UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Enginyeria Electrònica

Proposal and development of a highly modular and scalable self-adaptive hardware architecture with parallel processing capability.

Thesis submitted in partial fulfillment of the requirement for the PhD Degree issued by the Universitat Politècnica de Catalunya, in its Electronic Engineering Program.

Javier E. Soto Vargas

Director: Juan Manuel Moreno Arostegui

Barcelona, España July , 2014

This document is prepared to be printed double-sided.

To my wife Carol,
for her immense love and unconditional support,
thanks for always being there for me.

To my two little dinosaurs: Gabriela and Daniel,
for the games that we will play,
and for the life that you will teach me to discover.

This thesis is for you, my beautiful family.

Con mucho cariño para mis padres: Evandro ♣ y Nair.
Gracias por el gran ezfuerzo y sacrificio
que le han brindado a todos sus hijos.

Para toda mi famila, a la que siempre tengo presente.
A mis hermanos: Lina, Diana y Raúl,
a mi sobrino Nicolás.
y a la memoria de mi tio Enrique ♣.

Para Amira, Dalila y Sarah.
Gracias por el apoyo y cariño que le han brindado a la familia
durante esta etapa de nuestras vidas, lejos de casa.

Acknowledgements / Agradecimientos

I consider myself the best friend of my friends,
and I think any of them love me as much
as I love the friend I love less.

Me considero el mejor amigo de mis amigos,
y creo que ninguno de ellos me quiere tanto
como yo quiero al amigo que quiero menos.

Gabriel García Márquez (1927 – 2014)

First, I would like to thank my thesis supervisor, Professor Juan Manuel Moreno Arostegui, for their guidance during my studies in the Universitat Politècnica de Catalunya. Thank you for your professional and appropriate corrections in the preparation of this work. I would also like to thank the professors Joan Cabestany and Jordi Madrenas for kindly receiving me in the AHA research group. Thank you all for opening the doors of this university, and give me the opportunity of professional growth during the development of my doctoral studies.

Quiero agradecer a todos los que me han brindado su apoyo directo e indirecto durante el desarrollo de mis estudios de doctorado. Escribiendo estas líneas siento deseos de mencionar a muchas personas que he conocido y con quien he compartido algún momento de mi vida, tanto de carácter personal como profesional, con aquellos que han sido parte mi formación como persona y como ingeniero. Con ellos seguramente podremos recordar alguna anécdota o algún momento especial que se mantenga presente en la memoria, uno de esos que siempre se recuerdan con una sonrisa. Para todos los que aparecen en estas líneas y los que no, muchas gracias.

Gracias a la Escuela Colombiana de Ingeniería por formarme como Ingeniero y permitirme ser parte de su comunidad. Gracias a mis compañeros de universidad: Angela, Gustavo, Jaime, Angel y Andrés, con quienes compartí y espero seguir compartiendo, la profesión que ha marcado el futuro de nuestras vidas. Gracias a mis profesores, algunos de los cuales tengo el privilegio de tener como compañeros y amigos. Gracias a Henry por su amistad y apoyo en este proceso. Gracias a Alex y Gina, con quien compartí (sufrí) en paralelo este proceso de formación científica y de crecimiento familiar.

Gracias a mis grandes amigos del colegio, los fetzes: Luis Guillermo, José Alberto, Mauricio, Gustavo, Carlos, Julio y Luis Alfonso. Con quienes viví las aventuras más emocionantes de mi vida antes de salir de Colombia. Gracias a todos ellos, a los que que considero grandes amigos, con los que he compartido muchas fiestas, innumerables carcajadas y alguna copa de aguardiente, estoy seguro vendrán nuevas aventuras y seguiremos viviendo grandes momentos que serán inolvidables, ahora en su gran mayoría con familia incluida.

Gracias a Mauricio (Mao), que me brindó su generosa e incondicional amistad durante estos casi siete años en Barcelona. Agradezco también a los demás amigos en España: el siempre amable Rodrigo, los compañeros doctorandos en la UPC, los compañeros de trabajo en Delphi y Wonesys. Todos ellos, *culés* o *merengues* en su mayoría, con quienes compartí muchos momentos de carácter académico, laboral y de ocio, que sin duda recordaré tanto como este periodo en la universidad. Podría asegurar que con la gran mayoría compartí charlas personales, académicas, profesionales y sobre todo deportivas, incluyendo especulaciones pre y post partidos de liga, felicidad en las victorias y tristeza en las derrotas, pero siempre con respeto y amabilidad. También he tenido la oportunidad de compartir con todos ellos claritas, medianas, mojitos, cortados, cafés, biquinis, bocadillos, combinados, menús, días de marcha, partidos de fútbol, ping-pong, futbolín y muchos otros, pero sobre todo siempre brindándome buenos momentos que nunca olvidaré en mi paso por Barcelona.

Agradecimientos muy especiales para mi suegra Amira (Aguila 3), mi cuñada Dalila y mi sobrina Sarah, quienes me han brindado un valioso apoyo durante el tiempo que llevamos en Barcelona, principalmente estando muy pendientes de la familia, situación que me permitía trabajar en el desarrollo de este proyecto de tesis. Su cercanía a pesar de la distancia, nos ayudo a consolidar una unión familiar que fue y es aún un importante soporte para mí y sobretodo para mí familia, y que por lo tanto me ayudó a mantener la esperanza y tranquilidad de que este trabajo finalmente se lograría. Gracias también a todos los demás miembros de la familia Sánchez, que siempre nos han acompañado en este proceso.

Quiero agradecer a mi familia en Colombia, a quienes siempre tengo presentes a pesar de la distancia y del poco contacto que he podido brindarles en este tiempo. Gracias a mi Padre, luchador incansable y gran formador de mi carácter, al que recuerdo con profundo sentimiento y al que pido disculpas por no estar junto a él en los momentos más duros. Gracias a mi madre, el pilar de la familia, que con sus sonrisas y lágrimas ha llenado siempre a su hogar de amor y alegría. Gracias a cada uno de mis hermanos: Lina, Diana y Raúl, y a mi sobrino Nicolás; recordarlos y saber que siempre estarán allí, me llenó de buena energía para seguir adelante en el que fue un largo, intermitente y duro proceso. Gracias a mi tío Enrique por su eterna bondad y su su sonrisa siempre amigable. Gracias también a quienes rodearon constantemente a mi madre y hermanos: familiares, amigos, las tías (y tíos) Vargas, sobrinos y primos. El solo hecho de saber que mi madre estaba o estuvo con alguno de ellos cada vez que me contaba algo de lo que hacia, me tranquilizaba y me hacia olvidar por un momento de lo triste que es no estar a su lado para compartir tantos momentos importantes que me perdí por estar lejos de ellos, intentando conseguir un título que no me pertenece solo a mí.

Por último y sin duda, los agradecimientos mas importantes son para mi esposa e hijos: Gabriela y Daniel. Gracias Carol por compartir conmigo este largo proceso, tu presencia y compañía ha sido fundamental en todos los aspectos, agradezco profundamente tu apoyo y paciencia en el desarrollo de este trabajo. Son ellos por los que era impensable rendirse o dejar atrás algo que durante mucho tiempo pareció una meta lejana e incierta. También aprovecho para pedirles disculpas por gastar gran parte de mi tiempo libre, parte de mi atención y concentración en el desarrollo de algoritmos, lineas de código, solución de innumerables inconvenientes o cualquier otro aspecto relacionado con el doctorado, el cual nunca supe con certeza como compartir con ustedes. Gracias hijos por recordarme lo divertido que es jugar a cualquier cosa, desordenar todo sin importar las consecuencias y sobre todo no hacerle caso a la mamá. Gracias por los grandes e inolvidables momentos que hemos compartido en Barcelona, los dos más importantes cuando Millonarios y el Real Madrid . . . , mentiras, el nacimiento de nuestros dos pequeños, a quienes pondré a leer algunas lineas de este trabajo cuando se porten mal y quiera castigarles, y en el supuesto que esto continuara, tendrían que explicarme alguno de los algoritmos propuestos de la Arquitectura de Hardware Auto-adaptable con Capacidad de Procesamiento en Paralelo

que propuse en esta tesis doctoral. Este trabajo es para todos ustedes.

Agradezco a la persona que de manera anónima recuperó el morral que contenía el portátil con toda la información referente al trabajo de doctorado mientras inexplicablemente lo descuidé haciendo un trámite. Gracias a él o ella puedo escribir estas líneas y por lo tanto evitar una situación que recordaré como una anécdota y una lección más en la vida: “el gran susto del último día de tesis”.

Moltes Gràcies Barcelona, Ciutat Comtal.

Abstract

Education is what remains after one has forgotten what one has learned in school.

La educación es lo que queda después de que uno ha olvidado lo que se ha aprendido en la escuela.

Albert Einstein (1879 – 1955)

This dissertation describes a novel unconventional self-adaptive hardware architecture with capacity for parallel processing. For scalability issues, this bioinspired architecture is based on a regular array of homogeneous cells. The proposed programmable architecture implements in a distributed way self-adaptive capabilities including self-placement and self-routing which, due to its intrinsic design, enable the development of systems with runtime reconfiguration, self-repair and/or fault tolerance capabilities. Additionally, this work defines the *configuration* and the *functional* units of the elementary cell, which implements the self-adaptive and parallel processing capabilities respectively.

The physical implementation of this architecture is composed of two-layers, interconnected cells in the first level and interconnected switch and pin matrices in the second level. Several chips can be interconnected for enlarging the cell array. Any application scheduled to the system has to be organized in components, where each component is composed by one or more interconnected cells. The interconnection of cells inside a component is made at cell level (first layer), while the physical interconnections of components are made in the second layer. Additionally, two layers are defined as conceptual organization for the implementation of general purpose applications: the SANE and the SANE assembly. The SANE (Self-Adaptive Networked Entity) is composed by a group of components. This is the basic self-adaptive computing system. It has the ability to monitor its local environment and its internal computation process. The SANE ASSEMBLY (SANE-ASM) is composed by a group of interconnected SANEs.

The processing capabilities of the cell are included in its Functional Unit (FU), which can be described as a four-core configurable multicomputer. The FU includes twelve programmable configuration modes , i.e., each cell permits to select from one to four processors working in parallel, with different size of program and data memories. The cores are grouped or not depending of the configuration mode, allowing program memory sizes of 64, 128, 192 or 256 instructions. Similarly, the data memory can be combined in width and length, achieving combinations for data processing of 8, 16, 24 and 32 bits.

The self-adaptive capabilities of the cell are executed mainly by the Cell Configuration Unit (CCU). The self-placement algorithm is responsible for finding out the most suitable position in the cell array to insert the new cell of a component. The self-routing algorithm is executed since the insertion of the second cell of a component, each time that the self-placement process ends. This algorithm allows interconnecting the ports of the FU of two cells through the cell ports. The self-placement and self-routing processes allow for performing complex functionality changes in real time, these processes endow the system with enhanced functionality, enabling the system

to change itself, this allows for the implementation of run-time self-configuration, without the need for any configuration manager. The absence of a centralized supervision system permits cells to perform some of the tasks in a distributed way.

The architecture proposed includes two mechanisms of fault tolerance. One of these is the Dynamic Fault Tolerance Scaling Technique, that has the ability to create and eliminate the redundant copies of the functional section of a specific application. This is possible due to the ability of runt-time self-configuration included in the architecture. The other mechanism of fault tolerance is a dedicated or static Fault Tolerance System. It provides redundant processing capabilities that are working continuously. When a failure in the execution of a program is detected, the processors of the cell are stopped and the self-elimination and self-replication processes start for the cell (or cells) involved in the failure. This cell(s) will be self-discarded for future self-placement processes.

