(a) shows the speedup of the 10-processors DAP design for different POOL sizes when compared to the `soft_CNFG` and figure 5.18(b) shows the speedup for the same design, for different POOL sizes when compared to the `soft_CFG`. In these figures the speedup factor is represented as a function of sentence length. It comes out from this figures that the design performance improves as the POOL size increases from 1 to 8. However, as the POOL size changes from 8 to 16 no further increase in the design performance is observed. This leads us to conclude that in the particular case of a 10-processors DAP system a POOL of size 8 is optimal as it does not waste hardware resources. It is likely that if more processors are used a larger POOL size is required to achieve the best performance. The explanation is that a larger window may be necessary in order to search deeper in the chart for finding cell-combinations to process as there are more processors available for processing cell-combinations .

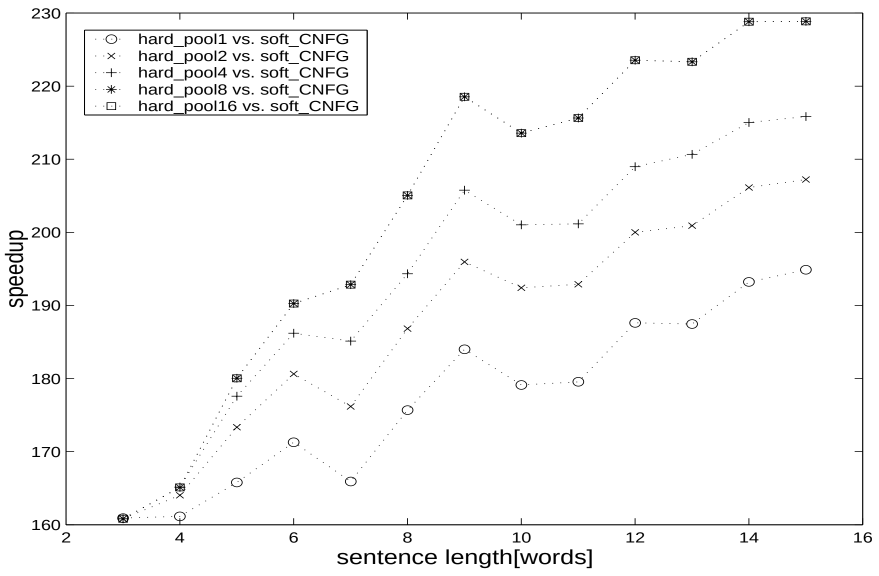
5.7 Conclusions

This chapter proposes an improved FPGA-based hardware implementation of the CYK algorithm, adapted for word lattice parsing that can deal with large-size real-life Chomsky Normal Form.

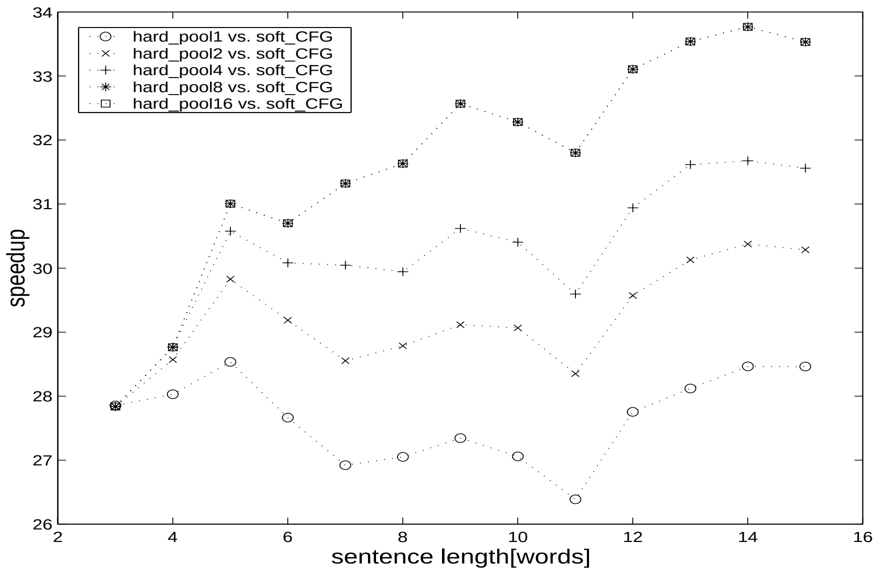
The proposed design – called DAP – implements the dynamic allocation method and the maximal sentence length it can parse is independent of the number of processors in the system. The maximal sentence length only depends on the size of the chart memory and does not depend anymore on the resources available in the used FPGA. In other words, if the syntactic analysis of a sentence fits in the chart memory, then any number of processors can parse the sentence and the number of processors we can fit in the FPGA will only influence the parsing performance. This is in contrast to the LAP design for which a Xilinx Virtex XCV1000 FPGA can only fit a LAP design with maximum $n = 25$ processors – in the particular case of the SUSANNE grammar – which is particularly restrictive when parsing word lattices.

The parsing results are available on-line, during the parsing, on some FPGA pins as a compact parsing forest and can be used for further processing (e.g. semantic module). For the implemented DAP design the performance measurements show an average speedup of 206.45 when compared to a software implementation of the enhanced-CYK algorithm – run on a SUN (Ultra-Sparc 60) with 1 processor at 360 MHz – using a CNF of the SUSANNE grammar and an average speedup of 31.90 when using a general CFG representation of the SUSANNE grammar.

All the improvements addressed by the DAP design did not require to change the data-



(a)



(b)

Figure 5.18: The speedup for the 10-processors DAP system for several POOL sizes when compared to (a) `soft_CNFG` and (b) `soft_CFG` software as a function of sentence length. For each sentence length more than 100 sentences were parsed.

structures used to represent the chart and the grammar memories, which are the same as those used for the LAP design. This rendered possible an accurate comparison between the performance of the LAP and DAP designs. Also, the DAP interface and initialization requirements are identical to those required for the LAP design. This allows an easy integration of a commercial FPGA such as the Xilinx Virtex XCV2000efg1156-6 – containing the DAP design – within a larger system. The larger system can be for instance an FPGA-board working as an accelerator in an application framework (e.g. NLP) requiring efficient parsing of context-free languages. An example of such an application, a Vocal Interface Server, is described in section 1.4.

The DAP design was the second step of our design methodology during which:

- we studied a better processor task allocation method;
- we refined and extended our background knowledge about the real-life behaviour of the CYK algorithm, on which the enhanced-CYK implementation will be built upon;
- we improved the speed-up factor.

The first point addresses a major drawback of the previous LAP design by investigating the dynamic allocation method – presented in appendix B.2 – which is also a key feature for a design able to efficiently exploit the parallelism available in the CYK algorithm. The second point was useful for identifying critical regions and features of the DAP design that have to be improved when implementing the enhanced-CYK algorithm. The third point was an argument for the advantage offered by the dynamic allocation method.

From the performed experiments and performance measurements of the DAP design we remark a number of drawbacks:

- for every parsed sentence there is an optimal number of processors in the system for which the speedup is maximal. A smaller/larger number of processors may result in a performance depreciation. The explanation is that when there are more processors than needed – for parsing a particular sentence – in the system, the time lost by the system for managing the unneeded processors becomes an overhead.
- as the number of working processors in the system grows the collisions among processors – that require arbitration – becomes significant resulting in a performance depreciation. This phenomena was illustrated in section 5.6.2. While a significant amount of chart memory accesses are performed for checking the guard-vectors a solution for solving this problem would be to use a data-structure for the chart that does not rely on the use of guard-vectors;
- although the processor utilization was improved for the DAP design it is still low – between $25 - 50\%$ – for sentences of real-life length. The reason is that the amount of processing is not evenly distributed among the processors. Concretely, as illustrated in the processor activity diagrams (see figure 5.15), there are processors carrying on cell-combinations that are very demanding in terms of processing, while other processors work on less demanding cell-combinations and finish quicker. Moreover, it is often the case that the processors that finish will also wait – due to data-dependency constraints – for the hard working processor(s) to finish. A solution to this problem that can increase even more the processor utilization would be to cluster several processors and assign them to process the same cell-combination. Such a solution will better distribute the overall amount of processing among the processors, resulting in a better average processor utilization and shorter parsing times;

- the initialization procedure – the same required for the LAP design – is relatively complex and time consuming. This comes from the fact that the guard-vectors are initialized in two steps: (1) all the guard-vectors in the chart require a cleaning step (i.e. "0"-filling) and (2) an initialization with the new content. The second step is required only when parsing word lattices;
- the designs can only deal with CNF grammars. The rewriting of a general CFG in a CNF grammar, although always possible, obfuscates the syntactic structure of the described language. Concretely, this refers to the fact that the parsing trees produced with the rewritten CNF grammar have a different structure when compared to those produced by the original general CFG. A design able to cope with general CFGs is required;
- the designs do not have yet the ability to recover from fatal errors (e.g. exceeded number of non-terminals in a cell, a unit crash). A mechanism for monitoring the normal system operation and that can reset the design is a stable state from which other parsings can start is also required;
- the designs do not integrate a unit for on-line extraction of parsing forest while this was not key feature for a non-final version of the design;
- only 30% of the resources available in a Xilinx Virtex XCV1000 were used in the DAP design. A lot of resources are therefore still available for further improvements .

Chapter 6

The Hardware Design of the enhanced-CYK Algorithm

This chapter presents a design architecture implementing the enhanced-CYK algorithm adapted for word lattice parsing (see section 2.3, page 17). The proposed architecture is the third step – and the final – of our design methodology, during which:

- we propose a design that can deal with almost unrestricted general CFGs;
- simplify the chart initialization procedure;
- propose a method called tiling that improves the average processor utilization;
- integrate an on-line parse extraction module;
- integrate a module that monitors the normal system operation during runtime;

The first point mentioned above addresses a major drawback of the previous designs by allowing the enhanced-CYK design to cope with general (almost unrestricted) CFGs, not only with Chomsky Normal Form CFGs. In fact the design can only handle a large subclass of CFGs, called "non partially lexicalized" CFGs (see section 2.3) that also does not contain unitary rules. The restriction to "non partially lexicalized" CFGs simplifies the design's initialization step and the restriction to contain no unitary rules reduces significantly the hardware complexity.

The second improvement is a simplified initialization procedure which was made possible by eliminating the guard-vectors used in the former chart data-structure and by replacing their functionality with specialized hardware. This change substantially reduces the memory space requirements per chart cell, resulting in the design's ability to parse, for the same size of the chart memory, longer sentences or word-lattices with more time-stamps. For instance, with the current implementation, 256 KBytes memory is enough to parse sentences with up to 32 words and 16 MBytes memory will be enough to parse sentences with up to 256 words.

The third improvement is a method called tiling that improves the average processor utilization. With the tiling method the processors can be assigned to process chunks of a cell-combination – henceforth referred as tiles – unlike the processors in the previous designs that were processing entire cell-combinations.

The last two improvements are required for an implementation of the enhanced-CYK algorithm that will be integrated in a larger application framework. Both the ability to extract the compact parse forest and to monitor the system state during runtime are essential. The later functionality is required in order to create a reliable environment.

The main features of this design are: (1) an average speed-up factor of about $65 - 85$ when compared to a software implementation of the enhanced-CYK algorithm[1] and (2) the ability to parse real-life sentences of up to 256 words (or time-stamps).

This chapter starts with a general and functional system description. The data-structures used to represent the chart and the grammar are next presented. It continues by presenting the system units in detail. The design performance measurements and the design analysis – discussing some key features of the proposed design – are given before concluding this chapter.

6.1 General system description

The enhanced-CYK design proposed in this chapter can parse sentences of any length, regardless of the number of processors in the system, given that the chart memory is large enough to store the chart data-structure. As for the DAP design, the input to the enhanced-CYK design is an initialized chart and grammar lookup tables and the output is a compact parse forest. The enhanced-CYK design uses SRAMs for storing the data-structures representing the chart (stored in the chart memory) and the "non partially lexicalized" grammar (stored in grammar memories).

Like for the DAP design, for the current design the maximal length of the sentence that can be parsed does not depend on the number of processors in the system and therefore the processors are not clustered around the grammar memories. Instead, each processor has its own (local) grammar memory for best performance. As the number of pins available for an FPGA is limited, only a limited number of grammar memories can be connected to the FPGA which will also limit the number of processors.

The chart memory is shared by all the processors in the system as it was the case for the previous designs. However, within the enhanced-CYK design the processors do not directly access the chart memory. An interface is used for this purpose in order to reduce the number of access collisions when accessing the chart memory and to increase in consequence the overall system performance.

The design's interface with the external world is identical with that of the previous designs (`startPARSE`, `overPARSE` and `SLEN` signals). The initialization procedure is almost the same but was simplified in some aspects and both the chart and the grammar data-structures changed. The grammar data-structure changed in order to represent the nplCFGs that cannot employ the same data-structure as the one used for representing Chomsky Normal Form CFGs. On the other hand, the chart data-structure changed, rendering its initialization significantly less time-consuming when compared to the previous designs. Concretely, the guard-vectors were eliminated and their functionality (i.e. non-terminals lookup) was undertaken by an associative memory.

The processor architecture and the grammar data-structure were jointly designed in order to allow a tighter coupling and for improving the access time to data. A 16-bit databus is used to link each processor to its grammar memory – in comparison to the 32-bit databus used in the DAP design – that allows a number of 16 processors to be placed inside the XCV2000efg1156-6 FPGA. A more sophisticated processor architecture compensates the smaller bandwidth between the processor and its grammar memory. Like the DAP design, the enhanced-CYK design implements the dynamic processor allocation method. However, unlike the processors in the previous designs that were processing entire cell-combinations, the processors in the current design can be assigned to process smaller chunks of a cell-combination. Such an approach is

[1] Implemented in the SlpToolKit (see appendix A.1)

useful in the case when a cell-combination is demanding in terms of processing in which case several processors can team-up and process the same cell-combination. This leads to a better average processor utilization. The tile size can be configured in the VHDL code describing the design which offers a very powerful method for investigating the design's behaviour under different tile size configurations.

Two new functionalities were integrated in the design architecture implementing the enhanced-CYK algorithm. The first functionality gives the system the ability to monitor, trace (and recover from) fatal errors such as exceeded number of non-terminals in a chart cell. Whenever a fatal error occurs during runtime the system is reset in a stable state from which normal operation can restart. The second functionality gives the system the ability to extract on-line the parsing forest.

The general system architecture for an n-processor system implementing the enhanced-CYK algorithm using the dynamic processor allocation method is depicted in the block diagram in figure 6.1. Again as for the previously presented designs the elements inside the dashed line are implemented with hardware resources available within the FPGA. The other elements (chart and grammar memories GMi) are implemented in SRAM chips present on the system board. An FPGA-board containing the hardware resources required for implementing a 16-processors enhanced-CYK system proposed in this chapter is presented in chapter 7.

6.2 Functional description

Before any parsing can start the system requires to be initialized. The system initialization procedure consists of initializing the:

- grammar memories: the binary image of the nplCFG data-structure is loaded in each grammar memory in the system. The grammar memories are configured once before any parsing can start and their content stays unchanged during successive parsings. The grammar memories require reconfiguration only if a different grammar (i.e. nplCFG without unitary rules) has to be used;
- chart memory: the initialization of the chart memory is done by some software running on the on-board processor or on the host system. It consists in initializing for certain cells (i, j) of the chart the set of non-terminals $N1_{i,j}$ and respectively the set of partial right-hand sides $N2_{i,j}$ by making use of the lexical rules;
- sentence length: a register of the I/O controller IO-CTRL is initialised before every parsing with the length of the sentence to be parsed.

Once the system was initialized the parsing can start by activating the signal `startPARSE`. At this moment the sequence generator unit SEQ_GEN (see section 6.4.1 for details) will start to generate triples $(S1,S2,D)$ of two source chart cells $S1$ and $S2$, that need to be combined (see lines $7-9$ of the enhanced-CYK algorithm on page 17) along with their corresponding destination cell D in which the cell-combination result will be stored.

Note: A source/destination cell in a triplet is in fact a pair of coordinates representing the row and the column of that cell. For instance, $S1$ is written as (r_{s1}, c_{s1}), where r_{s1} is the row and c_{s1} is the column of the first source cell.

The generated triples depend (only) on the length of the parsed sentence and the same sequence

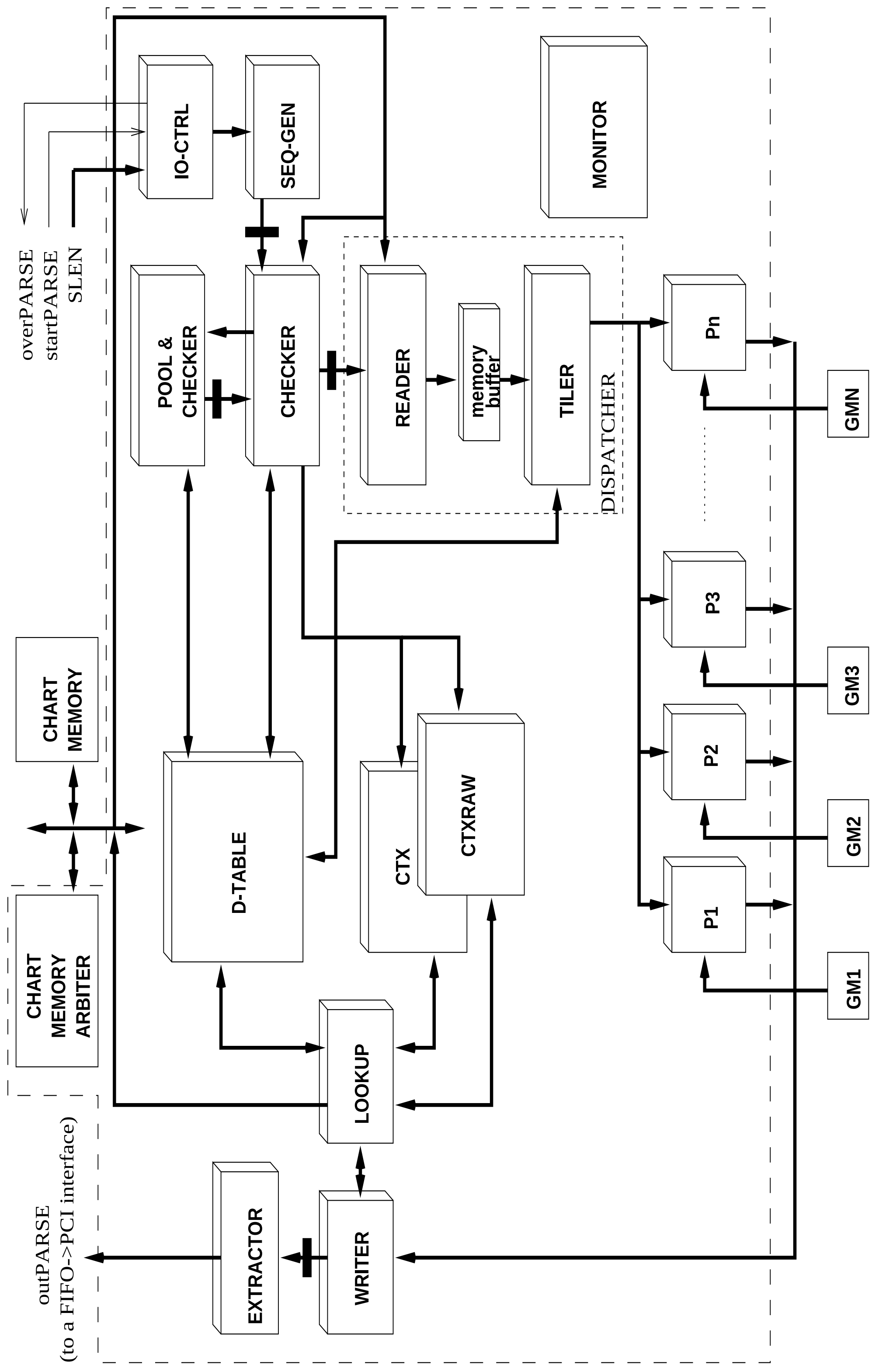


Figure 6.1: The block diagram of an n-processor enhanced-CYK design.

of triples is generated for any two sentences of the same length. The triples are then passed to the CHECKER unit (see section 6.4.2 for details) who's task is to check whether the source cells are available, or more precisely, that the source cells are not destinations of unfinished previous cell-combinations. In other words, the CHECKER verifies that the data-dependency in the chart, is satisfied. For doing this, the CHECKER makes use of the destination table D-TABLE (see section 6.4.3 for details) that stores all destination cells D, currently under processing. The CHECKER inserts a new destination cell D in the D-TABLE when it is encountered for the first time in a triplet and deletes a destination cell D from the D-TABLE when all the triples containing the destination cell D have been treated. Each time the CHECKER inserts a new destination cell D in the D-TABLE it does three things: (1) sets some "context information" for the destination cell D and stores it in the CTX-memory, (2) reads the content of the destination cell D (both the $N1$ and $N2$ sets) and stores them in the CTXRAW-memory and (3) allocates an unique identifier ID for the destination cell D. The CHECKER will tag all the subsequent triplets containing the destination cell D with the identifier ID before sending them further to the DISPATCHER.

A triplet that passes the CHECKER's test for data-dependency is forwarded to the task dispatching unit DISPATCHER, while a triplet that does not pass the CHECKER's test for data-dependency is stored in the POOL (see section 6.4.4 for details) where it waits for the data-dependency to be satisfied. In the POOL all triplets are continuously checked for data-dependency against the D-TABLE and as soon as a triplet passes the data-dependency test it is returned to the CHECKER. The CHECKER has therefore two inputs, one from the POOL and the other from the SEQ_GEN – both passing through FIFO memories (represented with small black boxes in figure 6.1). The triplets coming from the POOL have higher priority and are handled first. The reason for giving higher priority to the triplets coming from the POOL is that, for a SEQ_GEN generating triplets in a natural chronological order, the POOL will store old triplets that require to be treated first. Before forwarding a triplet to the DISPATCHER, the CHECKER will tag it with the ID associated with the destination cell D it contains.

The task of the DISPATCHER unit (see section 6.4.5 for details) is to assign (i.e. dispatch for processing) the incoming triplets to the available processors in the system. The DISPATCHER unit is one of the major changes of the new design. The new DISPATCHER unit has two stages: the first is the READER and the second is the TILER. The READER stage prefetches for each incoming triplet the set $N2$ for the source cell $S1$ and the $N1$ set for the source cell $S2$ and stores them in an FPGA internal buffer memory. The buffer memory – built with dual-port memory resources (see Xilinx's Virtex-E FPGA manual [30] for details) – is written by the READER stage and read by the TILER stage. The TILER stage reads the content of this buffer in a first-in-first-out fashion and splits each of the cross-products $N1 \times N2$ (i.e. cell-combinations) in smaller chunks of predefined size (that can be configured in the VHDL code) – referred henceforth as tiles. The tiles are further distributed to the processors as soon as the processors become available during runtime. As for the DAP design, the result of a processed tile (i.e. the processors output), is fetched by the WRITER (see section 6.4.7 for details), reassembled[2], and finally stored[3] in the chart memory. The WRITER uses the identifier ID that comes along a tile's processing result in order to identify and reassemble the destination cell D to which these results belong.

[2]All the tiles corresponding to a destination cell D are reassembled by the WRITER and the partial reassembled destination cell D is kept in the CTXRAW-memory for all destination cells D currently under processing.

[3]The content of the destination cell D stored in the CTXRAW-memory is dumped in the chart memory when the last tile for the destination cell D was reassembled.

After fetching the processor's output the WRITER updates accordingly the data-dependency information in the D-TABLE and forwards the data required for extracting the parsing forest to the EXTRACTOR (see section 6.4.8 for details). The EXTRACTOR extracts the compact parsing forest information and packs it in a compact format that can be efficiently (i.e. rapidly) transfered to the host machine. The MONITOR (see section 6.4.9) unit supervises correct system operation and in case of errors resets the system in an initial state from which new parsings can restart.

6.3 The enhanced-CYK algorithm data-structures

6.3.1 The chart data-structure

This section discusses the chart data-structure used for the hardware implementation of the enhanced-CYK algorithm. The factors that influence the chart data-structure organization are also discussed. Like for the CYK algorithm, the data-structure representing the chart used by the enhanced-CYK algorithm has to find a compromise between: (1) required functionality, (2) required memory space, (3) data access time and (4) data access circuit complexity.

The content of a cell for the enhanced-CYK algorithm is not the same with the content of a cell for the CYK algorithm and therefore the data-structure used to represent the chart changes. A cell (i, j) for the CYK algorithm contained a set $N_{i,j}$ of non-terminals. For the enhanced-CYK algorithm, a cell (i, j) contains two sets, a set $N1_{i,j}$ of non-terminals and a set $N2_{i,j}$ of partial rule right-hand sides. Nevertheless, the functionalities this new data-structure has to support, are essentially the same as for the CYK algorithm (see section 3.4.1) with some slight complications. Suppose we have the triplet $(S1\ , S2\ , D)$ in which the source cell $S1$ corresponds to cell (i, k), the source cell $S2$ to cell $(i + k, j - k)$ and the destination cell D to cell (i, j). Then the following functionalities require support:

- $eF1$: go through all elements of a set. Required in order to pair each partial rule right-hand side of the set $N2_{i,k}$, in the source cell (i, k) with each non-terminal of set $N1_{i+k,j-k}$ in the source cell $(i + k, j - k)$. For more details see the enhanced-CYK algorithm on page 17);

- $eF2$: is an element in a set ? Required in order to check the presence – in the destination cell (i, j) – of a non-terminal X in the set $N1_{i,j}$, or to check the presence of a partial rule right-hand side $\alpha\bullet$ in the set $N2_{i,j}$;

- $eF3$: insert an element in a set. Required in order to store – in the destination cell (i, j) – a non-terminal X in the set $N1_{i,j}$ or to store a partial rule right-hand side $\alpha\bullet$ in the set $N2_{i,j}$;

Where $eF\#$ stands for enhanced functionality. The $eF1$ and $eF3$ functionalities are supported by the same means as for the CYK algorithm, namely a list representations of the sets.

The previous $F2$ functionality was supported by means of guard-vectors that are not used anymore due to their bad influence on the overall system performance. The $eF2$ functionality is currently supported by means of an associative memory. The associative memory is loaded either with the set $N1_{i,j}$ or $N2_{i,j}$ of a destination cell (i, j). When the associative memory is loaded with the $N1_{i,j}$ set of a destination cell (i, j) any non-terminal X can be checked for occurrence against the $N1_{i,j}$ set. Identically, when the associative memory is loaded with the $N2_{i,j}$ set of a destination cell (i, j) any partial right-hand side $\alpha\bullet$ can be checked for occurrence against the $N2_{i,j}$ set. In order to be efficient (i.e. fast) it is imperative that the content of the associative memory can be changed (i.e. loaded) very fast with a new set.

Allocating for each set $N1_{i,j}$ an amount of memory proportional to N would represent an important memory waste, as in practice $|N_{i,j}| \ll |N|$. Identically, the size of each set $N2_{i,j}$ is much smaller in practice than its theoretical maximal size which depends on the grammar characteristics (i.e. number of rules, number of non-terminals). A maximal size K is assumed for the set $N1_{i,j}$ respectively the set $N2_{i,j}$. If however, during runtime, a cell receives in either of the sets more than K distinct elements, the hardware generates a fault signal and the parsing stops. This would be a very unlikely event for a well chosen value of K. The value of K depends on the considered grammar and can be found by investigating the characteristics of the grammar. In the particular case of the nplCGF without unitary-rules SUSANNE grammar (see appendix B.4) we have a value of $K = 128$ which requires a chart memory size of 256 KByte for parsing any sentence with up to 32 words and a memory size of 16 MByte for parsing any sentence with up to 256 words.

For understanding how the $eF1$, $eF2$ and $eF3$ functionalities are implemented, the chart data-structure organization is given in figure 6.2. The chart memory storing the chart data-

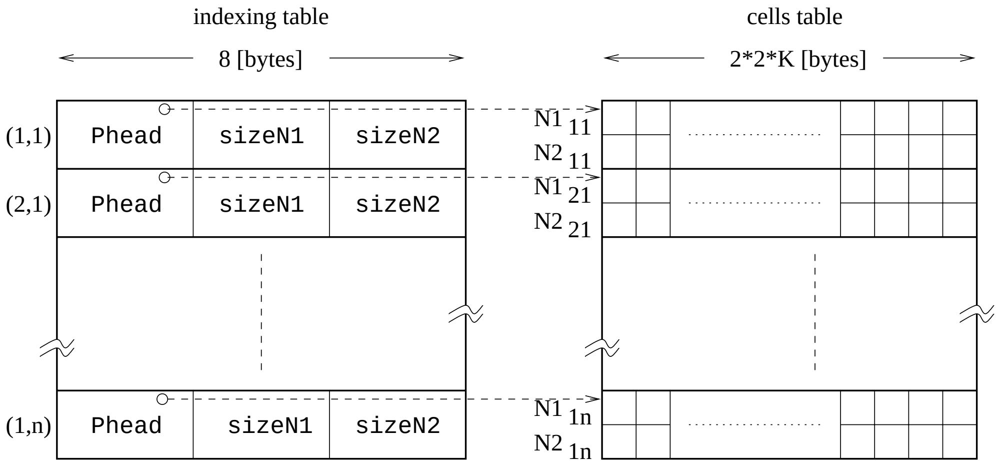


Figure 6.2: The enhanced-CYK chart data-structure memory organization. Indexing table entry is on 8 bytes and cells table entry is on $2*2*K$ bytes (each set $N1$ and $N2$ requires $2*K$ bytes).

structure is addressed on 8 byte words. Therefore, in order to efficiently access the data stored in the chart memory, the content of the data-structure is aligned to an 8 byte boundary. The chart data-structure is organized in memory as two distinct tables and is defined by some parameters that depend on the used grammar. The maximal size allowed for these parameters as well as the actual size of these parameters in the particular case of the SUSANNE grammar are tabulated in table 6.1. A grammar for which any of these parameters requires a size larger than its maximal allowed size cannot be accommodated within the proposed data-structure. These parameters are used by (1) the software used to initialise the chart data-structure and (2) the VHDL code used to synthesize the design in order to correctly access the chart data-structure. The left table in figure 6.2, called indexing table, contains for each chart cell (i, j) an entry that allows to retrieve (1) a pointer `Phead` to the memory location where the associated sets $N1_{i,j}$ and $N2_{i,j}$ are stored and (2) two values `sizeN1` and `sizeN2` representing the (current) number of elements in each of the two sets. Each entry in the indexing table is stored on 8 bytes and is organized as illustrated in figure 6.3. The physical address of the entry in the indexing table where the

parameter	maximal allowed size	SUSANNE size	meaning
NT	2^{16}	$1,912$	total # of non-terminals
N1_SIZE_BITS	16 [bits]	11 [bits]	bits for a non-terminal
N2_SIZE_BITS	16 [bits]	14 [bits]	bits for a partial right-hand side
CELLSIZE_BITS	16 [bits]	7 [bits]	bits for the constant K
CYK_ADR_SIZE	32 [bits]	16 [bits]	bits for a chart memory pointer

Table 6.1: The maximal size of the parameters that define the enhanced-CYK chart data-structure. Parameter values in the particular case of the SUSANNE grammar.

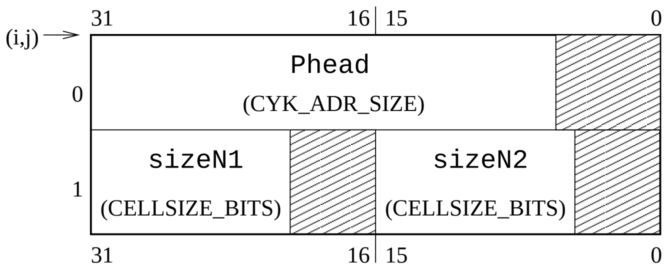


Figure 6.3: The organization of an entry (i, j) in the indexing table of the enhanced-CYK chart data-structure.

information about cell (i, j) resides is build by concatenating the binary representation of i and j and performing a left-shift with 3 positions (8 bytes aligned). The indexing table size is 8 KBytes for parsing sentences with up to 32 words and 512 KBytes for parsing sentences with up to 256 words.

The second table called the cell table contains for each chart cell (i, j) an entry storing the sets $N1_{i,j}$ and $N2_{i,j}$. While both the elements of the $N1$ set (i.e. non-terminals, represented on N1_SIZE_BITS) and the elements of the $N2$ set (i.e. partial rule right-hand sides, represented on N2_SIZE_BITS) are stored on 2 bytes, each entry in the cell table requires $2 * 2 * K$ bytes. In the particular case of the SUSANNE grammar the size of an entry in the cell table is 512 bytes. In the same particular case the size of the cell table is 264 KBytes for parsing sentences with up to 32 words and 16 MBytes for parsing sentences with up to 256 words.

Let's now look at how the functionalities $eF1$, $eF2$ and $eF3$ are implemented. For the implementation of $eF1$ the pointer Phead is the base memory address where the set $N1$ is stored and the pointer Phead $+ 2^{\texttt{CELLSIZE\_BITS}}$ is the base memory address where the set $N2$ is stored. Two displacements, i.e. indexes, are used for going through the non-terminals in the set $N1$ (0 to sizeN1 $- 1$) and respectively through the partial right-hand sides in the set $N2$ (0 to sizeN2 $- 1$). The addition of the base memory address and the displacement gives the physical memory location of the addressed item.

The functionality $eF2$ is supported by means of an associative memory that is loaded either with the content of the set $N1_{i,j}$ or $N2_{i,j}$ of a destination cell (i, j). A non-terminal X can be looked-up in the associative memory, when the associative memory is loaded with the set $N1_{i,j}$ of the destination cell (i, j). Identically, a partial right-hand side $\alpha\bullet$ can be looked-up in the associative memory, when the associative memory is loaded with the set $N2_{i,j}$ of the destination cell (i, j).

Finally, for implementing the functionality $eF3$, the $\texttt{Phead} + \texttt{sizeN1}$ points to the physical memory location where the next non-terminal has to be stored and $\texttt{Phead} + 2 * 2^{\texttt{CELLSIZE\_BITS}} + \texttt{sizeN2}$ points to the physical memory location where the next partial right-hand side has to be stored. Each time a non-terminal is stored in memory the value of sizeN1 is incremented to reflect the new size of the $N1$ set. Identically, each time a partial right-hand side is stored in memory the values of sizeN2 is incremented to reflect the new size of the $N2$ set. If during the parsing $\texttt{sizeN1} > K$ or $\texttt{sizeN2} > K$, a fault signal is raised to signal that the $N1_{i,j}$ or respectively the $N2_{i,j}$ set has to many elements. The parsing will stop in this case and a signal should tell the host computer (or local microprocessor) that the current parse could not finish and that the software has to redo it. When parsing sentences, the initial value of the sizeN1 and sizeN2 fields is always 0. However, for word lattices this in not always the case .

6.3.2 The nplCFG grammar data-structure

This section discusses the data-structure representation of the nplCFG (without unitary-rules) – referred henceforth simply as the grammar – used in the hardware implementation of the enhanced-CYK algorithm.

A copy of the grammar data-structure has to be loaded in each grammar memory in the system, before any parsing can start. The content of the grammar memory only has to be changed when a new grammar is to be used. Given that for real-life grammars the data-structure requires a large amount of storage memory, SRAM chips are used for this purpose. The SRAMs have several advantages: (1) are easy to control, (2) state-of-the-art SRAM chips have relatively large sizes, (3) fast access times and (4) require a minimum of interfacing signals with the FPGAs. The disadvantage is the high power (i.e. current) consumption.

Within the current design the processors do not share the available grammar memories. Each processor has its own grammar memory in order to achieve best performance. While such a configuration is efficient only if the processors are using the grammar memories intensively, the design of the grammar data-structure and of the processor aimed a tighter coupling of the two.

When working on the cell-combination $(S1, S2, D)$, a processor pairs each partial right-hand side $\alpha\bullet$ in the set $N2$ of the source cell $S1$ with each non-terminal Y of the source cell $S2$ and uses the grammar data-structure to check whether the $\alpha Y\bullet$ is a new partial rule right-hand side, and/or αY is a rule right-hand side and in each of the two cases to retrieve some information. Precisely, given a partial right-hand side $\alpha\bullet$ and a non-terminal Y the grammar data-structure should allow a processor to retrieve the following information:

1. the partial right-hand side $\beta\bullet = \alpha Y\bullet$ if there is one;

2. if αY is a rule right-hand side: all the non-terminals X_i that are left-hand sides of the grammar rules $X_i \rightarrow \alpha Y$ and all the $X_i\bullet$ in the grammar in this case;

3. a code that uniquely identifies the right-hand side αY for the grammar rules at point 2.

The example bellow illustrates the functionality required from the grammar data-structure when a cell-combination $(S1, S2, D)$ is performed by a processor.

Example 6.1 Let's consider the nplCFG without unitary rules given by:

$$
\begin{array}{rlll}
 & N = \{A, B, C, D, E, F, G, H, I, J, K, S\}, & & \\
 & \quad \Sigma = \{\ldots undefined \ldots\}, & & \\
R = \{ & A \rightarrow B\ C\ E\ , & B \rightarrow C\ E\ , & B \rightarrow C\ E\ G\ , \\
 & E \rightarrow H\ D\ , & F \rightarrow H\ D\ , & F \rightarrow B\ C\ E\ G\ , \\
 & H \rightarrow B\ C\ E\ , & I \rightarrow B\ C\ A\ E\ , & J \rightarrow B\ C\ F\ , \\
 & K \rightarrow A\ E\ , & K \rightarrow A\ B\ C\ , & S \rightarrow K\ J\ \ \}
\end{array}
$$

an perform the cell-combination $(S1, S2, D)$ given in figure 6.4. The cross-product of the set

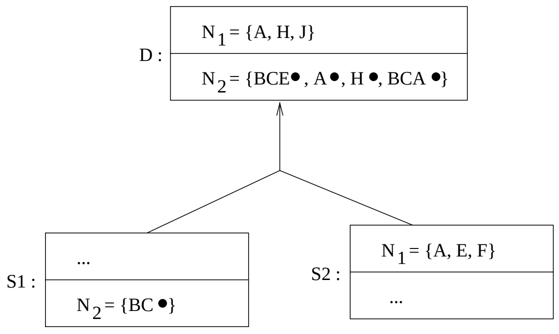


Figure 6.4: Example of cell-combination for the enhanced-CYK algorithm.

$N2$ (partial right-hand sides in source cell $S1$) and the set $N1$ (non-terminals in the set $S2$) is $\{(BC\bullet, A),\ (BC\bullet, E),\ (BC\bullet, F)\}$. The grammar data-structure should allow a processor to retrieve the following information:

- ▷ for $(BC\bullet, A)$: (1) the partial rule right hand side $BCA\bullet$ (e.g. the grammar rule $I \rightarrow BCAE$);
- ▷ for $(BC\bullet, E)$: (1) the partial rule right-hand side $BCE\bullet$ (e.g. the grammar rule $F \rightarrow BCEG$) and (2) the non-terminals A (the rule $A \rightarrow BCE$) and H ($H \rightarrow BCE$) respectively the partial right-hand sides $A\bullet$ ($K \rightarrow AE$) and $H\bullet$ ($E \rightarrow HD$);
- ▷ for $(BC\bullet, F)$: (2) the non-terminal J (the rule $J \rightarrow BCF$);

The partial rule right-hand sides found above are inserted in the set $N2$, and the non-terminals are inserted in the set $N1$ of the destination cell D. The result is illustrated in figure 6.4. $\diamondsuit$

The proposed grammar data-structure that allows a processor to perform the operation described in the example above is illustrated in figure 6.5. The grammar data-structure is aligned to a 4 byte boundary and therefore the grammar memories can be accessed with a 8-bit, 16-bit or 32-bit wide databus as needed. If the databus width is changed only the processor's interface with the grammar memories requires small changes. The grammar data-structure is defined by some parameters whose values depend on the used grammar. These parameters, their maximal allowed size and their actual size in the particular case of the SUSANNE grammar (without unitary rules) are tabulated in table 6.2. A grammar for which any of these parameters requires a size larger than its maximal allowed size cannot be accommodated within the proposed grammar data-structure.