An FPGA-based prototype and a software tool have been built for demonstration purposes. The prototype includes all the self-adaptive capabilities described in this document. The prototype has been developed in two chips, each one is a Virtex4 Xilinx FPGA (XC4VLX60). The physical design includes the cell array together with a component-level routing system. Additionally a Control Microprocessor (CμP) and other peripherals provide support for the implementation of general purpose applications in the prototype.

With the purpose of having a complete development system, the software tool SANE Project Developer (SPD) has been implemented. The SPD is an Integrated Development Environment (IDE) that allows generating the memory initialization data for the control microprocessor inside the prototype. The SPD allows the creation and edition of various files that describe the configuration of a SANE-ASM. The main (or top) file includes special SANE-ASM instructions (SASM files), which are equivalent to the assembler language for classic processors. The SPD allows the creation and edition of all related information of the SANE-ASM. In addition, the SPD automatically builds the final hexadecimal file with the configuration of the SANE-ASM that will be downloaded in the FPGA-based prototype.

Resumen

No man should escape our universities
without knowing how little he knows.

Ningún hombre debe escapar de nuestras
universidades sin saber lo poco que sabe.

Julius Robert Oppenheimer (1904 – 1967)

Esta tesis doctoral describe una arquitectura de hardware auto-adaptable novedosa y no convencional con capacidad de procesamiento en paralelo. Por razones de escalabilidad, esta arquitectura bioinspirada está basada en una matriz regular de células homogéneas. La arquitectura propuesta es programable, e implementa de manera distribuida diversas capacidades auto-adaptables incluyendo el auto-emplazamiento y auto-enrutamiento, los cuales debido a su diseño intrínseco, permiten el desarrollo de sistemas reconfigurables en tiempo de ejecución, así como de sistemas auto-reparables y/o con capacidades de tolerancia a fallos. Adicionalmente este trabajo define las unidades de *configuración* y *funcional* de la célula, las cuales implementan las capacidades auto-adaptables y de procesamiento en paralelo respectivamente.

La implementación física de esta arquitectura esta compuesta de dos capas, que incluyen células interconectadas en el primer nivel y matrices de conmutación y pines en el segundo nivel. Las células ejecutan la funcionalidad básica del sistema. Diversos chips pueden ser interconectados para aumentar la matriz de células en el sistema. Cualquier aplicación que se quiera programar en el sistema debe estar organizada en componentes, donde cada componente está compuesto por una o más células interconectadas. La interconexión de células dentro de un componente es realizado en el mismo nivel de la matriz de células (primera capa), mientras que la interconexión de componentes es realizada en la segunda capa. Adicionalmente, se definen dos capas conceptuales que son usadas con propósitos organizativos en aplicaciones de propósito general, estas son: el SANE y el SANE-assembly. La *entidad auto-adaptable interconectada* o SANE (Self-Adaptive Networked Entity) está compuesta por un grupo de componentes. Este es el sistema de computación auto-adaptable básico, el cual tiene la habilidad de monitorizar su entorno local y su proceso de computación interno. El *Conjunto de SANEs* o SANE ASSEMBLY (SANE-ASM) esta compuesto por un grupo de SANEs interconectados.

Las capacidades de procesamiento de la célula están incluidas en su unidad funcional o Functional Unit (FU). Esta puede ser definida como un multicomputador configurable con cuatro núcleos. La FU tiene doce modos de configuración programables, por lo que cada célula permite seleccionar entre uno y cuatro procesadores trabajando en paralelo con diversas capacidades en las memorias de programa y datos. Los núcleos son agrupados o no dependiendo del modo de configuración, permitiendo que la memoria de programa pueda implementar 64, 128, 192 o 256 instrucciones. De manera similar, la memoria de datos puede ser expandida a lo largo y/o ancho, permitiendo procesamiento de datos para 8, 16, 24 y 32 bits.

Las capacidades auto-adaptables de la célula son ejecutadas principalmente por la unidad de configuración de la célula o Cell Configuration Unit (CCU). El algoritmo de auto-emplazamiento

es el encargado de encontrar la posición mas adecuada dentro de la matriz de células para insertar la nueva célula de un componente. El algoritmo de auto-enrutamiento es ejecutado a partir de la inserción de la segunda célula de un componente, cada vez que el algoritmo de auto-emplazamiento termina. Este algoritmo permite interconectar los puertos de las FU de dos células a través de los puertos de la célula. Los procesos de auto-emplazamiento y auto-enrutamiento permiten realizar en tiempo real cambios funcionales complejos; estos procesos dotan al sistema de una mayor funcionalidad, permitiendo que el sistema cambie por si mismo, lo que permite la implementación de la auto-configuración en tiempo real, sin la necesidad de ningún gestor de configuración. La ausencia de un sistema de supervisión centralizado permite a las células realizar algunas de sus funciones de manera distribuida.

La arquitectura propuesta incluye dos mecanismos de tolerancia a fallos. Uno de estos es una técnica escalonada y dinámica de tolerancia a fallos, que tiene la habilidad de crear y eliminar copias redundantes de la unidad funcional (o de cómputo) de una aplicación específica. Esto es posible gracias a la capacidad de auto-configuración en tiempo real incluida en la arquitectura. El otro mecanismo de tolerancia a fallos es el Sistema de Tolerancia a Fallos dedicado o estático. Este provee capacidades de procesamiento redundante que están en funcionamiento continuamente. Cuando un fallo en la ejecución de un programa es detectado, los procesadores de la célula son detenidos y los procesos de auto-eliminación y auto-replicación se inician para la célula (o células) implicada en el fallo. Esta(s) célula(s) serán auto-descartadas para futuros procesos de auto-emplazamiento.

Se desarrolló un prototipo basado en FPGAs y una herramienta de software para comprobar la funcionalidad del sistema. El prototipo incluye todas las características de los sistemas auto-adaptable descritas en este documento. El prototipo ha sido desarrollado en dos chips, cada uno es una FPGA Virtex4 de Xilinx (XC4VLX60). El diseño físico incluye el arreglo de células junto a un sistema de enrutamiento a nivel de componentes. Adicionalmente incluye un microprocesador de control o CµP (Control Microprocessor) y otros periféricos, que dan soporte a la implementación de aplicaciones de propósito general en el prototipo.

Con el propósito de tener un sistema de desarrollo completo, la herramienta de software SPD (SANE Project Developer) ha sido desarrollada. El SPD es un ambiente integrado de desarrollo (Integrated Development Environment o IDE) que permite generar y descargar la memoria de inicialización de datos para el CµP dentro del prototipo. El SPD permite la creación y edición de archivos que describen la configuración de un SANE-ASM. El archivo principal incluye instrucciones especiales para el SANE-ASM (archivos SASM), el cual es equivalente al lenguaje de ensamblador de un procesador clásico. El SPD permite la creación y edición de toda la información relacionada con el SANE-ASM, así mismo construye de manera automática el archivo hexadecimal de configuración que será descargado a la FPGA del prototipo.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Fall Down Seven Times, Get Up Eight.
Cae siete veces, levántate ocho.
Japanese Proverb – Proverbio Japonés

Abstract: This chapter is an introduction to adaptive and parallel processing systems. It includes mainly a theoretical framework, architecture overview and contributions, preliminary and related works.

Self-adaptation is defined as the ability of a system to react to its environment in order to optimize its performance. The AETHER project (Self-Adaptive Embedded Technologies for Pervasive Computing Architectures) [1] was a notable initiative in the study of novel self-adaptive computing technologies for future embedded and pervasive applications.

This work started as a contribution of the hardware platform for the AETHER project. After finalization of this project, I continue with the investigation introducing additional contributions to the initial platform developed.

One of the purposes of the AETHER project (including this work) is to show that self-adaptive computing architectures can be a powerful approach to simultaneously addressing the major problems raised by pervasive computing. In particular, it aims to tackle the issues related to parallel processing, self-adaptive capabilities and technological scalability, increased complexity and programmability of future embedded computing architectures by introducing self-adaptive technologies in computing resources.

1.1 Adaptive and Bioinspired Systems

An adaptive system consists of a set of interacting entities, which form an integrated whole that is able to respond to environmental changes or changes in the interacting parts. Adaptive systems are closely tied with the concept of bioinspired systems, that are systems built using configurable hardware and electronic instruments, that emulates the capabilities of the biological systems to process information and solve problems.

These features are relevant in research areas such as embedded systems and pervasive computing among others [2]. Some important features like self-organization and self-configuration are linked to higher computational requirements, where adaptive system architectures are formed as a promise in the evolution of classical computing systems. Self-adaptive computing architectures can be a powerful approach to simultaneously address the major problems raised by pervasive computing.

In coming years virtually every object will have a processing power, where the processing resources will require greater flexibility and scalability to meet the various needs of users. Adaptive computing systems offer the ability to adequate all or part of its architecture with applications that include changing needs or changing environments.

Adaptable architectures are closely linked to the concept of parallelism and reconfigurability, theoretically allowing greater efficiency for development of general purpose applications. According to [2] adaptive systems should have the following characteristics of bioinspired systems: self-configuration, adaptivity, self-distribution, self-organization, self-healing, automatic parallelization, accounting, self-protection and protection of others.

Self-healing is a special feature of an adaptive system, where hardware failures should be detected, handled and corrected by the system automatically. A fault tolerance system in an adaptive system together with other self-adaptive capabilities could provide this functionality.

1.2 Self-Adaptive capabilities in the proposed architecture

Self-configuration is a basic principle that permits a programmable or configurable system to modify autonomously its functionality at a given time [3]. This modification is usually driven by an optimization process that tries to match the behavior of the system with the constraints posed to the application it is intended to solve. The main characteristic to be present in the actual self-adaptive system is the capability of determining its configuration at a given time in an autonomous and distributed way by the system members (cells). This implies that the following properties should be supported at the hardware level by any architecture intended to be used as an efficient platform for self-adaptive principles: dynamic and distributed self-routing [4] [5] [6], dynamic and distributed self-placement, scalability and distributed control.

The self-placement and self-routing processes, due to its nature, enable the systems with runtime reconfiguration, self-repair and/or fault tolerance capabilities. This processes allow for performing complex functionality changes in real time, beyond the programmed context changes, currently common in the FPGA domain. The proposed self-placement and self-routing processes endow the system with enhanced functionality, making it possible for the system to change by itself, without the need for any configuration manager, as needed in current FPGAs. The absence of a centralized supervision system allows performing some of the tasks in a distributed way.

1.3 Architectures for parallel computing

Flynn's taxonomy [7] is probably the most common way to classify computer architectures with respect to their parallelism, based on the instruction and data flows. These streams are independent, so there are four possible combinations in parallel computing (see Table 1.1).

The model SISD (Single Instruction Single Data) based on the Von Neumann machine is the classic computing architecture; it uses a processor that is capable of performing actions sequentially, making different types of operations (arithmetic, logical, shifts, etc.) between data memory and processor registers. Although significant improvements have been implemented, like pipelining, prefetching, RISC architectures, code optimizers and others, it is expected to slow the pace of improvement, mainly due to physical implementation constraints. On the other hand, the need to solve new problems has increased, which demands high computational loads, so that the development of new parallel processing system acquires significant importance.

In a SIMD (Single Instruction Multiple Data) model a single program controls the processors using multiple data streams to perform operations that can be parallelized in a natural way. This type of architecture is useful in uniform applications, as in image processing, where it is necessary to apply the same function to many pixels simultaneously.

Name	Instructions Flow	Data Flow	Example, application
SISD	1	1	Classic computing architectures: Von Neumann, Harvard, PCs.
SIMD	1	Multiple	Vector processors, graphics cards.
MISD	Multiple	1	Uncommon, it is used in situations of redundant parallelism (Air Navigation)
MIMD	**Multiple**	**Multiple**	**Multiprocessors, Multicomputer.**

Table 1.1: Flynn's taxonomy: classification of computer architectures with respect to its parallelism.

The model MISD (Multiple Instruction Single Data), where many functional units perform different operations on the same data, is often used in situations of redundant parallelism , as in the case of air navigation, but due of its features it has been poorly implemented by industry.

Architecture MIMD (Multiple Instruction Multiple Data) is characterized by a set of processors executing different instruction sequences simultaneously on different data sets. This architecture can be classified in two, depending on the type of memory access, so you can have MIMD for shared memory (multiprocessor) and MIMD for distributed memory (multicomputer) [8]. The main advantage of the architecture MIMD over the SIMD architecture is that they have greater flexibility and applicability. However, the architecture MIMD is harder to configure and control [9]. The architecture MIMD due to its high degree of parallelism is emerging as the most suitable for the implementation of bioinspired systems.

1.4 Preliminary work

1.4.1 POEtic

The POEtic project [10] [5] tackled the development of a flexible computational substrate inspired by the evolutionary, developmental and learning phases in biological systems. The device is organized as a custom 32-bit RISC microprocessor and a custom FPGA. The internal architecture developed for the device is scalable, making it possible to construct a physical hardware platform whose size matches the requirements of the application to be implemented.

This project is based in essentially three biological models [11]: phylogenesis (P), the history of the evolution of the species, ontogenesis (O), the development of an individual as orchestrated by his genetic code, and epigenesis (E), the development of an individual through learning processes.

The POEtic architecture is divided in three parts, the environment subsystem, the organic subsystem and system interface. The environment subsystem of the POEtic tissue has been built around a custom 32-bit microprocessor with an efficient and flexible system bus and several custom peripherals. The organic subsystem is composed of two layers, a two-dimensional array of basic elements, called molecules, and a two-dimensional array of routing units. Each molecule is connected to its four neighbors in a regular structure. It is composed of a 16-bit LUT and a Flip Flop (DFF), which has the ability to access the routing layer which is used for communication between molecules. The second layer implements a dynamic routing algorithm that enables the creation of data paths between molecules at runtime. The dynamic routing system is designed to automatically connect the inputs and outputs of the molecules. The system interface allows

the communication between the environment and the organic subsystem of the tissue.

1.4.2 PERPLEXUS

The Perplexus project [12] [13] aims to develop a scalable hardware platform made of custom reconfigurable devices endowed with bioinspired capabilities that will enable the simulation of large-scale complex systems and the study of emergent complex behaviors in a virtually unbounded wireless network of computing modules.

The Perplexus project defines its platform as a network of ubidules (ubiquitous computing modules), which are equipped with wireless communication capabilities and important sensory elements. The project identifies three areas where the modeled structure can provide its functionality as a new and powerful simulation tool; these are: neuro-genetic computational modeling, study of culture diffusion and social robots.

1.5 State of the art

This section describes some projects that include similar features to the one presented here.

1.5.1 Confetti

Among the projects that have proposed architectures for advanced multiprocessor systems it is worth mentioning Confetti [14] [15], that is based on a scalable array of homogeneous processing nodes physically arranged as a networking computer mesh. This architecture is a dual-layered array of FPGAs, with a layer dedicated to processing (ECell) and the other to networking (ERouting). The basic computation element (ECell) is implemented in an FPGA. The networking level is implemented in a board specifically designed for this purpose. The principal difference with the proposed architecture is the scalability. The ECell has a higher processing capacity than the processing elements (cells) presented in this document. The main limitation of the Confetti architecture is the physical implementation of a very large number of processing elements. The architecture presented here can work with hundreds or millions of cells without architectural modifications. However, the boards used for the prototype don't permit the implementation of a large number of cells.

1.5.2 eDNA

The eDNA [16] presents the concept of a bioinspired reconfigurable hardware cell architecture that supports self-organization and self-healing. In order to validate the algorithms for self-organization and self-healing, the authors wrote a simulator to provide a fast method to examine the behavior of the proposed algorithms. All algorithms were based on the idea that they should run on processing elements (eCells) which used a NoC as communication medium. This approach provides an interesting starting point for future study and possible implementation of other self-adaptive capabilities to the system proposed in this work.

1.5.3 Self-routing reconfigurable fault-tolerant cell array

The work presented in [17] represents a self-routing reconfigurable fault-tolerant cell array. The reconfigurable and fault-tolerant cellular structure is based on a cell array. It comprises functional cells with spare cells having the same hardware structure. The functional cells can be configured with arithmetic or logical functions. The interconnection of functional cells can thus accomplish a complex task, as specified by the user. Compared with this work, our architecture only implements

redundancy when needed, due to its dynamic fault tolerance capability. This permits to have free resources that could be used for other processes inside the system.

1.5.4 CEDAR

The Configurable Embedded Distributed Architecture (CEDAR) [18] implements an adaptive routing strategy based on ACO (Ant Colony Optimization). Similar to our architecture, the CEDAR platform consists of an array of homogeneous Processing Elements (PE). Each PE can be configured as a computing or routing node. The main difference with the SANE architecture presented here is that each processing element (cell) provides processing and routing capabilities for the interconnection of neighbors and/or remote cells.

1.5.5 Amorphous

The amorphous computing [19] medium is a system of irregularly placed, asynchronous, locally interacting computing elements. The system can model this medium as a collection of computational particles sprinkled irregularly on a surface or mixed throughout a volume. The physical implementation of this system is quite difficult and therefore prevents from a future physical realization.

1.5.6 Cell Processors - Sony-Toshiba-IBM team

Other commercial implementations, like the Cell Processors [20] developed by Sony-Toshiba-IBM team, implement a SIMD architecture processor that consists of a 64-bit Power microprocessor coupled with multiple processors, a flexible IO interface, and a memory interface controller that supports multiple operating systems. Despite their capacity, SIMD architectures show limitations in general-purpose computing. Our self-adaptive system implements a MIMD architecture, whose processing capacity is configurable between 8, 16 , 24 and 32 bits, and additionally each computing unit is able to execute small processing threads.

1.5.7 ADRES

The Architecture for Dynamically Reconfigurable Embedded System (ADRES) [21] [22] is an architecture that tightly couples a VLIW processor and a coarse-grained reconfigurable matrix.

The ADRES core consists of many basic components, e.g., Functional Units (FUs) and register files (RF). The whole ADRES core has two functional views: the VLIW processor and the reconfigurable matrix. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way, whereas the VLIW executes the non-kernel code by exploiting instruction-level parallelism (ILP). These two functional views share some resources because their executions will never overlap with each other thanks to the processor/co-processor model.

For the VLIW part, several FUs are allocated and connected together through one multi-port register file, which is typical for a VLIW architecture. For the reconfigurable matrix, apart from the FUs and RF shared with the VLIW processor, there are a number of reconfigurable cells (RC) which basically comprise FUs and RFs too.

1.5.8 MorphoSys

MorphoSys [23] [24] [25] is a reconfigurable processing system targeted at data-parallel and computation-intensive applications. The MorphoSys architecture comprises five major components: the Reconfigurable Cell Array (RC Array), control processor (TinyRISC), Context Memory, Frame Buffer and a DMA Controller.

The reconfigurable component of MorphoSys is an array of reconfigurable cells (RCs) or processing elements. The RC Array has 64 cells in a two-dimensional matrix (8x8). The RC Array follows the SIMD model of computation. All RCs in the same row/column share same configuration data (context). However, each RC operates on different data.

The major component of MorphoSys is the Reconfigurable Cell (RC). Each RC incorporates an ALU-multiplier, a shift unit, input muxes and a register file. In addition, there is a context register that is used to store the current context and provide control/configuration signals to the RC components.

The control processor (TinyRISC) is a MIPS-like processor with a 4-stage scalar pipeline. It has a 32-bit ALU, register file and an on-chip data cache memory. This processor also coordinates system operation and controls its interface with the external world. The Context Memory stores multiple planes of configuration data (context) for RC Array, thus providing depth of programmability. The system incorporates a high-speed memory interface consisting of a streaming buffer (Frame Buffer) and a DMA controller. The Frame Buffer has two sets, which work in complementary fashion to enable overlap of data transfers with RC Array execution.

1.5.9 REMARC

Reconfigurable Multimedia Array Coprocessor (REMARC) [26] is a reconfigurable coprocessor that is tightly coupled to a main RISC processor and consists of a global control unit and 64 16-bit programmable logic blocks called nano-processors. REMARC is a SIMD architecture that is designed to accelerate multimedia applications, such as video compression, decompression, and image processing.

The base architecture of REMARC uses MIPS-II ISA. Coprocessor 0 is used for memory management, coprocessor 1 is used for floating point and REMARC operates as coprocessor 2. With REMARC, users can define and configure their own instructions specialized for their application.

REMARC consist of an 8x8 array of nano-processors and a global configuration unit. Each nano-processor has a 32-entry instruction RAM, a 16 bit ALU, a 16-entry data RAM, and 13 16-bit data registers. The global control unit controls the nano-processors and the transfer of data between the main processor and the nano-processors.

1.5.10 XPP (eXtreme Processing Platform)

The eXtreme Processing Platform (XPP) [27], [28] is a new runtime-reconfigurable data processing architecture. It is based on a hierarchical array of coarse grain, adaptive computing elements called processing array elements (PAEs), and a packet-oriented communication network.

An XPP device contains one or several processing array clusters (PACs), which includes rectangular blocks of PAEs. A typical XPP device contains four PACs. Each PAC is attached to a Configuration Manager (CM) responsible for writing configuration data into the configurable objects of the PAC. The XPP architecture is also designed for cascading multiple devices in a multichip setup. The PAE contains back registers, forward registers and an ALU object which performs the actual computations. PAE objects communicate via a packet-oriented network.

According to the authors, the strength of the XPP technology originates from the combination of array processing with unique, powerful run-time reconfiguration mechanisms. Parts of the array can be configured rapidly in parallel while neighboring computing elements are processing data. Reconfiguration is triggered externally or even by special event signals originating within the array, enabling self-reconfiguring designs. The architecture is designed to support different types of parallelism: pipelining, instruction level, data flow, and task level parallelism.

The high-level compiler for this architecture is called XPP-VC (XPP Vectorizing C Compiler). It uses new mapping techniques, combined with efficient vectorization. A temporal partitioning phase guarantees the compilation of programs with unlimited complexity, provided that only the supported C subset is used.

1.5.11 The SANE Virtual Processor (SVP)

The SANE Virtual Processor (SVP) [29][30] was defined as a concurrent programming model developed and used at the University of Amsterdam as a basis for designing and programming many-core chips. The model is defined by a small number of actions used to create and asynchronously manage the execution of concurrent SVP programs. These actions capture concurrency, implicit communication and resource management, and using these abstractions they aim to develop an understanding of self-adaptive computational systems in the AETHER collaborative European project [1].