Let's see how the grammar data-structure is organized. Level 1 is a table with an entry for each distinct partial rule right-hand side $\alpha\bullet$ present in the grammar. An entry in this table contains a pointer `ptr_list` to a list stored at level 2, containing all the non-terminals that are

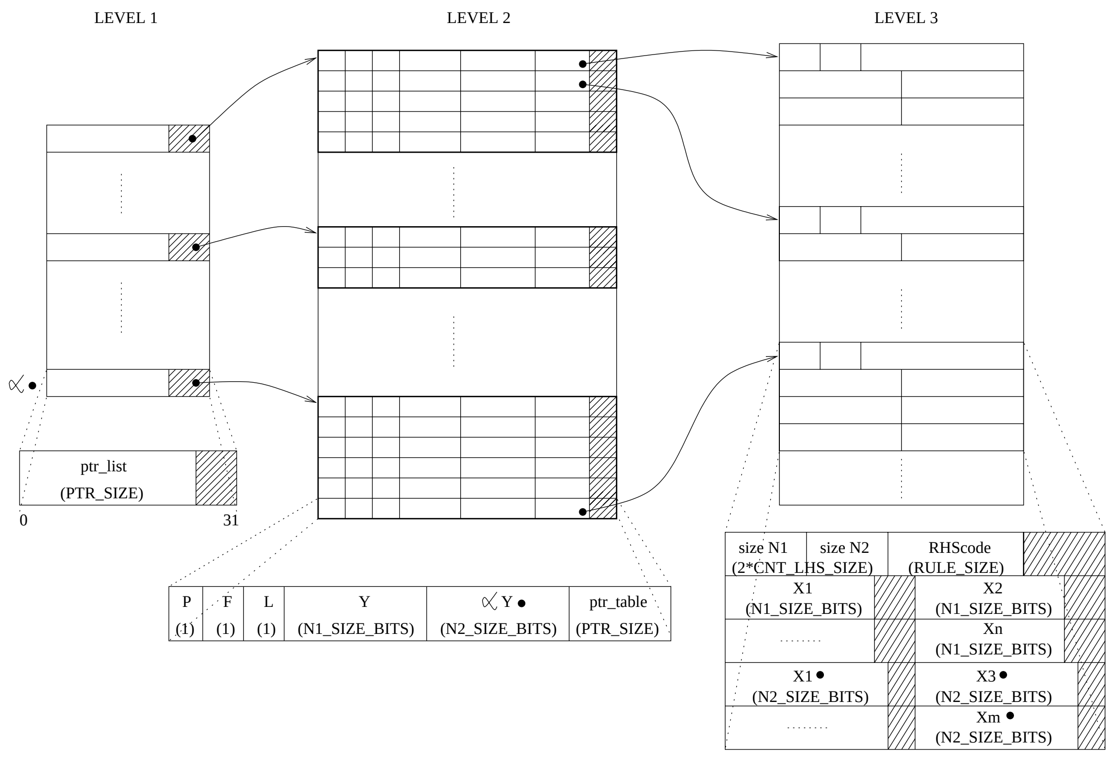


Figure 6.5: The data-structure used to represent a nplCFG without unitary rules.

parameter	maximal size [bit]	SUSANNE size	meaning
`PTR_SIZE`	up to 32	18	bits for a pointer
`N1_SIZE_BITS`	up to 16	11	bits for a non-terminal
`N2_SIZE_BITS`	up to 16	14	bits for a partial right-hand side
`CNT_LHS_BITS`	N/A	5	bits for the size of level3 lists
`RULE_SIZE`	N/A	5	bits for a rule right-hand side

Table 6.2: The maximal size of the parameters that define the nplCFG (without unitary rules) grammar data-structure. Parameter sizes in the particular case of the SUSANNE grammar.

possible continuations for the corresponding rule right-hand side $\alpha\bullet$. Note, that none of the table entries can contain a NULL pointer as a partial rule right-hand side always (i.e. by definition) has at least one non-terminal as possible continuation. Each table entry is represented on 4 bytes, that allows the construction of the physical address of the entry associated to the partial rule right-hand side $\alpha\bullet$ from its binary representation. Precisely, the physical memory address on the associated table entry is obtained by performing a 2 positions left-shift on the binary representation of the partial rule right-hand side $\alpha\bullet$. The number of bits required to represent a pointer is given by the parameter `PTR_SIZE` (see figure 6.5 and table 6.2). In the particular case of the SUSANNE grammar there are $15,851$ partial rule right-hand sides and the size of level 1 table is $63,404$ bytes.

The level 2 is a collection of lists, one for each partial rule right-hand side $\alpha\bullet$ in the grammar. The list associated to the partial rule right-hand side $\alpha\bullet$, has an entry for each non-terminal

Y for which there is a partial rule right-hand side $\beta\bullet = \alpha Y\bullet$ and/or rule right-hand side $\beta = \alpha Y$ in the grammar. The fields of an entry in a level 2 list (see figure 6.5) corresponding to the partial rule right-hand side $\alpha\bullet$ are:

- ▷ `P`: flag. '1' if $\alpha Y\bullet$ is a partial rule right-hand side, '0' otherwise;
- ▷ `F`: flag. '1' if αY is a rule right-hand side, '0' otherwise;
- ▷ `L`: flag. '1' the entry is the last in the list;
- ▷ `Y`: a non-terminal that is a possible continuation of the partial rule right-hand side $\alpha\bullet$.
- ▷ $\beta\bullet$: the new partial rule right-hand side $\alpha Y\bullet$;
- ▷ `ptr_table`: if $\mathtt{F} = 1$, a pointer to a table in level 3 containing (1) a code `RHScode` that uniquely identifies the rule right-hand side αY and (2) all the non-terminals X_i for which there is a grammar rule $X_i \to \alpha Y$ and all the $X_i\bullet$ in the grammar in this case;

Each entry in a level 2 list is represented on 8 bytes. Each flag requires 1 bit. The size in bits for the non-terminal Y is given by the parameter `N1_SIZE_BITS`, for the new partial right-hand side by the parameter `N2_SIZE_BITS` and for the pointer to the level 3 by the parameter `PTR_SIZE` (see figure 6.5 and table 6.2). While an entry in the level 2 is stored on 8 bytes, the following restriction should be satisfied: $3 + \mathtt{N1\_SIZE\_BITS} + \mathtt{N2\_SIZE\_BITS} + \mathtt{PTR\_SIZE} \leq 64$. In the particular case of the SUSANNE grammar the level 2 size is $217,080$ bytes.

Finally, level 3 is a collection of tables, one for each distinct rule right-hand side αY present in the grammar. Each level 3 table contains a header, a list of non-terminals and a list of partial rule right-hand sides. The fields of a level 3 table (see figure 6.5) corresponding to the rule right-hand side αY are:

- ▷ `sizeN1`: in header. Number of non-terminals in the list;
- ▷ `sizeN2`: in header. Number of partial rule right-hand sides in the list;
- ▷ `RHScode`: in header. A code that uniquely identifies the rule right-hand side αY to which the table corresponds;
- ▷ list of non-terminals: all the non-terminals X_i for which there is a grammar rule $X_i \to \alpha Y$;
- ▷ list of partial rule right-hand sides: all the partial rule right-hand sides $X_i\bullet$ in the grammar where X_i are elements of the non-terminals list;

A header in the level 3 table is represented on 4 bytes and both the non-terminals and the partial rule right-hand sides are represented on 2 bytes. The size in bits for the `sizeN1` and `sizeN2` is given by the parameter `CNT_LHS_SIZE`, for the `RHScode` by the parameter `RULE_SIZE`, for a non-terminal in the list of non-terminals by the parameter `N1_SIZE_BITS` and for the partial rule right-hand side by the parameter `N2_SIZE_BITS` (see figure 6.5 and table 6.2). While the header in the level 3 is stored on 4 bytes, the following restriction should be satisfied: $2 * \mathtt{CNT\_LHS\_SIZE} + \mathtt{RULE\_SIZE} \leq 32$. In the particular case of the SUSANNE grammar the level 3 size is $266,416$ bytes.

For a better understanding of the proposed data-structure the example bellow illustrates the data-structure organization for the nplCFG that was used in example 6.1 and explains how the search for a particular pair $(\alpha\bullet, Y)$ works .

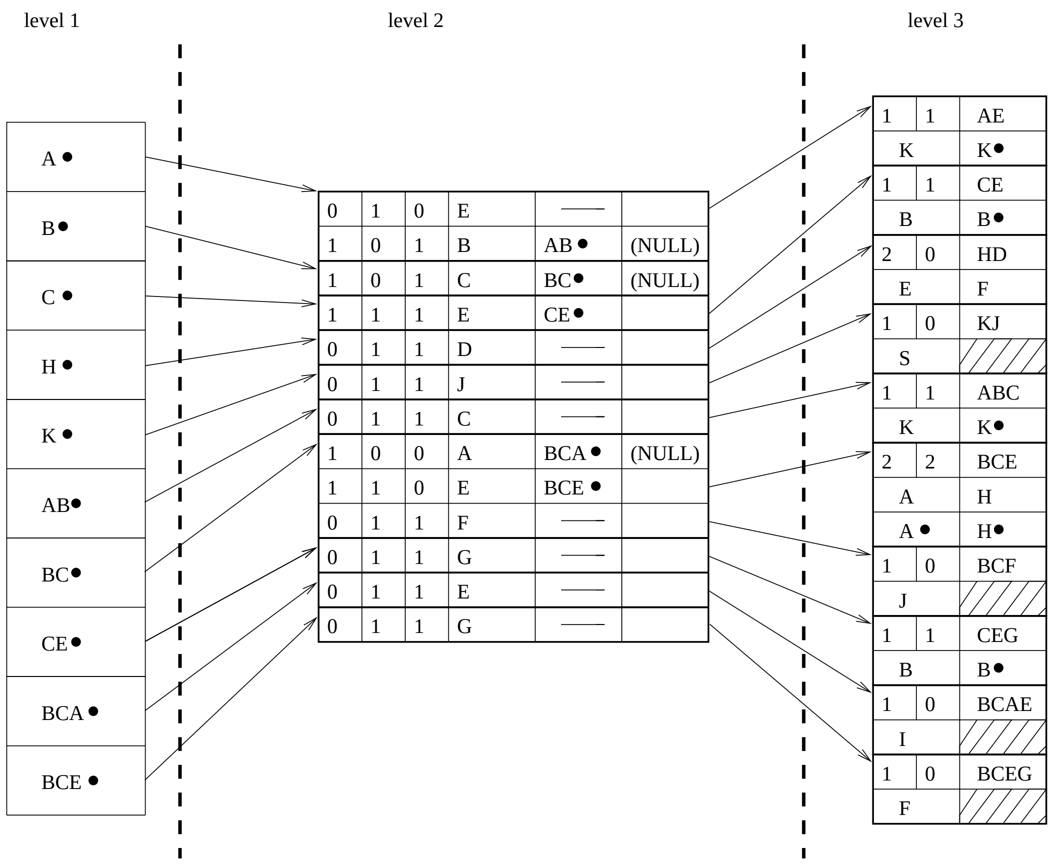


Figure 6.6: The data-structure representing the nplCFG without unitary rules used in example 6.1.

Example 6.2 We use the same grammar defined in example 6.1 – whose data-structure is illustrated in figure 6.6 – for illustrating how a processor uses the grammar data-structure to lookup the pair $(BC\bullet, E)$. The partial rule right-hand side $BC\bullet$ is used to index the level 1 table and retrieve a pointer to a level 2 list. The pointed level 2 list contains three items, and the second corresponds to the continuation of $BC\bullet$ with the non-terminal E. According to this list entry ($\texttt{P} = 1, \texttt{F} = 1$) there is both a partial right-hand side $BCE\bullet$ and a final BCE in the grammar. The pointer `ptr_table` in this list entry allows to retrieve from the level 3 table the set of non-terminals $\{A, H\}$ and the set of partial right-hand sides $\{A\bullet, H\bullet\}$. $\diamond$

6.4 Design units

6.4.1 The sequence generator (SEQ_GEN) unit

The sequence generator is presented in the enhanced-CYK block diagram (see figure 6.1) as the SEQ_GEN unit. The task of this unit is to generate all triples $(S1,S2,D)$ of two chart source cells $S1$ and $S2$ that need to be combined, along with their corresponding destination chart cell D, where the cell-combination result will be stored. The unit's functionality is identical with that described for the SEQ_GEN unit of the DAP design in section 5.3.1. The SEQ_GEN implemented within the enhanced-CYK design generates triplets with chronologically ordered source cells.

The output of the SEQ_GEN unit is forwarded to the CHECKER through a FIFO memory for decoupling the two units. A small FIFO is required (i.e. 2 to 4 words) as the SEQ_GEN unit is fast and will always keep the FIFO memory full. The size of this FIFO is configured in the VHDL code with the generic `OUT_GEN_DEPTH`.

6.4.2 The data-dependency checking (CHECKER) unit

The checker is presented in the enhanced-CYK block diagram (see figure 6.1) as the CHECKER unit. The unit's functionality is identical with that described for the DAP design in section 5.3.2 with some differences as follows.

As for the DAP design, when the CHECKER inserts a destination cell D in the D-TABLE (1) the CTX-memory is initialized with some "context information" related to the destination cell D and (2) a unique identifier is allocated to the destination cell D. In addition to these operations the CHECKER implemented within the enhanced-CYK design also initializes the CTXRAW-memory with the content of the destination cell D (both the $N1$ and the $N2$ sets).

<u>Note</u>: When parsing sentences, the sets $N1$ and $N2$ in all destination cells are empty and therefore the CTXRAW-memory is only initialized when parsing word lattices.

The information stored in the CTX-memory, CTXRAW-memory and D-TABLE associated to a destination cell D can be retrieved by using the unique identifier ID assigned to the destination cell D. The WRITER uses this information for reassembling the tile processing results.

As for the DAP design the output of the CHECKER is forwarded to the DISPATCHER unit through a FIFO memory for decoupling the two units. The size of this FIFO is configured in the VHDL code with the generic `OUT_CHK_DEPTH`.

6.4.3 The destination cells table (D-TABLE) unit

The destination cells table is represented in the enhanced-CYK block diagram (see figure 6.1) as the D-TABLE unit. The unit's functionality is identical with that described for the DAP design in section 5.3.6 with some differences as follows.

An entry in the D-TABLE has three fields: (1) a register $\texttt{TBL\_D\_REG}_\texttt{i}$ storing the row and column of a destination cell, (2) the counter `CNT_PROCESSING` and (3) the counter `CNT_WAITING`. The difference between the DAP and enhanced-CYK designs is related to the usage of the `CNT_PROCESSING` counter. Precisely, while for the DAP design the `CNT_PROCESSING` counter is counting the cell-combinations issued for that entry's destination cell, in the case of the enhanced-CYK design it is counting the issued tiles. Note, that when counting tiles the value of the `CNT_PROCESSING` counter cannot exceed the number of processors in the system.

The insertion and the deletion of D-TABLE entries works similarly for the DAP and enhanced-CYK designs. Each time a destination cell is inserted in the D-TABLE the `CNT_PROCESSING` counter is initialized on 0 and the `CNT_WAITING` counter is initialized with the row of the inserted destination cell D. Like for the DAP design, an entry in the D-TABLE is deleted (i.e. released) when both counters become 0.

6.4.4 The triplets buffer (POOL) unit

The POOL is presented in the enhanced-CYK block diagram (see figure 6.1) as the POOL unit. The unit's functionality is similar to the POOL unit used in the DAP design and presented in section 5.3.3.

In both the DAP and the current design, the POOL unit is key to the system's ability to dynamically allocate the processors to process cell-combinations. Without the POOL unit the system will hang on the first cell-combination that does not satisfy a data-dependency constraint and keep all the remaining cell-combinations – even if they satisfy their data-dependency constraints – unissued. On the other hand, by using the POOL unit the system can search over a larger number of cell-combination that can be potentially issued and eventually keep all the processors busy.

However, for the enhanced-CYK design it seems a priori that the POOL unit is less important – when comparing to the DAP design – due to the design's ability to balance the processor load by using the tiling mechanism. Concretely, there are less chances for a cell-combinations not to satisfy a data-dependency constraint that was usually the case when large cell-combinations were processed and therefore to require to be stored in the POOL. This will however be investigated in section 6.6.5.

6.4.5 The task dispatching (DISPATCHER) unit

The task dispatcher is represented in the enhanced-CYK block diagram (see figure 6.1) as the DISPATCHER unit. Its purpose is the same as for the DAP design, namely to dispatch tasks to the processors in the system. However, in order to implement the ability of tiling (large) cell-combinations the DISPATCHER has been redesigned from scratch, being one of the major changes in the new design. The new DISPATCHER unit is built of two stages: the first stage is called READER and the second TILER. The separation of the DISPATCHER in two stages aims to pipeline the source cell prefetching with their tiling and distribution to the processors. The time spent for accessing the source cells is made thus transparent, partly by the pipeline and partly by the high bandwidth communication – due to a 64-bit databus – with the chart memory. We will further discuss the two stages of the DISPATCHER.

6.4.5.1 The prefetching (READER) stage

The READER stage reads from the chart memory, for each incoming triplet $(S1, S2, D)$, the set $N2$ for the source cell $S1$ and the $N1$ set for the source cell $S2$ and further stored them in a buffer memory. The enhanced-CYK design uses 16-bit to represent a non-terminal ($N1$-item), respectively a partial right-hand side ($N2$-item), and therefore the 64-bit databus with the chart memory allows the READER to fetch 4 $N1$ or $N2$-items per read cycle. In the particular case of SUSANNE grammar, the maximal size for the sets $N1$ and $N2$ was established at 128 (see appendix B.4), and therefore a number of $128/4 = 32$ chart memory read cycles are sufficient to fetch a set $N1$ or $N2$ of any size and 64 read cycles to fetch both the $N2$ and $N1$ sets for a triplet. However, for real-life grammars less read cycles are usually sufficient for fetching these sets, as they do not contain a large number of elements.

The READER stage stores the set $N2$ from the source cell $S1$ and the set $N1$ from the source cell $S2$ in a dual-port buffer memory which is organized in two banks $N1$-bank and $N2$-bank as depicted in figure 6.7. For a given triplet $(S1, S2, D)$, the $N1$-bank stores the $N1$ set from the source cell S2, and the $N2$-bank stores the $N2$ set from the source cell S1.

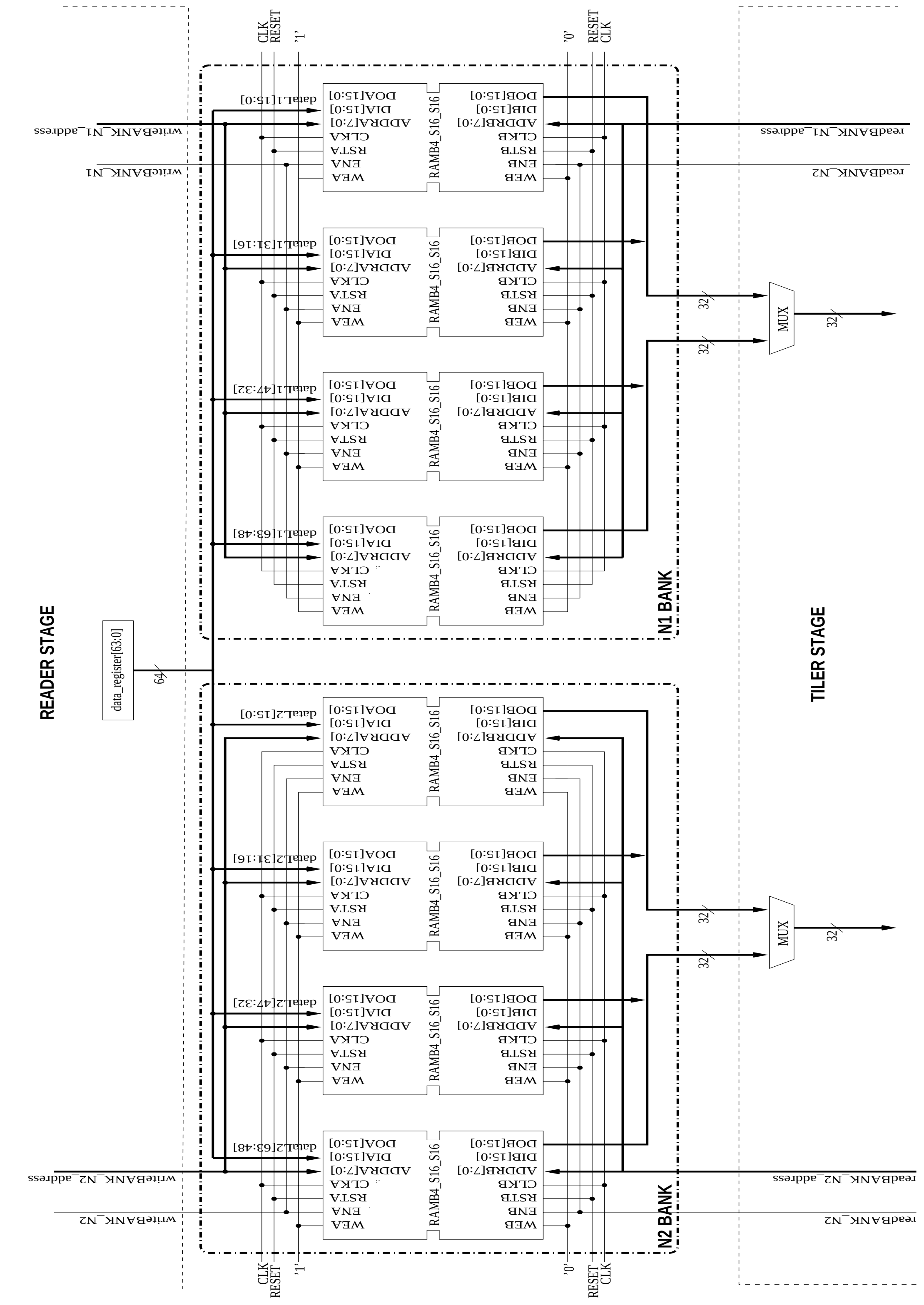


Figure 6.7: A dual-port memory buffer used for overlapping the triplet prefetching (READER stage) with their tiling and dispatching (TILER stage)

Both the $N1$-bank and the $N2$-bank are built with 4 `RAMB4_S16_S16` primitives[4]. The size of a `RAMB4_S16_S16` primitive is 4096 bit – organized as 256 words of 16 bit – and therefore both the $N1$-bank and the $N2$-bank have a size of 2 KByte.

The number of sets that can be stored in the $N1$-bank and $N2$-bank depends on the maximal size of the $N1$ and $N2$ set which also depends on the used grammar. For simplicity we consider that both the $N1$ and $N2$ sets have the same maximal size (i.e. the size of the largest). For instance, in the particular case of the SUSANNE grammar a maximal set size of 128 is used. The maximal size of these sets should be a power of 2 and is configured in the VHDL code by means of the `CELLSIZE_BITS` generic.

Table 6.3 tabulates for different maximal set sizes the value of parameter `CELLSIZE_BITS` and respectively the number of such sets that can be stored in the $N1$-bank, respectively $N2$-bank. In the particular case of the SUSANNE grammar a maximal set size of 128 is used (`CELLSIZE_BITS` $= 7$) and both the $N1$-bank and the $N2$-bank, can store a number of $256 * 4/128 = 8$ $N1$, respectively $N2$ sets.

$N1/N2$ set size	`CELLSIZE_BITS`	# of set entries/bank
32	5	32
64	6	16
128	7 (configured for SUSANNE)	8
256	8	4
512	9	2

Table 6.3: The value of parameter `CELLSIZE_BITS` to be configured for different maximal $N1/N2$ set sizes and the number of sets that can be accommodated in a bank.

<u>**Note:**</u> If it is necessary to store larger (or more) sets in the buffer memory more `RAMB4_S16_S16` primitives can be used for building the buffer memory. This requires, however, to slightly modify the READER and TILER in order to cope with such changes.

The internal organization of the $N1$-bank and $N2$-bank is similar, on the assumption that (1) both the $N1$-items and the $N2$-items are represented on 16 bit and (2) the size of the $N1$ and $N2$ sets are the same. Figure 6.8 illustrates the internal organization of the $N1$-bank (respectively $N2$-bank) in the particular case when the parameter `CELLSIZE_BITS` $= 7$. With this internal organization of the banks, each 64 bit word read by the READER from the chart memory, containing 4 $N1$-items (when the $N1$ set is read) is stored "at once" in the $N1$-bank, one $N1$-item in each `RAMB4_S16_S16` memory primitive. Similarly, each 64 bit word read by the READER from the chart memory, containing 4 $N2$-items (when the $N2$ set is read) is stored "at once" in the $N2$-bank, one $N2$-item in each `RAMB4_S16_S16` memory primitive.

Assuming that the set entry 3 is allocated to an incoming triplet $(S1, S2, D)$ the READER reads the set $N2$ from $S1$ and stores it in the set entry 3 of the $N2$-bank. Also, it reads the set $N1$ from $S2$ and stores it in the same set entry 3 of the $N1$-bank.

The buffer memory is supervised by a manager that keeps the evidence of the allocated set entries. If there are free set entries in the buffer memory, the READER reads the next triplet from the CHECKER's FIFO and asks the buffer memory manager to allocate one of the free

[4]We assume that a Xilinx Virtex-E XCV2000 FPGA is used. See details in [30].

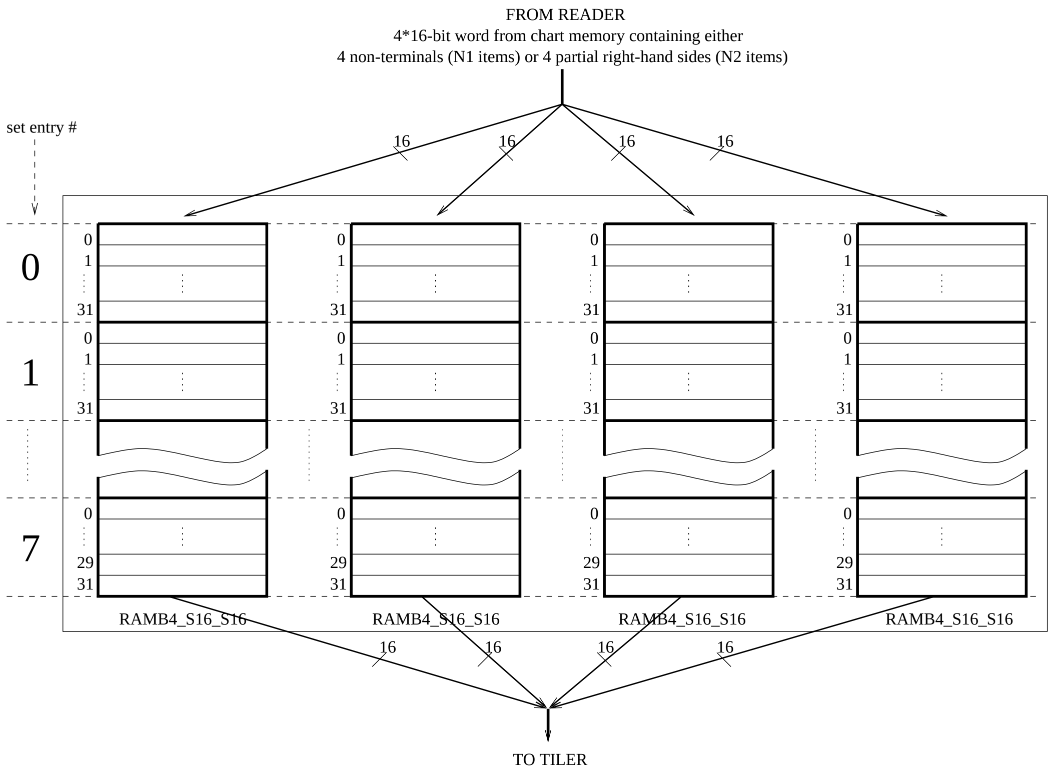

FROM READER
4*16-bit word from chart memory containing either
4 non-terminals (N1 items) or 4 partial right-hand sides (N2 items)
set entry #
16
16
16
16
0
1
7
0
1
31
29
RAMB4_S16_S16
RAMB4_S16_S16
RAMB4_S16_S16
RAMB4_S16_S16
16
16
16
16
TO TILER

Figure 6.8: Internal $N1$-bank/$N2$-bank organization in the particular case when the set size is 128 ($\texttt{CELLSIZE\_BITS} = 7$).

set entries for this triplet. On the other hand, if the buffer memory is full the READER will wait until the manager releases a set entry[5]. Each time the buffer memory manager allocates a set entry an identifier corresponding to the allocated set entry (i.e. the index) is returned. Once the sets in the source cells of the triplet are read and stored in the buffer memory, the triplet and the identifier is forwarded through an internal FIFO to the TILER stage. The TILER uses the identifier for accessing the sets stored in the buffer memory and the internal FIFO insures that the triplets are tiled and dispatched to the processors in the same order in which they come from the CHECKER, regardless of the place where they were stored in the buffer memory. The size of the internal FIFO memory is equal to the number of set entries in the buffer memory.

6.4.5.2 The cell-combinations tiling (TILER) stage

As seen in the design analysis of the DAP design (see section 5.6.1), the size of the cell-combinations may vary a lot and when forwarded as such to the processors and this typically results in an unbalanced processor load. Tiling is implemented within the current design architecture as a means of balancing the processor load. The basic idea is to split the (large) cell-combinations in smaller chunks, called tiles, and to send these tiles for processing to the processors. As the size of these tiles is more uniform the processor load has more chances to be balanced. One can see the tiling procedure as a dynamic clustering of the available processors around demanding (i.e. large) cell-combinations. That is, a way of concentrating the processing

[5]A set entry is released on the request of the tiler stage as soon as a triplet was tiled and dispatched.

resources when and where needed.

In general for a tile size $m \times n$, a cell-combination given by the sets $N2$ and $N1$ is divided in $\lceil |N2|/m \rceil \times \lceil |N1|/n \rceil$ tiles. The figure 6.9 illustrates two possible tilings of a cell-combination given by $|N2| = 23$ and $|N1| = 11$, one with a tile size 4×8 (left) and the second with a tile size 2×6 (right). Note, that the boundaries of the tiled cell-combination are not

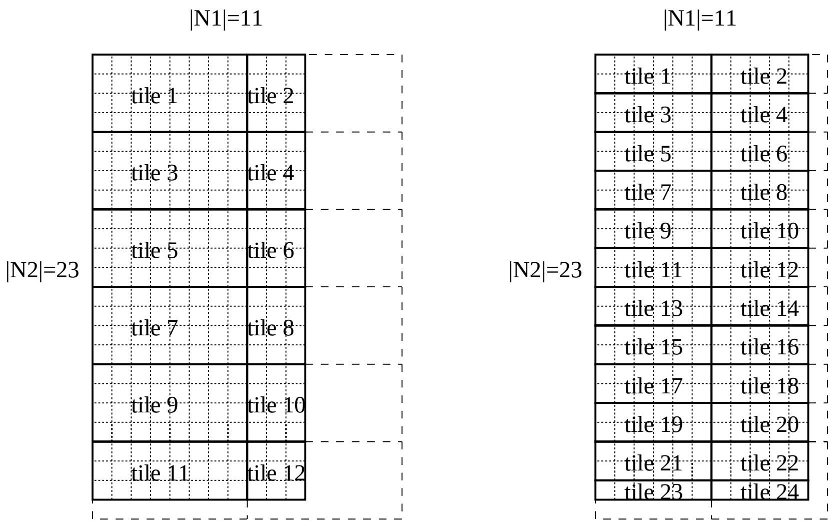


Figure 6.9: Two possible tilings for a cell-combination given by $|N2| = 23$ and $|N1| = 11$ when a tile size 4×8 is used (left) and when a tile size 2×6 is used (right).

necessarily matched and tiles smaller than the used tile size may result. The size of the tile can be configured in the VHDL code by means of two parameters `MODULO_N2` for the set $N2$ and `MODULO_N1` for the set $N1$. The only restriction on these two parameters is that they should be a multiple of 2. As a rule of thumb, in practice the table size should be neither too small nor too large. The influence of the tile size on the system performance is investigated in section 6.6.4

The second task of the TILER is to dispatch the tiles to the processors. The tile dispatching mechanism is similar with the one implemented in the DISPATCHER unit of the DAP design (see section 5.3.4 for details). Basically, for finding an idle processor the TILER performs a continuous polling over all the processors in the system. When an idle processor is found the TILER reads the content of the two banks from the buffer memory for building a tile. Due to the physical separation of the two banks in the buffer memory the TILER reads these banks in parallel. The tile is sent to the idle processor along the ID associated with the cell-combination to which the tile belongs. The D-TABLE entry associated with the destination cell D of the cell-combination to which the tile belongs is updated after each dispatched tile by incrementing the `CNT_PROCESSING` counter. Also, when the last tile of a cell-combination is dispatched the `CNT_WAITING` counter in D-TABLE entry associated to the destination cell D is decremented.

6.4.6 The processor

In the current design a processor is working on tiles as a means of balancing processor load. As illustrated in the enhanced-CYK block diagram (see figure 6.10) a processor is interfaced to the DISPATCHER which distributes the processing tasks (i.e. tiles), to the local grammar memory used for grammar rules lookup during the parsing and to the WRITER which is the interface used for storing the processing results into the chart memory.

The interface with the DISPATCHER: The processor receives tiles for processing from the

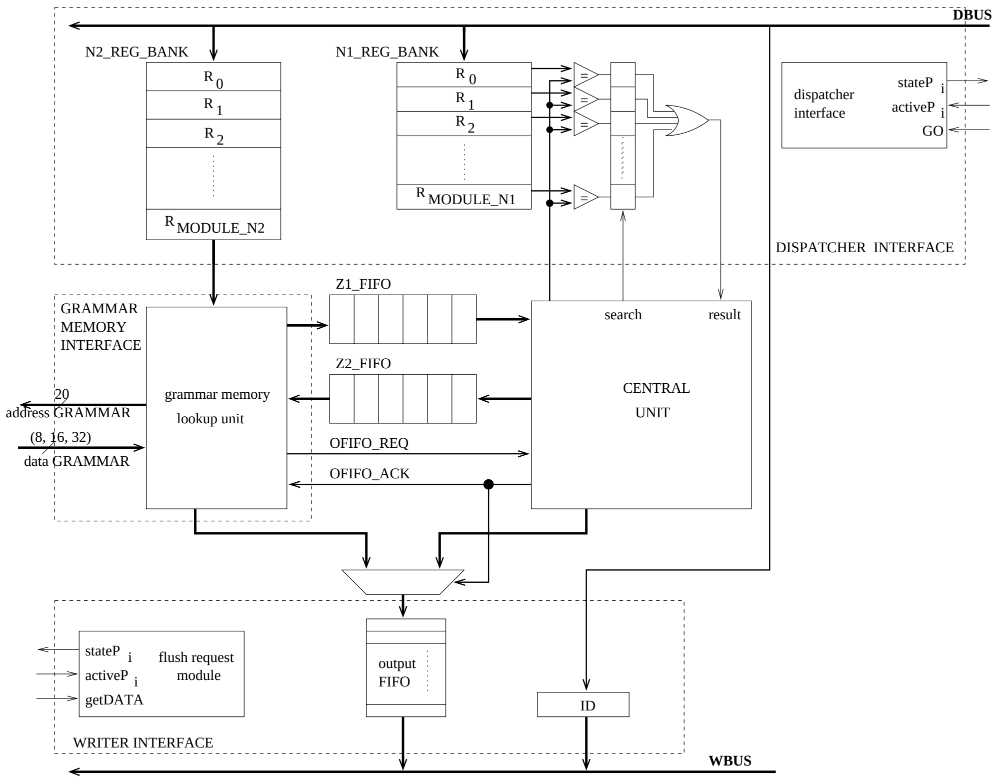


Figure 6.10: The enhanced-CYK processor datapath.

DISPATCHER. The later obtains the tiles for a certain cell-combination $(S1, S2, D)$, where $N2 \in S1$ and $N1 \in S2$, by rewriting $N2$ as $N2 = N2^{(1)} \bigcup N2^{(2)} \bigcup \ldots \bigcup N2^{(m)}$ and $N1$ as $N1 = N1^{(1)} \bigcup N1^{(2)} \bigcup \ldots \bigcup N1^{(n)}$ and taking all possible pairings $N2^{(i)} \times N1^{(j)}$ for $i \in \overline{1,m}$ and $j \in \overline{1,n}$. Remember that the tile size is configured in the VHDL code by means of two generics `MODULO_N2` for the set $N2$ and `MODULO_N1` for the set $N1$. The tile $N2^{(i)} \times N1^{(j)}$ received by a processor is stored in two banks of registers (see figure 6.10). The first bank `N2_REG_BANK` of size `MODULO_N2` stores the set $N2^{(i)}$ and the second bank `N1_REG_BANK` of size `MODULO_N1` stores the set $N1^{(j)}$. The tiles are transfered from the DISPATCHER to a processor by means of the `DBUS` bus (see section 5.3.4). Once the tile was transfered and loaded in the two banks of registers the signal `GO` issued by the DISPATCHER will start the processing of the tile.

The interface with the grammar memory: The processing of a tile $N2^{(i)} \times N1^{(j)}$ consists on grammar lookups and internal data movement and works as follows. The grammar memory lookup unit takes a partial right-hand side $\alpha\bullet$ stored in the `N2_REG_BANK`, and retrieves from the grammar memory all the entries[6] (`F`,`P`,`L`,Y,$\beta\bullet$,`ptr_table`) in the level 2 list (see section 6.3.2) corresponding to this partial right-hand side. Depending on the databus width of 8, 16 or 32-bits with the grammar memory, it takes 8, 4 and respectively 2 cycles to fetch a level 2 entry.

[6]The last entry is detected when the flag L is set on '1'.

Each level 2 entry is written as such (without the flag `L`) into the Z1-FIFO memory. The processor's central unit reads an item (`F`,`P`,Y,$\beta\bullet$,`ptr_table`) from the Z1-FIFO and compares the non-terminal Y "at once" against all the non-terminals stored in the `N1_REG_BANK` by activating the signal `search` (see figure 6.10) that latches in a register the comparison results of Y against each entry of the `N2_REG_BANK`. The result of this search is available on the signal `result` and if the non-terminal Y was found in the `N1_REG_BANK` one or even both of the cases listed bellow occur:

- ▷ if `P`='1': means that the partial right-hand side $\beta\bullet = \alpha Y\bullet$ is in the grammar. In this case the ($\alpha\bullet$, Y, $\beta\bullet$) item is stored in the output FIFO; Note that, the ID and the information contained in such an item allows the extraction of the compact parse forest;
- ▷ if `F`='1': means that the αY is a rule right-hand side. In this case the item ($\alpha\bullet$, Y, `ptr_table`) is returned through the Z2-FIFO to the grammar memory lookup unit;

If the non-terminal Y is not found in the `N1_REG_BANK`, nothing happens.

The field `ptr_table` in the items ($\alpha\bullet$, Y, `ptr_table`) returned to the grammar memory lookup unit is used to retrieve from the grammar memory level 3 tables all the non-terminals X_i for which there exists a grammar rule $X_i \to \alpha Y$ and all partial right-hand sides $X_i\bullet$ in this case. A code `RHScode` that uniquely identifies the rule right-hand side αY is also retrieved in this case. All the items retrieved from the grammar memory level 3 tables are stored in the output FIFO. While both the grammar memory lookup unit and the central unit may concurrently write data in the output FIFO, the access to the output FIFO is assigned permanently to the central unit and is requested when needed by the grammar memory lookup unit. The signal `OFIFO_REQ` and `OFIFO_ACK` are used for this purpose.

The interface with the WRITER Within the current design, the processors do not access directly the chart memory. Instead, they use the WRITER as an interface for storing the processing results into the chart memory. The reason is to reduce the number of collisions between processors and therefore to reduce the expected performance degradation phenomenon we have seen on the previous implementations of the CYK algorithm.