The SVP model provides five actions in order to create and manage concurrency. These actions replace those normally used to construct sequential programs (loops and calls). Three of these actions are used to parallelize sequential programs (create, sync and break), and the other two are used for concurrency engineering (squeeze, kill), i.e., the self-adaptive aspects of the model. The create action defines a family of threads based on a single fragment of code. The result of the create action is the creation of an ordered set of thread contexts defined by parameters to the action. These parameters define the code used, the size of the context required and the number of threads to be created.

A thread creating a subordinate family can detect its termination using an SVP sync action. This identifies the family by name so that multiple concurrent families can be created and synchronized from within a single thread. The sync action provides a return code that specifies how a family was terminated and can provide a return value in the case of a break action.

The break action is provided to allow for the creation of dynamically bounded families of threads. In such circumstances, a semi-infinite range of index values is specified in the family's parameters and any thread in the family may terminate the creation of new contexts using the break action and return a scalar value (e.g. an index or pointer) back to the creating thread via the sync action. This construct is the SVP concurrent equivalent of a while loop in a sequential program.

Any thread that can identify a family and the place where it is executing and can provide the capability generated on its creation, may send a kill signal to that family and force its termination. The squeeze and kill signals are similar, but the squeeze maintains the family's state. This is a form of preemption of the unit of work that the family represents and it allows the family to be restarted by re-creating it using the state captured when it was squeezed.

1.5.12 HTHREADS

Hthreads [31] [32] is a unifying programming model for specifying application threads running within a hybrid CPU/FPGA system. This system provides unique capabilities within the reconfigurable computing community by enabling concurrent execution of threads specified through a set of pthreads compatible API library routines to be automatically compiled, synthesized, and seamlessly executed on a CPU/FPGA hybrid chip.

The thread interface used by hthreads is based around three major tasks: management, scheduling, and synchronization. For its implementation, they have developed three hardware based state machines that are executed in true parallel with the others and with the application itself. This provides coarse-grained parallelism which is needed for high performance.

Although this is a CPU/FPGA hybrid architecture that allows the definition of threads, in both software and hardware level in the same code. The hthreads model does not provide the advantages of the microthreads model [29], due to its compatibility with the architecture presented in this document.

1.6 Architecture Overview and Contributions

This hardware architecture developed within the framework of the AETHER project [1] differs from other architectures due mainly to the self-adaptive features implemented, that are executed autonomously and in a distributed way by the system members (cells). Basically, this is a novel unconventional MIMD hardware architecture [7] with self-adaptive capabilities including self-placement and self-routing which, due to its intrinsic design, enable the development of systems with runtime reconfiguration, self-repair and/or fault tolerance capabilities

One of the main features of this architecture is its high degree of parallelism. The major drawback is the configuration of complex applications, where many processors have to be programmed and synchronized in order to accomplish a specific task. A new high-level programming paradigm has to be implemented, with the purpose of obtaining the maximum performance of the architecture.

The architecture proposed includes two mechanisms of fault tolerance. One of these is the static Fault Tolerance System [33] [34]. It provides redundant processing capabilities that are working continuously. When a failure in the execution of a program is detected, the processors of the cell are stopped and the self-elimination and self-replication processes starts for the cell (or cells) involved in the failure. This cell(s) will be self-discarded for future self-placement processes.

The other mechanism of fault tolerance is the Dynamic Fault Tolerance Scaling Technique [35], which permits a given subsystem to modify autonomously its structure in order to achieve fault detection and fault recovery. It has the ability to create and eliminate the redundant copies of the functional section of a specific application.

In this document we present a detailed description of the hardware architecture and the self-adaptive algorithms. An FPGA-based prototype has been built for demonstration purposes. Also we present the main features of the software tool *SANE Project Developer*, an integrated development environment that allows the creation, configuration, compilation, programing and test of general purpose applications in the FPGA-based prototype. Although this parallel processing architecture is appropriate for applications requiring fault tolerance mechanisms, it is also suitable for the development of any general purpose application that requires a high degree of parallel processing.

1.6.1 Scalability

The system presented here is widely scalable, theoretically it allows to deploy as many cells as its addressing system allows. This represents a number of cells in the system close to 2^{32}.

In a system with a large number of cells, the cost of this scalability is represented in the propagation time of a combinational signal in the system (one logic gate per cell), this is reflected in the operation frequency of the system. This cost in the propagation time across the system is represented by the addition of the number of rows and columns of the cell array.

The main problem for the implementation of a large amount of cells is the granularity of the system and the test tools available. The granularity of the prototype and the hardware tools available only allow for testing the architecture with few cells, but in a near future, with the evolution of the technology in the manufacturing processes for integrated circuits, the test of a large cell array can be performed.

1.7 Document Organization

This document is organized as follows:

- Chapter 1 is the introduction, which includes some architecture generalities, preliminary and related works, and a theoretical framework of relevant aspects of the dissertation.
- Chapter 2 presents a detailed description of the system architecture from the hardware point of view.
- Chapter 3 describes the architecture of the Functional Unit, which provides the processing capabilities to the system. In addition, annexes A and B presents in detail the instruction set and the description of data memory registers respectively.
- Chapter 4 details all the self-adaptive processes implemented in the system. Appendix C presents the flow diagrams for self-adaptive algorithms implemented in system.
- Chapter 5 presents the high-level instructions that permit the implementation of applications in the system. Additionally, two example applications are shown, which include all functionalities implemented in the system. Appendix D shows the listings of these examples, and appendix E presents the software tool developed for implementing applications in the system.

1.8 Conclusions

This chapter describes the general concepts of adaptive and bioinspired systems, as well as the typical classification of computer architectures with respect to its parallelism. These are basic concepts for the architecture proposed in this dissertation. Additionally, preliminary works are presented, which shows the starting point for the evolution of this project. The state of the art presents other projects with similar contributions to the presented here, at both hardware and software levels. Some of these projects can be useful as reference for future evolution of this self-adaptive architecture.

Chapter 2

System Architecture

In theory, there is no difference between theory and practice. But in practice, there is.

En teoría, no hay diferencia entre teoría y práctica. Pero en la práctica, sí que la hay.

Jan L.A. van de Snepscheut (1953 – 1994)

Abstract: This chapter presents a detailed description of all hardware components that compose the self-adaptive architecture presented in this dissertation. The system is presented in a top-down approach, starting for a general overview of the system and later specifying the subsystems or components involved in the architecture.

2.1 Conceptual organization

The proposed architecture consists of four conceptual layers (Figure 2.1). The bottom layer is composed of cells that implement the self-adaptive capabilities and provides the computing capacity of the system. The second one is the component layer, where each component is composed of interconnected cells. The third one is the Self-Adaptive Networked Entity (SANE) layer, which consists of a group of interconnected components, and the top layer, the SANE ASSEMBLY (SANE-ASM) is composed of a group of interconnected SANEs.

The SANE is the basic self-adaptive computing system; it has the ability of monitoring its local environment and its internal computation process.

2.2 Overview for the configuration of an application

Any application scheduled to the SANE has to be organized in components, where each component is composed by one or more interconnected cells. The interconnection of cells inside of a component is made at cell level, while the physical interconnections of components are made in another layer, at the Switch Matrix (SM) level. The connections between components can be inside a chip or may span several chips. The interconnection of SANEs is just conceptual, because it takes place at the same layer of components.

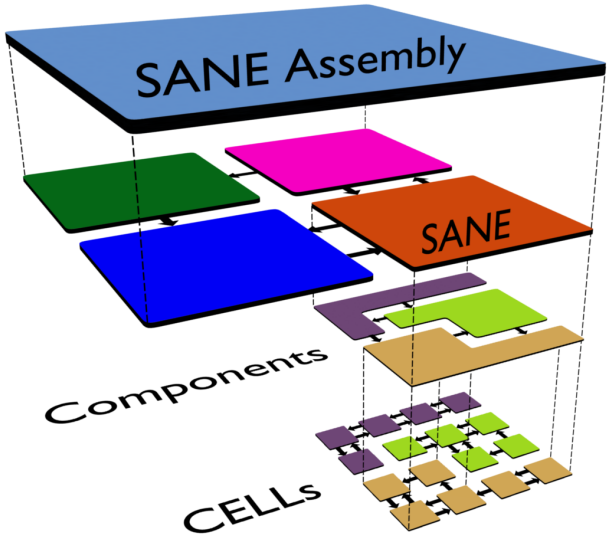


Figure 2.1: Conceptual layers of the self-adaptive hardware architecture.

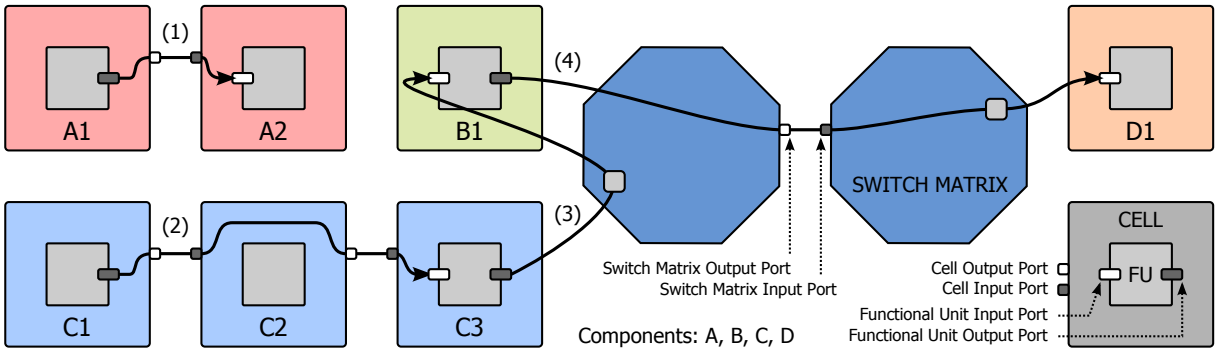


Figure 2.2: Possible connection between Functional Unit ports of two cells.

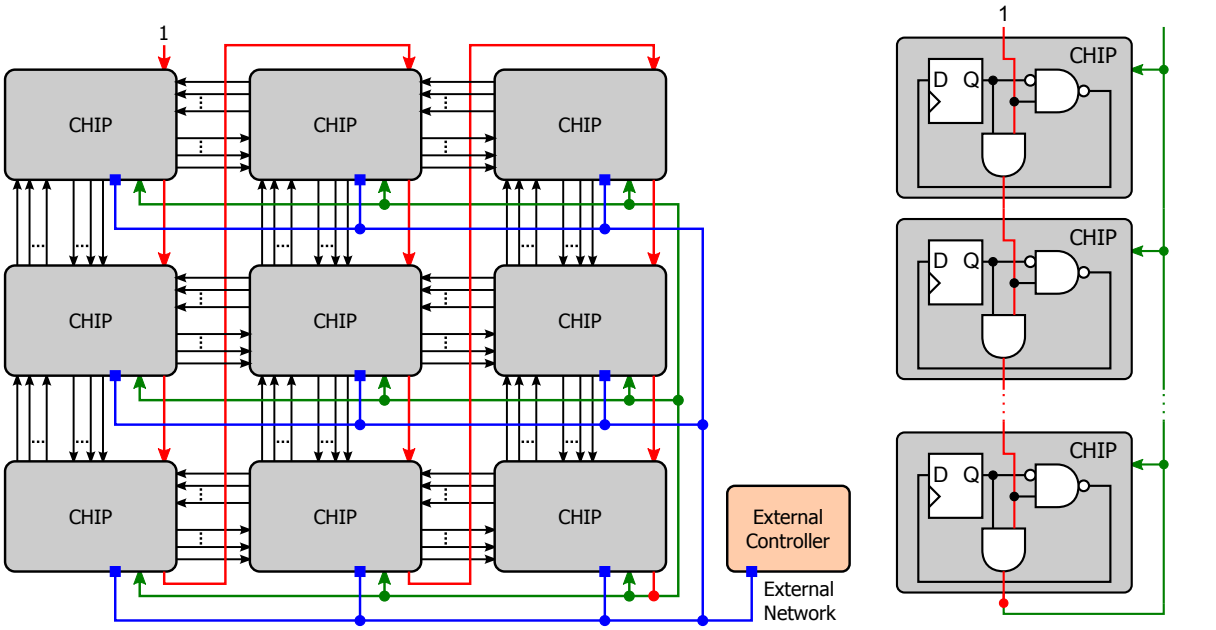
2.2.1 Connection of cells

Each cell includes a Functional Unit (FU), which provides the processing capabilities to the cell. The purpose of a *connection* between two cells is interconnect the output port of the FU of a cell with the input port of the FU of another cell. The system provides the necessary hardware for the interconnections of FU ports of two cells as shown in Figure 2.2. This figure presents the following scenarios:

(1) Connections between two neighbor cells of the same component.
(2) Connections between two not neighbor cells of the same component.
(3) Connections between two cells of different components that belong to the same SM.
(4) Connections between two cells of different components in different SMs.

Note that connections of cells in the same component are made through cell ports, while connections of cells of different components are made through one or more SMs.

During the initial stages of the architectural development a software tool was developed, which allowed us to determine through exhaustive simulations the most appropriate number of cell ports, the number of SM ports and the number of cells per SM [3]. The following sections details all these features.



(a) Global overview of the system.

(b) Circuit to select the winner chip.

Figure 2.3: System architecture

2.3 Overview of System Architecture

The physical implementation of this architecture is composed of one or more interconnected chips, each of which includes a two-layers implementation with interconnected cells in the first level and interconnected Switch Matrices in the second level (components layer). These represents the first two layer of the conceptual organization. The SANE and SANE-ASM are just conceptual and are implemented in the same layer of components.

The system architecture is composed of interconnected chips and an External Controller (EC) as depicted in Figure 2.3a. The ports that interconnect the chips and the EC can be divided in three groups: *data*, *network* and *chip_selection*. The *data* ports are composed of n buses of 9 bit each. The number of data buses that interconnect two chips depends of the internal capacity of the chip as will be explained later in this chapter. The *network* ports constitutes the External Network (ENET), which allows the information exchange between chips and EC. The *chip_selection* ports allows to select the chip with higher priority between several candidates for the execution of a specific process. Figure 2.3b shows the implementation of the selection process for chips. The chips that participate in the selection process set to zero its flip flop. After a clock pulse the flip flop that remains in low level will be the winner and the chip starts the execution of the scheduled process. If there is not a winner, the feedback line remains in high level and the process ends.

The EC broadcasts frames through the ENET that contains the necessary information for the implementation of an application, or configuration data for a specific cell. All chips receive the information and performs one of two operations: retransmit the information inside the chip, or evaluate if they are candidates for the execution of a specific process e.g. for the implementation of a new component. The chip candidates take part of the selection process and the winner chip executes the process.

2.4 Chip Architecture

The chip architecture is depicted in Figure 2.4, which shows the representation of a chip that includes an array of clusters, Pin Interconnection Matrices (PIMs) and a Global Configuration Unit (GCU). Figure 2.5 shows a 3D representation of an extensive array of clusters. This two-layer implementation is composed by interconnected cells at the first level and interconnected switch and pin matrices in the second level.

Several chips can be interconnected by means of input and output ports of the PIM and the ENET, which is connected to the GCU. Inside the chip, the GCU is interconnected with the cell array and PIM by means of an Internal Network (INET). This is used for the information exchange between cells, PIM and GCU, and is used to support all internal processes in the chip.

The Internal and External networks give support to all self-adaptive processes in execution time, like self-placement, self-routing and real-time self-configuration processes among others.

2.5 Global Configuration Unit

The Global Configuration Unit (GCU) is in charge of controlling the self-adaptive processes inside the chip. The GCU is connected with the EC through the ENET and with the internal components of the chip through the INET. The GCU receives from the EC information related to self-adaptive processes or configuration data. Depending on the information, the GCU could translate it and make a broadcast inside the chip, or start a negotiation between chips, e.g. for defining the destination of a component. After negotiation, the winner chip by means of its GCU controls the self-adaptive processes inside the chip and sends a confirmation command to the EC when the process ends.

The participation of the GCU in the self-adaptive processes will be treated subsequently throughout this document.

2.6 Cluster

Figure 2.6 shows a cluster, that is composed by a 3x3 cell array and a Switch Matrix (SM). Inside a cluster, each cell is identified by means of a letter as shown (A, B, C, D, E, F, G, H and I).

The cells are interconnected with their four direct neighbors in directions *North, East, South,* and *West*. The SMs are interconnected with their eight direct SMs neighbors or PIMs in directions *North, NorthEast, East, SouthEast, South, SoutWest, West,* and *NorthWest* as shown in Figure 2.4.

2.7 Cell Architecture

The cell is the basic element of the proposed self-adaptive architecture. Therefore, the cell has to include the necessary hardware to carry out the basic principles of self-adaptation; dynamic and distributed self-routing [4] [5] [6], dynamic and distributed self-placement, scalability and distributed control.

The cell architecture and port distribution are depicted in Figure 2.7. The cell consists of the Functional Unit (FU), the Cell Configuration Unit (CCU) and multiplexers that allow the interconnection between FU ports of two cells.

The cell is interconnected with its four direct neighbors by means of local, remote and expansion ports. The cell has eight local input ports, eight local output ports, twelve remote input ports and twelve remote output ports equally distributed in the four sides of the cell, each

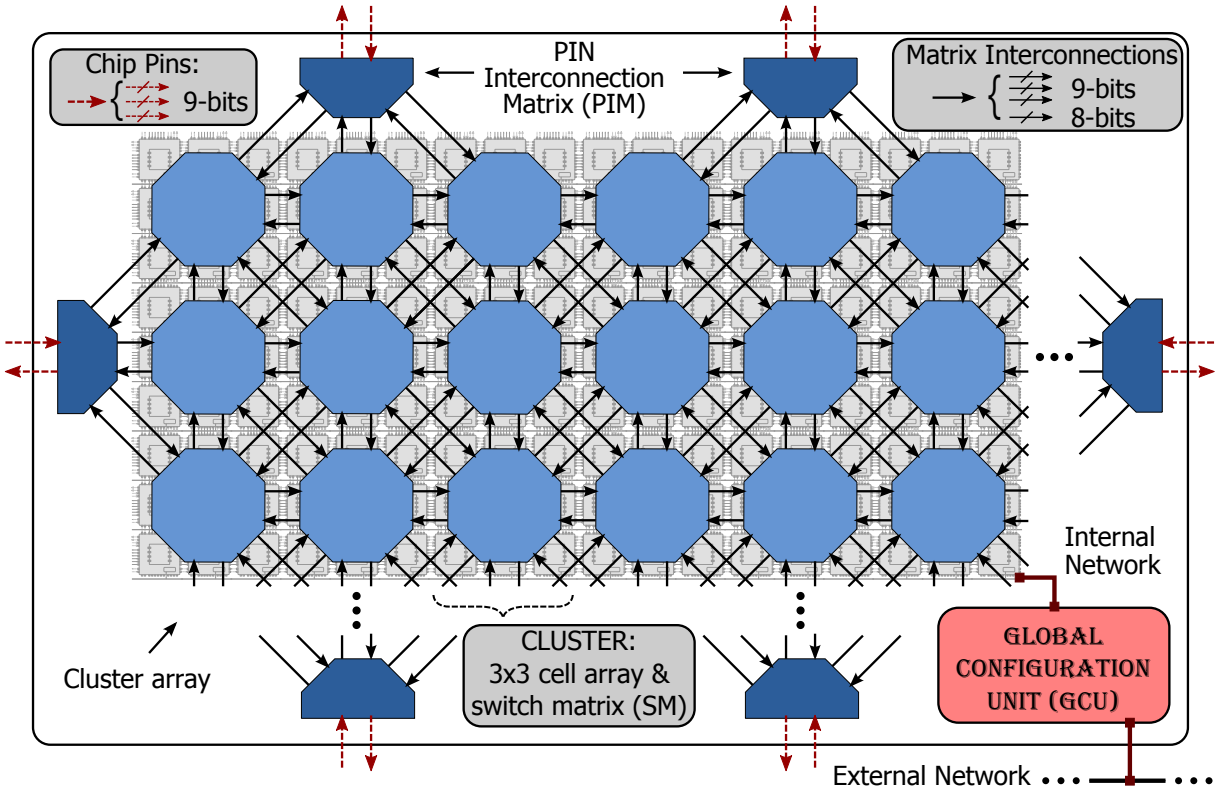


Figure 2.4: Organization of the proposed architecture inside a chip.

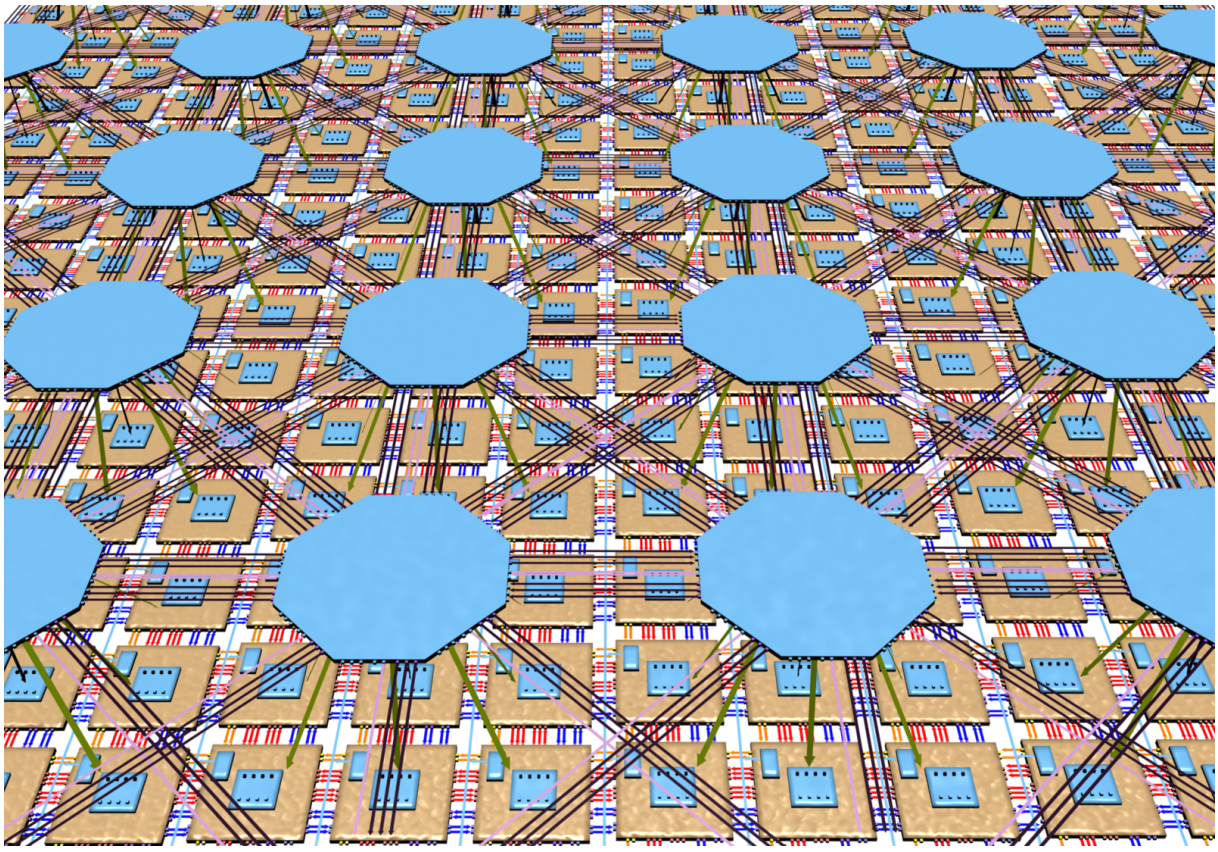


Figure 2.5: System architecture: 3D representation of an array of clusters.