When the output FIFO is full (or almost full) the flush request module asserts the $\mathtt{stateP_i}$ (on the WRITER interface) signal to request the output FIFO flush. The WRITER will flush the processor's output FIFO – using the `WBUS` – as soon as it pools the processor and detects the request. The $\mathtt{stateP_i}$ signal is also used to indicate that a processor has finished the processing of the tile. During a tile processing it is possible that the output FIFO becomes full even if the tile processing is not finished in which case the WRITER only flushes the content of the output FIFO. A flush is also requested when the output FIFO of the processor that finished to process a tile is not empty. A processor that finished to process a tile and was flushed becomes immediately available for processing other tiles.

6.4.7 The WRITER and LOOKUP units

The WRITER and LOOKUP units (see the enhanced-CYK block diagram figure 6.1) are used as an interface for storing the parsing results into the chart memory. The LOOKUP unit is under the control of the WRITER unit and for this reason the two units are discussed together. The task of the WRITER is to flush the processor parsing results – from the processor's output FIFO – and to store these results in the chart memory in the destination cell identified by the

ID associated to the flushed processor. Recall that this ID was allocated by the CHECKER and forwarded by the DISPATCHER to the processor along the dispatched tile.

A flushed item can be either a partial right-hand side or a non-terminal and in either of the two cases it should only be stored once in the destination cell set $N2$ or respectively $N1$. Checking for the occurrence of a partial right-hand side in the $N2$ set, and respectively for a non-terminal in the $N1$ set of a destination cell is implemented in the LOOKUP unit by means of an associative memory. The associative memory can be loaded either with the $N2$ set or with the $N1$ set of a destination cell, but not both at the same time given the large amount of required hardware resources. The sets $N1$ and $N2$ in a destination cell are retrieved from the CTXRAW-memory with the identifier ID associated to that destination cell. When the LOOKUP unit is loaded with the $N1$ set of a destination cell the WRITER can check "at once" for the unique occurrence of non-terminals and when loaded with the $N2$ set the WRITER can check for the unique occurrence of partial right-hand sides. The size of the associative memory is equal to the constant K representing the maximal size of a set $N1/N2$ in a chart cell. In the particular case of the SUSANNE grammar this constant is 128.

Due to the fact that at each moment during runtime there are several destination cells under processing and that there is no special order on the sequence of destination cells in which the processors results are flushed, the WRITER requires the ability to deal (i.e. write) with the destination cells in any order – the order in which the processors finish to process the tiles. In order to support this ability, when passing from a destination cell (ID1) to another (ID2) the WRITER performs the following three operations:

1. stores its current state – that of the destination cell (ID1) – in the CTX-memory;
2. restores the state of the destination cell (ID2) from the CTX-memory;
3. loads the LOOKUP with the $N1$ or $N2$ sets – in turn – of the new destination cell ID2. These sets are retrieved from the CTXRAW-memory;

The sequence of three operations (1)-(2)-(3) presented above is referred henceforth as a context switch. Note that a context switch is only required when two successively flushed processors have different IDs. The physical address in the CTX-memory and CTXRAW-memory where the information related to a destination cell is stored is built from the associated ID.

The WRITER employs two memory buffers in order to overlap the fetching of the parsing results (from processors) with their writing into the chart memory. The first buffer (FB, from Fetch Buffer) is used to store the parsing results fetched from a processor's output FIFO, while the second buffer (LB, from Lookup Buffer) is used by the WRITER with the LOOKUP unit. Each buffer is labelled with the ID of the processor whose parsing results it stores. The buffers are swapped as soon as the WRITER has finished to operate on the LB buffer and the FB contains new parsing results. A context switch takes place if after swapping the buffers the new ID of the LB buffer is different from the previous.

Once the WRITER's context is switched – if necessarily – the associative memory in the LOOKUP unit is already initialized with the $N1$ set of the destination cell (ID), where ID tags the content of the current LB buffer. The WRITER starts to process the non-terminals (i.e. the $N1$-items) stored in the LB buffer. If a non-terminal does not occur in the associative memory it is stored both in the associative memory and the CTXRAW-memory (this takes place in parallel). When the WRITER has finished to process the $N1$-items, it loads the associative memory with the $N2$ set of the destination cell (ID) and starts to process the partial right-hand sides (i.e. the $N2$-items) stored in the LB buffer. If a partial right-hand side does not occur in the associative memory it is stored both in the associative memory and the CTXRAW-memory (this takes place in parallel). When the WRITER has finished to process the $N2$-items, two situations may occur:

1. the tile from BL was not the last tile (to be processed) for the destination cell ID;
2. the tile from BL was the last tile (to be processed) for the destination cell ID;

In the first case the WRITER decrements the `CNT_PROCESSING` counter (in the D-TABLE). In the second case the content of the CTXRAW-memory (the $N1$ and the $N2$ sets) is dumped into the chart memory destination cell associated to the identifier ID. The ID, the entry in the D-TABLE, CTX-memory and CTXRAW-memory corresponding to the destination cell (ID) are released. The WRITER detects that the flushed processor processed the last tile if both the counters `CNT_PROCESSING` and `CNT_WAITING` in the D-TABLE are 0, that is, no cell-combinations are left for processing and no other tiles are under processing (for details on the D-TABLE see section 5.3.6).

At this point, if the FB contains new processing results, the FB will be swapped with the LB buffer and the procedure described above repeats.

6.4.8 The compact parse trees extractor (EXTRACTOR) unit

The EXTRACTOR unit (see the enhanced-CYK block diagram in figure 6.1) is the interface between the enhanced-CYK design and the host system and the information it sends to the host machine allows the later to rebuild the parsing forest.

When the WRITER flushes the output FIFO, i.e. parsing results, of a processor it forwards a copy of each item to the EXTRACTOR. The EXTRACTOR further packs the parsing results according to the width of the databus used to interface the enhanced-CYK design and the host machine in order to best exploit the available bandwidth. For instance if a 32-bit PCI interface is used the parsing results should be packed on 32-bit words and if a 16-bit SCSI interface is used the parsing results should be packed on 16-bit words.

6.4.9 The enhanced-CYK system monitor (MONITOR) unit

During the physical testing of the previous designs on the RC1000-PP board it was noted that it is useful to have a mechanism that can monitor the state (i.e. normal or faulty) of the system and allow to recover from faults and eventually track their source. The absence of such a mechanism, made very difficult the task of detecting a faulty system and the only way of recovering from such a state was to perform a manual system reset.

Such a monitoring mechanism is integrated within the enhanced-CYK design and is implemented in the MONITOR unit (see figure 6.1). The internal structure of the MONITOR unit is illustrated in figure 6.11 and allows to monitor the system's state and in the case of a faulty system to eventually track the unit which is the cause of the disfunctionality. The monitor unit is very useful during design testing and allows an easier and faster integration of extension units in the design. In order to be integrated (i.e. cope with) within the monitoring system each design unit uses an output error signal that is activated only if an abnormal internal condition is encountered. A unit may actually generate several error signals – one for each particular error condition. For instance, the WRITER activates the error signal if the allocated maximal cell size is exceeded for a certain cell destination and a processor activates the error signal if an entry in the Z1-FIFO has both the `P` and `F` flags set on '0'. A unit may activate as many error signals as needed, .

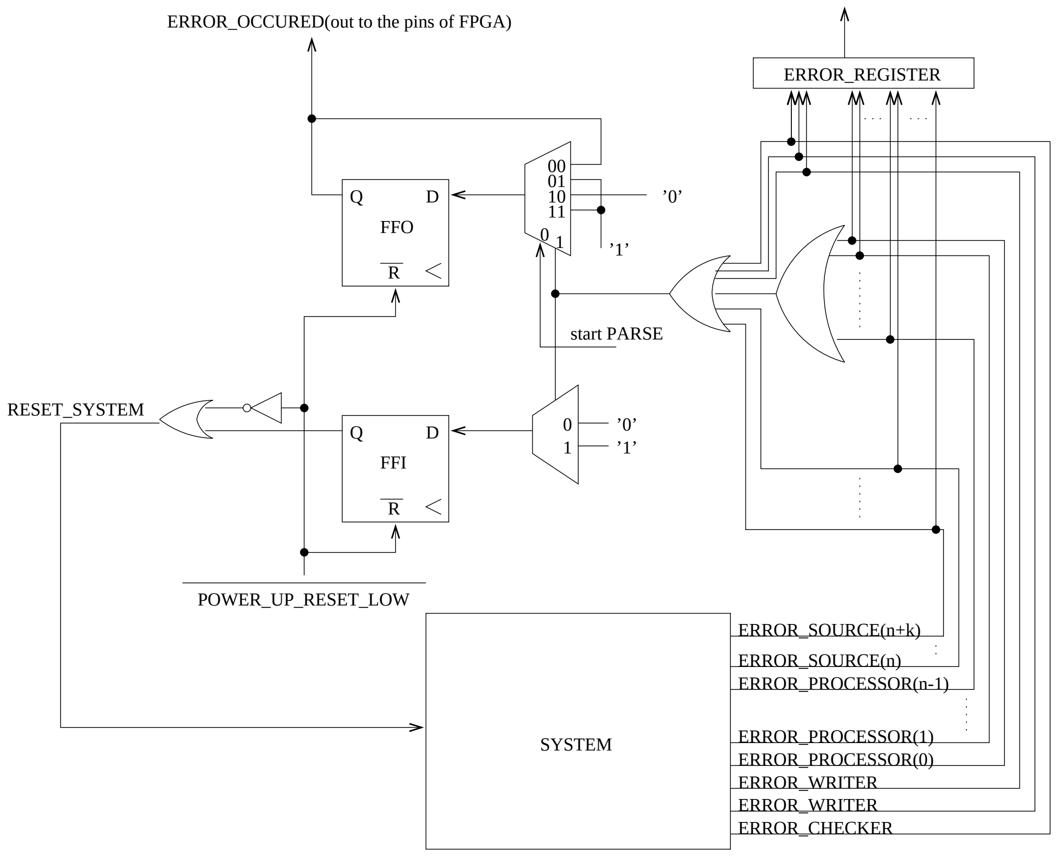


Figure 6.11: The monitoring system implemented within the enhanced-CYK design.

If a unit – integrated within the monitoring system – activates an error signal both the internal flip-flop `FFI` and the external flip-flop `FFO` are set. The `FFI` flip-flop is used to synchronize the error signal to the clock and its output is routed to the global system reset `RESET` signal used to bring the system in an initial state – from which new parsings can (re)start. The `RESET` signal is activated either by the output of the `FFI` flip-flop or by the FPGA's global initialization signal $\overline{\texttt{POWER\_UP\_RESET}}$. Once the system is reset by the internal `RESET` signal, the error signal is deactivated (due to the system reset) and the `FFI` flip-flop is reset. The `FFO` flip-flop routes the error signal to an FPGA pin that is used to signal the host system about the faulty state. The `FFO` flip-flop is not reset by the internal `RESET` signal which gives the required time to the host machine to react at the encountered error. The `FFO` flip-flop is reset when a new parsing starts – when the signal `startPARSE` is activated.

In order to track the source of errors, the error signals are latched in a register each time the `FFI` and `FFO` flip-flops are set. They can be read by the host system in order to check which error signal was activated and therefore to track its source.

The proposed mechanism is useful for monitoring the state of the system and for creating a reliable environment.

6.5 Performance measurements

The tests and performance measurements presented in this section are performed with the SUSANNE grammar without unitary rules – containing $1,912$ non-terminals and $66,123$ rules. The data-structures used to represent the chart and the SUSANNE grammar without unitary rules in the current design are presented in section 6.3.1 and section 6.3.2 respectively. The size of the memory required to store the SUSANNE grammar data-structure is $546,900$ bytes and the size of the chart memory depends on the length of the sentence we want to parse (e.g. 256 KBytes for parsing sentences with up to 32 words or 16 MBytes for parsing sentences with up to 256 words). The maximal length of the sentences that can be parsed with the enhanced-CYK design is independent of the number of processors in the system. In other words, any number of processors can be used for parsing sentences of any length in the condition that the amount of memory available for storing the chart is enough.

In order to determine the size and the clock frequency at which the system is able to work, a 14-processors system configuration was synthesized[7] and placed&routed[8] in a Xilinx FPGA, Virtex XCV2000efg-1156. The synthesis of the 14-processors system was made on a design instance characterised by the parameters given in table 6.4. The table 6.5 gives for a Xilinx Virtex XCV2000efg-1156 FPGA a summary of the resources used by each unit in the synthesized 14-processors system. The overall amount of resources required by a 14-processors system are also given. The hardware run-times presented were obtained by simulating[9] the VHDL model of an enhanced-CYK system with 4, 7, 10 and respectively 14 processors. Each of the simulated systems is running at a 50 MHz clock frequency, although the clock frequency at which the systems can work (i.e. the clock reported by the place&route tool) is 60 MHz. A 50 MHz clock is used in order to compare the enhanced-CYK design performance with the performance of the previous CYK designs. A 6×12 tile size and a POOL with 16 entries are used.

The software used for comparison is an implementation of the enhanced-CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The software uses the SUSANNE grammar in its original context-free form, while the hardware uses the same grammar but without unitary rules. In order to compare the performance of the enhanced-CYK hardware design with the performance of the previous CYK hardware designs we use two platforms for running the software: the first is a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz – that was also used to benchmark the CYK designs, and the second is a PC (DELL Dimension 4300) running the Linux OS (Mandrake 8.1) with 128 MByte memory, 576 MByte virtual memory and a PENTIUM Intel IV processor at a clock frequency of 1.5 GHz. For these simulation, the initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* `C` library function and not by profiling the code.

For the purpose of the comparison, $2,043$ sentences were parsed and validated[10]. Among these sentences 22 did not parse because at some point during the parsing the size allocated for a chart cell was exceeded (see section 6.3.1). The sentences have a length ranging from 3 to 15 and were all taken from the SUSANNE corpus. The hardware performance (i.e. run-times) of an enhanced-CYK system containing 4 processors (denoted as `hard_P4`), 7 processors (denoted as `hard_P7`), 10 processors (denoted as `hard_P10`) and respectively 14 processors (denoted as `hard_P14`) were compared against the software run-times.

[7]With LeonardoSpectrum v2000.1a2

[8]With Design Manager (Xilinx Alliance Series 2.1i)

[9]With ModelSim EE/Plus 5.5c

[10]The hardware output was compared to the software output for detecting mismatches

	Parameter	Size[units]	Details
system	CLOCK	50 [MHz]	the system clock
	COORD_BITS	5 [bits]	bits used to represent a row/column in the chart
	PROCESSORS	14	processors in the system
	MODULO_N1	12	tile size, $N1$ set, see section 6.4.5.2
	MODULO_N2	6	tile size, $N2$ set, see section 6.4.5.2
chart memory	CYKMEMSIZE	278,528 [bytes]	the required size for the chart memory for sentences of maximum 32 words
	CYKLATENCY	15 [ns]	the access-time of the SRAM memory used to store the chart
	CELLSIZE_BITS	7 [bits]	maximal size of a chart $N2/N1$ set, see section 6.3.1
	CYK_ADR_SIZE	16 [bits]	chart data-structure pointer size, see section 6.3.1
	CYK_WAIT_CYCLES	2	delay until chart memory data lines are stable
	GMEMSIZE	$546,900$ [bytes]	the size of each grammar memory
grammar memory	GLATENCY	15 [ns]	the access-time of the SRAM memory used to store the grammar memory
	N1_SIZE_BITS	11 [bits]	number of bits representing a $N1$-item, see section 6.3.2
	N2_SIZE_BITS	14 [bits]	number of bits representing a $N2$-item, see section 6.3.2
	NT	1,912	number of non-terminals ($N1$-items)
	RULE_SIZE	15 [bits]	bits used to represent a distinct right-hand side, see section 6.3.2
	CNT_LHS_SIZE	5 [bits]	bits used to count the number of $N1/N2$-items that may occur in a level 3 grammar table, see section 6.3.2
processor	FIFO_Z1_DEPTH	8	processor internal FIFO, see section 6.4.6
	FIFO_Z2_DEPTH	8	processor internal FIFO, see section 6.4.6
	OFIFO_DEPTH	16	processor output FIFO, see section 6.4.6
other	ENTRIES	16	entries in D-TABLE
	OUT_GEN_DEPTH	16	size of FIFO at output of SEQ_GEN
	OUT_CHK_DEPTH	16	size of FIFO at output of CHECKER
	OUT_POOL_DEPTH	16	size of FIFO from POOL to CHECKER
	OUT_WRITER_DEPTH	16	size of FIFO from WRITER to EXTRACTOR
	POOL_ENTRIES	8	size of the POOL

Table 6.4: The parameter values used to configure (i.e. instantiate) a 14-processors enhanced-CYK system.

component	DFFs/Latches	FGs	CLBs	XCV2000 area utilization (%)
IOctrl	23	18	12	0.06
SEQ_GEN	57	82	41	0.21
SEQ_GEN out FIFO	15	23	12	0.06
CHECKER	1139	1498	749	3.90
CHECKER out FIFO	23	24	12	0.06
DISPATCHER	258	303	152	0.79
POOL	213	393	197	1.03
POOL out FIFO	15	23	12	0.06
D-TABLE	1169	2677	1339	6.97
CTX	(uses 6 Block SelectRAMs/see Xilinx's XCV2000 Manual [30])			
CTXRAW	(uses 32 Block SelectRAMs/see Xilinx's XCV2000 Manual [30])			
WRITER	227	510	255	1.33
LOOKUP	2694	9766	4883	25.43
EXTRACTOR	8	18	9	0.05
MONITOR	4	12	12	0.03
processor	576	735	368	1.92
14-processors system	13976	25025	12513	**65.17**

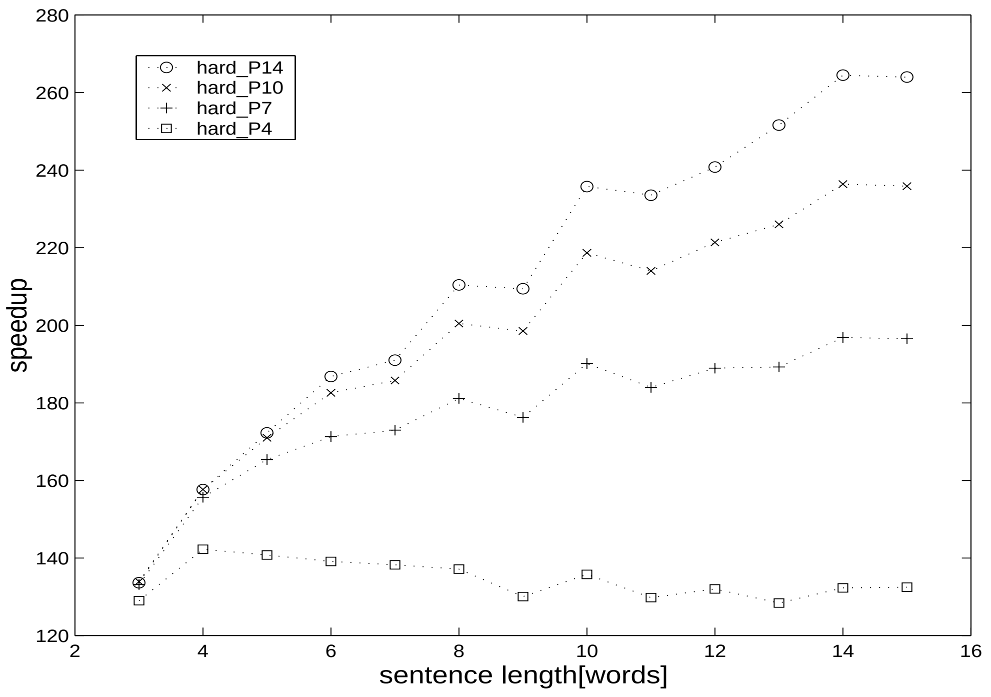
Table 6.5: Virtex XCV2000 FPGA resource utilization per enhanced-CYK design unit in terms of Flip-Flops/Latches (DFFs/Latches), function generators (FGs) and configurable logic blocks (CLBs) for the enhanced-CYK design.

When using the SUN machine that was also used to benchmark the previous CYK designs the average speedup factor of the `hard_P14` system against the software is $E(S) = 216.34$. The figure 6.12(a) shows the hardware speedup, for the `hard_P4`, `hard_P7`, `hard_P10` and `hard_P14` in comparison with the software run on the SUN as a function of the sentence length. When using the PC machine, the average speedup factor of the `hard_P14` system against the software is in this case $E(S) = 68.75$. The figure 6.12(b) shows the hardware speedup, for the `hard_P4`, `hard_P7`, `hard_P10` and `hard_P14` in comparison with the same software run on the PC, as a function of the sentence length.

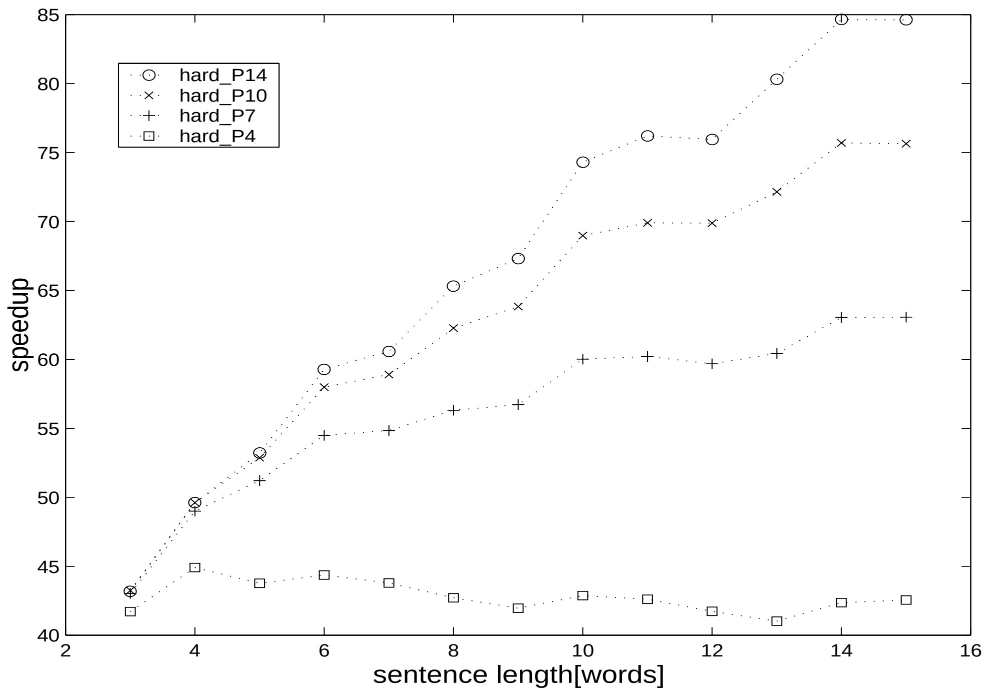
Note that the speedup factor is increasing with the sentence length (see figure 6.12(b)), up to 85 for 15 words long sentences. This behaviour is very important when dealing with real-life size sentences. There are several other factors that can increase the system performance among which we mention (1) the clock frequency can be increase to 60 MHz instead of 50 MHz, resulting in a 20% speedup improvement and (2) the number of processors can be increased to 16 which may lead to a further increase in performance and (3) using a more performant FPGA technology such as the XCV2000efg1156-8 instead of an XCV2000efg1156-6 which will allow the system to run at an even higher clock frequency – about 20% higher. The design can be further improved as well .

6.6 Design analysis

In this section some important aspects of the enhanced-CYK design are analysed and discussed. First, in order to get a general idea about the design, we look at the processor activity and



(a)



(b)

Figure 6.12: Hardware speedup for the `hard_P14`, `hard_P10`, `hard_P7` and `hard_P4` enhanced-CYK systems against the software running on (a) a SUN machine and (b) a PC machine, as a function of sentence length. For each sentence length more than 100 sentences were parsed.

measure the average processor utilization for some arbitrary sentences extracted from the SUSANNE corpus. As the processor activity mainly consists of two tasks (1) data-processing and (2) waiting for flushing the processing results, we investigate – as in the case of the DAP design – at the fraction of time spent by the processors in performing each of the two tasks. The effect of an increased number of processors requesting the flushing of parsing results – as the sentence length increases – on the overall design performance is next investigated. The influence of the tile size and POOL size on the design performance are also investigated. Finally, the system throughput – for sending the compact parse forest to the host machine – is investigated in order to select the bus that will interface the FPGA-board and the host machine.

6.6.1 Average processor utilization

In order to get a general idea about the processor activity during the parsing process we illustrate the processor activity for three sentences (given in table 6.6) of length 4, 10 and 15 words in figure 6.13(a), (b) and respectively (c). In this figures the processor activity is depicted for a 14-processors system with a 16 entries POOL, in two instances (1) for a tile size 6×12 in the left column and (2) for a tile size 128×128 in the right column. The 128×128 tile size makes the enhanced-CYK design to work like the DAP design in which a cell-combination is computed by a single processor. The speedup factor for the DAP design and enhanced-CYK design using a 6×12 tile size and respectively a 128×128 tile size when parsing these sentences are also given in table 6.6. The first important thing to note from the results in this table is the large

len	sentence	DAP speedup	enhanced-CYK speedup	
			6×12	128×128
4	*“One wing stood open”*	22.41	282.75	215.51
10	*“In fact our whole defensive unit did a good job”*	34.18	328.25	280.43
15	*“In societies like ours , however , its place is less clear and more complex”*	36.31	247.67	222.03

Table 6.6: Three sentences parsed with the enhanced-CYK design for which the processor activity is given in figure 6.13. Speedup factor is given against the software implementation.

difference between the speedup factors of the DAP and respectively enhanced-CYK (using an 128×128 tile size) designs. The difference comes mostly from an improved hardware design (i.e. processor, DISPATCHER, WRITER and LOOKUP units). The effect of tilling although significant is not the main factor for the observed speedup improvement. The second important thing to note is the difference between the gray area of the DAP (see figure 5.15 on page 86) – when parsing the same sentences – and of the enhanced-CYK design using an 128×128 tile size (see figure 6.13, right column). A much smaller gray area, which means a faster handling of the processing results is observed for the enhanced-CYK design (128×128 tile size) even if the parsing time diminished drastically with the enhanced-CYK design. For comparing the average processor utilization of the enhanced-CYK design using a 6×12 tile size with the enhanced-CYK design using a 128×128 tile size, we use some sentences that were arbitrarily extracted from the SUSANNE corpus. These sentences are tabulated in table 6.7. The average processor utilisation is computed in the same way as it was computed for the LAP design (see section 4.1, page 52). The results in this table show that the average processor utilization increases when

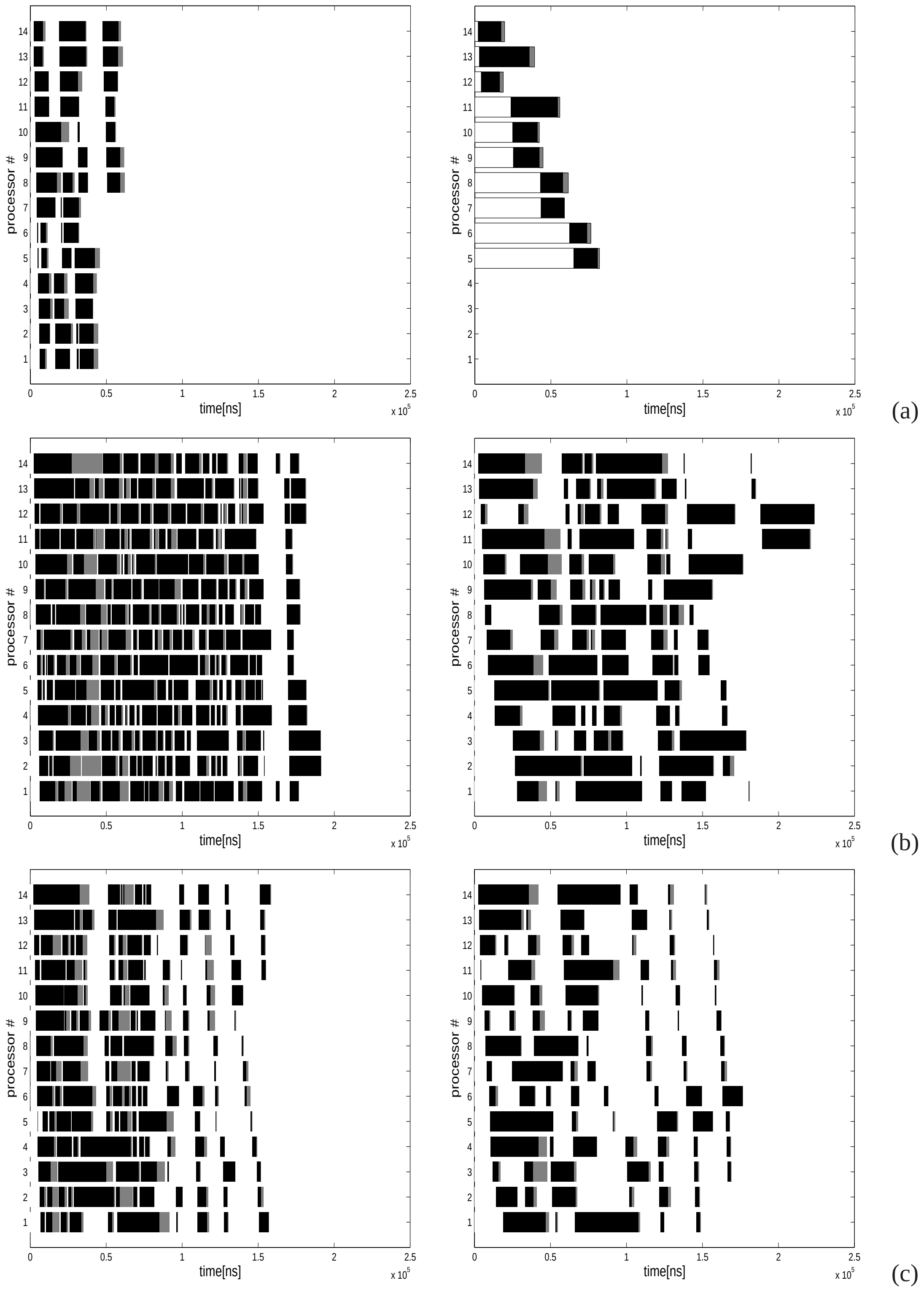


Figure 6.13: Enhanced-CYK processor activity in BLACK+GRAY when parsing a sentence of length (a) 4 words, (b) 10 words and (c) 15 words. GRAY: represents the time for flushing processing results. BLACK: represents the time spend for processing data.

len	sentence	enhanced-CYK U[%]	
		6x12	128x128
3	*“One pass only”*	15.34	8.88
4	*“There was no moon”*	9.94	7.18
5	*“She too began to weep”*	23.21	14.68
6	*“She must not think about time”*	16.36	13.02
7	*“The form and the chaos remain separate”*	26.09	19.02
8	*“The games were over , this was life”*	17.95	13.68
9	*“Like Napoleon , he was the worst of losers”*	19.53	14.64
10	*“In fact our whole defensive unit did a good job”*	62.53	42.90
11	*“I told him who I was and he was quite cold”*	14.31	12.72
12	*“Nerves tight as a bowstring , he paused to gather his wits”*	50.55	36.60
13	*“I told him no , that I had had a very happy childhood”*	14.76	13.40
14	*“As he had longed to be , he became the echo of a saga”*	40.06	30.03
15	*“It is all around us and our only chance now is to let it in”*	27.43	21.45

Table 6.7: The average processor utilization for the enhanced-CYK design when using a 6×12 tile size and respectively a 128×128 tile size, for a set of sentences extracted from the SUSANNE corpus with lengths between 3 and 15 words.

using a 6×12 tile size in comparison with the case when a 128×128 tile size is used. However the increase in the average processor utilization is not significant in some cases.

6.6.2 Expected performance depreciation

This section investigates the influence of an increasing number of working processors on the expected system performance. For the previous designs the expected performance depreciation phenomena was mainly caused by interprocessor collisions when accessing the chart memory. A number of improvements implemented within the current design (e.g. elimination of guard-vectors, a wider chart memory databus, a processor that does not access the chart memory and others) aim to limit the interprocessor collisions. In fact interprocessor collisions only occur within the enhanced-CYK design when flushing the processing results. Therefore we expect that the expected performance depreciation phenomena is less important within the enhanced-CYK design.

The fact that there are less interprocessor collisions in the enhanced-CYK design than in the previous DAP design can be observed by comparing the gray area in figures 6.13(a)-(c) (right column, 128×128 tile size) and respectively the gray area in figures 5.15, page 86. However, in order to better illustrate this phenomena we will use an own built grammar – the same that we used in example 4.1 – that has the particularity that any cell-combination in the chart requires the same amount of processing. The following example uses this grammar in an experiment that illustrates the expected performance depreciation phenomena.

Example 6.3 We use the grammar defined in example 4.1 on page 54 to parse the sentences: "a a", "a a a", "a a a a", . . . , up to a similar sentence of length 15.

This grammar has the property of generating the same sets $N1$ of non-terminals $\{X_1, X_2, \ldots, X_8\}$ and the same set $N2$ of partial right hand sides $\{X_1\bullet, X_2\bullet, \ldots, X_8\bullet\}$ for all cell-combination performed. This means that at the end of the parsing, each cell in the chart will contain these two sets and that during the parsing the processors work on the same data each time two cells are combined. In conclusion, if the time T spent by a processor for performing a cell-combination (i.e. the parsing time for the sentence "a a") is known we can compute the expected parsing time for a sentence of length L. We will also assume that a 128x128 tile size is used in order to simplify the computation of the expected parsing time for a sentence of arbitrary length. The method used to compute the expected parsing time is not analytic and we used a program for this purpose. The program computes the minimum number of steps S required for filling the chart when parsing a sentence of length L, using a given number of processors (e.g. 14 in our case). With S and T we compute the expected parsing time as $S * T$.

For each of these sentences the figure 6.14 illustrates the (computed) expected parsing time vs. the real parsing time. As we can see from this figure, the difference between the expected

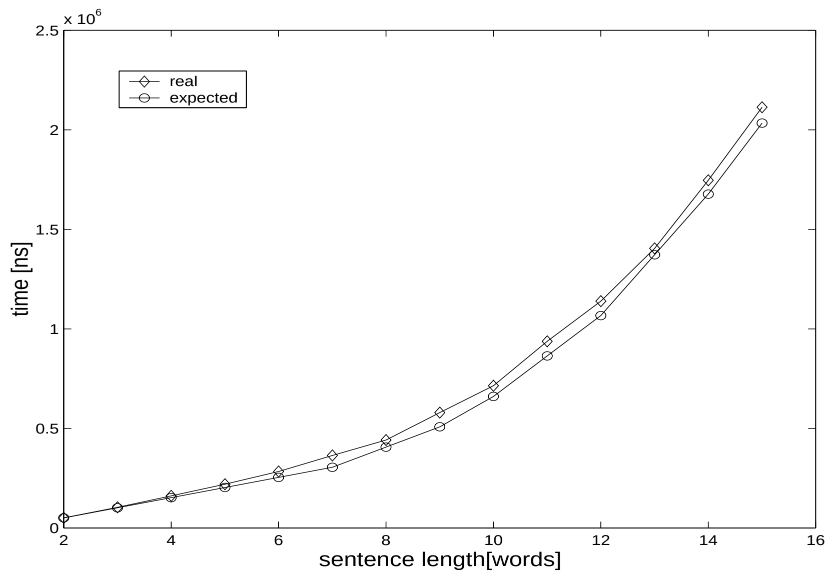


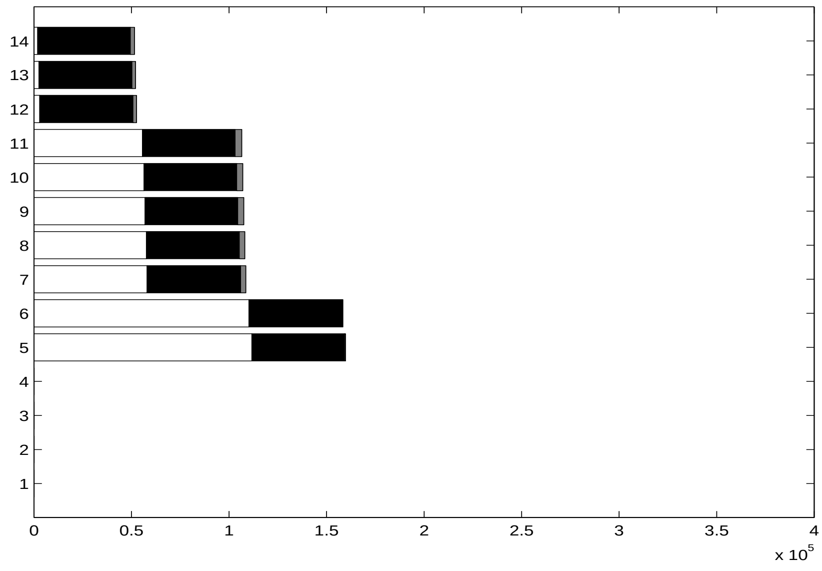
Figure 6.14: Enhanced-CYK design real vs. expected parsing time when parsing the sentences "a a", "a a a", "a a a a", . . . up to a similar sentence of length 15 with the enhanced-CYK design using a 128×128 tile size.

and the real enhanced-CYK design performance (i.e. parsing time) is not significant and does not increase with the sentence length. Therefore we can conclude that for the enhanced-CYK design an increasing number of processors does not depreciate the system performance as it was the case for the LAP and DAP designs.

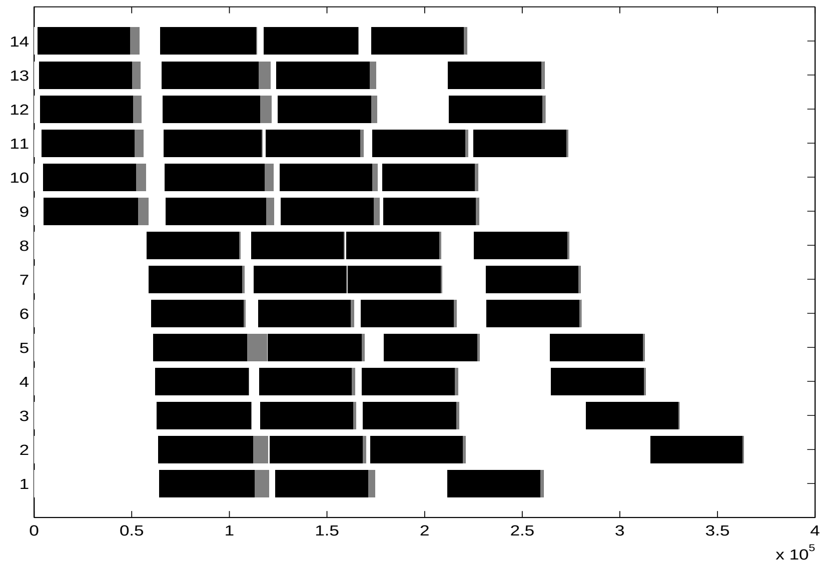
Figure 6.15 illustrates the enhanced-CYK processor activity when parsing a sentence of length 4 and respectively 7 "a"s. Note that the gray area does not change when parsing the sentence of length 4 and when parsing the sentence of length 7. $\diamondsuit$

6.6.3 Required bandwidth for transferring the compact parse forest

In this section we study (1) whether a PCI or SCSI interface is fast enough to transfer the compact parse forest to the host computer and (2) look at the average amount of data to be transfered during the parsing. The first point is important for choosing the type of interface that will be used to build the FPGA-board and the second for measuring the size of the FIFO required for buffering the data between the enhanced-CYK design output and the host machine.



(a)



(b)

Figure 6.15: Enhanced-CYK processor activity for a sentence of length (a) 4 "a"s and respectively (b) 7 "a"s. Note the small gray area in both cases.

All these measurements will be made in the particular case of the SUSANNE grammar without unitary rules.

The WRITER sends the parsing results fetched from the processors – without any change – to the EXTRACTOR . The later packs the parsing results according to the databus width of the interface used between the FPGA-board and the host machine in order to best exploit the available bandwidth. For instance if a 32-bit PCI interface is used the parsing results are packed on 32-bit words and if a 16-bit SCSI interface is used the parsing results are packed on 16-bit words. The rate, i.e. throughput, at which the FPGA-board is shipping the parsing results is given by the amount of data to be shipped during the parsing time. The bus connecting the FPGA-board to the host machine is required to have a bandwidth bigger than the FPGA-board throughput in order not to be a system bottleneck. While the FPGA-board throughput depends on the parsed sentence and the grammar used for parsing we investigate the throughput of the enhanced-CYK design in the particular real-life case of the SUSANNE grammar without unitary rules. For this purpose we will use (i.e. parse) the same $2,021$ sentences from the SUSANNE corpus used to benchmark the enhanced-CYK design. Two buses are investigated, a 32-bit 33 MHz PCI bus and a 16-bit Ultra 2 Wide SCSI bus.