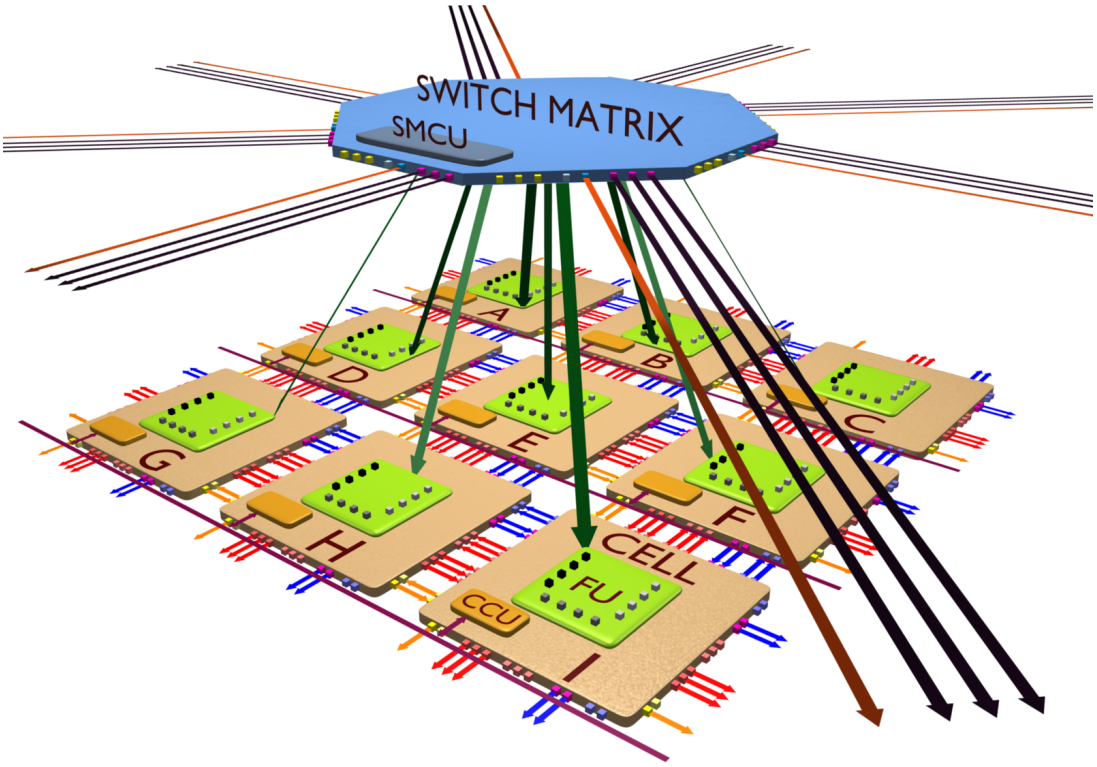


Figure 2.6: Cluster: 3x3 cell array and switch matrix.

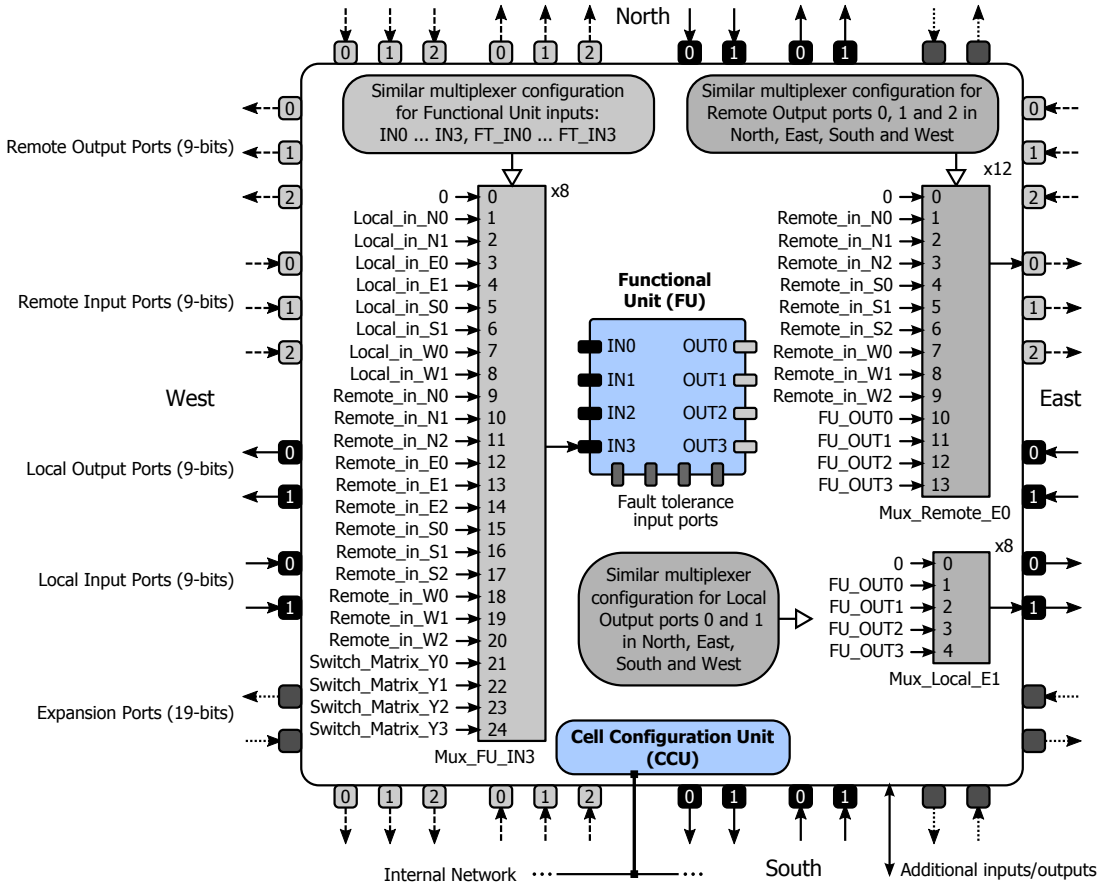


Figure 2.7: Cell architecture: internal hardware and ports.

one being a 9-bit wide bus, 8-bit for data and 1 bit for Read Enable (RE) flag. Additionally, the cell has four expansion input ports, four expansion output ports, a connection with the INET and additional input/output ports.

The cell includes additional ports interconnected with the SM: four 9-bits input ports, four 9-bit output ports (FU output ports), one expansion input port and one expansion output port.

The cell includes 28 multiplexers divided in three groups:

- 8 multiplexers connected to the local output ports. These multiplexers allow connecting the FU output ports with the local output port of the cell. These multiplexers are used exclusively to connect neighboring cells.
- 12 multiplexers connected to the remote output ports. These multiplexers allow connecting the remote output port with: (1) the FU output ports and (2) the remote input ports of the other sides of the cell, e.g., if the remote output port is in east, the multiplexer allows connecting the remote input ports from north, south and west. The remote port is used for connecting neighboring or remote cells.
- 8 multiplexers connected to the FU input ports. These multiplexers are used to connect local and remote cell input ports when the connection between two cells is in the same component (inputs 1 to 20). They additionally allow connecting FU ports of two different components using the inputs coming from the SM (inputs 21 to 24).

2.7.1 Functional Unit (FU)

The FU is in charge of executing the processes scheduled to the cell. The FU can be described as a four-core configurable multicomputer [8]. The FU has four 9-bit input ports and four 9-bit output ports. There are four additional input ports (*ft_input_ports*) used exclusively for the static Fault Tolerance System (FTS) [33].

The internal architecture of the FU is detailed in chapter 3 and annexes A and B. The FTS is described in section 3.7.

2.7.2 Cell Configuration Unit (CCU)

Using a distributed working principle, the CCUs of the cells in the array are responsible for the execution of the required algorithms for the implementation of the system self-adaptive capabilities, specifically the self-placement and self-routing algorithms. These algorithms are executed by the CCU using the INET and the expansion ports, and they are explained in detail in chapter 4.

2.8 Switch Matrix

The Switch Matrix (SM) allows connecting cells from two different components. Figure 2.8 shows the port distribution and the internal hardware that is included in the SM.

The SM is connected to its eight adjacent neighbors, each through three input ports and three output ports (9 bits each). It additionally includes one input and one output expansion port (8 bits each).

The SM is connected to the nine cells (cell_A... cell_I) belonging to the cluster, each one by means of four input ports and four output ports (9 bits each). The input ports correspond to the FU output ports of cells, and the output ports correspond to the output of internal multiplexers. In addition, the SM includes nine input and nine output expansion port (8 bits each) connected to each cell in the cluster.

The Switch Matrix Configuration Unit (SMCU) participates in the component self-routing process (section 4.8). In this process, the FU output port of a cell in a given component is connected to the FU input port of a cell in a different component. The process configures the multiplexers included in the SM.

The SM includes 60 multiplexers divided in two groups:

- 24 multiplexers connected to the SM output ports. These multiplexers are used as start point of a connection between two components or when a connection has to cross the SM. These multiplexers allow connecting the output port with: (1) the FU output port of any cell in the cluster (inputs 1 to 36), and (2) the input ports of other sides of the SM (inputs 37 to 57), e.g., if the output port is in east, the multiplexer allows connecting the input ports form north, northeast, southeast, south, southwest, west and northwest.

- 36 multiplexers connected to the FU input ports of cells. These multiplexers are used to direct the end point of a connection between two components to one of the nine cells belonging to the cluster. These multiplexers allow connecting the FU input port with: (1) the FU output port of other cells in the cluster (inputs 25 to 56), e.g., if the output port is in cell_A, the multiplexer allows connecting the output ports form cell_B. . . cell_I, and (2) the input ports of any side of the SM (inputs 1 to 24).

2.9 Pin Interconnection Matrix

The Pin Interconnection Matrix (PIM) is used exclusively for the port interconnection between cells of two components in different chips. Figure 2.9 shows the port distribution and the internal hardware that is included in the PIM.

The PIM is connected to its three adjacent clusters, each through three input ports and three output ports (9 bits each). It additionally includes one input and one output expansion port (8 bits each). The PIM is connected to the INET.

The Pin Interconnection Matrix Configuration Unit (PIMCU) configures the multiplexers in the component self-routing process, to allow for the interconnection of the FU port of a cell with a pin of the chip. Previously to this configuration, the GCU undertook a negotiation process with other chips, with the aim of assigning a pin of the chip to connect these components.

The PIM includes 12 multiplexers divided in two groups:

- 9 multiplexers connected to the PIM output ports. These multiplexers are used to connect a input pin with an output port. This is the start point of a connection between a input pin and the FU input port of the target cell.

- 3 multiplexers connected to the output pins of the PIM. These multiplexers are used to connect a input port with an output pin. These multiplexers are the last resource used to interconnect the FU output port of the source cell with the pin of the chip.

2.10 Expansion Signals

The expansion signals are included in the expansion ports of cells, SMs and PIMs. The expansion signals are used by self-placement and self-routing processes.

The signals con be divided in three categories as outlined below:

- **Cell to Cell:** The expansion port between cells includes four 19-bit input ports and four 19-bit output ports per cell, distributed in north, east, south and west. Each input and output

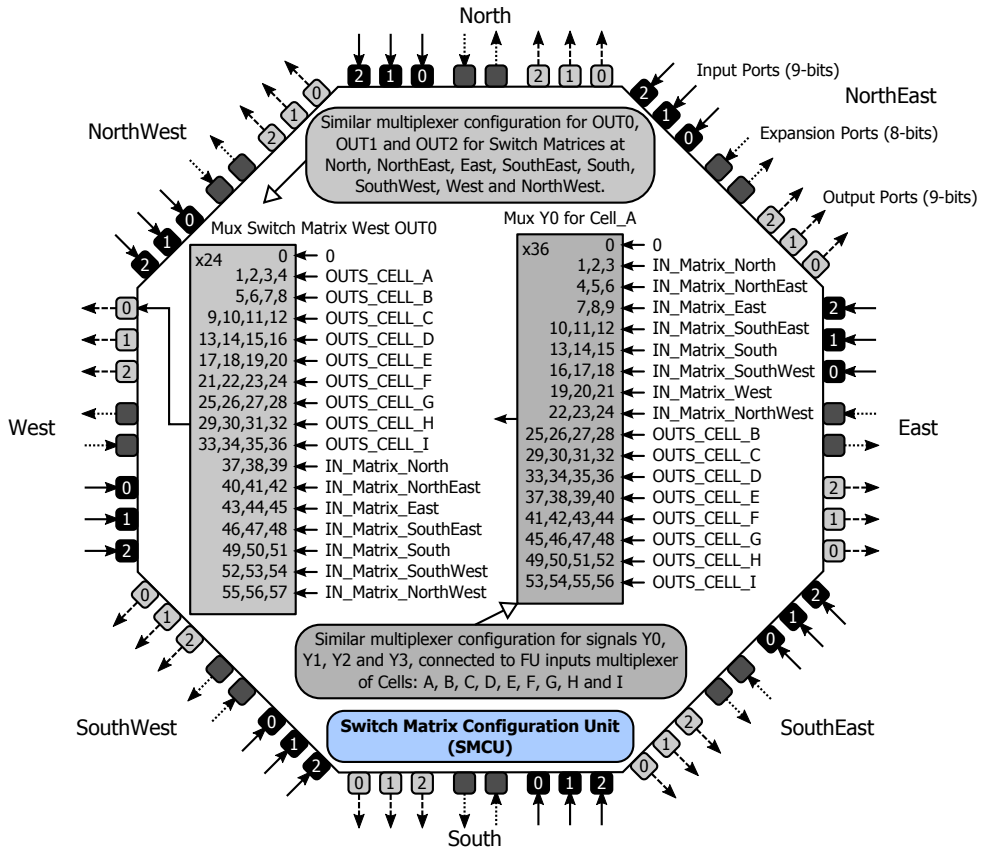


Figure 2.8: Architecture of Switch Matrix.

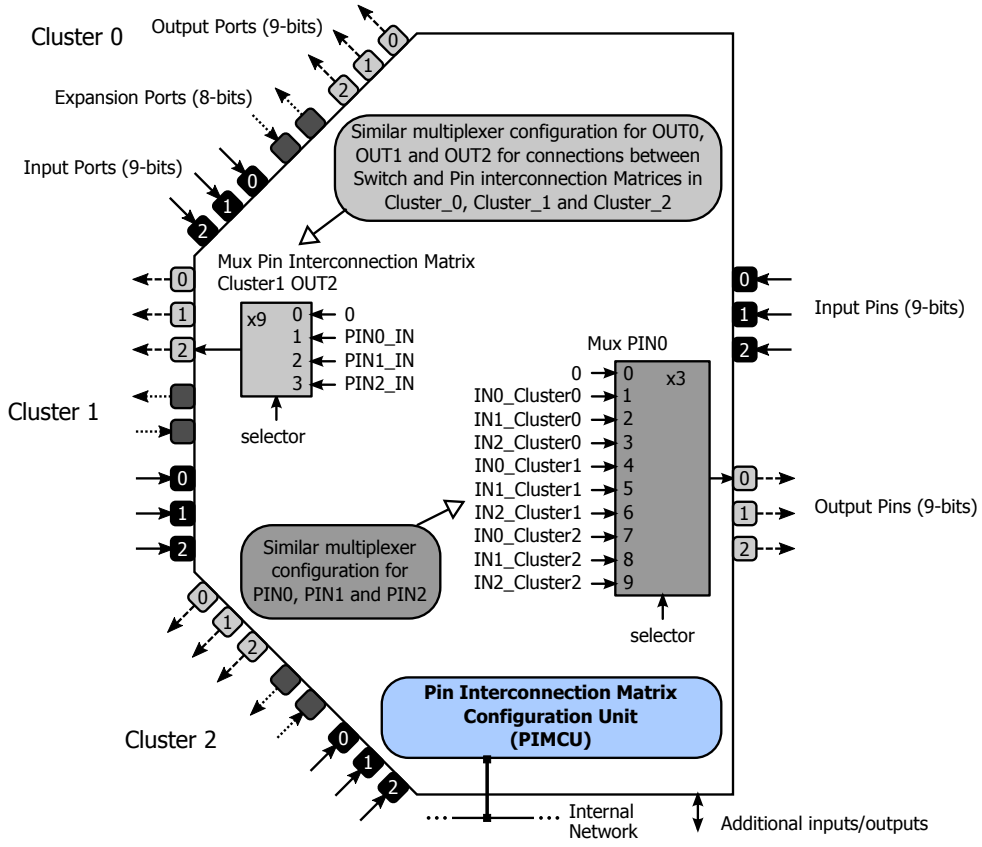


Figure 2.9: Architecture of Pin Interconnection Matrix.

port includes 11 signals. Figure 2.10 shows the expansion signals between two cells connected in east and west ports.

- **Cell to Switch Matrix:** Each SM includes nine 8-bit input ports and nine 8-bit output ports, for the connection of expansion signals between SM and the cells that belong to the cluster (cell_A. . . cell_I). Therefore, each cell includes additionally one 8-bit expansion input port and one 8-bit expansion output port. Figure 2.11 shows the expansion signals between cell A and SM inside the cluster.

- **Switch Matrix to Switch Matrix (or Pin Interconnection Matrix):** Each SM includes eight 8-bit expansion input ports and eight 8-bit expansion output ports, for the connection of expansion signals between SMs across clusters. Figure 2.12 shows the expansion signals between two SMs. The expansion ports between SM and PIMs are the same.

2.10.1 Global Signals for Self-routing Process

Figure 2.13 shows the additional global signals: *Routing_complete* and *Enable_routing*. These signals are the result of a logic OR function between all members of the system that could start or stop a routing process, i.e., CCUs and PIMCUs. When one of these CUs wants to start a routing process, it sets its *enable_routing_out* signal, enabling the routing capabilities for all members in the system. Similarly, when any CU wants to terminate the process, it sets the *routing_complete_out* signal. The self-routing process is detailed in section 4.6.

2.11 Internal and External Networks

The Internal Network (INET) and External Network (ENET) have been designed to provide the system with the necessary functionality to carry out the self-adaptive capabilities, specifically the self-placement and self-routing processes. Since several CUs of the system (CCUs, PIMCUs and GCU) should be able to send and simultaneously receive messages, the interface communication system is based on an adaptation of the I2C Bus Specification [36].

2.11.1 Communication Interface

The INET and ENET are based on two basic signals: serial clock line (SCL) and serial data line (SDA). These signals are the result of a logic AND (organized in rows and columns) between the output signals of all CUs of the system with networking capabilities. When the transmission (tx) process of a CU is in standby, the signals are at high logic level, thus, if all CU are in standby, the result on the SDA and SCL lines will be high. The Figure 2.14 shows the hardware implementation for the CUs involved in the INET.

When a CU needs to send a message, the transmission process starts, so that it has to set or clear the appropriate output signals. This way, the message will be visible in SDA and SCL and can be read simultaneously by all CUs of the system, including the transmitter. This characteristic is important in the execution of the self-adaptive algorithms implemented in the system. This implies that source cell(s) can get information of other cells without the need to transmit an answer from the target cell(s).

The transmission process could be started by one or more CUs in the chip (simultaneously), this depends on the algorithm that is being executed.

The ENET has a similar configuration to that of the INET. The ENET interconnects the GCUs with the EC.

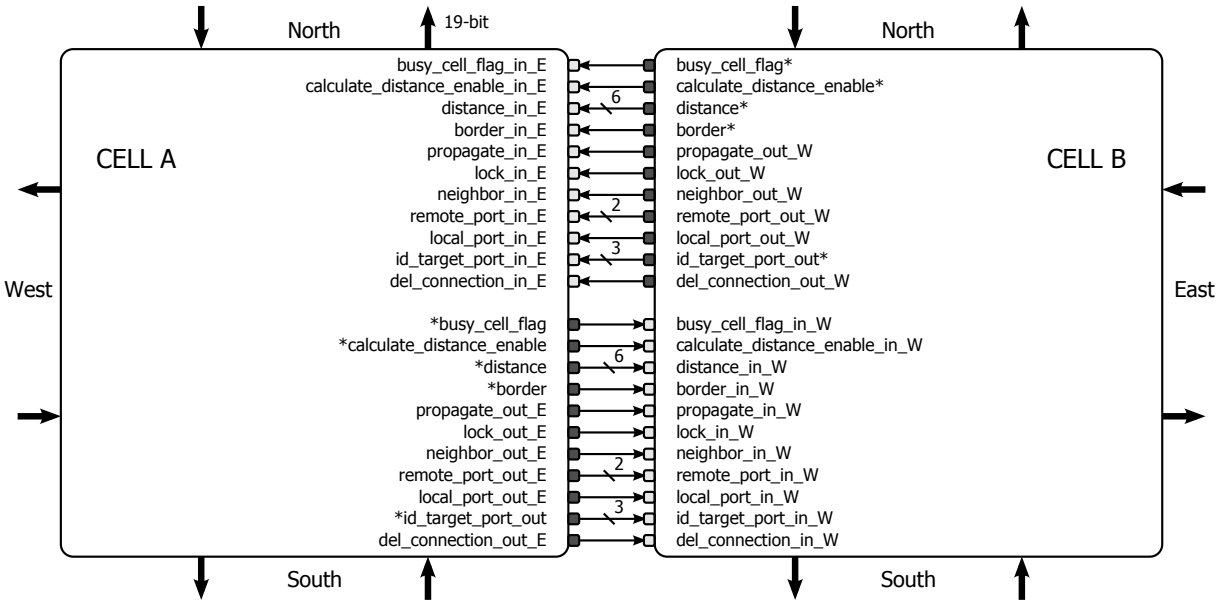


Figure 2.10: Expansion signals between cells[1].