In order to get a general idea about the information transfered from the FPGA-board to the host computer, we use the following example illustrating the output produced by the enhanced-CYK design when the sentence "It was a box" is parsed with the SUSANNE grammar without unitary rules.

Example 6.4 The output produced when parsing the sentence "It was a box" – with the SUSANNE grammar without unitary rules – is given in table 6.8. Each line (i.e. parser result item) in this table has the form:

$$S_{1r}\ S_{1c}\ (\alpha\bullet) \quad S_{2r}\ S_{2c}\ X \quad D_r\ D_c\ (\alpha X\bullet)$$
or
$$S_{1r}\ S_{1c}\ (\alpha\bullet) \quad S_{2r}\ S_{2c}\ X \quad D_r\ D_c\ Y$$

where $S1$ is the first source cell (S_{1r} is the row and S_{1c} is the column of this source cell), $S2$ is the second source cell (S_{2r} is the row and S_{2c} is the column of this source cell) and D is the destination cell (D_r is the row and D_c is the column of the destination cell). Parentheses are used to mark $N2$-items. The parsing result represented in this format allows the reconstruction of the parsing trees on the host machine. Concretely, the chart can be reconstructed on-line during the parsing and used later to extract the parsing trees. $\diamondsuit$

The FPGA-board throughput illustrated in figure 6.16(a) (left) corresponds to the case when all the parse result items – as for the example given in table 6.8 – are sent to the host. This figure illustrates for each sentence in the bench the required throughput. The sentences are ordered in increasing order of their length. The figure 6.16(a) (right) illustrates for the same sentences the size (in KBytes) of the parse result. Note, that even if the parse result size is relatively low (i.e. several KBytes) the required throughput is high. This is due to the fast parsing times. The figure 6.16(a) (left) shows that neither the SCSI interface nor the PCI interface is fast enough to transfer efficiently the parsing results. Therefore, it is not possible to send all the parse result items to the host. A solution is to eliminate the redundant parts from the parse result items. The squared items illustrated in table 6.8 correspond to such a solution. The FPGA-board throughput illustrated in figure 6.16(b) (left) corresponds to this case. The figure 6.16(b) (right) illustrates in this case the size (in KBytes) of the parse result. In this case we can see that the PCI bus is fast enough in most of the cases.

An even more aggressive solution would be to send only the minimum information that allows to retrieve the parsing forest. The oval-squared items illustrated in table 6.8 correspond to such a solution. In this case the parse result only contains the pairs of $N2$-items (in the source cell $S1$) and $N1$-items (in the source cell $S2$) that produce output results (in their associated destination cell D), but does not include these output results. The output results can be nevertheless reconstructed on the host side by performing supplementary grammar lookups. The required grammar lookups may be an overhead in this case. The FPGA-board throughput illustrated in figure 6.16(c) (left) corresponds to this case. The figure 6.16(c) (right) illustrates in this case the size (in KBytes) of the parse result. In this case we can see that the SCSI is fast enough in most of the cases and that the PCI bus is fast enough in all the cases.

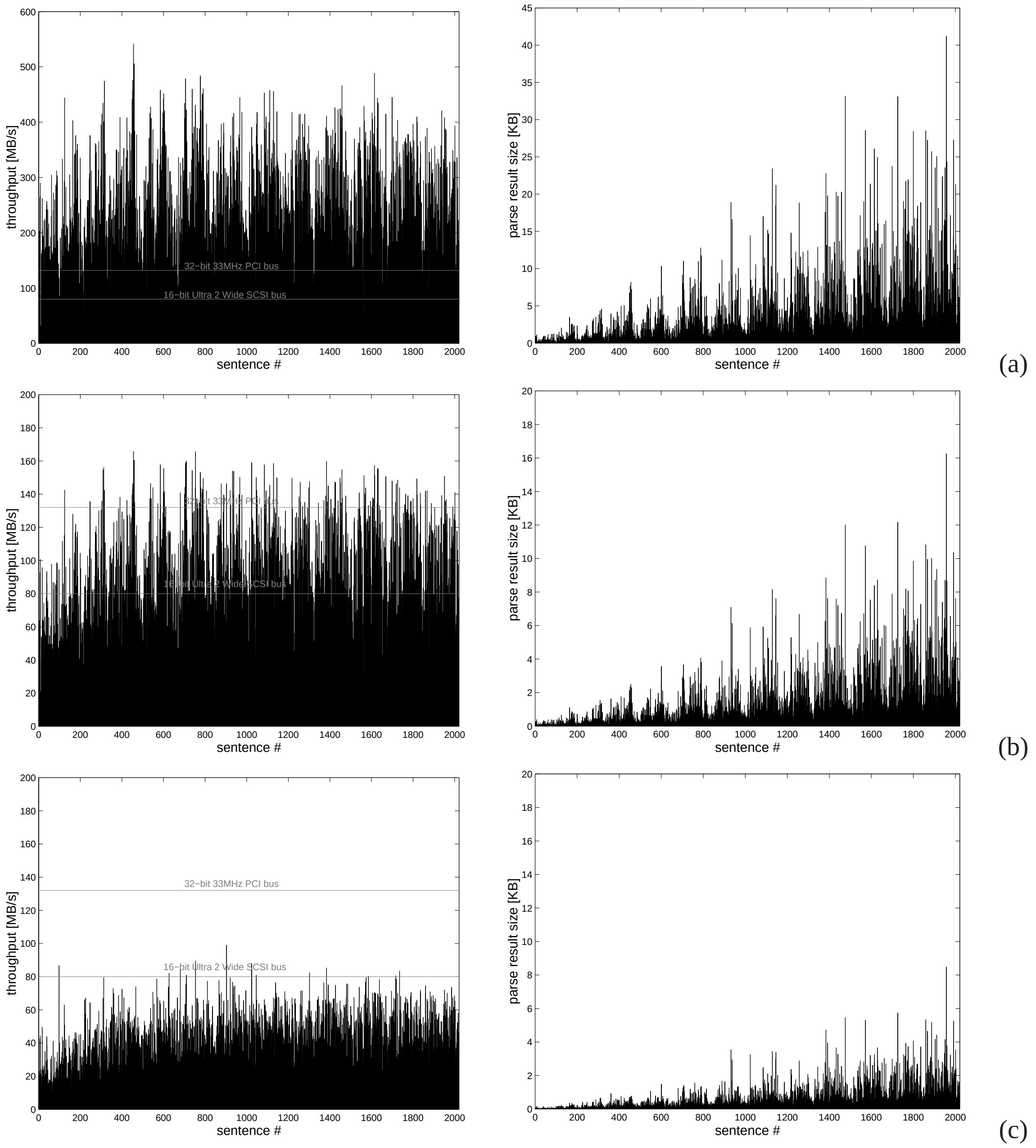


Figure 6.16: Parse result output rate (left column) and output size (right column) for three possible ways of formating the parsing results.

item #	parser result item
1-	$\boxed{\boxed{\text{1 1 (:1226) 1 2 :1509}}\text{ 2 1 (:1226 :1509)}}$
2-	$\boxed{\boxed{\text{1 1 (:1223) 1 2 :1509}}\text{ 2 1 (:1223 :1509)}}$
3-	$\boxed{\boxed{\text{1 1 (:1222) 1 2 :1509}}\text{ 2 1 (:1222 :1509)}}$
4-	$\boxed{\boxed{\text{1 3 (:9) 1 4 :167}}\text{ 2 3 :1238}}$
5-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1341}}$
6-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1342}}$
7-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1344}}$
8-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1347}}$
9-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1351}}$
10-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1355}}$
11-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1363}}$
12-	1 3 (:9) 1 4 :167 2 3 $\boxed{\text{:1373}}$
13-	$\boxed{\boxed{\text{2 1 (:1226 :1509) 2 3 :1347}}\text{ 4 1 :1170}}$
14-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1238}}$
15-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1363}}$
16-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1368}}$
17-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1369}}$
18-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1371}}$
19-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1372}}$
20-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1373}}$
21-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1375}}$
22-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1450}}$
23-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1451}}$
24-	2 1 (:1226 :1509) 2 3 :1347 4 1 $\boxed{\text{:1517}}$

Table 6.8: Output for the enhanced-CYK design when parsing the sentence "It was a box".

There are faster interfaces than the 32-bit, 33 MHz PCI bus (132 MB/s) or the 16-bit Ultra 2 Wide SCSI (80 MB/s) considered in this section. For instance the 64-bit 66 MHz PCI bus (528 MB/s) or the Ultra 160 SCSI (160 MB/s), Ultra 320 SCSI (320 MB/s) offer larger bandwidths but by the time this thesis is written these technologies are not in common use and we did not consider them as options for the FPGA-board implementation.

In conclusion we select the 32-bit 33MHz PCI bus for interfacing the FPGA-board to the host system and use the second solution proposed above for sending the parsing results to the host system.

6.6.4 Tile size influence on the design performance

In this section we study the influence of the tile size on the overall system performance. The tiling method implemented within the enhanced-CYK design aims to balance the processor load in order to achieve a better average processor utilization and therefore better performance. In order to study the influence of the tile size on the design performance a number of benchmarks have been performed on $2,021$ sentences of length 3 to 15 from the SUSANNE corpus – the same sentences used for all the benchmarks in this chapter. The benchmarks were performed by simulating the VHDL model of the enhanced-CYK design (configured for the SUSANNE grammar without unitary rules) with 14-processors and a POOL with 16 entries for several tile sizes. A tile size of 2×12, 6×12, 12×12 and 128×128 were used for these simulations. The later tile size, corresponds to the case when no tilling is actually performed.

The software used for comparison is an implementation of the enhanced-CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The hardware performance (i.e. run-times) of the 14-processors enhanced-CYK design and a tile size 2×12 (denoted as `hard_tile2x12`), a tile size 6×12 (denoted as `hard_tile6x12`), a tile size 12×12 (denoted as `hard_tile12x12`) and a tile size 128×128 (denoted as `hard_tile128x128`) was compared against the software run-times. The software was run on a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* `C` library function and not by profiling the code.

The results are illustrated in figure 6.17. It can be seen from this figure that for short sentences (up to 8 words) the system using a 2×12 tile size gives best performance. However, as the sentence length increases the performance of the system using the 2×12 tile size decays and becomes the worse for sentences above 12 words.

For short sentences, the explanation is that without tiling not all the processors are used during the parsing while with tiling the design will use more processors. Thus for short sentences (e.g. 3,4) the smallest the tile size and more processors are used the better.

The explanations becomes more complex as the sentence length increases. Basically, we can distinguish two different regions in the chart in which the tiling has a different effect. The first region is the bottom of the chart where all the processors will be busy and the tilling only introduces an unnecessary overhead by splitting the cell-combinations. The second region is towards the top of the chart where the number of interpretations (i.e. size of the sets $N1$ and $N2$) becomes smaller and smaller. The direct consequence of this fact is that the processors spend a shorter time for processing the cells towards the top of the chart. In this context the overhead introduced for sending the tiles prevails over the gain of processing the tiles in parallel on several processors. Concretely, there are several tiles for a cell-combinations but only few or none of the processors processing these tiles produce results. In this case it is better to send the entire cell-combination to a processor instead of loosing time for tiling it.

There are (at least) two solutions for this problem. The first solution should aim an adaptive sizing of the tile during the parsing according to the sentence length and the size of the cells that are combined. A second solution would be to improve the hardware used for dispatching the tiles in order to hide the dispatching overhead.

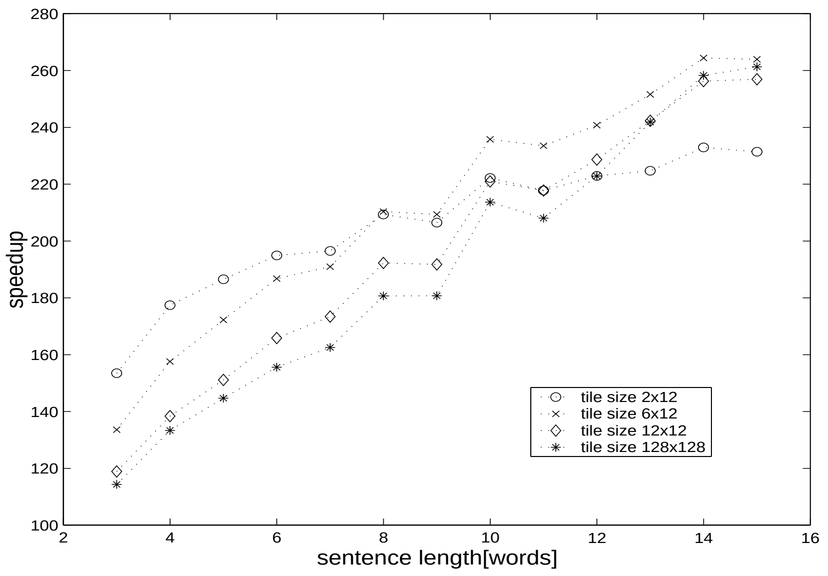


Figure 6.17: The speedup for a 14-processors enhanced-CYK system for several tile sizes as a function of sentence length. For each sentence length more than 100 sentences were parsed.

6.6.5 POOL influence on the design performance

In both the DAP and the enhanced-CYK design, the POOL unit is key to the system's ability to dynamically allocate the cell-combinations to the processors. Without the POOL unit the system will hang on the first cell-combination that does not satisfy a data-dependency constraint and keep all the remaining cell-combinations – even if they satisfy their data-dependency constraints – unissued. On the other hand, a POOL unit allows the system to search over a larger number of cell-combination that can be potentially issued. One can interpret the POOL unit functionality as a window sliding over the unprocessed cells of the chart that allows the design to issue the cell-combinations that satisfy the data-dependency constraints while keeping under observation those that do not satisfy the data-dependency constraints. In this context, a larger POOL size (i.e. a larger sliding window) is supposed to be better as it allows the design to search over a larger number of cell-combination that can be potentially issued.

In order to investigate the influence of the POOL size on the enhanced-CYK design performance a number of benchmarks have been performed on $2,021$ sentences of length 3 to 15 from the SUSANNE corpus – the same sentences used for all the benchmarks in this chapter. The benchmarks were performed by simulating the VHDL model of the enhanced-CYK design (configured for the SUSANNE grammar without unitary rules) with 14-processors and a 6x12 tile size for several POOL sizes. A POOL size of 8, 4, 2 and respectively 1 was used for the purpose of these simulations.

The software used for comparison is an implementation of the enhanced-CYK algorithm and is part of the SlpToolKit that was developed in our laboratory. The hardware performance (i.e. run-times) of the 14-processor enhanced-CYK design with POOL size 1 (denoted as `hard_pool1`), POOL size 2 (denoted as `hard_pool2`), POOL size 4 (denoted as `hard_pool4`), POOL size 8 (denoted as `hard_pool8`) was compared against the software run-times. The software was run on a SUN (Ultra-Sparc 60) with 512 MBytes memory, 770 MBytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the chart was not taken into account for the computation of the run-times. For accuracy, the timing was done with the *times()* C library function and not by profiling the code. The results are illustrated in figure 6.18.

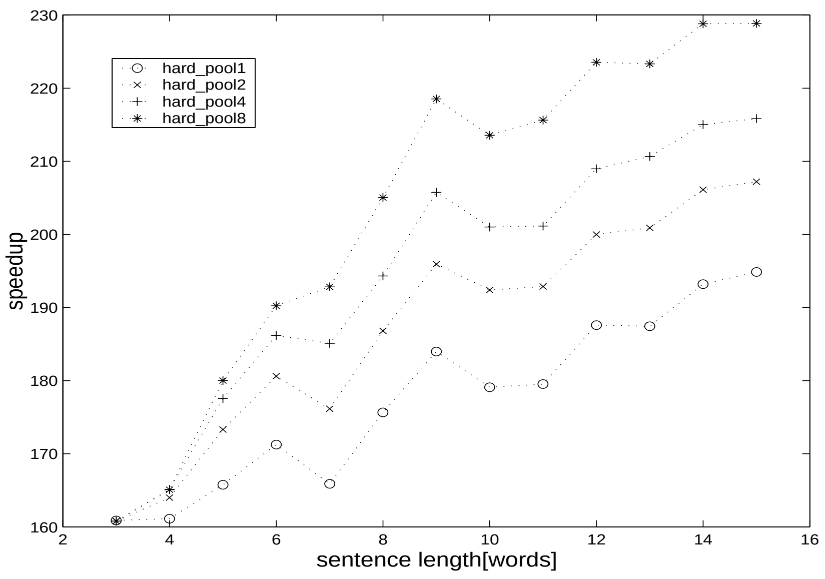


Figure 6.18: The speedup for the 14-processors enhanced-CYK system for several POOL sizes as a function of sentence length. For each sentence length more than 100 sentences were parsed.

Contrary to the expectations, the size of the POOL has a significant influence on the system performance. It was expected that the system performance will not increase when passing – for instance – from a POOL size 4 to 8. It was expected that the role of the POOL unit is less important due to the design's ability to balance the processor load by performing tiling. A balanced processor load, further means less chances for a cell-combination not to satisfy a data-dependency constraint – that was usually the case when large cell-combinations were processed – and therefore to require to be stored in the POOL.

The problem comes from the fact that the tiling does not equilibrate as much as expected the processor load .

6.7 Conclusions

This chapter proposes a design architecture implementing the enhanced-CYK algorithm adapted for word lattice parsing. The presented enhanced-CYK design architecture can deal with real-life large-size almost unrestricted CFGs. It can parse sentences of any length if enough memory is available for storing the chart. For instance, in the particular case of the SUSANNE grammar (without unitary rules) a chart memory size of 256 KBytes allows to parse sentences with up to 32 words and a chart memory size of 16 MBytes allows to parse sentences with up to 256 words. The speedup obtained when compared against a software implementation of the enhanced-CYK algorithm run on a Intel PENTIUM IV processor at 1.5 GHz is $85\times$ for sentences of real-life size (15 words).

For all the tests, performance measurements and design analysis performed in the current chapter the databus with the chart memory is 64-bit and the databus with each grammar memory is 16-bit. However, the databus with the grammar memories or the chart memory can be easily changed as needed – according to the targeted system (e.g. FPGA-board). For instance the databus with the grammar memories can be changed by using a different VHDL module description for the (plug-in) grammar memory interface module, while the rest of the processor stays unchanged. With such an approach we can use a 8-bit, 16-bit or 32-bit databus with the grammar memories or even a 1-bit, 2-bit or 4-bit databus. The same remark applies to the databus with the chart memory. Even more, a mixed databus environment is also allowed. For instance some grammar memories may use a 16-bit databus and the rest a 8-bit databus. Such an approach makes feasible the mapping of the proposed enhanced-CYK design on any system containing the FPGA resources available in a Xilinx Virtex XCV2000 FPGA and enough external memory resources.

Precisely, the proposed design architecture is the third step and the final of our design methodology during which:

- we proposed a design that can deal with almost unrestricted general CFGs;
- simplify the (chart) initialization procedure;
- integrate a method called tiling that improves the average processor utilization and improves the performance;
- integrate an on-line parse extraction module;
- integrate a module that monitors the normal system operation during runtime;

The enhanced-CYK design can deal with almost unrestricted general CFGs. Precisely, it can deal with "non partially lexicalized" CFGs (nplCFG) that do not contain unitary rules. The restriction to the nplCFG subclass of the CFGs, was introduced because it corresponds to the kind of grammars we are actually dealing with in practice, and also because it restricts the processing of "lexicalized rules" to the initialization step. On the other hand, in order to improve the hardware performance and to reduce the hardware complexity we also restricted the grammars not to contain unitary rules.

The simplification of the initialization step aims to insure that this step represents a insignificant amount in the overall parsing time.

The tiling method implemented within the enhanced-CYK design aims at improving the average processor utilization and therefore the design performance. With the tiling method the processors can be assigned to process chunks of a cell-combination (i.e. the tiles) unlike the processors in the previous designs that were processing entire cell-combinations. One can see the tiling method as the design's ability to dynamically cluster the available processors around demanding (i.e. large) cell-combinations. That is, a way of concentrating the processing resources when and where needed.

The integration of an on-line parse extraction module finally renders feasible the extraction of the compact parse forest. The information sent by the parse extraction module allows a host machine to rebuild (on-line) the chart table that can be further used for extracting parsing trees.

The monitoring module creates a reliable environment that allows the design to continue, i.e. restart, with a new parsing whenever a fatal error occurs. It also allows the host machine to check which unit in the system is the source of the error, ability which is especially useful when debugging or when integrating new functionalities in the enhanced-CYK design.

From the performed experiments and performance measurements of the enhanced-CYK design we make the following remarks:

- increasing the number of processors in the system can only increase the system performance. The overhead introduced for managing more processors then actually needed – especially when parsing short sentences – does not result in a performance depreciation as in the case of the DAP design;
- when the number of processors in the system increases no performance depreciation is observed as in the case of the DAP design. This is mainly the result of eliminating the guard-vectors and using the DISPATCHER and the WRITER as interfaces for accessing the chart memory;
- the average processor utilization is (still) relatively low. For instance in the particular case of the SUSANNE grammar the observed average processor utilization for some arbitrary sentences extracted from the SUSANNE corpus is between 7 - 45%, when tiling is not used. In the same particular case, when using tiling a better average processor utilization is observed – between 10-62%;
- the tiling mechanism does not balance as much as expected the processor load. The explanation is that for longer sentences, the number of interpretations (i.e. size of the sets $N1$ and $N2$) for the cells towards the top of the chart becomes smaller and smaller. The direct consequence of this fact is that the processors spend less time for processing the cells towards the top of the chart. In this context the overhead introduced for sending the tiles prevails over the gain of processing the tiles in parallel on several processors and in this case it is better to send the entire cell-combination to a processor instead of loosing time for tiling it. This also explains the relatively low average processor utilization mentioned above.

  Two solutions to this problem are: (1) to use an adaptive tile size during the parsing. It should start with a small tile size and increase it as the filled row is closer to the top of the chart and (2) improve the hardware used for dispatching the tiles in order to hide the dispatching overhead;
- the throughput of the enhanced-CYK design is relatively high and if the parsing time shrinks even more it will soon become to large for a standard 32-bit 33 MHz PCI bus. A solution in this case would be to use a faster bus;
- less than 65% of the resources available in a Xilinx Virtex XCV2000 FPGA were used for a 14-processors enhanced-CYK design. Resources are still available for further improvements ;

Chapter 7

An accelerator FPGA-board for running the enhanced-CYK algorithm

This chapter presents an accelerator FPGA-board that implements the enhanced-CYK design presented in chapter 6. The proposed FPGA-board contains the necessary hardware resources for implementing a 16-processors enhanced-CYK design and features a PCI interface that allows it to be integrated and work as an accelerator within a host machine (e.g. desktop PC).

The chapter starts with a general and functional description of the proposed FPGA-board. Next it presents in greater detail the purpose of the components on the FPGA-board.

7.1 General description

As the analysis of the enhanced-CYK design (see section 6.6.3) shows, a common 32-bit 33 MHz PCI bus has a bandwidth (132 MBytes/s) which is large enough in most of the cases[1] for transferring the compact parse forest to a host machine.

However, building a PCI interface is not an easy task by itself and we used for this purpose an of-the-shelf generic PCI interface board which is the development board provided by PLX Technology Inc. in the Reference Design Kit (RDK). This PCI interface board is build around the PLX's IOP 480 I/O processor which integrates both a PCI-bus controller and a 32-bit, 66MHz PowerPC RISC core. The PCI interface board provides an expansion connector (PLX Option Module, POM) that can be used to plug-in custom build modules. We used the POM expansion connector to plug-in our FPGA-board expansion module implementing the enhanced-CYK design.

Figure 7.1 illustrates the FPGA-board which is built from the PCI interface board and the FPGA-board expansion module. Hereafter, we will use the terms:

- PCI interface board: to denote the PLX PCI development board;
- FPGA expansion board: to denote the FPGA-board implementing the hardware design of the enhanced-CYK algorithm;
- FPGA-board to denote the ensemble of the two above;

[1]This analysis was made in the particular case of the SUSANNE grammar.

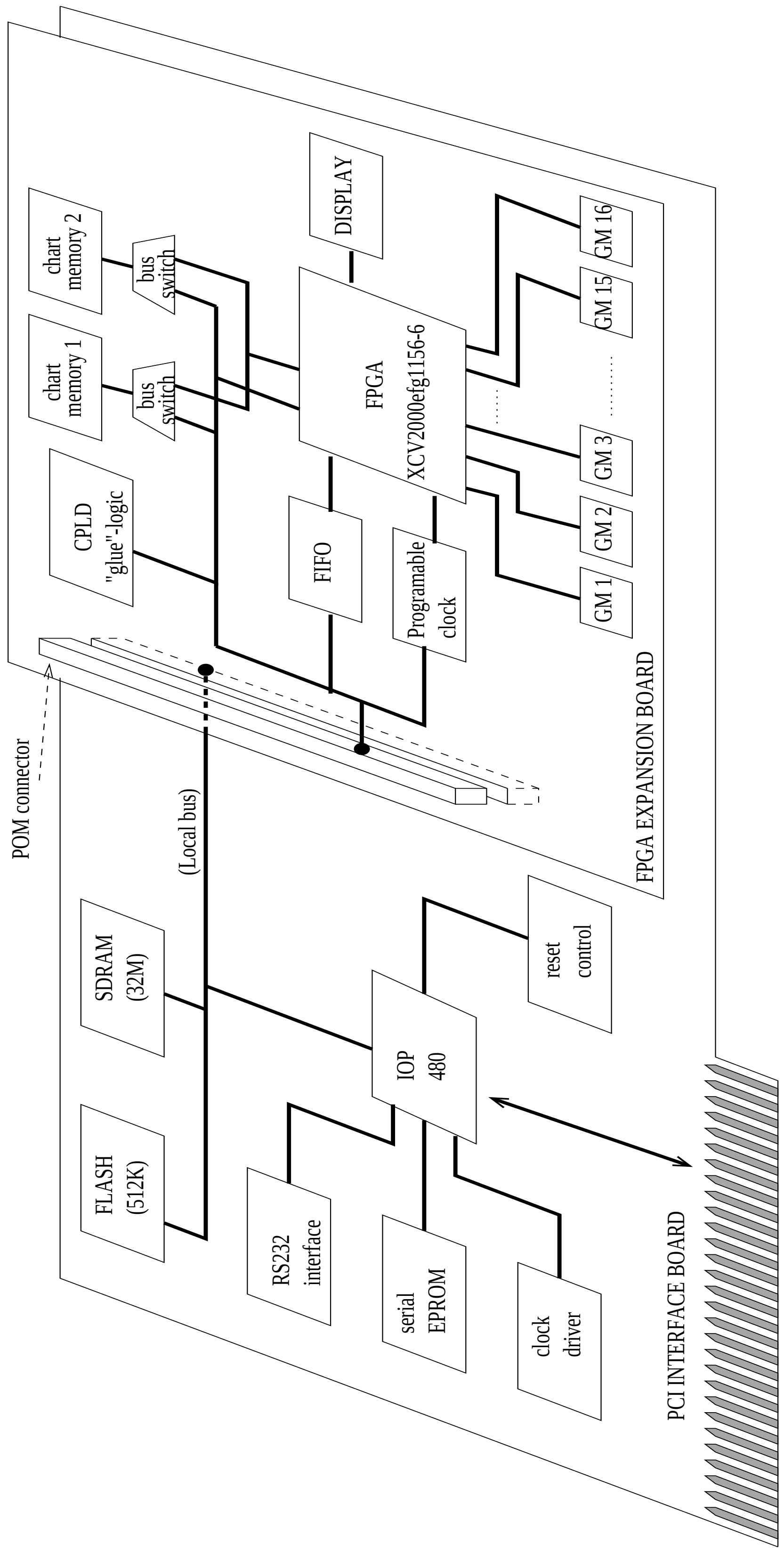


Figure 7.1: An FPGA-board implementing a 16-processors enhanced-CYK design.

The resources available on the PCI interface board are: 512 KBytes flash memory, 32 MBytes SDRAM memory and a RS-232 serial interface. The flash memory stores the power on initialization routines for the PCI interface board and several other initialization routines for the FPGA expansion board. The SDRAM stores the lexicon which is used to initialize the chart memory and some other (large) data-structures.

The FPGA expansion board uses a Xilinx Virtex-E XCV2000efg1156-6 FPGA for implementing a 16-processors enhanced-CYK design (see chapter 6) and also contains the required memory resources for (i.e. chart and grammar memories) this purpose. For the implementation of the 16-processors enhanced-CYK design a 64-bit databus with the chart memory and a 16-bit databus with the grammar memories are used.

The CPLD implements the "glue-logic" used to interface the FPGA expansion board and the PCI interface board and the FIFO memory is used for buffering the compact parse forest in DMA transfers over the PCI-bus. Note, that the FPGA expansion board uses two chart memories in order to overlap the chart memory initialization and the parsing in order to make the initialization transparent.

7.2 Functional description

This section describes the steps required for initializing the FPGA-board and how the FPGA-board can be used for parsing sentences or word lattices.

The FPGA-board is a slave device under the full control of the host system. In other words, any parsing request or operation initiates on the host system. In order to execute the commands issued by the host system the FPGA-board relies on a software component running on the IOP 480 processor. This software – stored in the flash memory on the PCI interface board – is the interface between the host machine and the FPGA-board and contains routines for executing commands such as: initializing the FPGA on the FPGA expansion board, initializing the grammar memories, initializing the chart memories, initializing the lexicon, starting the parsing process and others.

Three initialization steps are required for initializing the FPGA-board before any parsing can start. These steps are required for configuring the FPGA on the FPGA expansion board, for initializing the lexicon and respectively the grammar memories.

The FPGA is configured by means of a configuration file generated by a place&route tool (e.g. Design Manager, Xilinx Alliance Series 2.1i) that is provided by the host system to the FPGA-board. The FPGA on the FPGA expansion board is configured under the control of the IOP 480 processor that generates the required signals during the configuration process.

The initialization of the lexicon consists of storing the lexicon in the SDRAM memory available on the PCI interface board in a data-structure that allows fast words lookup (i.e. prefix trees). The lexicon is provided by the host system in a format compatible with the SlpToolKit software (see appendix A.1) and the lexicon data-structure is built locally by the IOP 480 processor. The time required for building the lexicon data-structure is not critical as it is done only once. The lexicon is specific to a grammar and does not change during successive parsings. A reinitialization of the lexicon data-structure is only required if the grammar changes.

The initialization of the grammar memories consists of storing a copy of the binary image of the grammar data-structure in each grammar memory available on the FPGA expansion board. The binary image of the grammar data-structure is provided by the host system and the IOP 480 processor only has to copy the grammar data-structure in the grammar memories. Note that the only access path for the IOP 480 processor to the grammar memories passes through the FPGA circuit and therefore the FPGA circuit has to be configured before the grammar memories can be initialized. Moreover the circuit configured in the FPGA should support the IOP 480 processor during the initialization of the grammar memories. For this purpose, the circuit inside the FPGA has two working states: (1) used for initializing the grammar memories and (2) implementing the enhanced-CYK algorithm. The working state is switched under the control of the IOP 480 processor whenever required.

Once the above three initializations have been performed the FPGA-board is ready to parse sentences and/or word lattices. The procedure given bellow describes how a sentence is initialized and parsed with the FPGA-board. First, the sentence is sent in text form by the host system to the FPGA-board where the IOP 480 processor splits the sentence in words and lookups the words in the lexicon data-structure in order to initialize the chart memory. Once the chart memory is initialized an FPGA internal register is configured with the length of the parsed sentence and the signal `PARSE_START`[2] is activated. The parsing result (i.e. the compact parse forest) is written by the FPGA into the FIFO memory from where the IOP 480 processor performs a DMA transfer to the host system (1) as soon as the FIFO is (almost) full or (2) the parsing is finished and the FIFO is not empty. When the parsing ends the FPGA activates the signal `PARSE_FINISHED`[3] which is used to signal the host that the parsing results are available and that a new parsing can start.

Note that two chart memories are used, which allows to overlap the chart memory initialization and the parsing. When one chart memory is initialized with the next sentence to be parsed the other is used to parse the current sentence. When the current parsing ends the two chart memories are swapped[4] and a new parsing can immediately start. Therefore, the initialization and parsing of successive sentences are overlapped and the time lost for chart memory initialization is made transparent.

7.3 The FPGA-board

7.3.1 The PCI interface board

7.3.1.1 IOP 480 I/O processor

The IOP 480 is an I/O processor with a 32-bit 66 MHz PowerPC RISC core, integrated 32-bit PCI bus interface, 3 DMA channels controller, memory controllers and UART. The RISC core is compatible with the PowerPC Book D architecture from an instruction and register-level perspective. The IOP 480 processor interfaces a Local bus running at 66 MHz (available on the POM connector) to the PCI bus running at 33 MHz. The Local bus is the standard J-Mode multiplexed bus. The integrated memory controller can support SDRAM, FLASH, ROM and other types of memory and peripherals.

In order to access the chart memory, the FIFO memory and the other devices on the FPGA expansion board, the IOP 480 processor uses 3 of the 4 available Local bus Chip Selects ($\overline{\mathtt{LCS}}[3:0]$) generated by the integrated memory controller. The Local bus Chip Selects can be configured by means of some IOP 480 internal registers that are used to control the bus width and the timing for the memory regions assigned to each Local bus Chip Select (see [24] for details). The Local bus Chip Selects are used as follows:

[2]the equivalent of the `startPARSE` signal used in the block diagram of the enhanced-CYK design, see page 98
[3]the equivalent of the `overPARSE` signal used in the block diagram of the enhanced-CYK design, see page 98
[4]The hardware supports the swapping of the chart memories by means of bidirectional bus-switches.

- $\overline{\texttt{LCS3}}$ for accessing general purpose FPGA user registers, for FPGA configuration, for swapping the chart memories and for programming the programmable clock;
- $\overline{\texttt{LCS2}}$ for accessing the chart memories (i.e. initialization and possibly readback during debugging);
- $\overline{\texttt{LCS1}}$ for reading the FIFO memory during DMA transfers;

The $\overline{\texttt{LCS0}}$ is already assigned to the 512 KBytes flash memory on the PCI interface board (see appendix C for a detailed IOP 480 memory map). The memory controller has bursting capabilities for supporting high data transfer rates. The FPGA-board uses burst transfers for transferring the parse results from the FIFO memory to the host machine. Such burst transfers take place under the control of the on-chip DMA controller.

The Local bus Chip Select ($\overline{\texttt{LCS}}[3:1]$) signals as well as all IOP 480 signals (i.e. address, data, control) on the Local bus are available on the POM connector and further to the FPGA expansion board.

7.3.1.2 The serial EEPROM

The serial EEPROM on the PCI interface board is a MN93CS66LEN 4 Kbit serial EEPROM, used as a boot device for initializing the IOP 480 processor internal registers – in particular those required to configure the PCI bus controller – after reset.

7.3.1.3 The flash memory

The flash memory is a 512 KByte memory addressed on 8-bit words, with 120 [ns] access time. This memory can be read and written and is selected by the Local bus Chip Select signal $\overline{\texttt{LCS0}}$. It stores the power on initialization routines and the software that allows the host to communicate with the FPGA-board. This software includes routines for configuring the Virtex XCV2000 FPGA on the FPGA expansion board, for initializing the chart and grammar memories and others.

Note: Due to the fact that it is relatively slow, the software it contains should be copied into the SDRAM memory and executed from there.

7.3.1.4 The SDRAM memory

The SDRAM memory consists of four 8Mx8 bit SDRAMs, providing a total of 32 MBytes on-board SDRAM with a 7.5 [ns] access time. The SDRAM controller is part of the integrated memory controller of the IOP 480 and is software programmable. The SDRAM memory is used for mass storage and in particular for storing the lexicon data-structure, the binary image of the grammar memories and the FPGA configuration byte-stream. It is also used to run the software that allows the host computer to communicate with the FPGA-board.

7.3.1.5 The Serial port

The serial port on the PCI interface board is a standard RS-232 DB-9 connector. The IOP 480 processor built-in UART is connected to this connector through a voltage level converter. This interface is very useful during the testing of the FPGA-board for debugging purposes.

7.3.1.6 The PLX Option Module Connector

The PLX Option Module (POM) connector is directly connected on the 32-bit multiplexed J-Mode Local bus.

Some signals used by the FPGA expansion board are not available on the POM connector but can be retrieved on two 2x10 headers (JP5 and JP8, see [23]) available on the PCI interface board. These signals are: $\overline{\texttt{LCS1}}$, $\overline{\texttt{MOE}}$, $\overline{\texttt{MWE}}$ and the latched address lines $\texttt{MA}[\texttt{16 : 0}]$.

7.3.2 The FPGA expansion board

The FPGA expansion board contains the hardware resources required for implementing a 16-processors enhanced-CYK design as presented in chapter 6. The FPGA expansion board is built as an expansion module for the PCI interface board and uses the POM connector for this purpose. The schematics for the FPGA expansion board are available in appendix C. For all the signals mentioned in this chapter please refer to these schematics.

7.3.2.1 The programmable clock

The circuit used is a dual programmable clock generator ICD 2051 with two independent clock outputs ranging from 320 KHz to 100 MHz. The circuit also requires a clock reference that is derived from a 10 MHz clock oscillator. The two clocks are fully user-programmable phase-locked loops. The circuit is required for driving the FPGA clock input(s) with a clock frequency that matches the working clock frequency of the design configured in the FPGA circuit. The two clock outputs are used to drive two (out of four) clock pins (i.e. the `GCLK2` and `GCLK3`) of the used Xilinx Virtex XCV2000 FPGA. The second clock may be eventually used to build a more sophisticated SRAM memory access module or for other purposes, but is currently unused.

The ICD 2051 circuit is powered from a 5V power supply while the rest of the FPGA expansion board is using 3.3V signalling. For this reason each of the two programmable clock outputs is passed through a zero delay clock buffer CY 2304 operating at 3.3V and having the clock input tolerant to 5V signalling. The operating range of this circuit is from 10 MHz to 133 MHz, thus the working frequency range of the FPGA expansion board will be restricted to the range from 10 MHz to 100 MHz.

Note: The observation above is important for the host-side software library function that is invoked when programming the programmable clock. This function has to check that the programmed frequency is within the above mentioned range.

The clock output driving the FPGA clock `GCLK2_FPGA` is also used to drive the FIFO memories (required for synchronous writes) available on the FPGA expansion board.

The programmable clock is controlled by the IOP 480 processor. The programmable clock is decoded in the memory space assigned to the Local bus Chip Select $\overline{\texttt{LCS3}}$. The port addresses used for this purpose are given in table 7.1 (see the ICD 2051 data-sheet for more details). A detailed memory map is given in appendix C.

7.3.2.2 The FPGA

The FPGA used on the FPGA expansion board is a Xilinx Virtex-E XCV2000efg1156-6. This FPGA was chosen due to the high number of user available pins (i.e. 804 pins), the internal

port	R/W	meaning
`2000,00C0`	W	pulse `SCLKA` to load `B_LAD0` input
`2000,00C4`	W	pulse `SCLKB` to load `B_LAD0` input
`2000,00C8`	W	set/reset $\overline{\texttt{MUXREFA}}$ for switching the clock output A
`2000,00CC`	W	set/reset $\overline{\texttt{MUXREFB}}$ for switching the clock output B

Table 7.1: Port addresses used to configure the dual programmable clock generator. For details see the ICD 2051 circuit data-sheet.

memory resources (160 Block SelectRAM) and its advanced technology which allows relatively high clock frequencies. Among these features, the internal memory resources were particularly important for the implementation of the enhanced-CYK design presented in chapter 6. The FPGA uses a core voltage of 1.8 V and the user pins are 3.3V LVTTL compatible as the rest of the circuits used on the FPGA expansion board.

The FPGA circuit is configured in the SelectMAP mode which is the fastest configuration option (see [30] for details). The FPGA configuration byte-stream is provided by the host machine and configured in the FPGA under the control of the IOP 480 processor. The signals required for configuring the FPGA are decoded in the memory space assigned to the Local bus Chip Select $\overline{\texttt{LCS3}}$. The port addresses used for controlling these signals are given in table 7.2 (see [30] for details). A detailed memory map is given in appendix C. The user also has access

port	R/W	meaning
`2000,0040`	R	reads the $\overline{\texttt{FPGACFG\_INIT}}$ and `FPGACFG_DONE` status signals
`2000,0050`	W	write the $\overline{\texttt{FPGACFG\_PROGRAM}}$ control signal
`2000,0080`	W	activates the $\overline{\texttt{FPGACFG\_CS}}$ FPGA chip select
`2000,0000`	R/W	FPGA general-purpose register R_0 (sentence length)
`2000,0004`	R/W	FPGA general-purpose register R_1 (fault code, MSB)
`2000,0008`	R/W	FPGA general-purpose register R_2 (fault code)
`2000,000C`	R/W	FPGA general-purpose register R_3 (fault code, LSB)
`2000,0010`	R/W	FPGA general-purpose register R_4 (grammar initialization support)
`2000,0014`	R/W	FPGA general-purpose register R_5 (grammar initialization support)
`2000,0018`	R/W	FPGA general-purpose register R_6 (not assigned)
`2000,001C`	R/W	FPGA general-purpose register R_7 (not assigned)
`2000,0020`	R/W	FPGA general-purpose register R_8 (not assigned)
`2000,0024`	R/W	FPGA general-purpose register R_9 (not assigned)
`2000,0028`	R/W	FPGA general-purpose register R_{10} (not assigned)
`2000,002C`	R/W	FPGA general-purpose register R_{11} (not assigned)
`2000,0030`	R/W	FPGA general-purpose register R_{12} (not assigned)
`2000,0034`	R/W	FPGA general-purpose register R_{13} (not assigned)
`2000,0038`	R/W	FPGA general-purpose register R_{14} (not assigned)
`2000,003C`	R/W	FPGA general-purpose register R_{15} (not assigned)

Table 7.2: Port addresses used to control the signals involved in the configuration of the Virtex-E XCV2000 FPGA and for accessing the 16 8-bit FPGA internal general-purpose registers.

to 16 8-bit FPGA internal general-purpose registers that can be used for data-communication, state readback and during the debugging phase of an implemented design. One of these registers is used for configuring the length of the parsed sentence. These registers are also decoded in the memory space assigned to the Local bus Chip Select $\overline{\texttt{LCS3}}$. The signals used to read/write the internal FPGA general-purpose registers are: $\texttt{MA}[3:0]$ (used to select one of these registers) , $\overline{\texttt{MOE}}$, $\overline{\texttt{MWE}}$ and $\overline{\texttt{FPGAUSER\_CS}}$. In table 7.2 is given the port addresses used to access the FPGA internal general-purpose registers. In particular the register $\texttt{R}_0$ is used for configuring the length of the sentence to parse and the registers $\texttt{R}_1$, $\texttt{R}_2$ and $\texttt{R}_3$ are used to store the error code in case a fault occurred during the parsing. The code stored in these registers allows the user to track the unit that generated the error signal (see section 6.4.9 for details). The registers $\texttt{R}_4$ and $\texttt{R}_5$ are used to support the grammar memories initialization.

7.3.2.3 FIFO memory

The FIFO memory used on the FPGA expansion board is implemented with 4 CY7C4291-10JI and works at a frequency up to 100 MHz. The CY7C4291-10JI is a 128K x 9bit FIFO memory and the 4 chips are used to build a 128K x 32bit FIFO memory. The FIFO is fully asynchronous and allows simultaneous read and write operations. Data is written in the FIFO from the FPGA in words of 32-bit and the clocks $\texttt{GCLK2\_FIFO}[4:1]$ (synchronous with $\texttt{GCLK2\_FPGA}$, generated by CY2304) are used for this purpose. Data is read from the FIFO during DMA transfers by the IOP 480 DMA controller at the 66 MHz clock frequency of the IOP 480 Local bus, available on the $\texttt{IOP\_CLK}$ on the POM connector. The "glue"-logic required for accessing the FIFO memory during DMA transfers is implemented in the ALTERA 7128AE CPLD. The DMA transfer is initiated when the flag $\overline{\texttt{PAE}}$ (FIFO almost empty) goes high, meaning that a significant amount of data is already in the FIFO and is stopped when the empty flags $\texttt{EF}[4:1]$ are activated. Using DMA transfers on 32-bit insures that the full bandwidth available on the PCI bus is used. The FIFO memory is mapped in the IOP 480 memory space $\texttt{B000},\texttt{0000} - \texttt{EFFF},\texttt{FFFF}$ and is decoded by the Local bus Chip Select signal $\overline{\texttt{LCS1}}$. See appendix C for a detailed memory map.

7.3.2.4 Chart memory

The chart memory used on the FPGA expansion board is implemented with 8 TC55V16100FT-12. Each TC55V16100FT-12 is a 1M x 16bit static RAM. As figure 7.2 illustrates, two chart memories are used, which allows to overlap the chart memory initialization and the parsing. When the chart memory 1 is initialized with the next sentence to be parsed, the chart memory 2 is used for parsing the current sentence. When the parsing ends a new one can immediately begin on the chart memory 1 which has already been initialized, and so on.

Both the chart memory 1 and the chart memory 2 are build from 4 TC55V16100FT-12 which are organized in two blocks, low L and high H (see figure 7.2). The L and H blocks are build from 2 TC55V16100FT-12 and organized as 1M x 32bit memories which can be directly accessed by the IOP 480 processor on the Local bus which is a 32-bit databus. On the other hand, the FPGA accesses the L and H blocks of a chart memory in parallel on a 64-bit databus. The IOP 480 processor uses the Local bus Chip Select $\overline{\texttt{LCS2}}$, mapped in the memory space $\texttt{6000},\texttt{0000} - \texttt{60FF},\texttt{FFFF}$ to access the chart memories. More precisely, the selection signals used by the IOP 480 to access the chart memories and the mapping of the chart memories in the IOP 480 memory space is given bellow:

- chart memory 1, L: selected by $\overline{\texttt{CYKSRAM1\_LCS}}$, mapped in $\texttt{6000},\texttt{0000} - \texttt{603F},\texttt{FFFF}$;

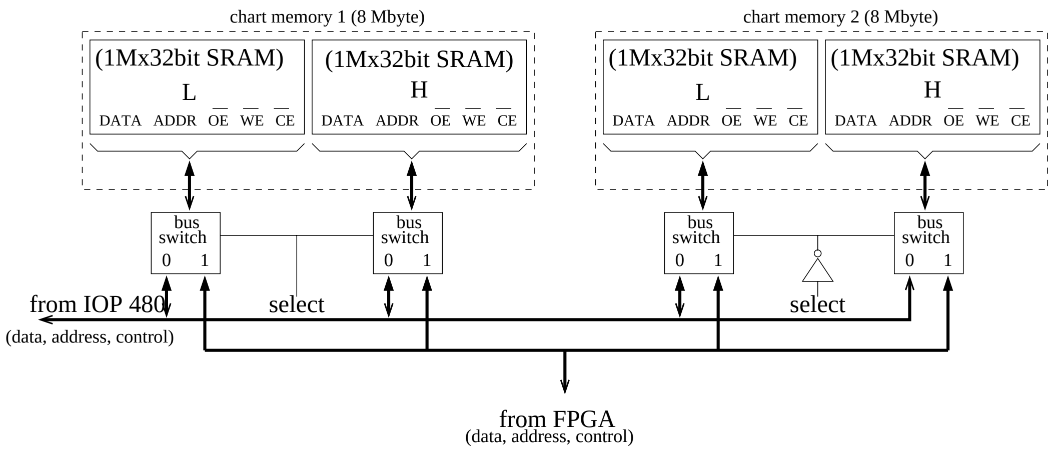


Figure 7.2: The organization of the chart memories on the FPGA expansion board. Two chart memories are used in order to overlap the parsing and the chart initialization.

- chart memory 1, H: selected by $\overline{\texttt{CYKSRAM1\_HCS}}$, mapped in $\texttt{6040},\texttt{0000} - \texttt{607F},\texttt{FFFF}$;
- chart memory 2, L: selected by $\overline{\texttt{CYKSRAM2\_LCS}}$, mapped in $\texttt{6080},\texttt{0000} - \texttt{60BF},\texttt{FFFF}$;
- chart memory 2, H: selected by $\overline{\texttt{CYKSRAM2\_HCS}}$, mapped in $\texttt{60C0},\texttt{0000} - \texttt{60FF},\texttt{FFFF}$;

On the other hand, the FPGA sees the chart memories starting at the physical address 0.

Passing the control of a chart memory to the IOP 480 or the FPGA means that full control is given over the databus DATA, address bus ADDR and control signals ($\overline{\texttt{WE}}$, $\overline{\texttt{OE}}$ and $\overline{\texttt{CE}}$) of that chart memory. This is implemented by means of 12 bidirectional "near-zero delay" bus-switches IDTQS34XVH245. The SRAMSEL signal – controlled by the IOP 480[5] – is used to control the bus-switches in order to assign the chart memories ownership to the IOP 480 processor or respectively the FPGA. Table 7.3 gives the routing of the IOP 480 Local bus signals and respectively the FPGA bus signals to the L and H banks of the chart memories under the control of the SRAMSEL signal. Note that when SRAMSEL = ”0” the chart memory 1 is under the control of the IOP 480 while the chart memory 2 is under the control of the FPGA and vice-versa.

7.3.2.5 Grammar memories

Each grammar memory is implemented with a 1M x 16bit TC55V16100FT-12 static RAM. This amount of memory is considered enough for storing large-size real-life CFGs. For instance, the SUSANNE grammar without unitary rules only uses 1/4 of this memory space.

Before the parsing starts, each grammar memory is initialized by the IOP 480 from a binary image of the grammar data-structure provided by the host machine. In order to reduce the number of components on the FPGA expansion board the grammar memories are only connected to the FPGA and not to the IOP 480 Local bus. The only access for the IOP 480 to the grammar memories is through the FPGA. Thus, the initialization of the grammar memories can only be done with the support of the design (i.e. enhanced-CYK) configured in the FPGA. The enhanced-CYK design should be extended to include a mechanism that supports the IOP

[5]The SRAMSEL signal is decoded in the memory space assigned to $\overline{\texttt{LCS3}}$. The port address 2000, 00B0 is used to control this signal.

chart memory 1						
		|DATA	ADDR	$\overline{\texttt{OE}}$	$\overline{\texttt{WE}}$	$\overline{\texttt{CE}}$
SRAMSEL = "0"	L	LAD[31:0]	LATCH_LAD[21:19] & MA[16:0]	$\overline{\texttt{MOE}}$	$\overline{\texttt{MWE}}$	$\overline{\texttt{CYKSRAM1\_LCS}}$
	H	LAD[31:0]				$\overline{\texttt{CYKSRAM1\_HCS}}$
SRAMSEL = "1"	L	FPGA_DATA[31:0]	FPGA_ADDR[19:0]	$\overline{\texttt{FPGA\_OE}}$	$\overline{\texttt{FPGA\_WE}}$	$\overline{\texttt{FPGA\_CE}}$
	H	FPGA_DATA[63:32]				$\overline{\texttt{FPGA\_CE}}$

chart memory 2						
		|DATA	ADDR	$\overline{\texttt{OE}}$	$\overline{\texttt{WE}}$	$\overline{\texttt{CE}}$
SRAMSEL = "0"	L	FPGA_DATA[31 : 0]	FPGA_ADDR[19 : 0]	$\overline{\texttt{FPGA\_OE}}$	$\overline{\texttt{FPGA\_WE}}$	$\overline{\texttt{FPGA\_CE}}$
	H	FPGA_DATA[63 : 32]				$\overline{\texttt{FPGA\_CE}}$
SRAMSEL = "1"	L	LAD[31:0]	LATCH_LAD[21:19] & MA[16:0]	$\overline{\texttt{MOE}}$	$\overline{\texttt{MWE}}$	$\overline{\texttt{CYKSRAM2\_LCS}}$
	H	LAD[31:0]				$\overline{\texttt{CYKSRAM2\_HCS}}$

Table 7.3: Routing of the IOP 480 Local bus signals and respectively the FPGA bus signals to the L and H banks of the chart memories under the control of the SRAMSEL signal. The organization of the chart memory 1 and the chart memory 2 is illustrated in figure 7.2.

480 during the initialization of the grammar memories. For this purpose the FPGA internal general-purpose registers R_2 and R_3 are used (see section 7.3.2.2 and table 7.2). The R_2 register is used for data transfers and the R_3 register is used for signalling (e.g. marking the end of the initialization, handshake).

Thus, the initialization of the grammar memories is done in 3 steps: (1) the FPGA is configured with the enhanced-CYK design extended to support the initialization of the grammar memories, (2) the enhanced-CYK design uses the grammar memory initialization mechanism for initializing the grammar memories and (3) the enhanced-CYK design is switched to normal operation and waits for sentences to parse. The extension on the enhanced-CYK design is relatively easy to implement and is not discussed here.

7.3.2.6 "Glue"-logic

Implemented with a ALTERA EPM7128AE CPLD that can work up to 100 MHz. The circuit implements all the "glue"-logic required for interfacing: the FIFO to the IOP 480 DMA controller, the IOP 480 to the FPGA as well as for generating the port and chart memories selection signals. It can be programmed in-system via the industry standard 4-pin IEEE 1149.1 (JTAG) interface.

7.3.2.7 Display

Implemented with a DLG1414 circuit. A general purpose display that can be used, for instance, during debugging to display FPGA internal status information.

7.3.2.8 Power supply

The FPGA expansion board uses three power supplies: 5V for the dual-programmable clock generator and the display, 3.3V for most of the components and 1.8V for the FPGA core voltage. The power consumption on the 5V supply and 1.8V supply is not critical and does not require special precautions. However, on the 3.3V supply the peak current consumption is around 7A in the assumption that all the SRAMs on the FPGA expansion board work at maximum frequency. For this reason the power supply used on the FPGA expansion board is powered directly from a 5V source available on the host machine and not through the PCI interface board. The 3.3V and 1.8V are derived locally with a LT1584CT and respectively LT1764 circuits.

7.4 Conclusions

This chapter presented an FPGA-board that implements the enhanced-CYK design presented in chapter 6. The proposed FPGA-board contains the necessary hardware resources for implementing a 16-processors enhanced-CYK design and features a PCI interface that allows it to be integrated and work as an accelerator within a host machine (e.g. desktop PC).

Chapter 8

Conclusions

In this thesis we have presented an FPGA-based hardware implementation of a syntactic parsing algorithm – an enhanced version of the CYK algorithm – that can deal with large-size real-life context-free grammars and is adapted for word lattice parsing. In this final chapter, we first summarize the results obtained during our research. Then, we propose some improvements to the current implementation and outline some future research directions.

8.1 Analysis of the results

Our work started by the investigation of a 2D-array of processors implementing the CYK algorithm presented in [8]. This approach proved to be much too demanding in terms of hardware resources for being implemented within state-of-the-art FPGAs.

We further investigated whether a 1D-array of processors implementing the CYK algorithm is feasible – in terms of required hardware resources and data-structures – for being implemented within state-of-the-art FPGAs. The results of this study was presented in chapter 3 in which a linear array of processors (LAP) architecture implementing the CYK algorithm was proposed and tested for validation on a commercial FPGA-board. The proposed LAP architecture uses $n - 1$ processors for parsing sentences with up to n words and word lattices with up to $n + 1$ time stamps. The implementation of the LAP design was the first step of our design methodology during which we proved the feasibility of an FPGA-based hardware implementation of the CYK algorithm, both in terms of required hardware resources and size of data-structures. The LAP design also exhibited a significant $16\times$ speedup factor when compared against a software implementation of the enhanced-CYK algorithm.

However, the LAP architecture neither efficiently exploits the parallelism available in the CYK algorithm nor the available processors. We therefore investigated a more aggressive and efficient processor allocation mechanism for further increasing the speedup factor. This new approach for processor allocation mechanism was investigated in chapter 5 in which a dynamic array of processors (DAP) architecture implementing the CYK algorithm was proposed. We called the design a dynamic array of processors, due to the fact that the processors are dynamically allocated to process cell-combinations in the chart. The DAP architecture has a better average processor utilization in comparison to the LAP design and a better 29 to $34\times$ speedup factor while using a slightly modified processor and the same data-structures for the chart and the grammar.

Next, we investigated (in chapter 6) a dynamic array of processors architecture implementing the enhanced-CYK algorithm, called enhanced-CYK. The enhanced-CYK architecture proposes a method called tiling that can further increase the average processor utilization and the speedup factor. The idea of tiling was suggested by the observation that there are large cell-combinations that require a long time to process and that all cell-combinations that are data-depend on the results of these large cell-combinations cannot be processed having to wait for these results to be available. The tiling mechanism then consists of splitting expensive (i.e. large) cell-combinations in smaller chunks that can be more efficiently processed by the processors. The tiling mechanism can therefore be seen as a dynamic clustering of the available processors around expensive cell-combinations. In other words, it is a way of concentrating the processing resources when and where they are needed. The enhanced-CYK architecture yields a 65 to $85\times$ speedup factor and can parse sentences with up to 256 words.

In brief, the main results achieved in this thesis are:

- a processor allocation method called dynamic processor allocation that better exploits the parallelism available in the (enhanced-)CYK algorithm(s). The dynamic processor allocation method allows the design to dynamically assign the processors to compute cells in the chart, which was not the case in the LAP design that statically assigns the processors to cells in the chart;

- a mechanism called tiling that allows several processors to work on the same (large) cell-combination. This mechanism is useful for small sentence lengths when in general not all the processors are busy during parsing or for large sentence lengths when filling the cells towards the top of the chart;

- an FPGA-based hardware implementation of the enhanced-CYK algorithm that integrates the dynamic processor allocation method and the tiling mechanism. The enhanced-CYK design has the following features: it parses word lattices, can deal with large-size context-free grammars and can parse long sentences (up to 256 words or time-stamps). These features, along with the achieved speedup factor, render the hardware implementation well suited for being integrated in NLP applications that have strong data-size and/or real-time constraints or within a speech recognition application framework;

- an accelerator FPGA-board that implements the enhanced-CYK design is proposed in chapter 7. The proposed FPGA-board contains all the necessary hardware resources for implementing a 16-processors enhanced-CYK design and features a PCI interface that allows it to be integrated and work as an accelerator within an application framework as illustrated in figure 1.2 at page 8;

An important feature of the proposed enhanced-CYK design is its ability to be configured (as an IP core) – by means of generics in the VHDL code – according to the CFG considered. The synthesis will therefore produce a design with optimal resources (e.g. registers, counters) sizes for the particular considered CFG. The modularity of the VHDL code also allows the enhanced-CYK design to be targeted on "any" FPGA-board that contains a Virtex-E family FPGA and enough memory resources for storing the chart and the grammar data-structures. This modularity allows us to use a 8-bit, 16-bit or 32-bit databus with the grammar memories or even a 1-bit, 2-bit or 4-bit databus. The same remark applies to the databus with the chart memory. Even more, a mixed databus environment is also allowed. For instance some grammar memories may use a 16-bit databus while the others use an 8-bit databus.

8.2 Future work

Several improvements can be added to the current enhanced-CYK design. Among these, we mention (1) a faster tile dispatching mechanism, (2) an adaptive tile sizing that will take into account the size of each individual cell-combination and the number of processors available in the system and (3) a time-efficient mechanism for compressing/decompressing the compact parse forest before shipping it to the host machine. The integration of the FPGA-board within an application framework will also be required for testing the utility of such a tool.

Another research direction – which is also interesting as a standalone research topic – is to investigate the possibility to use FPGA internal resources for implementing (i.e. representing) the CFGs. While certain CFGs may require less resources (e.g. combinatorial logic) to be implemented than others an interesting investigation will be to seek a particularly synthesis-friendly CFG form.

Finally, to fully exploit the FPGA-board that will result of this research, a software library will need to be written to implement the communication interface between the host machine and the FPGA-board.

The FPGA-board will then allow to perform tests and performance measurements in a very efficient way when compared to the time required to carry the VHDL simulations (usually taking about 1 week). The fast feedback will allow to explore several design variations and occasionally to perform exhaustive analyses on the tested designs.

Appendix A

Transformation steps towards CNF grammars

A.1 SlpToolKit organisation of SUSANNE grammar

The SUSANNE grammar we used is extracted from the SUSANNE corpus. Due to the fact that this grammar is used by SlpToolKit it has some particularities that we briefly describe in the following lines. When transforming the SUSANNE grammar in its equivalent Chomsky Normal Form (CNF), special attention should be given to these particularities.

We saw that a CFG is defined as $G = \{V, \Sigma, P, S\}$. In the context of NLP the terminals (Σ) are the words used in the language generated by grammar G. From the definition of the CFG we can see that the grammar can contain words in the rules. However, this kind of grammars – called lexicalized grammars – are not often used when dealing with natural languages for practical reasons. This is also the case of the SUSANNE grammar which is not a lexicalized grammar.

SlpToolKit uses two files for storing a grammar, in particular the SUSANNE grammar. The first file, called the lexicon file, stores all (and only) the rules such as $A \rightarrow w$ (called lexical rules) where w is a terminal (word). The second file stores all the other remaining grammar rules and is called, for convenience, the grammar file. Both the lexicon and the grammar files are text files.

Another particularity of the grammars used by SlpToolKit is a category of non-terminals, called pre-terminals, used to represent sets of words in the lexicon. For example, in the grammar rule $A \rightarrow w$, where w is a word, the left-hand side A is a pre-terminal. The pre-terminals are associated, in general, with morpho-syntactic categories of the lexicon (e.g. noun, verb). The pre-terminals use a special notation *:nnn* (e.g. :1415, :19, ...). For more details regarding these files see the documentation of SlpToolKit[1].

Every line in the grammar file is a grammar rule. The SUSANNE grammar file [2], used by SlpToolKit has the following format:

$\vdots$
$A \rightarrow B\,C\ :14$
$X \rightarrow :1123\ A$
$\vdots$

[1]available under request from {chaps, webmaster}@lia.di.epfl.ch

[2]the grammars considered in this document are assumed non probabilistic

Every line in the lexicon file stores a word and an associated pre-terminal (without ':') separated by '|=|'. A certain word may have associated several pre-terminals (e.g. the word 'can' is a noun and a verb). In this case it occures in several lines of the lexicon file. The SUSANNE lexicon file used by SlpToolKit has the following format:

$$\begin{array}{l} \vdots \\ camps| = |1369 \\ camps| = |1373 \\ can| = |386 \\ can| = |388 \\ can| = |1170 \\ \vdots \end{array}$$

A.2 Transforming the general SUSANNE CFG to its CNF.

During the extraction of the SUSANNE grammar from the SUSANNE corpus, no ϵ-rules (i.e. rules like $A \rightarrow \epsilon$ where ϵ is the empty string) are introduced. We also assume that no useless rules are introduced.

With these assumptions, in order to transform a CFG to an equivalent CNF one should apply the following transformation steps:

- step 1: eliminate the unitary rules (i.e. $X \rightarrow Y$)
- step 2: tranform all rules with more than 2 non-terminals in the right-hand-side, in rules of the form $X \rightarrow Y\ Z$

The grammar and lexicon files that result after each transformation step are illustrated in figure A.1. These transformation steps are detailed bellow.

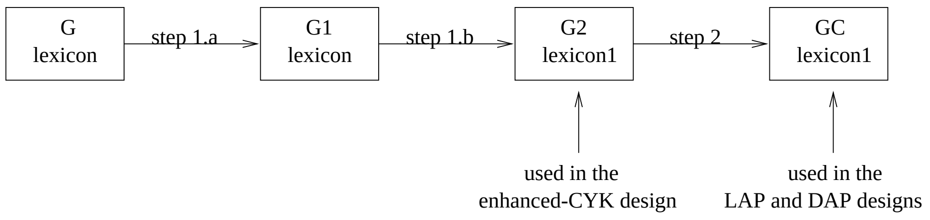


Figure A.1: Transformation steps for the SUSANNE grammar for obtaining the equivalent Chomsky Normal Form.

step 1.a: The unitary rules are eliminated using for instance the algorithm given in [20]. After this transformation the original grammar file `G` becomes the grammar file `G1`. The lexicon does not change.

step 1.b: A particularity of the SUSANNE grammar is that it contains rules such as $A \rightarrow : nnn$ that cannot be eliminated with the same algorithm by treating them as unitary rules because they will make the number of grammar rules grow too much (i.e. explode). In a test we made on an algorithm that treats the rules of type $A \rightarrow : nnn$ as unitary rules we got as result for the transformed grammar a number of more than $1,300,000$ new rules[3]! This number of

[3]the number of rules in the original grammar is $17,669$

rules is not resonable and in conclusion we should find another way to eliminate these rules. A solution to eliminate rules like $A \rightarrow : n_1 n_1 n_1| : n_2 n_2 n_2| \ldots$ is to modify both the lexicon and the grammar files as described bellow:

1. code A with $: kkk$ (i.e. A becomes a new 'pre-terminal') and replace all occurences of A in the grammar with its code $: kkk$ [4]

2. for every word in the lexicon like:
   word|=|$n_1 n_1 n_1$ or word|=|$n_2 n_2 n_2$ or ... we introduce *only once* a new line:
   word|=|kkk

After these transformations we can remove the rules $A \rightarrow : n_1 n_1 n_1| : n_2 n_2 n_2| \ldots$. The obtained grammar file `G2` plus lexicon file `lexicon1` (see figure A.1) is weakly equivalent to the previous grammar file `G1` plus lexicon file `lexicon`. The inconvenience with this procedure is the growth in size for the `lexicon` file from $18,273$ lines to $228,297$ lines in `lexicon1` file, which is however reasonable.

optional step: At this point we may consider to eliminate the useless rules in the grammar `G2`. The useless rules are those rules that never take part in a derivation. This step is however not necessary and may be skipped. An algorithm can be found in [20].

step 2 : In order to transform the grammar `G2` in an equivalent CNF grammar file `GC`, the algorithm bellow is aplyied:

- 1: cnt=0;

- 2: remove from `G2` and put in the output grammar `GC` the rules that are CNF

- 3: if `G2` is empty (no rules left) then **STOP**

- 4: build a table with all the distinct pairs of non-terminals and their frequency that occur in the right hand-side of the remaining rules in `G2`

- 5 : take the most frequent of these pairs and replace it everywhere in `G2` with a new non-terminal. This new non-terminal has to be unique and we built the symbol as A_cnt (e.g. A_1, A_2, ...).
  Increment cnt
  Go to 2

By replacing the most frequent pair in step 3 of the algorithm above we have a greedy algorithm that tries to reduce as much as possible the size of the grammar. This algorithm is not optimal in the sense that it does not produce necessarily the smallest possible grammar `GC`. The lexicon does not change during this transformation step.

A.3 Transformation steps detailed.

All the steps described bellow are discussed in the context of the particular grammar format used for representing a CFG for the SlpToolKit. The processing steps are:

[4]this may result in occurence of 'pre-terminal' symbols in the left hand-side of the new grammar rules, and therefore the term 'pre-terminal' loses its meaning.

- step 1.a (elimination of unitary rules): The program that makes the first transformation takes as input the SUSANNE grammar `G`. Bellow are given some characteristics of `G`:

  # of $RULES = 17,669$
  # of non-terminals = 1,920
  # of unitary rules = 206 (to be eliminated in this step)
  # of non unitary rules = 17,463

  During the transformation a number of $49,652$ new rules are added. The output of this program is the transformed grammar `G1` that does not contain unitary rules. Some characteristics of the grammar `G1` are:

  # of $RULES = 67,115 = 17,669 + 49,652 - 206$
  # of non-terminals = 1,920

- step 1.b (elimination of rules like $A \rightarrow: n_1n_1n_1| : n_2n_2n_2| \ldots$): The program that makes this transformation takes as input the grammar `G1` resulted in the previous step and the `lexicon`. The output is a text file `G2` containing the transformed grammar that does not contain rules like $A \rightarrow : n_1n_1n_1| : n_2n_2n_2| \ldots$ and a new lexicon file `lexicon1`. Some characteristics of the grammar `G2` are:

  # of $RULES = 66,123$
  # of non-terminals = 1,912
  # of Chomsky Normal Form rules = 8,587
  # of non Chomsky Normal Form rules = 66,123 - 8,587= 57,536

  The `lexicon1` contains $228,297$ lines in comparison to only $18,273$ in the `lexicon`.

- step 2 (rules transformation) : The program that makes this transformation takes as input the grammar file `G2`. The output is a grammar file `GC` that contains only CNF rules. The `lexicon1` does not require changes.

  Some characteristics of the grammar `GC` are:

  # of $RULES = 74,340 = 66,123 + 8,217$
  # of non-terminals =10,129 = 1,912 + 8,217

  A number of $8,217$ new non-terminals (A_1, A_2, $\ldots$, A_{8217}) were introduced in this transformation step and of course the same number of rules.

All the algorithms implemented for transforming the SUSANNE grammar in CNF grammar were written in the Python language.

A.4 Creating and validating the memory image

Three kind of elements can be distinguished in the data-structure used for representing the CNF grammar in section 3.4.2, namely (1) the non-terminal, (2) the pointer, and (3) the right hand-side code. Due to constraints imposed by this data-structure which are:

parameter	size (bits)
`NT_SIZE`	14
`PTR_SIZE`	19
`RULE_SIZE`	15

Table A.1: Grammar data-structure parameters sizes for the SUSANNE CNF grammar (`GC`.

- an entry in level 1 is 4 bytes (32 bits);
- an entry in level 2 is 4, 8 or 12 bytes (maximum 96 bits);
- an entry in level 3 is 4 bytes (32 bits) for the header and 4 bytes (32 bits) for a pair of non-terminals;

a maximum number of 14 bits can be used for representing a non-terminal (the restriction actually comes from the chart memory representation, see section 3.4.1), 26 bits for a pointer, and 32 bits for a right hand-side code. The CNF grammar data-structure is alligned to a 4-byte boundary.

These maximal parameter sizes can accomodate (most) real-life CNF grammars. A CNF grammar that requires a larger size for any of these parameters cannot be accomodated within the proposed data-structure. The actual sizes for the non-terminal, pointer and right hand-side code depend on the considered CNF grammar. We will denote henceforth by `NT_SIZE` the size of the non-terminal, by `PTR_SIZE` the size of the pointer and by `RULE_SIZE` the size of the right hand-side code.

The value of the `PTR_SIZE` parameter is not known prior to memory image creation and the maximum value (i.e. 26 bits) for this parameter is assumed. The `NT_SIZE` and `RULE_SIZE` parameters can be easily evaluated. We used for this purpose the Python programming language, that can easily handle grammars stored in text files (see section A.1). Once a first estimate for the memory image size is obtained, the real value for the `PTR_SIZE` parameter is obtained and the pointers in the data-structure can be correctly (re)alligned according to the new value. The created CNF grammar memory image is stored in a dump file in a big-endian format.

In particular for the SUSANNE CNF grammar the size of the CNF grammar memory image is $558,576$ bytes and the values for the `NT_SIZE`, `RULE_SIZE` and `PTR_SIZE` are given in table A.1. These values are used to configure the VHDL code (i.e. using generics) to generate the right register sizes, counter sizes, comparator sizes and buses sizes that match the characteristics of the used CNF grammar. In order to validate the created CNF grammar memory image, a program that can extract the CNF grammar from the dump file was also writen. The extracted CNF grammar was compared to the original CNF grammar for validation.

Appendix B

Facts about the (enhanced)-CYK algorithm

B.1 The number of cell combinations for an n word sentence

A cell of the chart is filled by combining several pairs of already filled cells (precisely $i-1$ cell pairs for a cell in row i). The operation of combining a pair of cells (i.e. cell-combination) is the basic operation performed by a processor and we assume that a processor performs one such basic operation in a time-step. For a sentence with n words, the recognition/parsing result is available after a certain number C of cell-combinations has been performed. The number C can be computed as follows:

$$C \quad = \quad (n-1)*1+(n-2)*2+\cdots+(n-(n-1))*(n-1)=\frac{n^3-n}{6}$$

The first term in the summ represents the number of cell-combinations performed for filling the first row of the CYK chart ($n-1$ times 1 cell-combination), the second term is the number of cell-combinations performed for filling the second row ($n-2$ times 2 cell-combinations) and so on.

B.2 The *dynamic* processor allocation method

The linear array of processors (LAP) architecture presented in chapter 3 has a low processor utilization due to the method used for allocating cell-combinations to processors. The allocation of cell-combinations to processors within the LAP design is intrinsic to the design which processes the chart cells in a row-by-row manner and has the advantage of taking into account – without requiring aditional hardware – the data-dependencies among the cells.

In order to increase the processor utilization we introduce a method called *dynamic* processor allocation. A hardware architecture using the *dynamic* processor allocation method would be able to accomodate any number of processors and can be seen as a general case for the 2D-array of processors presented in [8] if enough processors are available. The method works as follows: at each time step allocate as many processors as possible (given the data-dependency restrictions) for performing cell-combination in the chart. If a number P of processors are available, ideally at each time step all P processors will be allocated to cell-combinations. However, this will not always be possible due to data-dependency constraints in the chart. On the other hand, in practice, a design based on the *dynamic* processor allocation method may suffer due to the delays introduced for cell-combination dispatching. These delays are a direct consequence of the fact that the cell-combination dispatcher has to take into account dynamic data-dependency constraints that is not an easy task and may therefore introduce some significant overhead.

By assuming that the overhead introduced for dispatching cell-combinations is insignificant, the *dynamic* processor allocation method can be used for computing the minimum number of processors required for parsing a sentence of length n within $n-1$ time-steps – as if a 2D-array of processors is used. Given the sentence length n, the procedure implementing the dynamic processor allocation method used to compute the minimum number of processors P is given bellow: The above algorithm starts with a tight assumption on the number of processors

Algorithm 3 Minimum number of processors required for filling the chart in linear time ($n-1$, for a given sentence length n) when using the dynamic processor allocation method

```
1:  $found = false$
2:  $P = 1$
3:  **while** (not $found$) **do**
4:     empty chart
5:     **while** ((chart not filled) and ($step \leq n-1$)) **do**
6:        allocate at most $P$ if possible
7:        $step++$
8:     **end while**
9:     **if** (chart not filled) **then**
10:       $P++$
11:    **else**
12:       $found = true$
13:    **end if**
14: **end while**
15: return $P$
```

($P = 1$) required to parse (i.e. fill the chart) a sentence of length n within $n-1$ time-steps. Then, it tries to fill the chart within $n-1$ time-steps using P processors, while taking into account the data-dependencies among cells during chart filling. If more then $n-1$ steps are required for filling the chart, the number of processors P is increased and the procedure retries to fill the chart. The chart is emptied each time a new filling is retried. If the chart is filled within $n-1$ time-steps the returned value of P represents the minimum number of processors required for parsing a sentence of length n within $n-1$ time-steps. For a sentence length n taking values between 2 and 64, the number of processors computed with the procedure above is tabulated in table B.1 and compared against the number of processors required in a 2D-array of processors, given by the formula $n(n-1)/2$. The number of processors computed with the given procedure corresponds to the worst-case assumption when all the cells in the chart are non-empty. However, this is not the case for real-life CFGs when quite a significant number of cells in the chart may be empty. A direct consequence is that for real-life CFGs even less processors are actually required for parsing a sentence of length n within $n-1$ time-steps than computed for the worst-case assumption.

The number of processors actually required when parsing real-life sentences (extracted from the SUSANNE corpus) with the CYK, respectivelly enhanced-CYK algorithms is measured in the following sections.

In the following two sections we will compare the theoretical (i.e. worst-case) number of

n	# processors	n	# processors	n	# processors	n	# processors
–	–	17	62 (136)	33	228 (528)	49	498 (1176)
2	1 (1)	18	69 (153)	34	242 (561)	50	518 (1225)
3	2 (3)	19	77 (171)	35	256 (595)	51	539 (1275)
4	4 (6)	20	85 (190)	36	270 (630)	52	560 (1326)
5	6 (10)	21	94 (210)	37	286 (666)	53	581 (1378)
6	9 (15)	22	103 (231)	38	301 (703)	54	603 (1431)
7	12 (21)	23	112 (253)	39	317 (741)	55	626 (1485)
8	15 (28)	24	122 (276)	40	333 (780)	56	649 (1540)
9	18 (36)	25	132 (300)	41	350 (820)	57	672 (1596)
10	22 (45)	26	142 (325)	42	367 (861)	58	695 (1653)
11	27 (55)	27	153 (351)	43	384 (903)	59	719 (1711)
12	32 (66)	28	165 (378)	44	402 (946)	60	744 (1770)
13	37 (78)	29	177 (406)	45	420 (990)	61	768 (1830)
14	43 (91)	30	189 (435)	46	439 (1035)	62	794 (1891)
15	49 (105)	31	202 (465)	47	458 (1081)	63	819 (1953)
16	55 (120)	32	215 (496)	48	478 (1128)	64	845 (2016)

Table B.1: The worst-case number of processors required for filling the chart within $n-1$ time-steps, given the sentence length n, when using the *dynamic* allocation method. The number in parantheses ($n(n-1)/2$) is the number of processors required by a 2D-array.

processors required for filling the chart within $n-1$ time-steps, where n is the length of the sentence (as given in table B.1), to the number of processors that are actually required when parsing with a concrete real-life CFG. The real-life CFG used in our case-studies is the SUSANNE grammar. For the purpose of this comparison a number of $3,788$ sentences, consisting of all the sentences with lengths between 2 and 32 from the SUSANNE corpus, were parsed with the CYK algorithm and respectivelly with the enhanced-CYK algorithm. For these algorithms, two values are computed for the sentences with the same length n: (1) the maximal number of processors ever needed during a time-step for parsing the sentences and (2) the average number of processors during all time-steps for parsing the sentences.

B.2.1 Case-study on the CYK algorithm

As the CYK algorithm requires a grammar written in CNF, the SUSANNE grammar was first transformed into an equivalent CNF grammar. The CNF equivalent of the SUSANNE grammar consists of $74,340$ grammar rules and $10,129$ distinct non-terminals. When using the CYK algorithm every cell in the chart contains a set of non-terminals that can derive the subsentence associated with that cell.

As we know, a processor combines the sets in two source cells and stores the result in the set of a destination cell. For a given sentence, the set sizes as well as their distribution within the chart during the parsing depends on the used grammar. Whenever one of the sets in the source cells is empty the processor does not have any work to do being therefore unnecessary during that time-step. Figure B.1 depicts for each sentence length n (varying between 2 and 32) the worst-case required number of processors along with the real number of processors actually required (both the maximal and the average number). The same values illustrated in figure B.1

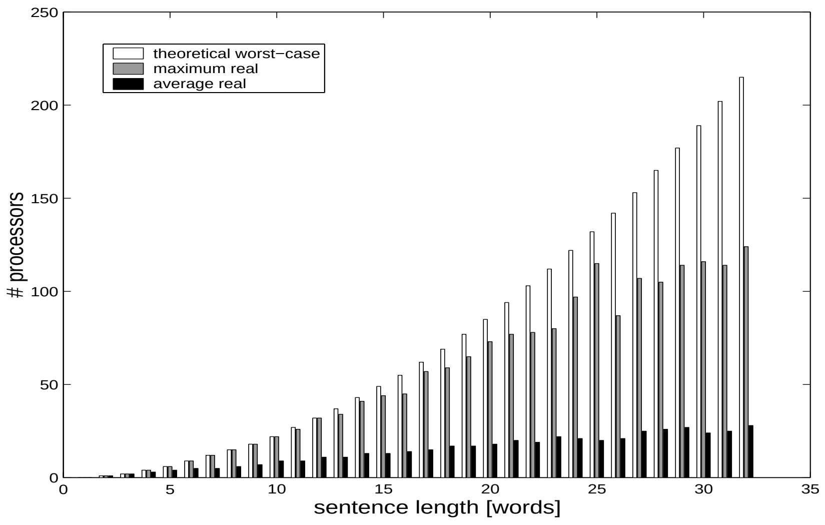


Figure B.1: The theoretical (worst-case) and real (average and maximal) number of processors required to parse sentences of length $2 \leq n \leq 32$, extracted from the SUSANNE corpus, when using the CYK algorithm in a case-study on the SUSANNE grammar. A number of at least 100 sentences were parsed for each value of n.

are tabulated in table B.2. The average number of processors required for parsing any sentence with length n ranging from 2 to 32 within $n-1$ time-steps is 28 and is significantly less than the number of processors required in the worst-case assumption.

B.2.2 Case-study on the enhanced-CYK algorithm

The grammar used in this experiment is a version of the SUSANNE grammar from which we have eliminated all unitary rules (i.e. rules of the form $X \rightarrow Y$). The new grammar contains a number of $66,123$ grammar rules and $1,912$ distinct non-terminals.

When using the enhanced-CYK algorithm every cell in the chart contains two sets of items. One set contains non-terminals (the $N1$ sets) and the second set (the $N2$ sets) contains partial derivations that can derive the subsentence associated with that cell. A processor combines the set $N2$ of the first source cell with the set $N1$ of the second source cell and stores the result of this combination in the set $N1$ and/or $N2$ of a destination cell. Whenever one of the sets in the source cells (either $N2$ for the first or $N1$ for the second) are empty the processor does not have any work to do, being therefore unnecessary during that time-step. Figure B.2 depicts for each sentence length n (varying between 2 and 32) the worst-case required number of processors along with the real number of processors actually required (both the maximal and the average number). The same values illustrated in figure B.2 are tabulated in table B.2.2. The average number of processors (i.e. 18) required for parsing any sentence with length n ranging from 2 to 32 within $n-1$ time-steps is significantly less (actually even smaller when compared to the CYK algorithm) than the number of processors required in the worst-case assumption.

n	worst-case	real (average/maximal)	n	worst-case	real (average/maximal)
-	-	-	17	62	15 / 57
2	1	1 / 1	18	69	17 / 59
3	2	2 / 2	19	77	17 / 65
4	4	3 / 4	20	85	18 / 73
5	6	4 / 6	21	94	20 / 77
6	9	5 / 9	22	103	19 / 78
7	12	5 / 12	23	112	22 / 80
8	15	6 / 15	24	122	21 / 97
9	18	7 / 18	25	132	20 / 115
10	22	9 / 22	26	142	21 / 87
11	27	9 / 26	27	153	25 / 107
12	32	11 / 32	28	165	26 / 105
13	37	11 / 34	29	177	27 / 114
14	43	13 / 41	30	189	24 / 116
15	49	13 / 44	31	202	25 / 114
16	55	14 / 45	32	215	28 / 124

Table B.2: The values for the theoretical (worst-case) and real (average and maximal) number of processors as illustrated in figure B.1.

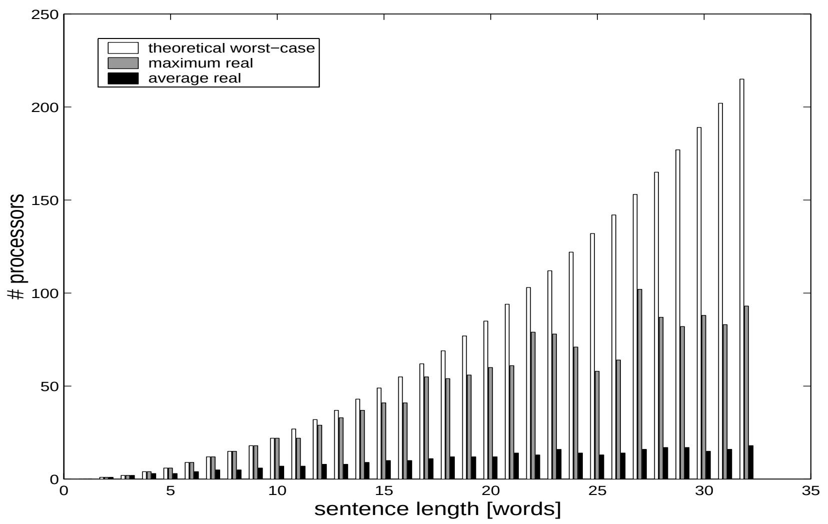


Figure B.2: The theoretical (worst-case) and real (average and maximal) number of processors required to parse sentences of length $2 \leq n \leq 32$, extracted from the SUSANNE corpus, when using the enhanced-CYK algorithm in a case-study on the SUSANNE grammar. A number of at least 100 sentences were parsed for each value of n.

n	worst-case	real (average/maximal)	n	worst-case	real (average/maximal)
-	-	-	17	62	11 / 55
2	1	1 / 1	18	69	12 / 54
3	2	2 / 2	19	77	12 / 56
4	4	3 / 4	20	85	12 / 60
5	6	3 / 6	21	94	14 / 61
6	9	4 / 9	22	103	13 / 79
7	12	5 / 12	23	112	16 / 78
8	15	5 / 15	24	122	14 / 71
9	18	6 / 18	25	132	13 / 58
10	22	7 / 22	26	142	14 / 64
11	27	7 / 22	27	153	16 / 102
12	32	8 / 29	28	165	17 / 87
13	37	8 / 33	29	177	17 / 82
14	43	9 / 37	30	189	15 / 88
15	49	10 / 41	31	202	16 / 83
16	55	10 / 41	32	215	18 / 93

Table B.3: The values for the theoretical (worst-case) and real (average and maximal) number of processors as illustrated in figure B.2.

B.3 The chart cell size for the CYK algorithm

In this section the chart cell size (i.e. the number of non-terminals) for the CYK algorithm is profiled in the real-life case of the CNF SUSANNE grammar. More precisely, the chart cell size distribution is investigated. Such a profiling is useful in order to choose the right chart memory size. In other words, to allocate for a chart cell an amount of memory smaller than the distinct number of non-terminals in the grammar which tends to waste memory space.

For the purpose of these measurements, a number of $4,292$ sentences, with lengths between 2 to 64 words, extracted from the SUSANNE corpus were parsed. For these sentences the number of occurences of each cell size is counted and used to compute the overall cell size distribution. Figure B.3 depicts the frequency with which each cell size occures within the charts.

The figure shows that most of the cells have a size between 1 and 60 non-terminals and that only few cells have a size larger than 128 (exactly 0.027%). This means that, if a cell size of 128 is used, about 0.027% sentences will fail to parse. The maximum cell size is 165 and therefore, a cell size of 256 is sufficient for parsing any sentence from the SUSANNE corpus. This number is significantly smaller than the number $10,129$ of distinct non-terminals in the CNF SUSANNE grammar. A non-terminal is represented on 2 bytes which means that 512 bytes are used for storing a chart cell. It also turns out that in average 63.77% of the overall (over all the parsed sentences) number of cells are empty.

B.4 The chart cell size for the enhanced-CYK algorithm

In this section the chart cell size (i.e. the number of non-terminals and the number of partial rule right-hand sides) for the enhanced-CYK algorithm is profiled in the real-life case of the

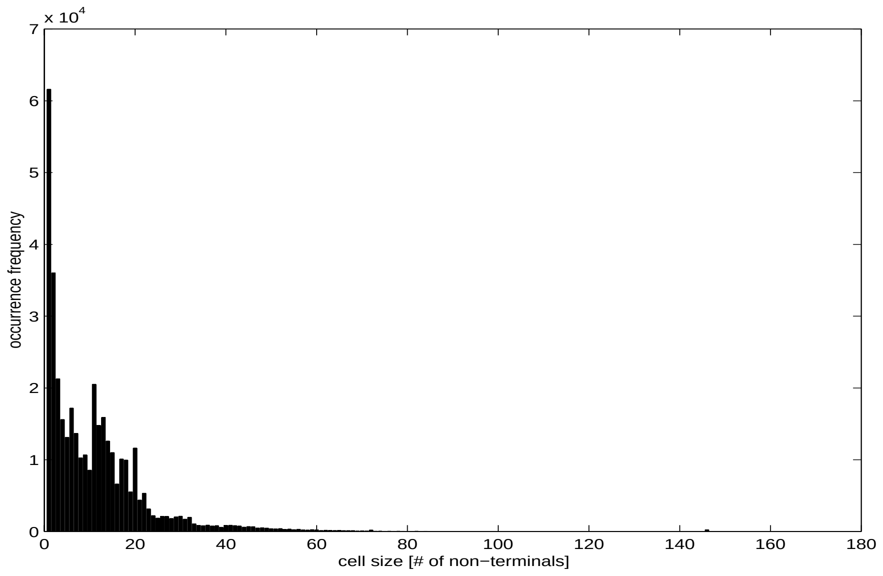


Figure B.3: Cell size distribution for the CYK algorithm in the particular case of the CNF SUSANNE grammar. Computed for a number of $4,292$ sentences with lengths between 2 to 64 words, extracted from the SUSANNE corpus.

SUSANNE grammar without unitary rules. More precisely, the $N1$ set size and respectivelly the $N2$ set size distributions are investigated. Such a profiling is useful in order to choose the right chart memory size for not wasting memory space.

For the purpose of these measurements, a number of $4,292$ sentences, with lengths between 2 to 64 words, extracted from the SUSANNE corpus were parsed. For these sentences the number of occurences of each $N1$ set size (and respectivelly each $N2$ set size) is counted and used to compute the overall $N1$ set size (and respectivelly $N2$ set size) distribution. Figure B.4(a) depicts the frequency with which each $N1$ set size occures within the charts and figure B.4(b) the frequency with which each $N2$ set size occures within the charts.

The figure B.4(a) shows that most of the $N1$ sets have a size between 1 and 128 non-terminals and that only few $N1$ sets have a size larger than 128 (exacly 0.095%). This means that, if a $N1$ set size of 128 is used, about 0.095% sentences will fail to parse. The maximum size of the $N1$ sets is 146 and therefore, a set size of 256 is sufficient for parsing any sentence from the SUSANNE corpus. This number is significantly smaller than the number $1,912$ of distinct non-terminals in the SUSANNE grammar without unitary-rules. A non-terminal is represented on 2 bytes which means that $1,024$ bytes are used for storing a $N1$ set in a chart cell. It also turns out that in average 73.67% of the overall (over all the parsed sentences) number of sets $N1$ are empty.

On the other hand, the figure B.4(a) shows that most of the $N2$ sets have a maximum size of 104 partial rule right-hand sides. Therefore, if a $N2$ set size of 256 is used any sentence from the SUSANNE corpus can be parsed.

<u>Note</u>: using the same set size for the $N1$ sets and for the $N2$ sets is important in order to (1) simplify the retrieval of the sets in the chart memory and (2) to simplify the hardware required for processing the sets.

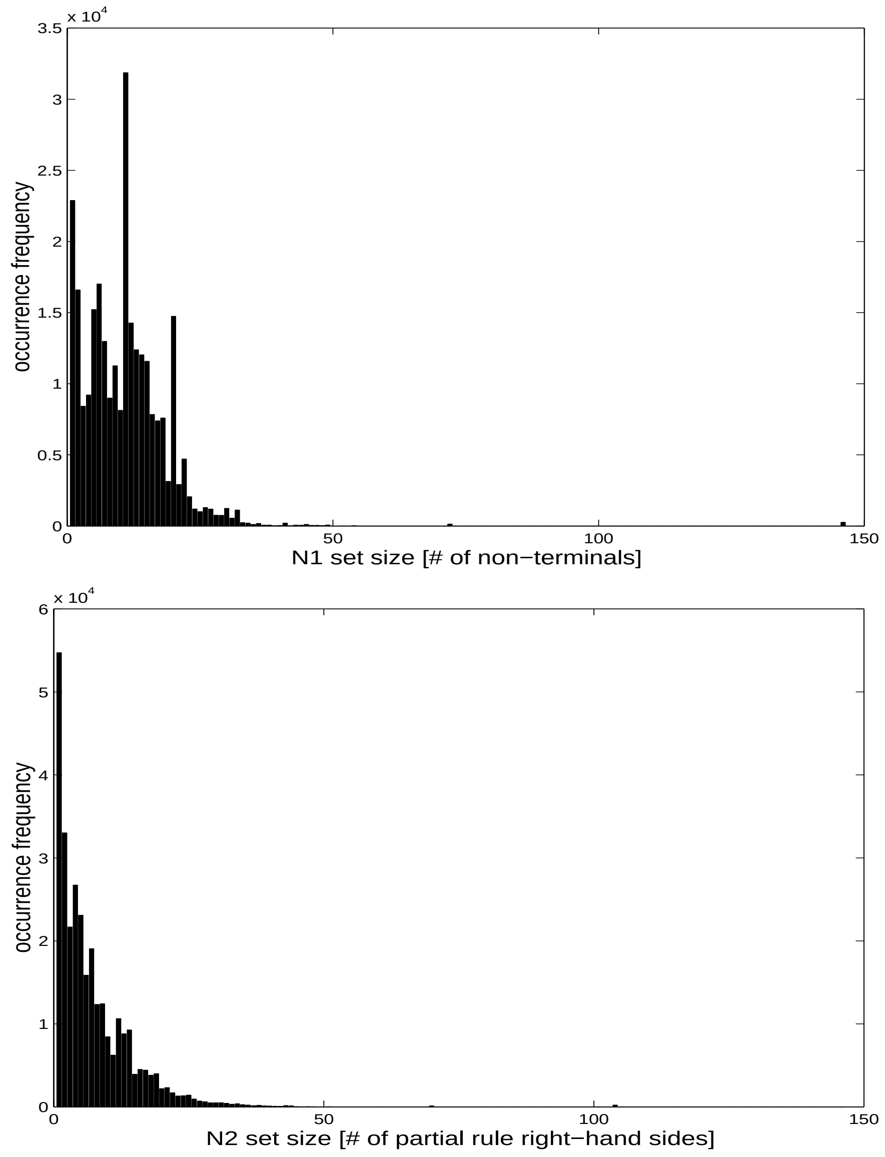


Figure B.4: (a) $N1$ set size distribution and (b) $N2$ set size distribution for the enhanced-CYK algorithm in the particular case of the SUSANNE grammar without unitary rules. Computed for a number of $4,292$ sentences with lengths between 2 to 64 words, extracted from the SUSANNE corpus.

This number is significantly smaller than the number $28,111$ of distinct partial rule right-hand sides in the SUSANNE grammar without unitary-rules. A partial rule right-hand side is represented on 2 bytes which means that 512 bytes are used for storing a set $N2$ in a chart cell. It also turns out that in average 71.18% of the overall (over all the parsed sentences) number of sets $N2$ are empty.

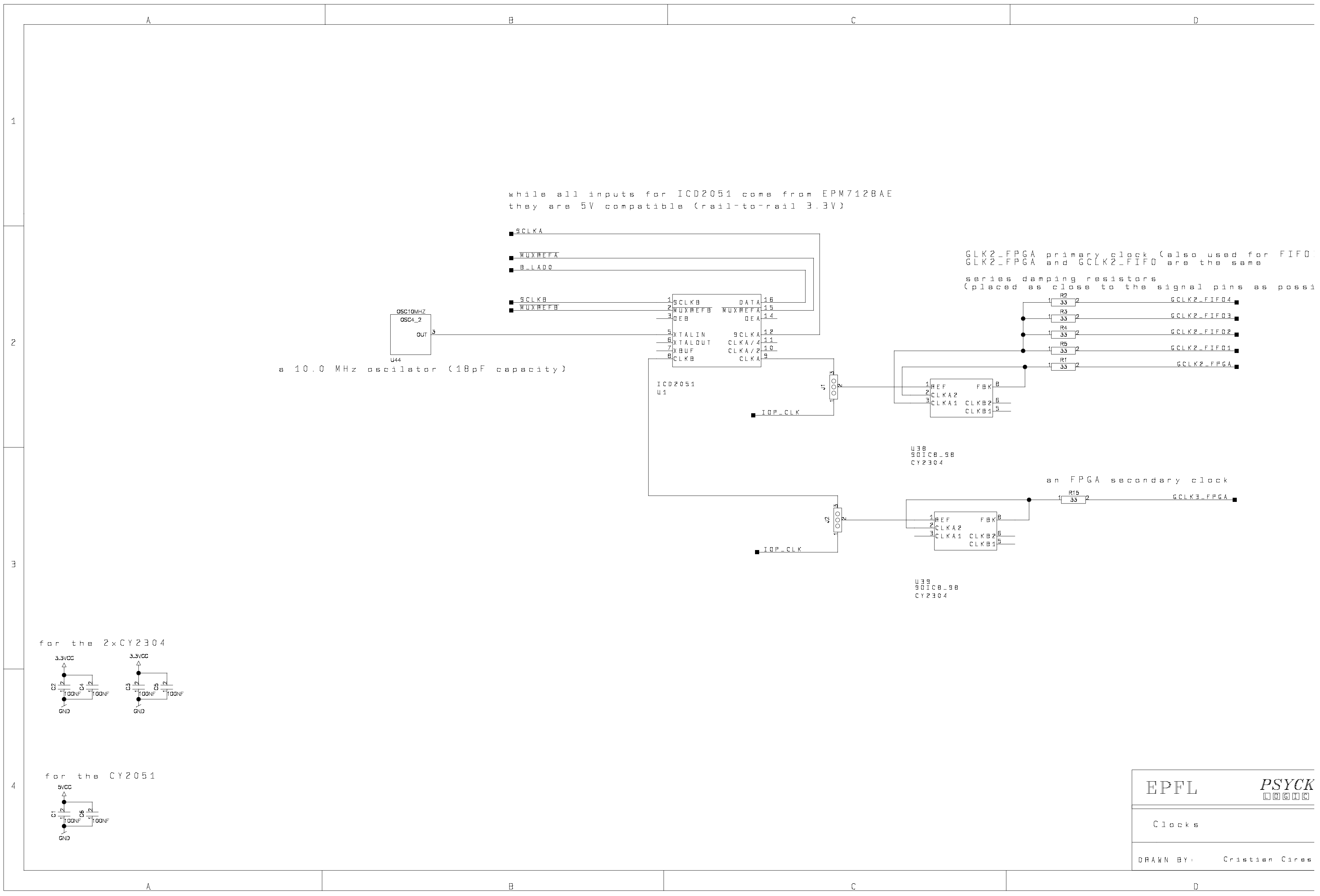
In conclusion if a set size of 128 is used, about 0.095% sentences will fail to parse. If a set size of 256 is used any sentence from the SUSANNE corpus can be parsed.

Appendix C

FPGA expansion board schematics

Address Range	Device	Chip Select
`FFFF,FFFF` `FFF8,0000`	PCI interface board, flash memory 512 KBytes	$\overline{\mathtt{LCS0}}$
`FFF7,FFFF` `F000,0000`	unused	-
`EFFF,FFFF` `B000,0000`	FIFO memory (DMA transfers)	$\overline{\mathtt{LCS1}}$
`AFFF,FFFF` `8000,0000`	unused	-
`7FFF,FFFF` `7000,0000`	unused	-
`6FFF,FFFF` `6000,0000`	chart memories 1 & 2	$\overline{\mathtt{LCS2}}$
`5FFF,FFFF` `5000,0000`	Local bus-to-PCI bus, Memory Mapped	-
`4FFF,FFFF` `4000,0000`	Local bus-to-PCI bus, I/O Mapped	-
`3FFF,FFFF` `3000,0000`	IOP 480 Internal Config. Registers	-
`2FFF,FFFF` `2000,0000`	FPGA configuration, FPGA general-purpose user registers, swapping the chart memories and for programming the programmable clock	$\overline{\mathtt{LCS3}}$
`1000,0024` `1000,0000`	IOP 480 Internal UART	-
`0FFF,FFFF` `0200,0000`	unused	-
`01FF,FFFF` `0000,0000`	SDRAM 32MBytes	-

Table C.1: FPGA-board Memory Map (see [23] for more details)

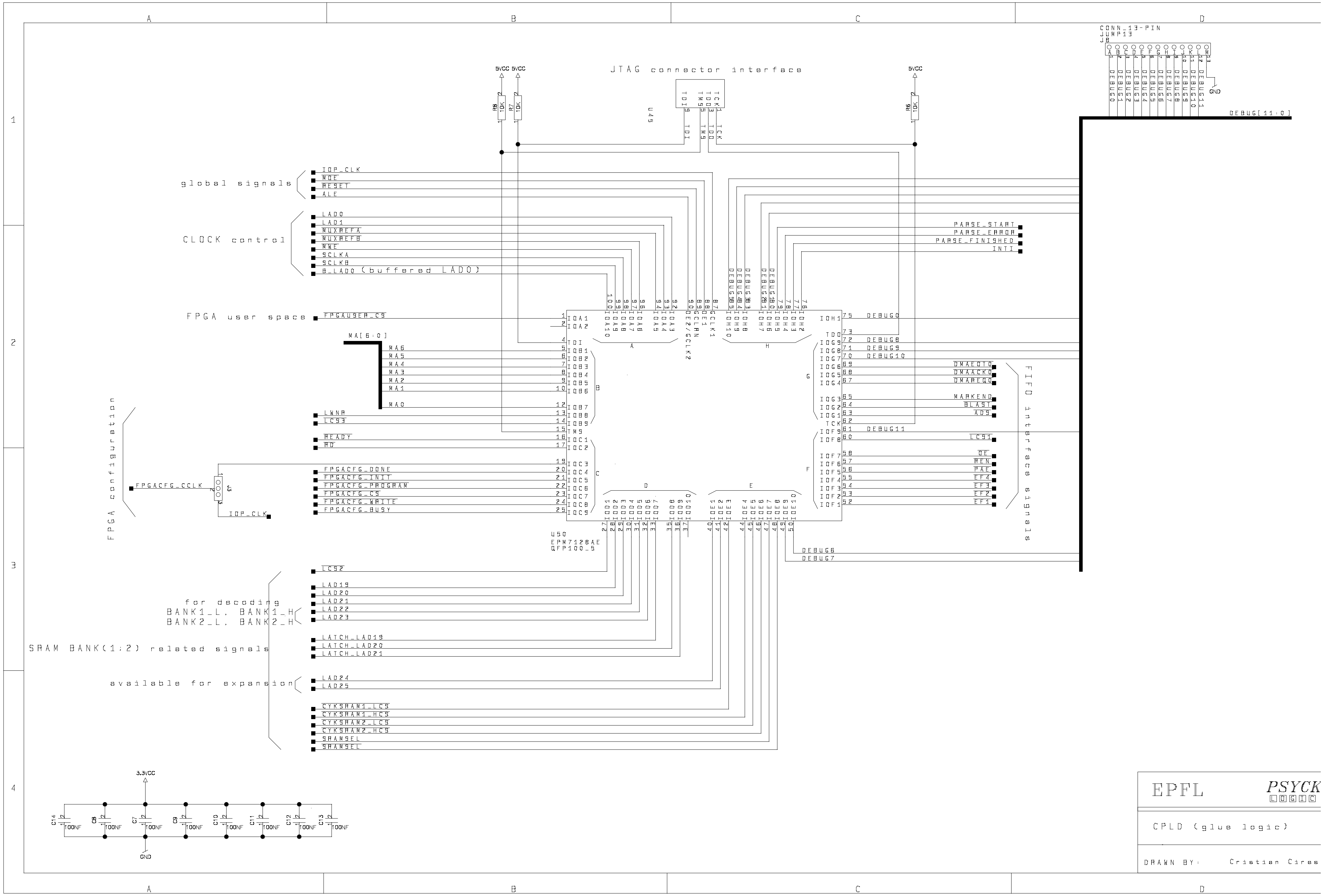


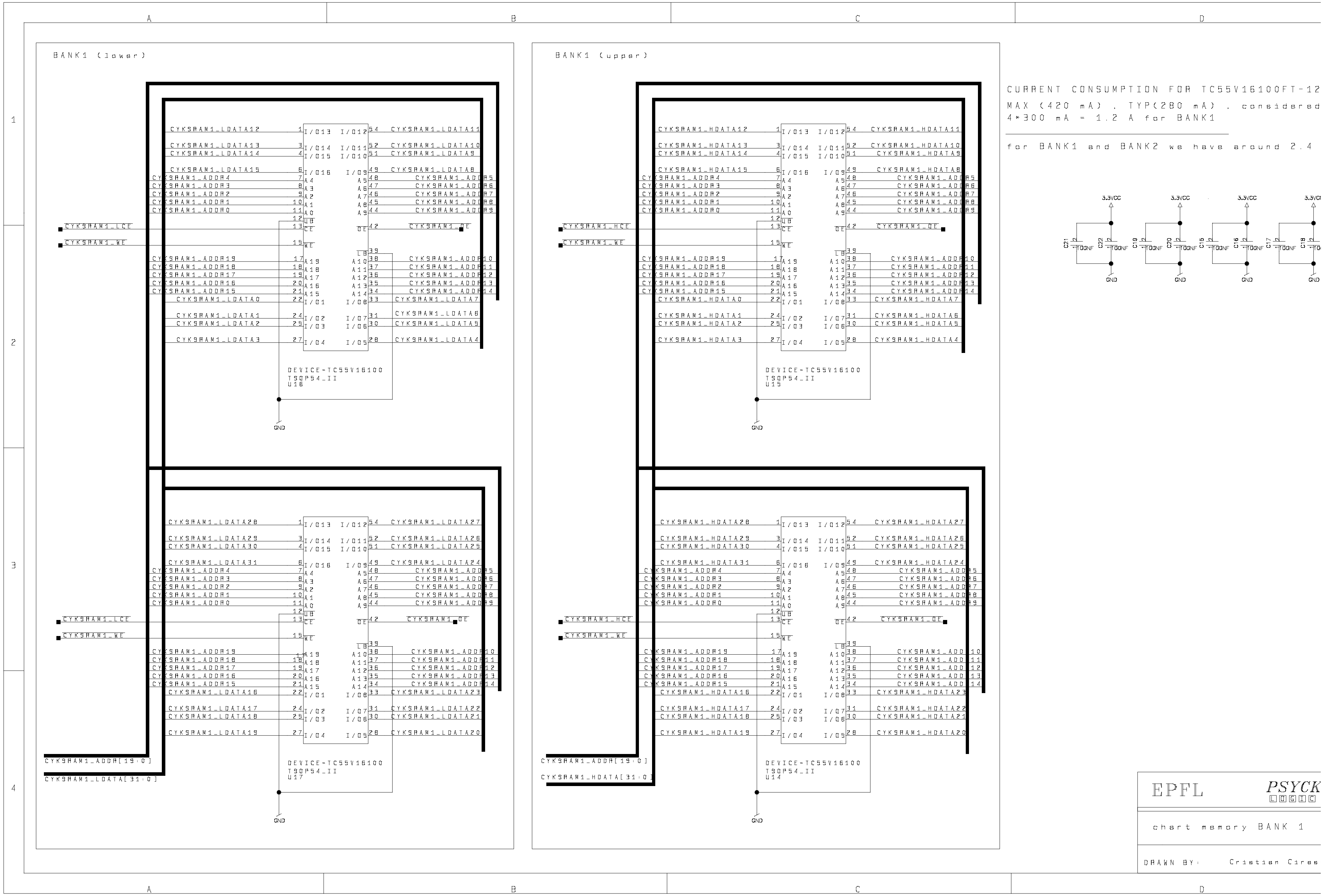
EPFL

PSYCK LOGIC

Clocks

DRAWN BY: Cristian Cires


JTAG connector interface
U45
TCK TDO TMS TDI
R6 R7 R8 10K
5VCC
global signals
IOP_CLK
MOE
RESET
ALE
CLOCK control
LAD0
LAD1
MUXREFA
MUXREFB
MWE
SCLKA
SCLKB
B_LAD0 (buffered LAD0)
FPGA user space
FPGAUSER_CS
MA[6:0]
MA6 MA5 MA4 MA3 MA2 MA1 MA0
LWNR
LCS3
READY
RD
FPGA configuration
FPGACFG_CCLK
J3
FPGACFG_DONE
FPGACFG_INIT
FPGACFG_PROGRAM
FPGACFG_CS
FPGACFG_WRITE
FPGACFG_BUSY
U50
EPM7128AE
QFP100_5
for decoding
BANK1_L, BANK1_H
BANK2_L, BANK2_H
SRAM BANK(1:2) related signals
LCS2
LAD19 LAD20 LAD21 LAD22 LAD23
LATCH_LAD19
LATCH_LAD20
LATCH_LAD21
available for expansion
LAD24
LAD25
CYKSRAM1_LCS
CYKSRAM1_HCS
CYKSRAM2_LCS
CYKSRAM2_HCS
SRAMSEL
SRAMSEL
PARSE_START
PARSE_ERROR
PARSE_FINISHED
INTI
FIFO interface signals
DMAEOT0
DMAACK0
DMAREQ0
MARKEND
BLAST
ADS
LCS1
OE
REN
PAE
EF4 EF3 EF2 EF1
DEBUG0 … DEBUG11
DEBUG[11:0]
CONN_13-PIN
JUMP13
J8
GND
3.3VCC
C7 C8 C9 C10 C11 C12 C13 C14
100NF
EPFL
PSYCK LOGIC
CPLD (glue logic)
DRAWN BY: Cristian Ciresan



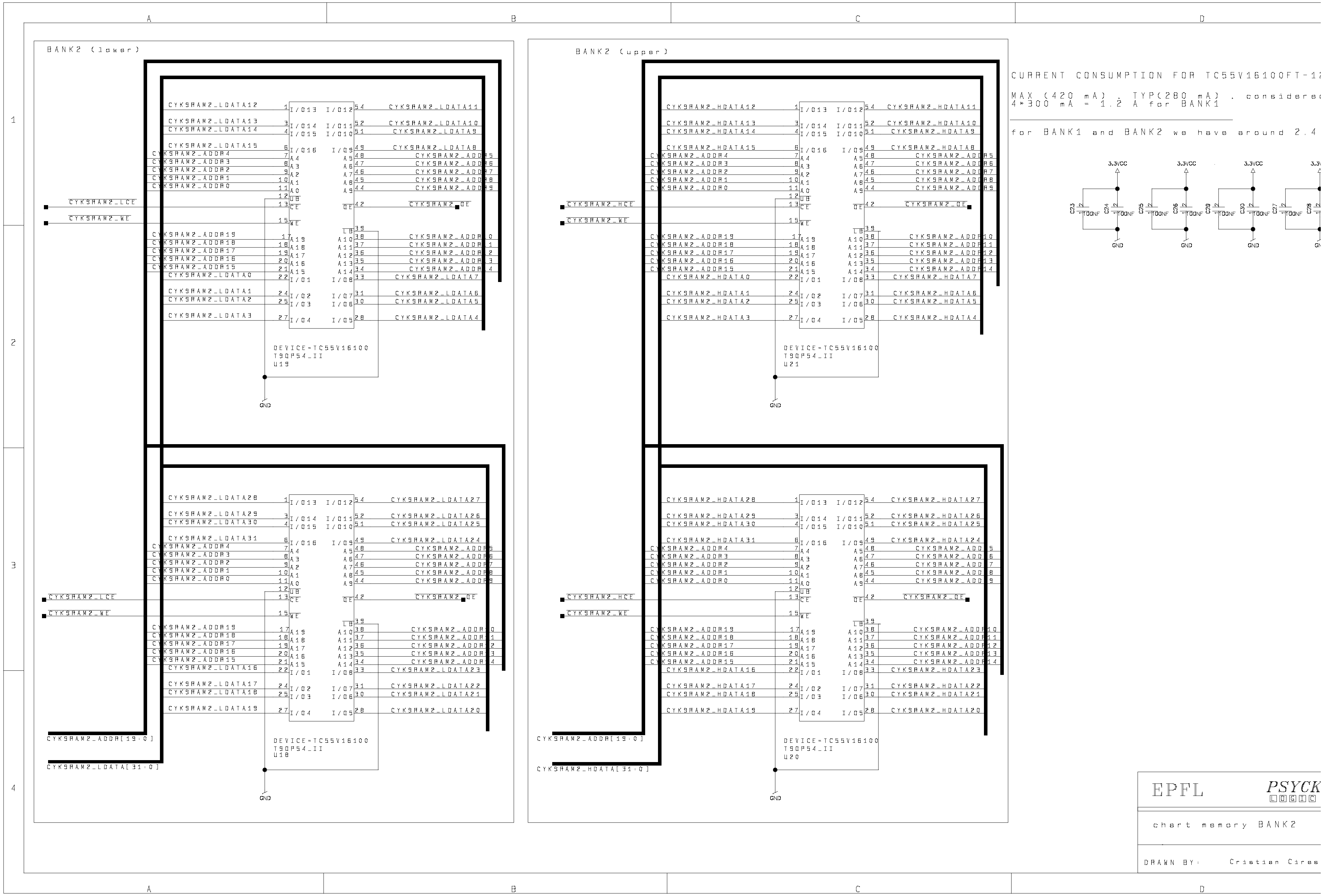
CURRENT CONSUMPTION FOR TC55V16100FT-12
MAX (420 mA), TYP(280 mA), considered
4*300 mA = 1.2 A for BANK1

for BANK1 and BANK2 we have around 2.4

EPFL PSYCK LOGIC

chart memory BANK 1

DRAWN BY: Cristian Cires



CURRENT CONSUMPTION FOR TC55V16100FT-12

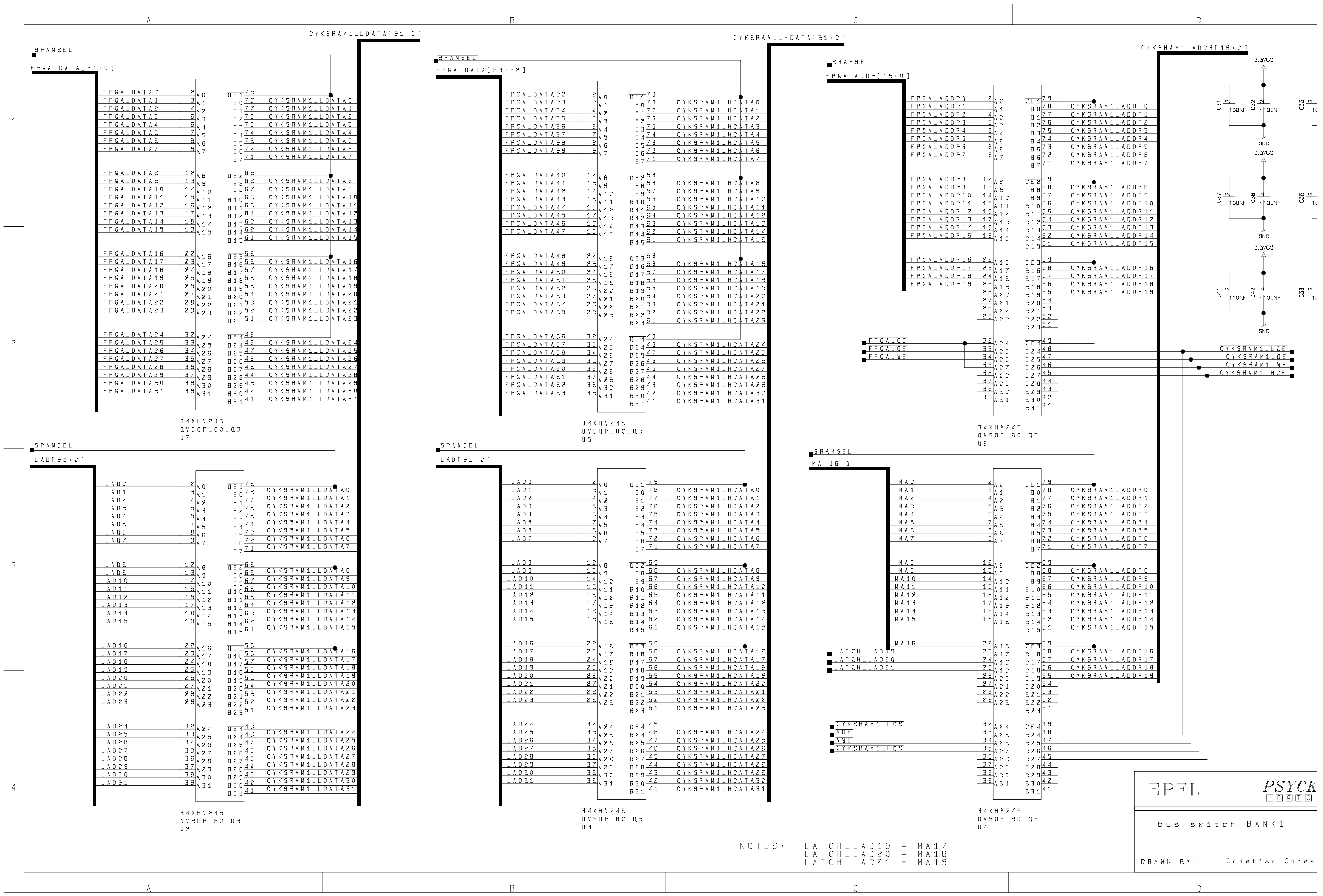
MAX (420 mA) , TYP(280 mA) , considered
4*300 mA = 1.2 A for BANK1

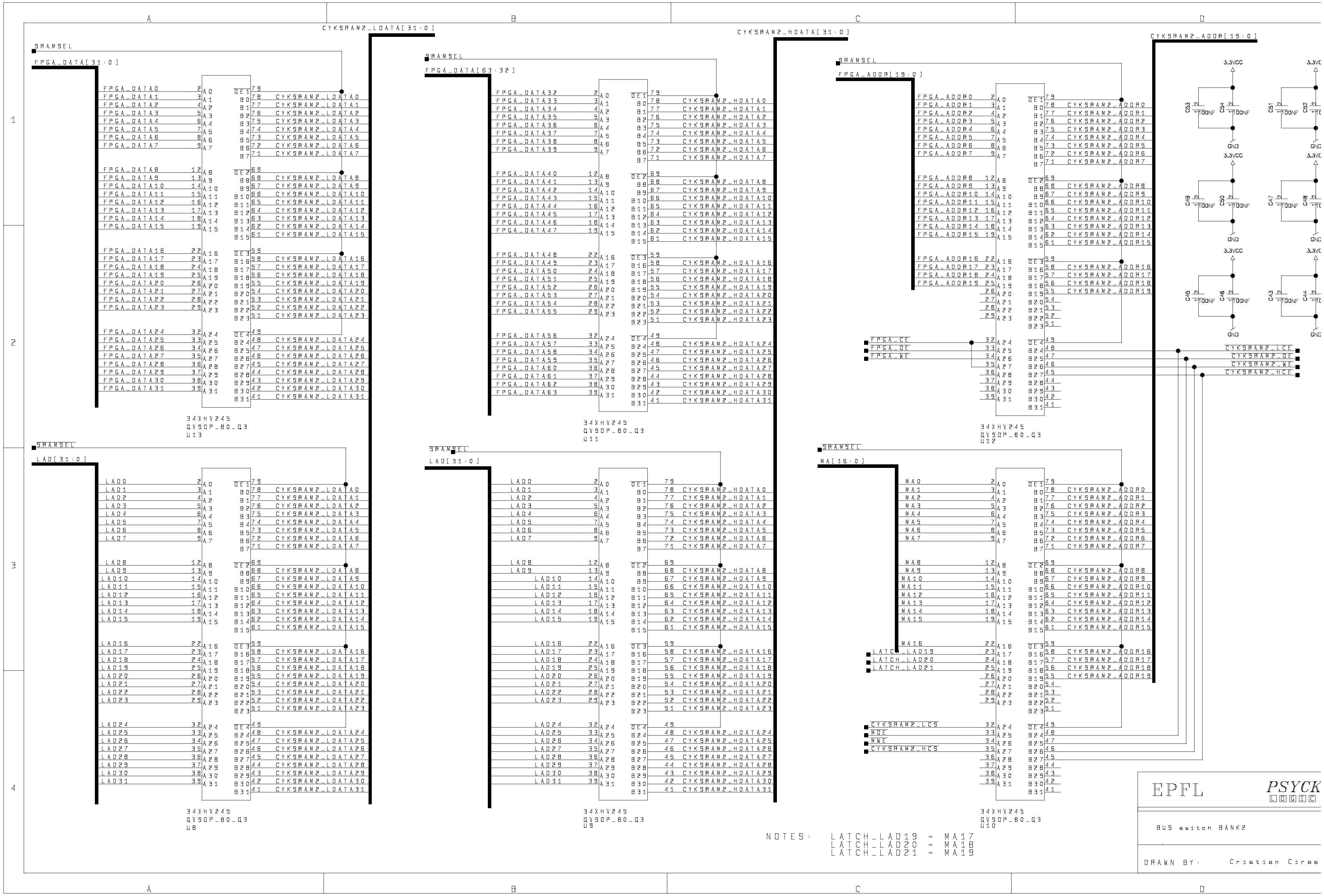
for BANK1 and BANK2 we have around 2.4

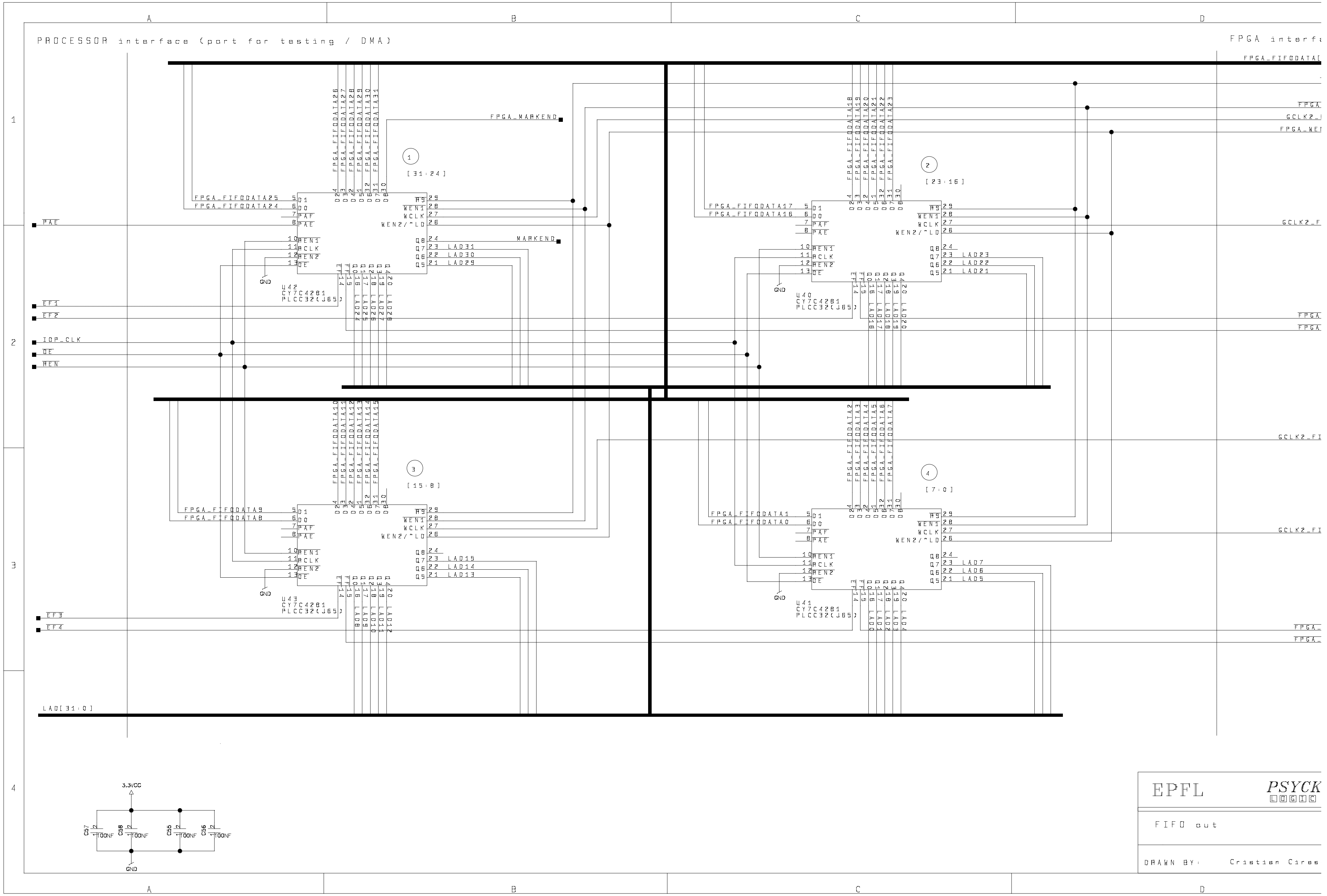
EPFL PSYCK LOGIC

chart memory BANK2

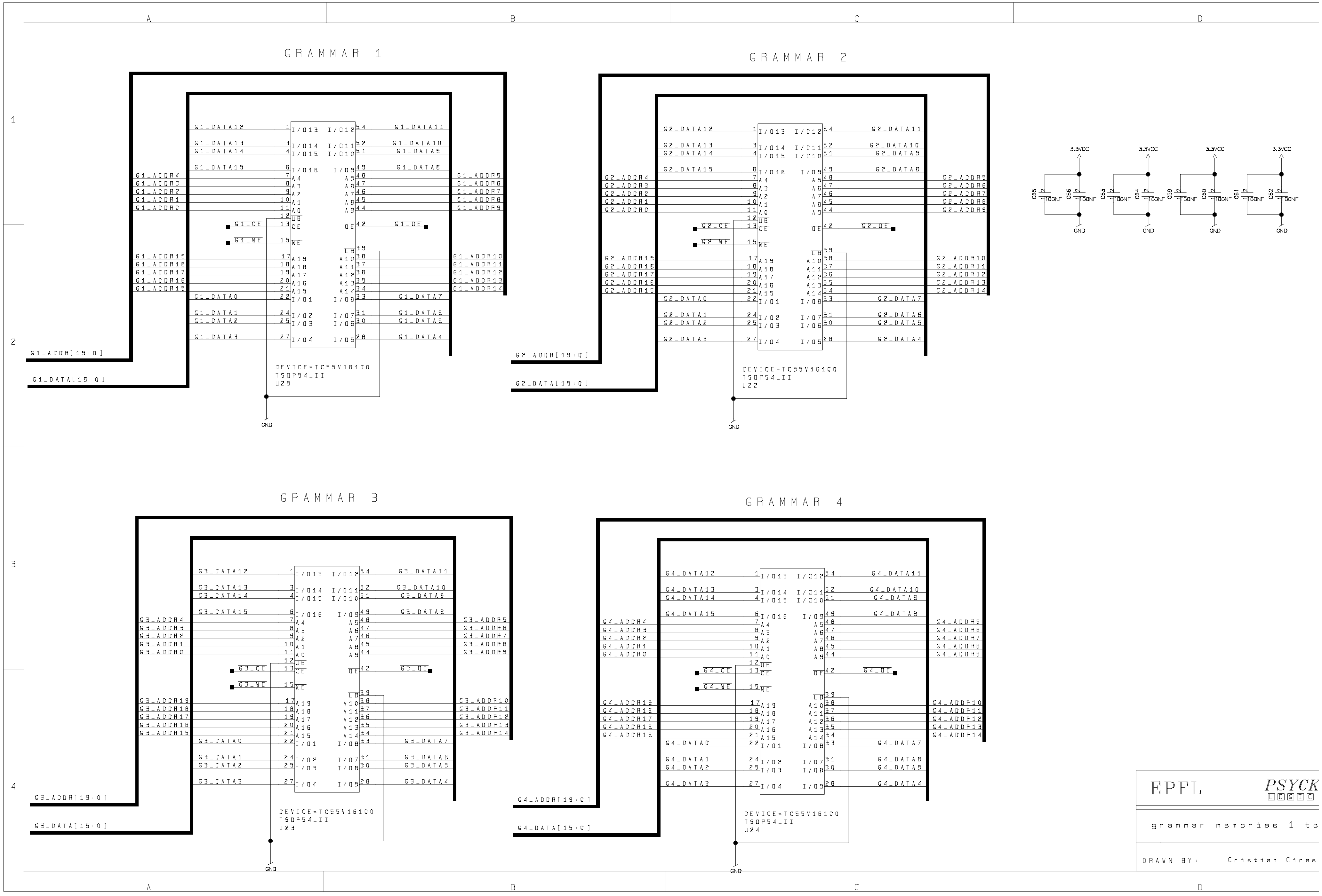
DRAWN BY: Cristian Cires


CYKSRAM1_LDATA[31:0]
CYKSRAM1_HDATA[31:0]
CYKSRAM1_ADDR[19:0]
SRAMSEL
FPGA_DATA[31:0]
FPGA_DATA[63:32]
FPGA_ADDR[19:0]
LAD[31:0]
MA[16:0]
FPGA_CE
FPGA_OE
FPGA_WE
CYKSRAM1_LCE
CYKSRAM1_OE
CYKSRAM1_WE
CYKSRAM1_HCE
LATCH_LAD19
LATCH_LAD20
LATCH_LAD21
CYKSRAM1_LCS
MOE
MWE
CYKSRAM1_HCS
34XHV245
QVSOP_80_Q3
U7
U5
U6
U2
U3
U4
3.3VCC
GND
100NF
C31
C32
C33
C37
C38
C35
C41
C42
C39
NOTES: LATCH_LAD19 = MA17
LATCH_LAD20 = MA18
LATCH_LAD21 = MA19
EPFL
PSYCK
LOGIC
bus switch BANK1
DRAWN BY: Cristian Cires


CYKSRAM2_LDATA[31:0]
CYKSRAM2_HDATA[31:0]
CYKSRAM2_ADDR[19:0]
SRAMSEL
FPGA_DATA[31:0]
FPGA_DATA[63:32]
FPGA_ADDR[19:0]
LAD[31:0]
MA[16:0]
34XHV245
QVSOP_80_Q3
U13
U11
U12
U8
U9
U10
FPGA_CE
FPGA_OE
FPGA_WE
CYKSRAM2_LCE
CYKSRAM2_OE
CYKSRAM2_WE
CYKSRAM2_HCE
LATCH_LAD19
LATCH_LAD20
LATCH_LAD21
CYKSRAM2_LCS
MOE
MWE
CYKSRAM2_HCS
3.3VCC
GND
100NF
C53
C54
C51
C52
C49
C50
C47
C48
C45
C46
C43
C44
NOTES: LATCH_LAD19 = MA17
LATCH_LAD20 = MA18
LATCH_LAD21 = MA19
EPFL
PSYCK
LOGIC
BUS switch BANK2
DRAWN BY: Cristian Ciress



PROCESSOR interface (port for testing / DMA)
FPGA interfa
FPGA_FIFODATA[
FPGA_
GCLK2_
FPGA_WE
GCLK2_F
GCLK2_FI
FPGA_MARKEND
MARKEND
PAE
EF1
EF2
IOP_CLK
OE
REN
EF3
EF4
LAD[31:0]
[31:24]
[23:16]
[15:8]
[7:0]
U42
U40
U43
U41
CY7C4281
PLCC32(J65)
FPGA_FIFODATA25
FPGA_FIFODATA24
FPGA_FIFODATA17
FPGA_FIFODATA16
FPGA_FIFODATA9
FPGA_FIFODATA8
FPGA_FIFODATA1
FPGA_FIFODATA0
RS
WEN1
WCLK
WEN2/~LD
REN1
RCLK
REN2
OE
PAF
PAE
EF
FF
GND
3.3VCC
C57
C58
C55
C56
100NF
EPFL
PSYCK LOGIC
FIFO out
DRAWN BY:
Cristian Cires

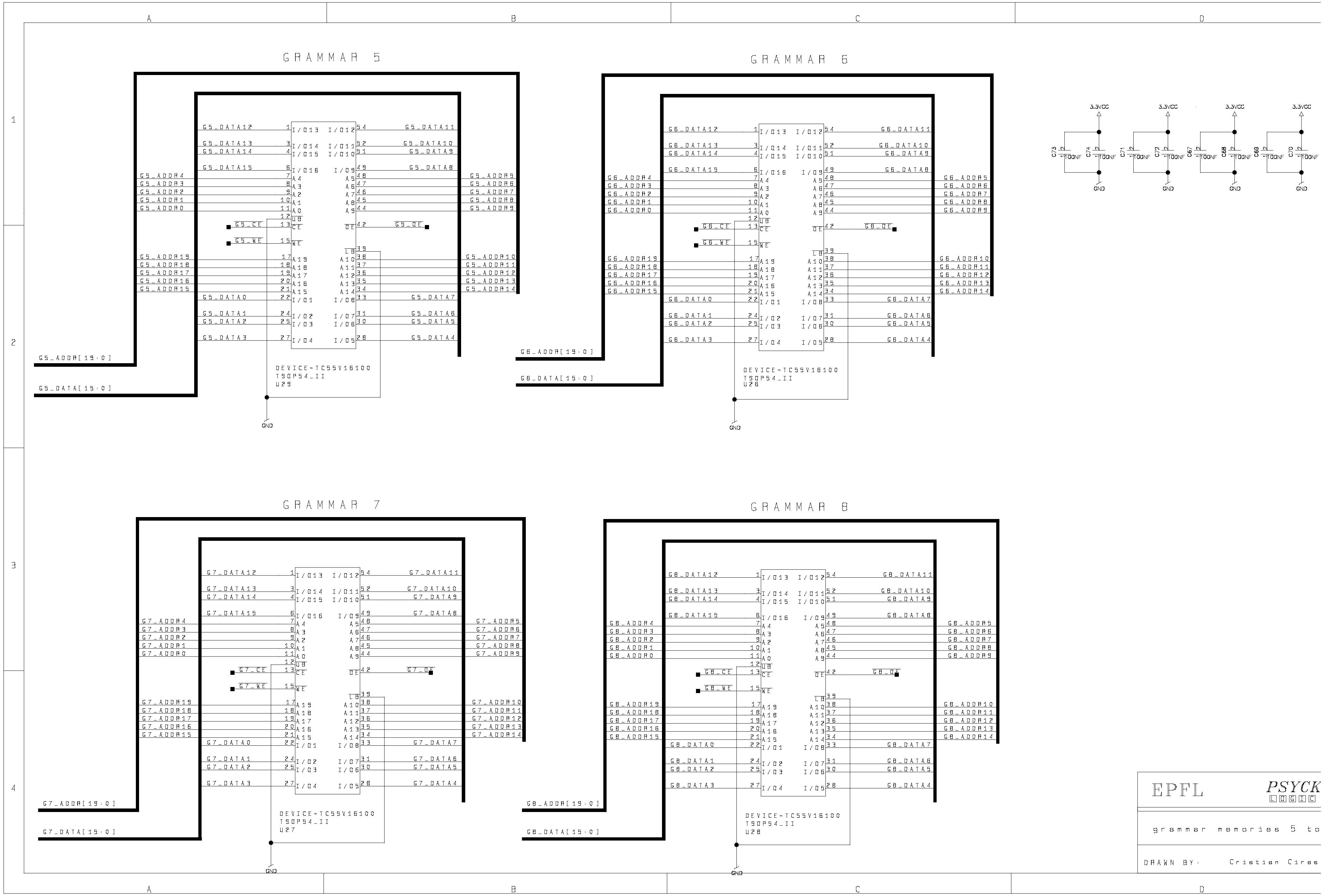


EPFL

PSYCK LOGIC

grammar memories 1 to

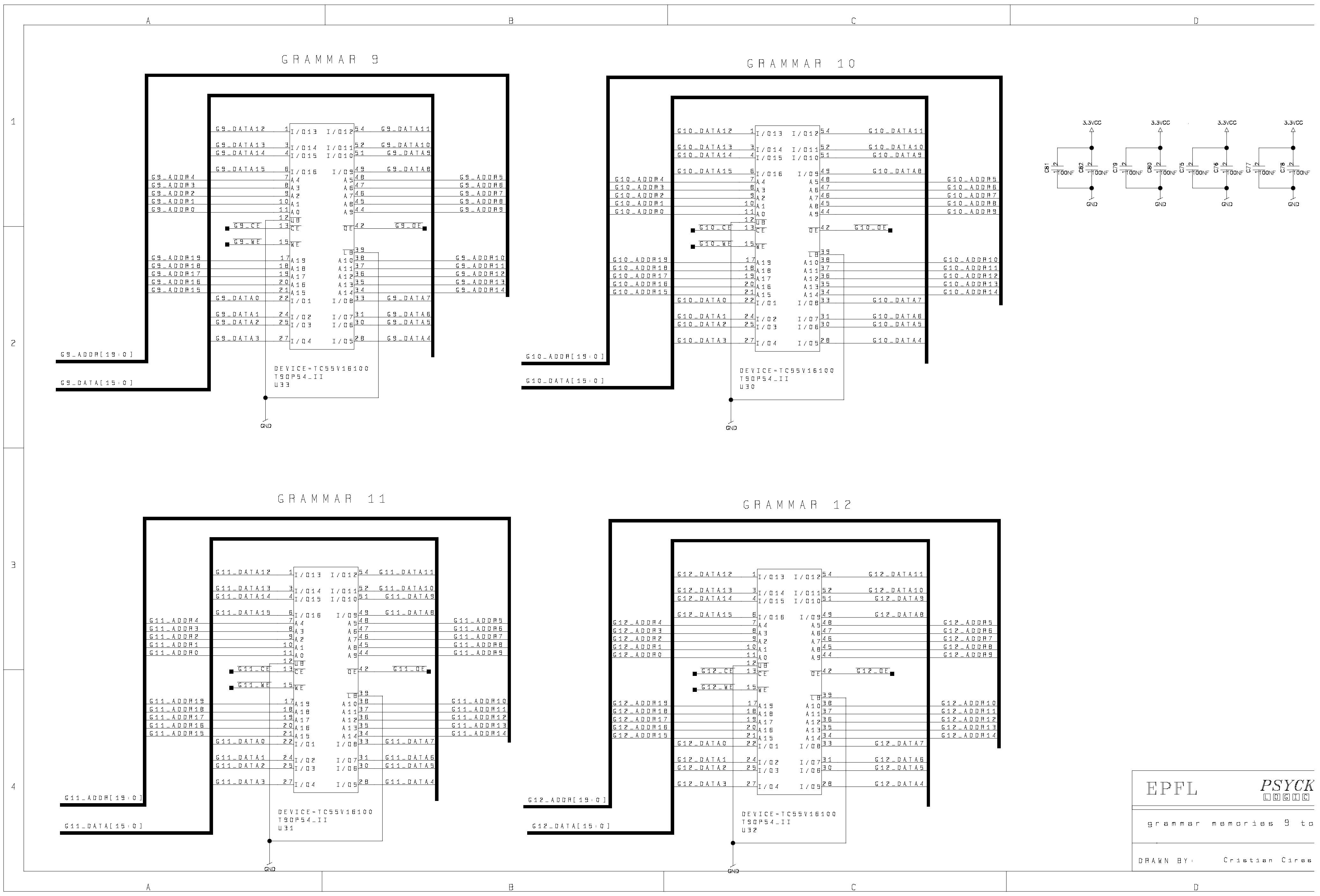
DRAWN BY: Cristian Cires

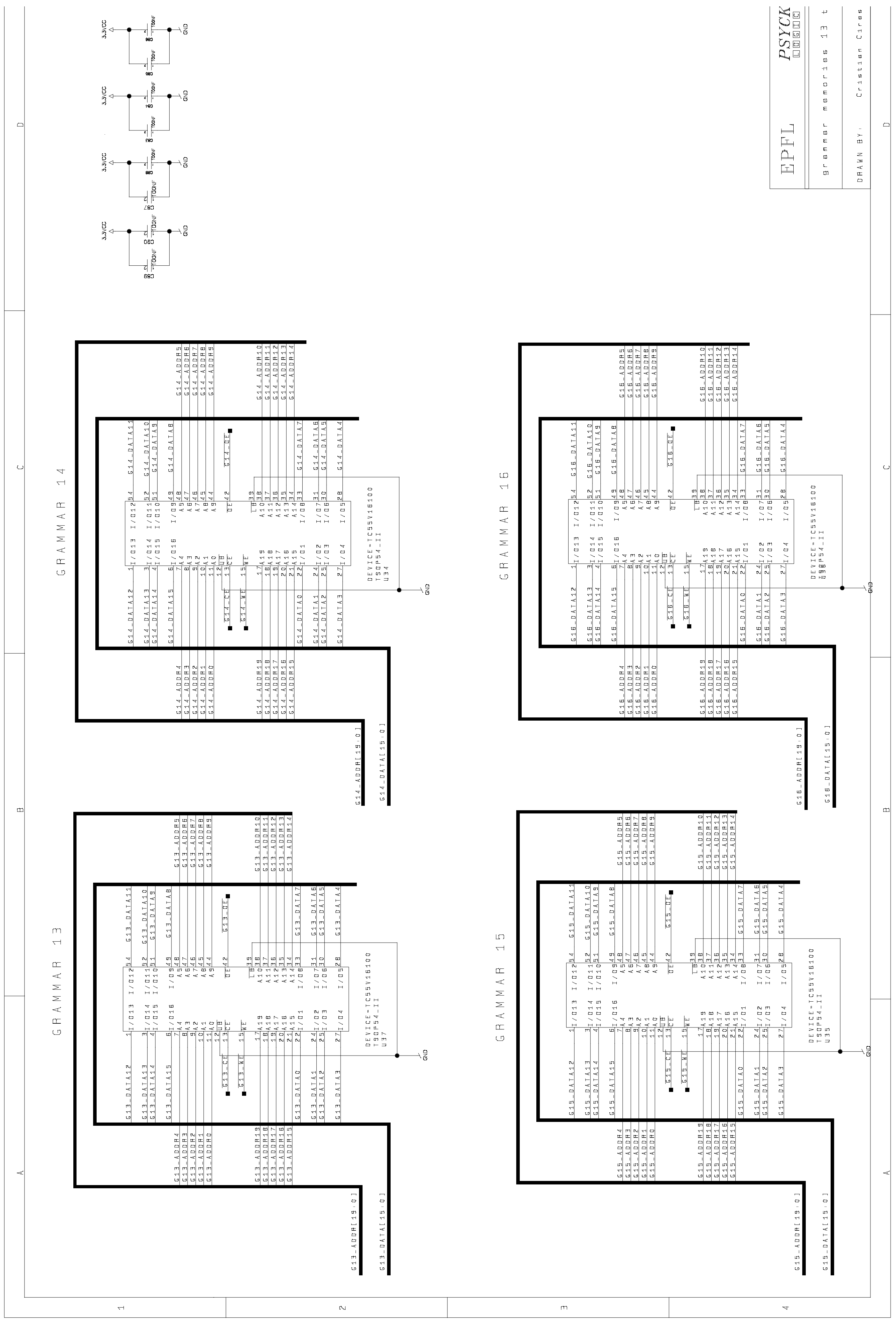


EPFL PSYCK LOGIC

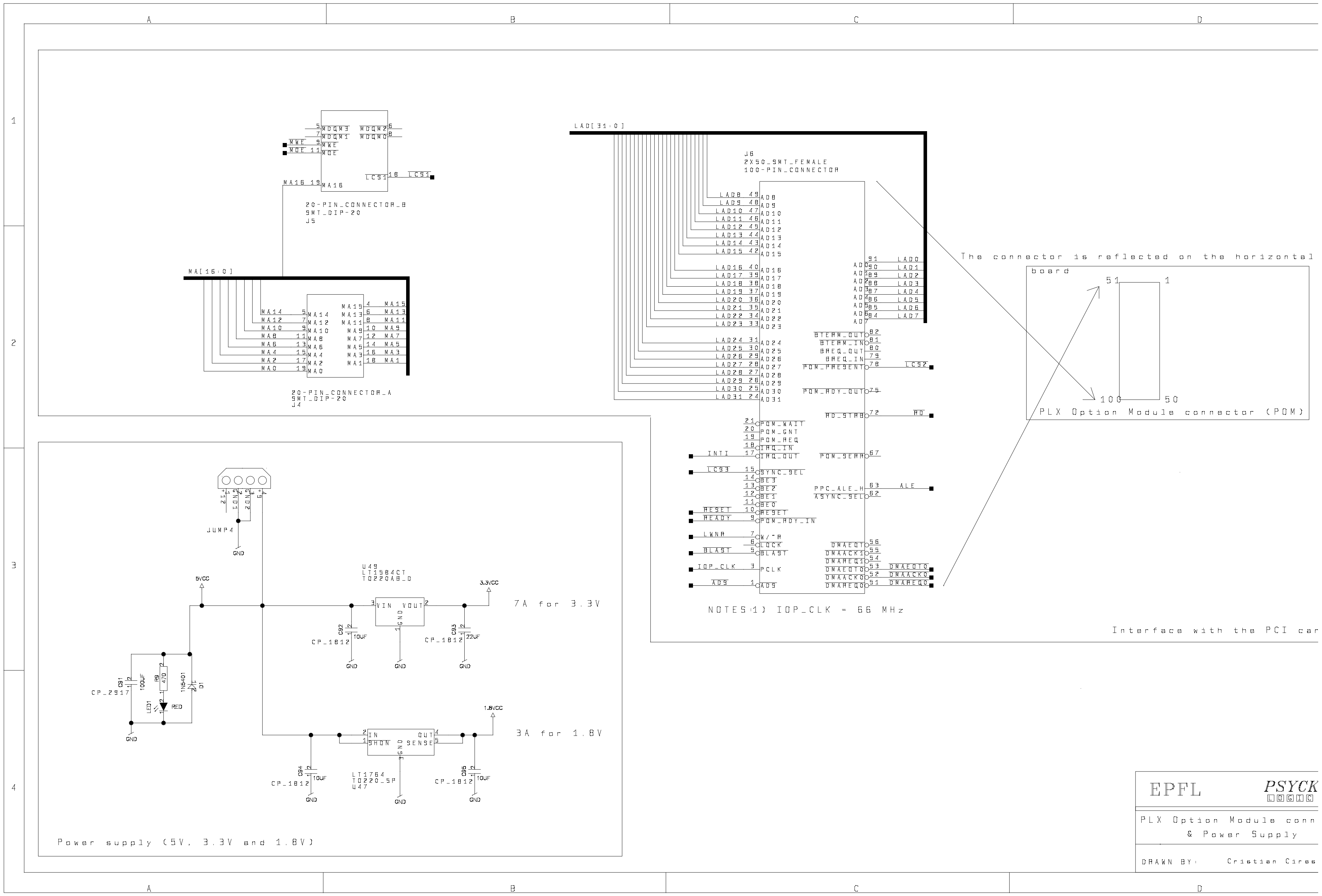
grammar memories 5 to

DRAWN BY: Cristian Cires

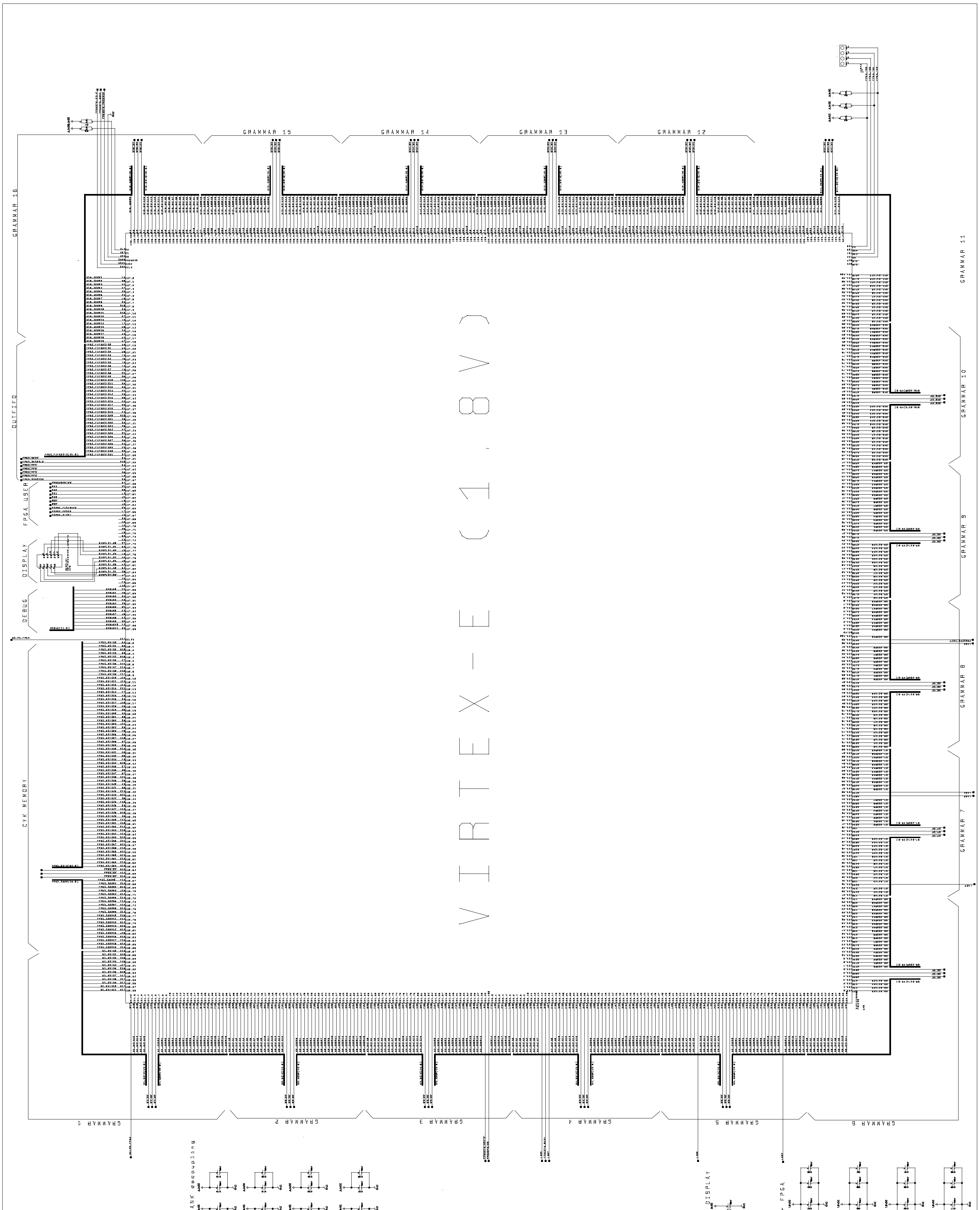

GRAMMAR 9
GRAMMAR 10
GRAMMAR 11
GRAMMAR 12
DEVICE=TC55V16100
TSOP54_II
U33
U30
U31
U32
G9_ADDR[19:0]
G9_DATA[15:0]
G10_ADDR[19:0]
G10_DATA[15:0]
G11_ADDR[19:0]
G11_DATA[15:0]
G12_ADDR[19:0]
G12_DATA[15:0]
3.3VCC
GND
C81
C82
C79
C80
C75
C76
C77
C78
100NF
EPFL
PSYCK
LOGIC
grammar memories 9 to
DRAWN BY:
Cristian Cires



EPFL
PSYCK LOGIC
grammar memories 13 t
DRAWN BY: Cristian Ciras
GRAMMAR 13
GRAMMAR 14
GRAMMAR 15
GRAMMAR 16
DEVICE=TC55V16100
TSOP54-II
U37
U34
U35
U36
G13_ADDR[19:0]
G13_DATA[15:0]
G14_ADDR[19:0]
G14_DATA[15:0]
G15_ADDR[19:0]
G15_DATA[15:0]
G16_ADDR[19:0]
G16_DATA[15:0]
3.3VCC
GND
100NF



20-PIN_CONNECTOR_B
SMT_DIP-20
J5
20-PIN_CONNECTOR_A
SMT_DIP-20
J4
MA[16:0]
LAD[31:0]
J6
2X50_SMT_FEMALE
100-PIN_CONNECTOR
The connector is reflected on the horizontal
board
PLX Option Module connector (POM)
NOTES:1) IOP_CLK = 66 MHz
Interface with the PCI car
JUMP4
U49
LT1584CT
TO220AB_D
7A for 3.3V
LT1764
TO220_5P
U47
3A for 1.8V
5VCC
3.3VCC
1.8VCC
GND
Power supply (5V, 3.3V and 1.8V)
EPFL
PSYCK LOGIC
PLX Option Module conn
& Power Supply
DRAWN BY: Cristian Cires


VIRTEX-E (1.8V)
GRAMMAR 15
GRAMMAR 14
GRAMMAR 13
GRAMMAR 12
GRAMMAR 16
GRAMMAR 11
GRAMMAR 10
GRAMMAR 9
GRAMMAR 8
GRAMMAR 7
OUTFIFO
FPGA USER
DISPLAY
DEBUG
CYK MEMORY
GRAMMAR 1
GRAMMAR 2
GRAMMAR 3
GRAMMAR 4
GRAMMAR 5
GRAMMAR 6
For VBANK decoupling
For DISPLAY
For FPGA

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
1	CCLK	C31	FPGACFG_CCLK	26	IO0_9	F17	FPGA_DATA9
2	DONE	AM31	FPGACFG_DONE	27	IO0_10	J12	FPGA_DATA10
3	DXN	AJ5	(not used)	28	IO0_11	J13	FPGA_DATA11
4	DXP	AL5	(not used)	29	IO0_12	J14	FPGA_DATA12
5	GCLK0	AH18	(not used)	30	IO0_13	K11	FPGA_DATA13
6	GCLK1	AL19	(not used)	31	IO0_14	F7	FPGA_DATA14
7	GCLK2	D17	GCLK2_FPGA	32	IO0_15	H9	FPGA_DATA15
8	GCLK3	E17	GCLK3_FPGA	33	IO0_16	C5	FPGA_DATA16
9	M0	AK4	pull-down	34	IO0_17	J10	FPGA_DATA17
10	M1	AG7	pull-up	35	IO0_18	E6	FPGA_DATA18
11	M2	AL3	pull-up	36	IO0_19	D6	FPGA_DATA19
12	PROGRAM	AG28	/FPGACFG_PROGRAM	37	IO0_20	A4	FPGA_DATA20
13	TCK	D5	FPGA_TCK	38	IO0_21	G8	FPGA_DATA21
14	TDI	C30	FPGA_TDI	39	IO0_22	C6	FPGA_DATA22
15	TDO	K26	FPGA_TDO	40	IO0_23	J11	FPGA_DATA23
16	TMS	C4	FPGA_TMS	41	IO0_24	G9	FPGA_DATA24
17	IO0_0	B4	FPGA_DATA0	42	IO0_25	F8	FPGA_DATA25
18	IO0_1	B9	FPGA_DATA1	43	IO0_26	A5	FPGA_DATA26
19	IO0_2	B10	FPGA_DATA2	44	IO0_27	H10	FPGA_DATA27
20	IO0_3	D9	FPGA_DATA3	45	IO0_28	D7	FPGA_DATA28
21	IO0_4	D16	FPGA_DATA4	46	IO0_29	B5	FPGA_DATA29
22	IO0_5	E7	FPGA_DATA5	47	IO0_30	K12	FPGA_DATA30
23	IO0_6	E11	FPGA_DATA6	48	IO0_31	E8	FPGA_DATA31
24	IO0_7	E13	FPGA_DATA7	49	IO0_32	B6	FPGA_DATA32
25	IO0_8	E16	FPGA_DATA8	50	IO0_33	F9	FPGA_DATA33

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 1 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
51	IO0_34	G10	FPGA_DATA34	76	IO0_59	D11	FPGA_DATA59
52	IO0_35	C7	FPGA_DATA35	77	IO0_60	G13	FPGA_DATA60
53	IO0_36	D8	FPGA_DATA36	78	IO0_61	C12	FPGA_DATA61
54	IO0_37	B7	FPGA_DATA37	79	IO0_62	K15	FPGA_DATA62
55	IO0_38	H11	FPGA_DATA38	80	IO0_63	A12	FPGA_DATA63
56	IO0_39	C8	FPGA_DATA39	81	IO0_64	B12	/FPGA_CE
57	IO0_40	E9	FPGA_DATA40	82	IO0_65	H14	/FPGA_OE
58	IO0_41	B8	FPGA_DATA41	83	IO0_66	D12	/FPGA_WE
59	IO0_42	K13	FPGA_DATA42	84	IO0_67	F13	FPGA_ADDR0
60	IO0_43	G11	FPGA_DATA43	85	IO0_68	A13	FPGA_ADDR1
61	IO0_44	A8	FPGA_DATA44	86	IO0_69	B13	FPGA_ADDR2
62	IO0_45	F10	FPGA_DATA45	87	IO0_70	J15	FPGA_ADDR3
63	IO0_46	C9	FPGA_DATA46	88	IO0_71	G14	FPGA_ADDR4
64	IO0_47	H12	FPGA_DATA47	89	IO0_72	C13	FPGA_ADDR5
65	IO0_48	D10	FPGA_DATA48	90	IO0_73	F14	FPGA_ADDR6
66	IO0_49	A9	FPGA_DATA49	91	IO0_74	H15	FPGA_ADDR7
67	IO0_50	F11	FPGA_DATA50	92	IO0_75	D13	FPGA_ADDR8
68	IO0_51	A10	FPGA_DATA51	93	IO0_76	A14	FPGA_ADDR9
69	IO0_52	K14	FPGA_DATA52	94	IO0_77	K16	FPGA_ADDR10
70	IO0_53	C10	FPGA_DATA53	95	IO0_78	E14	FPGA_ADDR11
71	IO0_54	H13	FPGA_DATA54	96	IO0_79	B14	FPGA_ADDR12
72	IO0_55	G12	FPGA_DATA55	97	IO0_80	G15	FPGA_ADDR13
73	IO0_56	A11	FPGA_DATA56	98	IO0_81	D14	FPGA_ADDR14
74	IO0_57	B11	FPGA_DATA57	99	IO0_82	J16	FPGA_ADDR15
75	IO0_58	E12	FPGA_DATA58	100	IO0_83	D15	FPGA_ADDR16

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 2 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
101	IO0_84	F15	FPGA_ADDR17	126	IO1_10	D28	G1_ADDR3
102	IO0_85	B15	FPGA_ADDR18	127	IO1_11	D29	G1_ADDR4
103	IO0_86	A15	FPGA_ADDR19	128	IO1_12	G23	G1_ADDR5
104	IO0_87	E15	G1_DATA0	129	IO1_13	J23	G1_ADDR6
105	IO0_88	G16	G1_DATA1	130	IO1_14	J18	G1_ADDR7
106	IO0_89	A16	G1_DATA2	131	IO1_15	G18	G1_ADDR8
107	IO0_90	F16	G1_DATA3	132	IO1_16	C18	G1_ADDR9
108	IO0_91	J17	G1_DATA4	133	IO1_17	H18	G1_ADDR10
109	IO0_92	C16	G1_DATA5	134	IO1_18	F18	G1_ADDR11
110	IO0_93	B16	G1_DATA6	135	IO1_19	B19	G1_ADDR12
111	IO0_94	H17	G1_DATA7	136	IO1_20	A19	G1_ADDR13
112	IO0_95	A17	G1_DATA8	137	IO1_21	K19	G1_ADDR14
113	IO0_96	G17	G1_DATA9	138	IO1_22	C19	G1_ADDR15
114	IO0_97	B17	G1_DATA10	139	IO1_23	F19	G1_ADDR16
115	IO0_98	C17	G1_DATA11	140	IO1_24	E19	G1_ADDR17
116	IO1_0	A18	G1_DATA12	141	IO1_25	G19	G1_ADDR18
117	IO1_1	B18	G1_DATA13	142	IO1_26	J19	G1_ADDR19
118	IO1_2	B24	G1_DATA14	143	IO1_27	A20	G2_DATA0
119	IO1_3	B25	G1_DATA15	144	IO1_28	G20	G2_DATA1
120	IO1_4	E22	/G1_CE	145	IO1_29	B20	G2_DATA2
121	IO1_5	E23	/G1_WE	146	IO1_30	F20	G2_DATA3
122	IO1_6	D18	/G1_OE	147	IO1_31	D20	G2_DATA4
123	IO1_7	D19	G1_ADDR0	148	IO1_32	E20	G2_DATA5
124	IO1_8	D25	G1_ADDR1	149	IO1_33	H20	G2_DATA6
125	IO1_9	D26	G1_ADDR2	150	IO1_34	A21	G2_DATA7

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 3 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
151	IO1_35	E21	G2_DATA8	176	IO1_60	D24	G2_ADDR14
152	IO1_36	J20	G2_DATA9	177	IO1_61	A25	G2_ADDR15
153	IO1_37	D21	G2_DATA10	178	IO1_62	E24	G2_ADDR16
154	IO1_38	K20	G2_DATA11	179	IO1_63	A26	G2_ADDR17
155	IO1_39	B21	G2_DATA12	180	IO1_64	C25	G2_ADDR18
156	IO1_40	H21	G2_DATA13	181	IO1_65	F24	G2_ADDR19
157	IO1_41	G21	G2_DATA14	182	IO1_66	B26	G3_DATA0
158	IO1_42	F21	G2_DATA15	183	IO1_67	K23	G3_DATA1
159	IO1_43	A22	/G2_CE	184	IO1_68	F25	G3_DATA2
160	IO1_44	B22	/G2_WE	185	IO1_69	C26	G3_DATA3
161	IO1_45	J21	/G2_OE	186	IO1_70	H24	G3_DATA4
162	IO1_46	C22	G2_ADDR0	187	IO1_71	G24	G3_DATA5
163	IO1_47	D22	G2_ADDR1	188	IO1_72	A27	G3_DATA6
164	IO1_48	G22	G2_ADDR2	189	IO1_73	B27	G3_DATA7
165	IO1_49	K21	G2_ADDR3	190	IO1_74	G25	G3_DATA8
166	IO1_50	A23	G2_ADDR4	191	IO1_75	E26	G3_DATA9
167	IO1_51	F22	G2_ADDR5	192	IO1_76	C27	G3_DATA10
168	IO1_52	B23	G2_ADDR6	193	IO1_77	J24	G3_DATA11
169	IO1_53	C23	G2_ADDR7	194	IO1_78	B28	G3_DATA12
170	IO1_54	H22	G2_ADDR8	195	IO1_79	K24	G3_DATA13
171	IO1_55	D23	G2_ADDR9	196	IO1_80	H25	G3_DATA14
172	IO1_56	K22	G2_ADDR10	197	IO1_81	D27	G3_DATA15
173	IO1_57	A24	G2_ADDR11	198	IO1_82	F26	/G3_CE
174	IO1_58	J22	G2_ADDR12	199	IO1_83	G26	/G3_WE
175	IO1_59	H23	G2_ADDR13	200	IO1_84	C28	/G3_OE

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 4 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
201	IO1_85	E27	G3_ADDR0	226	IO2_9	P25	G4_DATA3
202	IO1_86	J25	G3_ADDR1	227	IO2_10	U26	G4_DATA4
203	IO1_87	A30	G3_ADDR2	228	IO2_11	U30	G4_DATA5
204	IO1_88	H26	G3_ADDR3	229	IO2_12	U32	G4_DATA6
205	IO1_89	G27	G3_ADDR4	230	IO2_13	U34	G4_DATA7
206	IO1_90	B29	G3_ADDR5	231	IO2_14	M30	LAD5
207	IO1_91	F27	G3_ADDR6	232	IO2_15	D32	FPGACFG_BUSY
208	IO1_92	C29	G3_ADDR7	233	IO2_16	J27	LAD7
209	IO1_93	E28	G3_ADDR8	234	IO2_17	E31	G4_DATA8
210	IO1_94	F28	G3_ADDR9	235	IO2_18	F30	G4_DATA9
211	IO1_95	L25	G3_ADDR10	236	IO2_19	G29	G4_DATA10
212	IO1_96	B30	G3_ADDR11	237	IO2_20	F32	G4_DATA11
213	IO1_97	B31	G3_ADDR12	238	IO2_21	E32	G4_DATA12
214	IO1_98	E29	G3_ADDR13	239	IO2_22	G30	G4_DATA13
215	IO1_99	A31	/FPGACFG_WRITE	240	IO2_23	M25	G4_DATA14
216	IO1_100	D30	/FPGACFG_CS	241	IO2_24	G31	G4_DATA15
217	IO2_0	F31	G3_ADDR14	242	IO2_25	L26	/G4_CE
218	IO2_1	J32	G3_ADDR15	243	IO2_26	D33	/G4_WE
219	IO2_2	K27	G3_ADDR16	244	IO2_27	D34	/G4_OE
220	IO2_3	K31	G3_ADDR17	245	IO2_28	H29	G4_ADDR0
221	IO2_4	L28	G3_ADDR18	246	IO2_29	J28	G4_ADDR1
222	IO2_5	L30	G3_ADDR19	247	IO2_30	E33	G4_ADDR2
223	IO2_6	M32	G4_DATA0	248	IO2_31	H28	G4_ADDR3
224	IO2_7	N26	G4_DATA1	249	IO2_32	H30	G4_ADDR4
225	IO2_8	N28	G4_DATA2	250	IO2_33	H32	G4_ADDR5

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 5 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
251	IO2_34	K28	G4_ADDR6	276	IO2_59	R25	G5_DATA10
252	IO2_35	L27	G4_ADDR7	277	IO2_60	M34	G5_DATA11
253	IO2_36	F33	G4_ADDR8	278	IO2_61	L31	G5_DATA12
254	IO2_37	M26	G4_ADDR9	279	IO2_62	L33	G5_DATA13
255	IO2_38	E34	G4_ADDR10	280	IO2_63	P27	G5_DATA14
256	IO2_39	H31	G4_ADDR11	281	IO2_64	M33	G5_DATA15
257	IO2_40	G32	G4_ADDR12	282	IO2_65	M31	/G5_CE
258	IO2_41	N25	G4_ADDR13	283	IO2_66	R26	/G5_WE
259	IO2_42	J31	G4_ADDR14	284	IO2_67	N30	/G5_OE
260	IO2_43	J30	G4_ADDR15	285	IO2_68	P28	G5_ADDR0
261	IO2_44	G33	G4_ADDR16	286	IO2_69	N29	G5_ADDR1
262	IO2_45	H34	G4_ADDR17	287	IO2_70	N33	G5_ADDR2
263	IO2_46	J29	G4_ADDR18	288	IO2_71	T25	G5_ADDR3
264	IO2_47	M27	G4_ADDR19	289	IO2_72	N34	G5_ADDR4
265	IO2_48	H33	G5_DATA0	290	IO2_73	P34	G5_ADDR5
266	IO2_49	K29	G5_DATA1	291	IO2_74	R27	G5_ADDR6
267	IO2_50	J34	G5_DATA2	292	IO2_75	P29	G5_ADDR7
268	IO2_51	L29	G5_DATA3	293	IO2_76	P31	G5_ADDR8
269	IO2_52	J33	G5_DATA4	294	IO2_77	P33	G5_ADDR9
270	IO2_53	M28	G5_DATA5	295	IO2_78	T26	G5_ADDR10
271	IO2_54	K34	G5_DATA6	296	IO2_79	R34	G5_ADDR11
272	IO2_55	N27	G5_DATA7	297	IO2_80	R28	G5_ADDR12
273	IO2_56	L34	G5_DATA8	298	IO2_81	N31	G5_ADDR13
274	IO2_57	K33	G5_DATA9	299	IO2_82	N32	LAD4
275	IO2_58	P26	LAD6	300	IO2_83	P30	G5_ADDR14

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 6 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
301	IO2_84	R33	G5_ADDR15	326	IO3_8	AF34	G6_ADDR1
302	IO2_85	R29	G5_ADDR16	327	IO3_9	AG31	G6_ADDR2
303	IO2_86	T34	G5_ADDR17	328	IO3_10	AG33	G6_ADDR3
304	IO2_87	R30	G5_ADDR18	329	IO3_11	AG34	G6_ADDR4
305	IO2_88	T30	G5_ADDR19	330	IO3_12	AH29	G6_ADDR5
306	IO2_89	T28	G6_DATA0	331	IO3_13	AJ30	G6_ADDR6
307	IO2_90	R31	G6_DATA1	332	IO3_14	V26	G6_ADDR7
308	IO2_91	T29	G6_DATA2	333	IO3_15	V30	G6_ADDR8
309	IO2_92	U27	G6_DATA3	334	IO3_16	W34	G6_ADDR9
310	IO2_93	T31	G6_DATA4	335	IO3_17	V28	G6_ADDR10
311	IO2_94	T33	G6_DATA5	336	IO3_18	W32	G6_ADDR11
312	IO2_95	U28	G6_DATA6	337	IO3_19	W30	G6_ADDR12
313	IO2_96	T32	G6_DATA7	338	IO3_20	V29	G6_ADDR13
314	IO2_97	U29	G6_DATA8	339	IO3_21	Y34	G6_ADDR14
315	IO2_98	U33	G6_DATA9	340	IO3_22	W29	G6_ADDR15
316	IO2_99	V33	G6_DATA10	341	IO3_23	Y33	G6_ADDR16
317	IO2_100	U31	G6_DATA11	342	IO3_24	W26	G6_ADDR17
318	IO3_0	V27	G6_DATA12	343	IO3_25	W28	G6_ADDR18
319	IO3_1	V31	G6_DATA13	344	IO3_26	Y31	G6_ADDR19
320	IO3_2	V32	G6_DATA14	345	IO3_27	Y30	G7_DATA0
321	IO3_3	W33	G6_DATA15	346	IO3_28	AA34	G7_DATA1
322	IO3_4	AB25	/G6_CE	347	IO3_29	W31	G7_DATA2
323	IO3_5	AB26	/G6_WE	348	IO3_30	AA33	LAD3
324	IO3_6	AB31	/G6_OE	349	IO3_31	Y29	G7_DATA3
325	IO3_7	AC31	G6_ADDR0	350	IO3_32	W25	G7_DATA4

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 7 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
351	IO3_33	AB34	G7_DATA5	376	IO3_58	AA25	G7_ADDR9
352	IO3_34	Y28	G7_DATA6	377	IO3_59	AE32	G7_ADDR10
353	IO3_35	AB33	G7_DATA7	378	IO3_60	AE31	G7_ADDR11
354	IO3_36	AA30	G7_DATA8	379	IO3_61	AD29	G7_ADDR12
355	IO3_37	Y26	G7_DATA9	380	IO3_62	AD31	G7_ADDR13
356	IO3_38	Y27	G7_DATA10	381	IO3_63	AF33	G7_ADDR14
357	IO3_39	AA31	G7_DATA11	382	IO3_64	AC28	G7_ADDR15
358	IO3_40	AA27	G7_DATA12	383	IO3_65	AF31	G7_ADDR16
359	IO3_41	AA29	G7_DATA13	384	IO3_66	AC27	G7_ADDR17
360	IO3_42	AB32	G7_DATA14	385	IO3_67	AF32	G7_ADDR18
361	IO3_43	AB29	G7_DATA15	386	IO3_68	AE29	G7_ADDR19
362	IO3_44	AA28	/G7_CE	387	IO3_69	AD28	G8_DATA0
363	IO3_45	AC34	/G7_WE	388	IO3_70	AD30	G8_DATA1
364	IO3_46	Y25	/G7_OE	389	IO3_71	AG32	G8_DATA2
365	IO3_47	AD34	G7_ADDR0	390	IO3_72	AC26	G8_DATA3
366	IO3_48	AB30	G7_ADDR1	391	IO3_73	AH33	G8_DATA4
367	IO3_49	AC33	G7_ADDR2	392	IO3_74	AD26	G8_DATA5
368	IO3_50	AA26	G7_ADDR3	393	IO3_75	AF30	G8_DATA6
369	IO3_51	AC32	G7_ADDR4	394	IO3_76	AC25	G8_DATA7
370	IO3_52	AD33	G7_ADDR5	395	IO3_77	AH32	G8_DATA8
371	IO3_53	AB28	G7_ADDR6	396	IO3_78	AE28	G8_DATA9
372	IO3_54	AE34	G7_ADDR7	397	IO3_79	AL34	G8_DATA10
373	IO3_55	AB27	LAD2	398	IO3_80	AG30	G8_DATA11
374	IO3_56	AE33	LAD1	399	IO3_81	AD27	G8_DATA12
375	IO3_57	AC30	G7_ADDR8	400	IO3_82	AF29	G8_DATA13

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 8 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
401	IO3_83	AK34	G8_DATA14	426	IO4_7	AK19	G8_ADDR18
402	IO3_84	AD25	G8_DATA15	427	IO4_8	AL25	G8_ADDR19
403	IO3_85	AE27	/G8_CE	428	IO4_9	AL27	G9_DATA0
404	IO3_86	AJ33	/G8_WE	429	IO4_10	AL30	G9_DATA1
405	IO3_87	AH31	/G8_OE	430	IO4_11	AN18	G9_DATA2
406	IO3_88	AE26	G8_ADDR0	431	IO4_12	AN22	G9_DATA3
407	IO3_89	AL33	G8_ADDR1	432	IO4_13	AN24	G9_DATA4
408	IO3_90	AF28	G8_ADDR2	433	IO4_14	AP31	G9_DATA5
409	IO3_91	AL32	G8_ADDR3	434	IO4_15	AK29	G9_DATA6
410	IO3_92	AJ31	G8_ADDR4	435	IO4_16	AP30	G9_DATA7
411	IO3_93	AF27	G8_ADDR5	436	IO4_17	AN31	G9_DATA8
412	IO3_94	AG29	G8_ADDR6	437	IO4_18	AH27	G9_DATA9
413	IO3_95	AJ32	G8_ADDR7	438	IO4_19	AN30	G9_DATA10
414	IO3_96	AK33	G8_ADDR8	439	IO4_20	AM30	G9_DATA11
415	IO3_97	AH30	G8_ADDR9	440	IO4_21	AK28	G9_DATA12
416	IO3_98	AK32	LAD0	441	IO4_22	AG26	G9_DATA13
417	IO3_99	AK31	/FPGACFG_INIT	442	IO4_23	AN29	G9_DATA14
418	IO3_100	V34	G8_ADDR10	443	IO4_24	AF25	G9_DATA15
419	IO4_0	AE21	G8_ADDR11	444	IO4_25	AM29	/G9_CE
420	IO4_1	AG18	G8_ADDR12	445	IO4_26	AL29	/G9_WE
421	IO4_2	AG23	G8_ADDR13	446	IO4_27	AL28	/G9_OE
422	IO4_3	AH24	G8_ADDR14	447	IO4_28	AE24	G9_ADDR0
423	IO4_4	AH25	G8_ADDR15	448	IO4_29	AN28	G9_ADDR1
424	IO4_5	AJ28	G8_ADDR16	449	IO4_30	AJ27	G9_ADDR2
425	IO4_6	AK18	G8_ADDR17	450	IO4_31	AH26	G9_ADDR3

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 9 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
451	IO4_32	AG25	G9_ADDR4	476	IO4_57	AP24	G10_DATA9
452	IO4_33	AK27	G9_ADDR5	477	IO4_58	AL24	G10_DATA10
453	IO4_34	AM28	G9_ADDR6	478	IO4_59	AK23	G10_DATA11
454	IO4_35	AF24	G9_ADDR7	479	IO4_60	AG22	G10_DATA12
455	IO4_36	AJ26	G9_ADDR8	480	IO4_61	AN23	G10_DATA13
456	IO4_37	AP27	G9_ADDR9	481	IO4_62	AP23	G10_DATA14
457	IO4_38	AK26	G9_ADDR10	482	IO4_63	AM23	G10_DATA15
458	IO4_39	AN27	G9_ADDR11	483	IO4_64	AH22	/G10_CE
459	IO4_40	AE23	G9_ADDR12	484	IO4_65	AP22	/G10_WE
460	IO4_41	AM27	G9_ADDR13	485	IO4_66	AL23	/G10_OE
461	IO4_42	AL26	G9_ADDR14	486	IO4_67	AF21	G10_ADDR0
462	IO4_43	AP26	G9_ADDR15	487	IO4_68	AL22	G10_ADDR1
463	IO4_44	AN26	G9_ADDR16	488	IO4_69	AJ22	G10_ADDR2
464	IO4_45	AJ25	G9_ADDR17	489	IO4_70	AK22	G10_ADDR3
465	IO4_46	AG24	G9_ADDR18	490	IO4_71	AM22	G10_ADDR4
466	IO4_47	AP25	G9_ADDR19	491	IO4_72	AG21	G10_ADDR5
467	IO4_48	AF23	G10_DATA0	492	IO4_73	AJ21	G10_ADDR6
468	IO4_49	AM26	G10_DATA1	493	IO4_74	AP21	G10_ADDR7
469	IO4_50	AJ24	G10_DATA2	494	IO4_75	AE20	G10_ADDR8
470	IO4_51	AN25	G10_DATA3	495	IO4_76	AH21	G10_ADDR9
471	IO4_52	AE22	G10_DATA4	496	IO4_77	AL21	G10_ADDR10
472	IO4_53	AM25	G10_DATA5	497	IO4_78	AN21	G10_ADDR11
473	IO4_54	AK24	G10_DATA6	498	IO4_79	AF20	G10_ADDR12
474	IO4_55	AH23	G10_DATA7	499	IO4_80	AK21	G10_ADDR13
475	IO4_56	AF22	G10_DATA8	500	IO4_81	AP20	G10_ADDR14

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 10 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
501	IO4_82	AE19	G10_ADDR15	526	IO5_6	AK13	G11_ADDR1
502	IO4_83	AN20	G10_ADDR16	527	IO5_7	AL13	G11_ADDR2
503	IO4_84	AG20	G10_ADDR17	528	IO5_8	AM4	G11_ADDR3
504	IO4_85	AL20	G10_ADDR18	529	IO5_9	AN9	G11_ADDR4
505	IO4_86	AH20	G10_ADDR19	530	IO5_10	AN10	G11_ADDR5
506	IO4_87	AK20	G11_DATA0	531	IO5_11	AN16	G11_ADDR6
507	IO4_88	AN19	G11_DATA1	532	IO5_12	AN17	G11_ADDR7
508	IO4_89	AJ20	G11_DATA2	533	IO5_13	AL17	G11_ADDR8
509	IO4_90	AF19	G11_DATA3	534	IO5_14	AH17	G11_ADDR9
510	IO4_91	AP19	G11_DATA4	535	IO5_15	AM17	G11_ADDR10
511	IO4_92	AM19	G11_DATA5	536	IO5_16	AJ17	G11_ADDR11
512	IO4_93	AH19	G11_DATA6	537	IO5_17	AG17	G11_ADDR12
513	IO4_94	AJ19	G11_DATA7	538	IO5_18	AP16	G11_ADDR13
514	IO4_95	AP18	G11_DATA8	539	IO5_19	AL16	G11_ADDR14
515	IO4_96	AF18	G11_DATA9	540	IO5_20	AJ16	G11_ADDR15
516	IO4_97	AP17	G11_DATA10	541	IO5_21	AM16	G11_ADDR16
517	IO4_98	AJ18	G11_DATA11	542	IO5_22	AK16	G11_ADDR17
518	IO4_99	AL18	G11_DATA12	543	IO5_23	AP15	G11_ADDR18
519	IO4_100	AM18	G11_DATA13	544	IO5_24	AL15	G11_ADDR19
520	IO5_0	AF17	G11_DATA14	545	IO5_25	AH16	G12_DATA0
521	IO5_1	AG12	G11_DATA15	546	IO5_26	AN15	G12_DATA1
522	IO5_2	AH12	/G11_CE	547	IO5_27	AF16	G12_DATA2
523	IO5_3	AJ10	/G11_WE	548	IO5_28	AP14	G12_DATA3
524	IO5_4	AJ11	/G11_OE	549	IO5_29	AE16	G12_DATA4
525	IO5_5	AK7	G11_ADDR0	550	IO5_30	AK15	G12_DATA5

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 11 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
551	IO5_31	AJ15	G12_DATA6	576	IO5_56	AJ13	G12_ADDR12
552	IO5_32	AH15	G12_DATA7	577	IO5_57	AP10	G12_ADDR13
553	IO5_33	AN14	G12_DATA8	578	IO5_58	AK12	G12_ADDR14
554	IO5_34	AK14	G12_DATA9	579	IO5_59	AM10	G12_ADDR15
555	IO5_35	AG15	G12_DATA10	580	IO5_60	AP9	G12_ADDR16
556	IO5_36	AM13	G12_DATA11	581	IO5_61	AK11	G12_ADDR17
557	IO5_37	AF15	G12_DATA12	582	IO5_62	AL11	G12_ADDR18
558	IO5_38	AG14	G12_DATA13	583	IO5_63	AL10	G12_ADDR19
559	IO5_39	AP13	G12_DATA14	584	IO5_64	AE13	G13_DATA0
560	IO5_40	AE14	G12_DATA15	585	IO5_65	AM9	G13_DATA1
561	IO5_41	AE15	/G12_CE	586	IO5_66	AF12	G13_DATA2
562	IO5_42	AN13	/G12_WE	587	IO5_67	AP8	G13_DATA3
563	IO5_43	AG13	/G12_OE	588	IO5_68	AL9	G13_DATA4
564	IO5_44	AH14	G12_ADDR0	589	IO5_69	AH11	G13_DATA5
565	IO5_45	AP12	G12_ADDR1	590	IO5_70	AF11	G13_DATA6
566	IO5_46	AJ14	G12_ADDR2	591	IO5_71	AN8	G13_DATA7
567	IO5_47	AL14	G12_ADDR3	592	IO5_72	AM8	G13_DATA8
568	IO5_48	AF13	G12_ADDR4	593	IO5_73	AG11	G13_DATA9
569	IO5_49	AN12	G12_ADDR5	594	IO5_74	AL8	G13_DATA10
570	IO5_50	AF14	G12_ADDR6	595	IO5_75	AK9	G13_DATA11
571	IO5_51	AP11	G12_ADDR7	596	IO5_76	AH10	G13_DATA12
572	IO5_52	AN11	G12_ADDR8	597	IO5_77	AN7	G13_DATA13
573	IO5_53	AH13	G12_ADDR9	598	IO5_78	AE12	G13_DATA14
574	IO5_54	AM12	G12_ADDR10	599	IO5_79	AJ9	G13_DATA15
575	IO5_55	AL12	G12_ADDR11	600	IO5_80	AM7	/G13_CE

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 12 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
601	IO5_81	AL7	/G13_WE	626	IO6_6	AB5	G14_DATA3
602	IO5_82	AG10	/G13_OE	627	IO6_7	AB7	G14_DATA4
603	IO5_83	AN6	G13_ADDR0	628	IO6_8	AB9	G14_DATA5
604	IO5_84	AK8	G13_ADDR1	629	IO6_9	AD7	G14_DATA6
605	IO5_85	AH9	G13_ADDR2	630	IO6_10	AD8	G14_DATA7
606	IO5_86	AP5	G13_ADDR3	631	IO6_11	AE2	G14_DATA8
607	IO5_87	AJ8	G13_ADDR4	632	IO6_12	AE4	G14_DATA9
608	IO5_88	AE11	G13_ADDR5	633	IO6_13	AJ4	G14_DATA10
609	IO5_89	AN5	G13_ADDR6	634	IO6_14	AH5	G14_DATA11
610	IO5_90	AF10	G13_ADDR7	635	IO6_15	AH6	G14_DATA12
611	IO5_91	AM6	G13_ADDR8	636	IO6_16	AF8	G14_DATA13
612	IO5_92	AL6	G13_ADDR9	637	IO6_17	AE9	G14_DATA14
613	IO5_93	AG9	G13_ADDR10	638	IO6_18	AK3	G14_DATA15
614	IO5_94	AH8	G13_ADDR11	639	IO6_19	AD10	/G14_CE
615	IO5_95	AP4	G13_ADDR12	640	IO6_20	AL2	/G14_WE
616	IO5_96	AN4	G13_ADDR13	641	IO6_21	AL1	/G14_OE
617	IO5_97	AJ7	G13_ADDR14	642	IO6_22	AH4	G14_ADDR0
618	IO5_98	AM5	G13_ADDR15	643	IO6_23	AG6	G14_ADDR1
619	IO5_99	AK6	G13_ADDR16	644	IO6_24	AK1	G14_ADDR2
620	IO6_0	T1	G13_ADDR17	645	IO6_25	AF7	G14_ADDR3
621	IO6_1	V2	G13_ADDR18	646	IO6_26	AK2	G14_ADDR4
622	IO6_2	V3	G13_ADDR19	647	IO6_27	AJ3	G14_ADDR5
623	IO6_3	V5	G14_DATA0	648	IO6_28	AG5	G14_ADDR6
624	IO6_4	V8	G14_DATA1	649	IO6_29	AD9	G14_ADDR7
625	IO6_5	AA10	G14_DATA2	650	IO6_30	AJ2	G14_ADDR8

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 13 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
651	IO6_31	AC10	G14_ADDR9	676	IO6_56	AD2	G15_DATA14
652	IO6_32	AH2	G14_ADDR10	677	IO6_57	AB8	G15_DATA15
653	IO6_33	AH3	G14_ADDR11	678	IO6_58	AC1	/G15_CE
654	IO6_34	AF5	G14_ADDR12	679	IO6_59	AC5	/G15_WE
655	IO6_35	AE8	G14_ADDR13	680	IO6_60	AC2	/G15_OE
656	IO6_36	AG3	G14_ADDR14	681	IO6_61	AA9	G15_ADDR0
657	IO6_37	AE7	G14_ADDR15	682	IO6_62	AC3	G15_ADDR1
658	IO6_38	AG2	G14_ADDR16	683	IO6_63	AC4	G15_ADDR2
659	IO6_39	AF6	G14_ADDR17	684	IO6_64	AD4	G15_ADDR3
660	IO6_40	AG1	G14_ADDR18	685	IO6_65	AA8	G15_ADDR4
661	IO6_41	AC9	G14_ADDR19	686	IO6_66	AB6	G15_ADDR5
662	IO6_42	AG4	G15_DATA0	687	IO6_67	AB1	G15_ADDR6
663	IO6_43	AE6	G15_DATA1	688	IO6_68	Y10	G15_ADDR7
664	IO6_44	AF3	G15_DATA2	689	IO6_69	AB2	G15_ADDR8
665	IO6_45	AF1	G15_DATA3	690	IO6_70	AA7	G15_ADDR9
666	IO6_46	AF4	G15_DATA4	691	IO6_71	AA4	G15_ADDR10
667	IO6_47	AB10	G15_DATA5	692	IO6_72	AA1	G15_ADDR11
668	IO6_48	AF2	G15_DATA6	693	IO6_73	Y9	G15_ADDR12
669	IO6_49	AC8	G15_DATA7	694	IO6_74	AB4	G15_ADDR13
670	IO6_50	AE1	G15_DATA8	695	IO6_75	AA2	G15_ADDR14
671	IO6_51	AD5	G15_DATA9	696	IO6_76	Y8	G15_ADDR15
672	IO6_52	AE3	G15_DATA10	697	IO6_77	AA6	G15_ADDR16
673	IO6_53	AC7	G15_DATA11	698	IO6_78	AA5	G15_ADDR17
674	IO6_54	AD1	G15_DATA12	699	IO6_79	AB3	G15_ADDR18
675	IO6_55	AD6	G15_DATA13	700	IO6_80	Y7	G15_ADDR19

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 14 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
701	IO6_81	Y1	G16_DATA0	726	IO7_5	K4	G16_ADDR6
702	IO6_82	W10	G16_DATA1	727	IO7_6	L6	G16_ADDR7
703	IO6_83	Y5	G16_DATA2	728	IO7_7	M5	G16_ADDR8
704	IO6_84	Y2	G16_DATA3	729	IO7_8	M10	G16_ADDR9
705	IO6_85	W9	G16_DATA4	730	IO7_9	N5	G16_ADDR10
706	IO6_86	W2	G16_DATA5	731	IO7_10	N10	G16_ADDR11
707	IO6_87	W7	G16_DATA6	732	IO7_11	R7	G16_ADDR12
708	IO6_88	Y4	G16_DATA7	733	IO7_12	T2	G16_ADDR13
709	IO6_89	W1	G16_DATA8	734	IO7_13	T7	G16_ADDR14
710	IO6_90	Y6	G16_DATA9	735	IO7_14	U8	G16_ADDR15
711	IO6_91	W6	G16_DATA10	736	IO7_15	V4	G16_ADDR16
712	IO6_92	W3	G16_DATA11	737	IO7_16	U9	G16_ADDR17
713	IO6_93	V9	G16_DATA12	738	IO7_17	U4	G16_ADDR18
714	IO6_94	W4	G16_DATA13	739	IO7_18	U7	G16_ADDR19
715	IO6_95	W5	G16_DATA14	740	IO7_19	U5	FPGA_FIFODATA0
716	IO6_96	V1	G16_DATA15	741	IO7_20	U3	FPGA_FIFODATA1
717	IO6_97	V7	/G16_CE	742	IO7_21	U6	FPGA_FIFODATA2
718	IO6_98	U2	/G16_WE	743	IO7_22	T3	FPGA_FIFODATA3
719	IO6_99	V6	/G16_OE	744	IO7_23	T6	FPGA_FIFODATA4
720	IO6_100	U1	G16_ADDR0	745	IO7_24	T9	FPGA_FIFODATA5
721	IO7_0	F5	G16_ADDR1	746	IO7_25	T4	FPGA_FIFODATA6
722	IO7_1	G6	G16_ADDR2	747	IO7_26	T5	FPGA_FIFODATA7
723	IO7_2	H1	G16_ADDR3	748	IO7_27	R1	FPGA_FIFODATA8
724	IO7_3	H7	G16_ADDR4	749	IO7_28	R6	FPGA_FIFODATA9
725	IO7_4	K2	G16_ADDR5	750	IO7_29	T10	FPGA_FIFODATA10

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 15 of 17)

#	FPGA signal	pin	design signal	#	FPGA signal	pin	design signal
751	IO7_30	R2	FPGA_FIFODATA11	776	IO7_55	N8	FPGA_FF3
752	IO7_31	R5	FPGA_FIFODATA12	777	IO7_56	L2	FPGA_FF4
753	IO7_32	P1	FPGA_FIFODATA13	778	IO7_57	N9	/FPGAUSER_CS
754	IO7_33	P5	FPGA_FIFODATA14	779	IO7_58	M7	FPGA_MARKEND
755	IO7_34	R8	FPGA_FIFODATA15	780	IO7_59	K1	MA3
756	IO7_35	P2	FPGA_FIFODATA16	781	IO7_60	M8	MA2
757	IO7_36	R9	FPGA_FIFODATA17	782	IO7_61	L4	MA1
758	IO7_37	N1	FPGA_FIFODATA18	783	IO7_62	J1	MA0
759	IO7_38	P4	FPGA_FIFODATA19	784	IO7_63	L5	/MOE
760	IO7_39	R10	FPGA_FIFODATA20	785	IO7_64	J2	/MWE
761	IO7_40	P8	FPGA_FIFODATA21	786	IO7_65	K3	PARSE_FINISHED
762	IO7_41	N2	FPGA_FIFODATA22	787	IO7_66	L7	PARSE_ERROR
763	IO7_42	P6	FPGA_FIFODATA23	788	IO7_67	J3	PARSE_START
764	IO7_43	P7	FPGA_FIFODATA24	789	IO7_68	M9	(not used)
765	IO7_44	M1	FPGA_FIFODATA25	790	IO7_69	H2	(not used)
766	IO7_45	N4	FPGA_FIFODATA26	791	IO7_70	J4	(not used)
767	IO7_46	N6	FPGA_FIFODATA27	792	IO7_71	K6	(not used)
768	IO7_47	N3	FPGA_FIFODATA28	793	IO7_72	L8	(not used)
769	IO7_48	P9	FPGA_FIFODATA29	794	IO7_73	G2	(not used)
770	IO7_49	M2	FPGA_FIFODATA30	795	IO7_74	H3	(not used)
771	IO7_50	N7	FPGA_FIFODATA31	796	IO7_75	K7	DISPLAY_D0
772	IO7_51	M3	/FPGA_WEN1	797	IO7_76	G3	DISPLAY_D1
773	IO7_52	P10	/FPGA_WEN2/ LD	798	IO7_77	J5	DISPLAY_D2
774	IO7_53	M4	FPGA_FF1	799	IO7_78	L9	DISPLAY_D3
775	IO7_54	L1	FPGA_FF2	800	IO7_79	H5	DISPLAY_D4

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 16 of 17)

#	FPGA signal	pin	design signal
801	IO7_80	J6	DISPLAY_D5
802	IO7_81	H4	DISPLAY_D6
803	IO7_82	G4	DISPLAY_A0
804	IO7_83	K8	DISPLAY_A1
805	IO7_84	J7	/DISPLAY_WR
806	IO7_85	F2	(not used)
807	IO7_86	F3	(not used)
808	IO7_87	L10	(not used)
809	IO7_88	E1	DEBUG0
810	IO7_89	H6	DEBUG1
811	IO7_90	G5	DEBUG2
812	IO7_91	E2	DEBUG3
813	IO7_92	K9	DEBUG4
814	IO7_93	D1	DEBUG5
815	IO7_94	E3	DEBUG6
816	IO7_95	J8	DEBUG7
817	IO7_96	E4	DEBUG8
818	IO7_97	D2	DEBUG9
819	IO7_98	F4	DEBUG10
820	IO7_99	D3	DEBUG11

Virtex-E XCV2000efg1156-6 FPGA pin assignment (page 17 of 17)

Bibliography

[1] A. V. Aho and J. D. Ullman. *"The Theory of Parsing, Translation and Compiling"*, volume 1. Prentice-Hall, 1972.

[2] V. L. Arlazarov, E. A. Dinic, M. A. Kronod, and I. A. Faradzev. "On economical construction of the transitive closure of a directed graph". In *Soviet Mathematics Doklady*, volume 11, pages 1209–1210, 1970.

[3] F. M. Barcal, O. Sacristan, and J. Grana. "Stochastic Parsing and Parallelism". In *Computational Linguistics and Intelligent Text Processing*, pages 401–410. Springer-Verlag Berlin, 2001.

[4] Von L. Burton. *"The Programmable Logic Device Handbook"*. TAB Professional an Reference Books, Blue Ridge Summit, 1990.

[5] J. C. Chappelier and M. Rajman. "A generalized CYK algorithm for parsing stochastic CFG". In *1st Workshop on Tabulation in Parsing and Deduction (TAPD98)*, pages 133–137, April 1998.

[6] J. C. Chappelier and M. Rajman. "A Practical Bottom-Up Algorithm for On-Line Parsing with Stochastic Context-Free Grammars". Technical Report 284, EPFL (DI-LIA), July 1998.

[7] Y. T. Chiang and K. S. Fu. "Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition". In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 6, May 1984.

[8] King-Hang Chu and King-Sun Fu. "VLSI architectures for high speed recognition of context-free languages and finite-state languages". In *Proceedings 9th Annual International Symposium on Computing Architecture*, April 1982.

[9] C. Ciressan. "Using FPGAs for designing an NLP coprocessor.". Technical Report TR-339, EPFL, October 2000.

[10] R. Dale, H. Moisl, and H. Somers, editors. *"Handbook of Natural Language Processing"*. Marcel Dekker, 2000.

[11] G. Erbach. "Bottom-up Earley deduction". In *Proceedings of the* 14*th International Conference on Computational Linguistics (COLING'94)*. Kyoto, Japan, 1994.

[12] M. J. Foster and H. T. Kung. "The design of special-purpose VLSI chips". *IEEE Computer*, 13, January 1980.

[13] S. L. Graham, M. A. Harrison, and R. L. Mercer. "An Improved Context-Free Recognizer". *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[14] J. Hopcroft and J. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley, 1979.

[15] O. H. Ibarra, T. Jiang, and H. Wang. "Parallel Parsing on a One-Way Linear Array of Finite-State Machines". In *Foundations of Software Technology and Theoretical Computer Science: Proc. of the Ninth Conference*, pages 291–300. Springer, 1989.

[16] O. H. Ibarra, T. C. Pong, and S. M. Sohn. "Parallel Recognition and Parsing on the Hypercube". *IEEE Transactions on Computers*, 40(6):764–770, June 1991.

[17] S. R. Kosaraju. "Speed of recognition of context-free languages by array automata". *SIAM Journal on Computing*, 4, September 1975.

[18] H. T. Kung. "Let's design algorithms for VLSI systems". In *Caltech Conference on VLSI*, January 1979.

[19] Parag K. Lala. *"Digital System Design Using Programmable Logic Devices"*. Prentice Halls, Englewood Cliffs, 1990.

[20] P. Linz. *"An Introduction to Formal Languages and Automata"*. Jones and Bartlett Publishers, 1997. Pages 155-168.

[21] T. Ninomiya, K. Torisawa, K. Taura, and J. Tsujii. "A Parallel CKY Parsing Algorithm on Large-Scale Distributed-Memory Parallel Machines". In *Proceedings of the Pacific Association for Computational Linguistics*, pages 232–243, September 1997.

[22] R. A. Peleato, M. Rajman, and J.-C. Chappelier. "Integration of syntactic constraints within a speech recognition system. Coupling a speech recognizer and a stochastic context-free parser". Technical report, EPFL (DI-LIA), February 1999.

[23] PLX Technology Inc. *"Hardware Reference Manual"*, v 1.0 edition, September 1999.

[24] PLX Technology Inc. *"IOP 480 Data Book"*, October 1999.

[25] G. Sampson. "The Susanne Corpus Release 3". School of Cognitive & Computing Sciences, University of Sussex, Falmer, Brighton, England, 1994.

[26] E. Sanchez and M. Tomassini. *"Towards Evolvable Hardware"*. Springer-Verlag Berlin, 1996.

[27] L. G. Valiant. "General Context-Free Recognition in Less than Cubic Time". *Journal of Computer and System Sciences*, 10:308–315, 1975.

[28] John Villasenor and William H. Mangione-Smith. "Configurable Computing". *Scientific American*, 1997.

[29] F. Voisin and J.-C. Raoult. "A new, Bottom-up, General Parsing Algorithm". In *Journees AFCET-GROPLAN, les Avancees en programmation*. Nice, 1990.

[30] Xilin Inc. $Virtex^{TM} - E$ *1.8V Field Programmable Gate Arrays*, v1.7 edition, September 2000.

[31] Xilin Inc. $Virtex^{TM}$ *2.5V Field Programmable Gate Arrays*, v2.5 edition, April 2001.

[32] D. H. Younger. "Recognition of context-free languages in time n^3". In *Information and Control*, pages 189–208, February 1967.

Index

Curriculum Vitae

Name:	Cristian CIRE S SAN
Date/Place of birth:	1973, August 11, born in Timisoara, Romania
Languages:	Romanian, English, French

Education:

1991-1996	Diploma in Computer Science (Dipl. Hardware Engineer, "Politechnica" University of Timisoara)
1994-1996	Research & Development, BEE-SPEED S.R.L., Timisoara, Romania.
1996	Diploma Project "Data transmission using the NICAM standard", Swiss Federal Institute of Technology in Zürich (ETHZ)
1996-1997	Doctoral School in Communication Systems, Swiss Federal Institute of Technology in Lausanne (EPFL)
1997	Diploma Project "Real-time 3D body tracking for Virtual Reality Environments" Swiss Federal Institute of Technology in Lausanne (EPFL)
1997-current	PhD. Student, Artificial Intelligence Laboratory, Dept. of Computer Science Swiss Federal Institute of Technology in Lausanne (EPFL)

PUBLICATIONS

1) C. Ciressan, M. Rajman, E. Sanchez and J.-C. Chappelier, "Towards NLP-coprocessing: An FPGA implementation of a context-free parser.", 7ème conférence sur le Traitement Automatique du Langage Naturel (TALN 2000), Lausanne, Suisse, Octobre, 2000, pp. 91-100.
2) C. Ciressan, E. Sanchez, M. Rajman and J.-C. Chappelier, "An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars.", 11th International Conference on Field Programmable Logic and Applications (FPL 2001), Lecture Notes in Computer Science, Belfast, Northern Ireland, August, 2001.
3) C. Ciressan, E. Sanchez, M. Rajman and J.-C. Chappelier, "An FPGA-based coprocessor for the parsing of context-free grammars.", 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, California, April, 2000.
4) C. Ciressan, E. Rajman, E. Sanchez and J.-C. Chappelier, "Using FPGAs for designing an NLP coprocessor", Technical Report No. 00/339, EPFL, Dept. Computer Science, DI-LIA, October 2000
5) C. Ciressan, E. Sanchez, M. Rajman and J.-C. Chappelier, "An FPGA-based syntactic parser for real-life unrestricted context-free grammars", Technical Report No. 01/373, EPFL, Dept. Computer Science, DI-LIA, October 2001