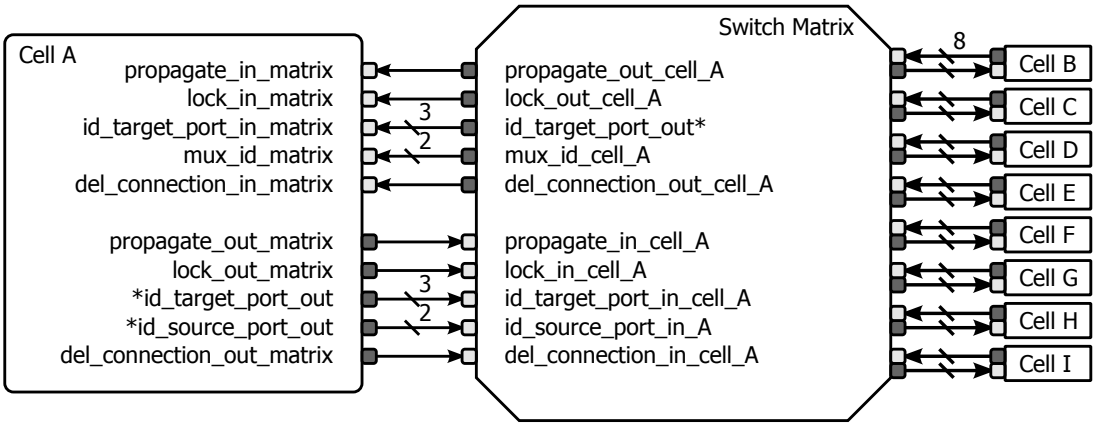


Figure 2.11: Expansion signals between cell and Switch Matrix[1].

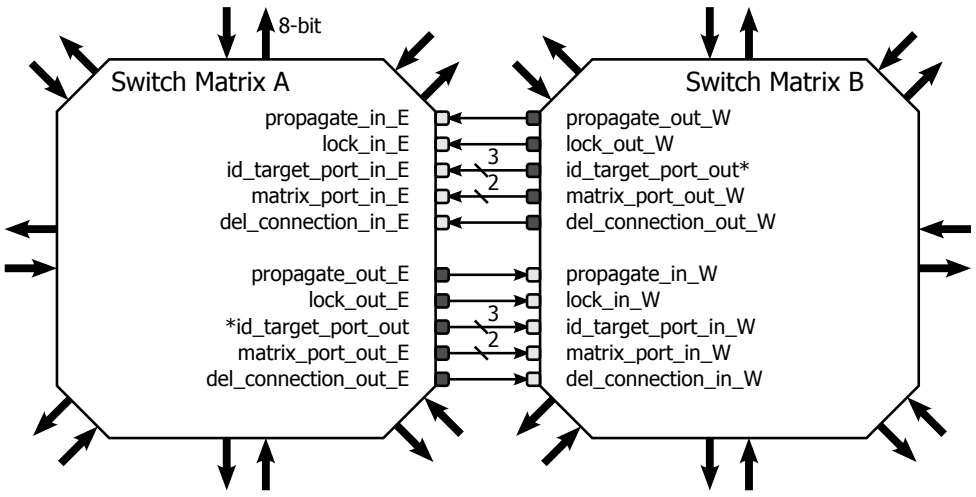


Figure 2.12: Expansion signals between Switch Matrices (including Pin Interconnection Matrix)[1].

[1]Signals marked with * are common for all expansion ports.

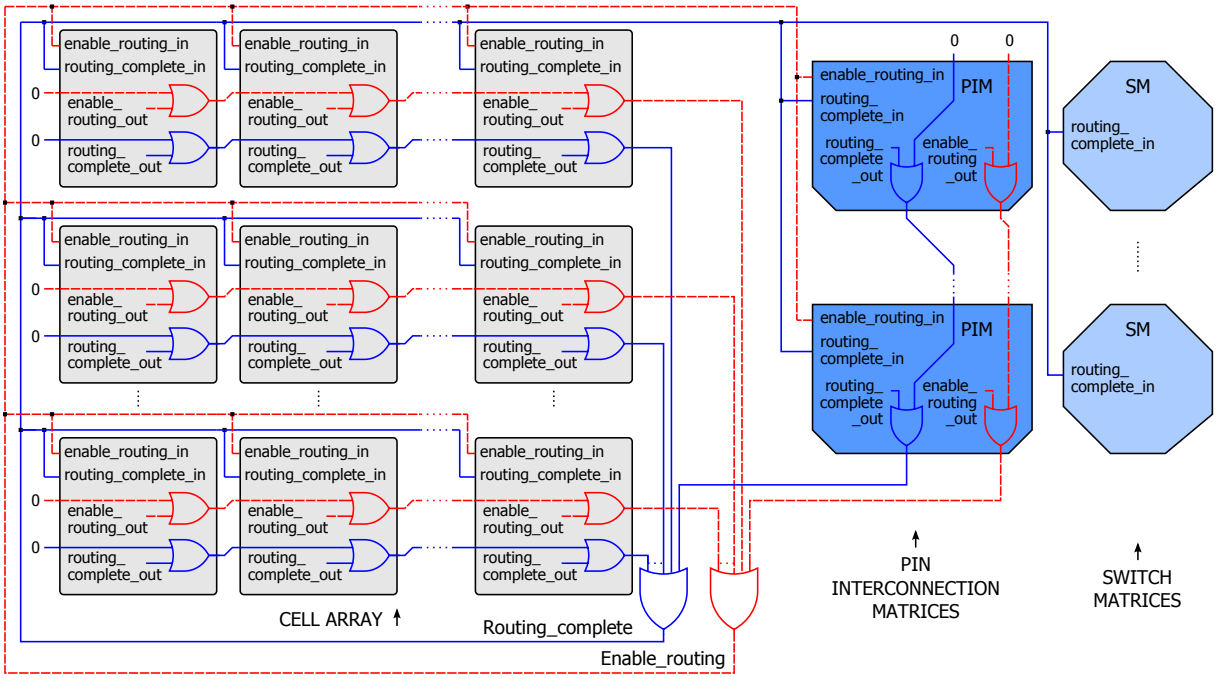


Figure 2.13: Routing signals implementation.

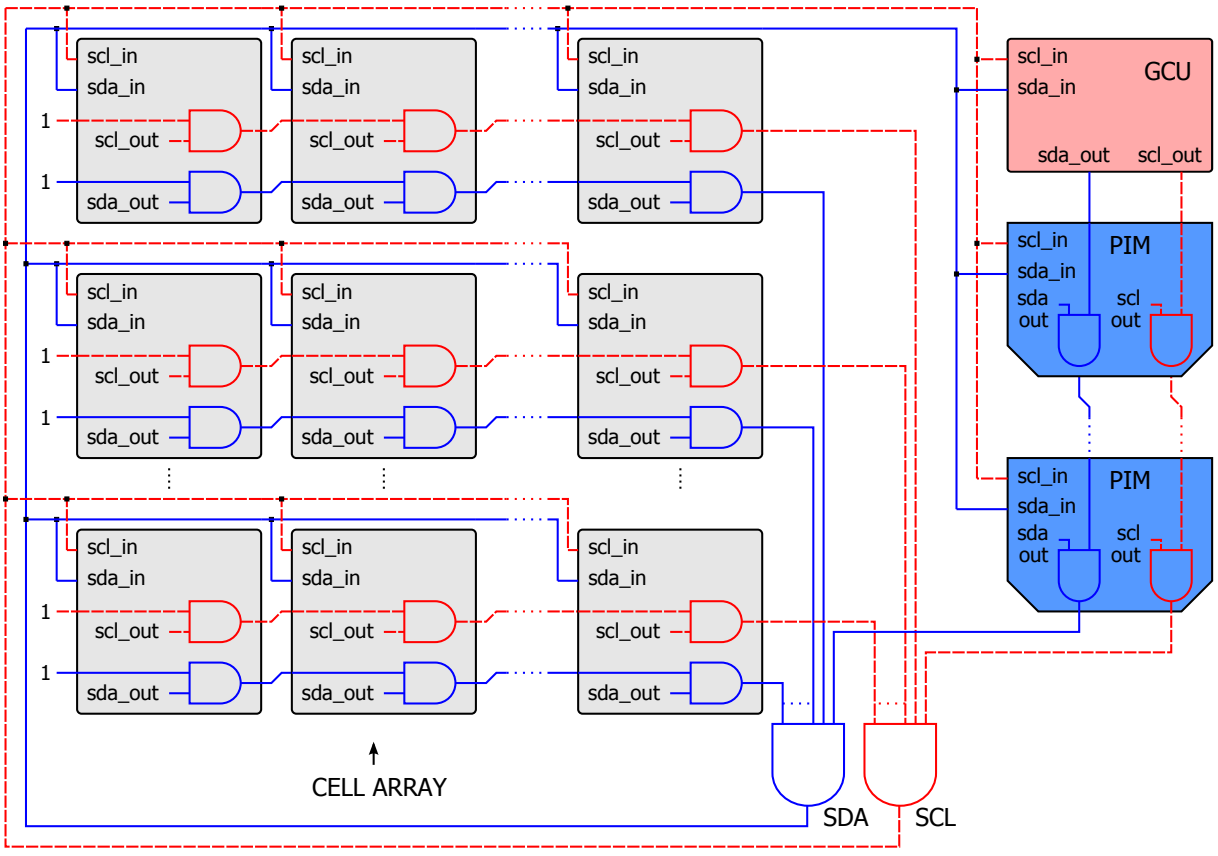


Figure 2.14: Internal Network implementation.

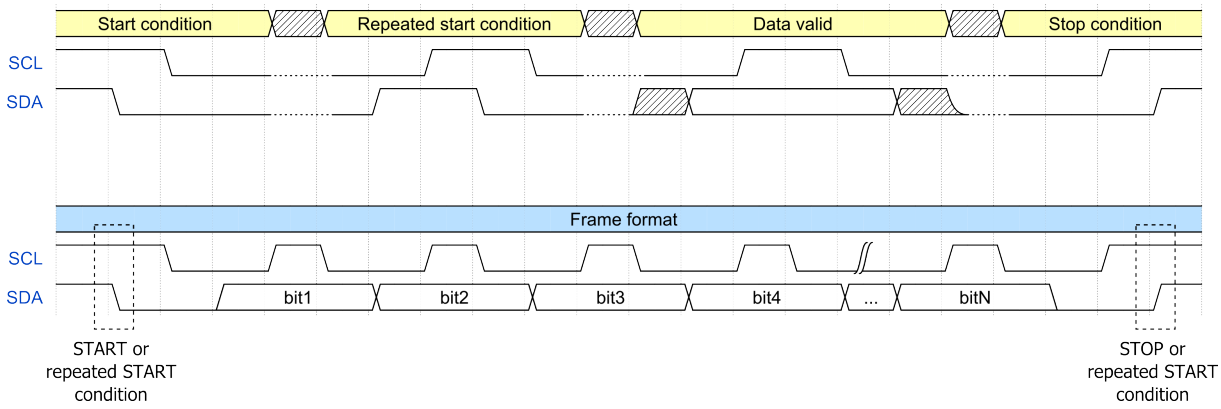


Figure 2.15: Considerations for Internal and External Networks.

2.11.2 Data Transmission

A transmission consists of three parts: generation of the start condition (or repeated start condition), data bits transmission and generation of the stop condition (or repeated start condition). Figure 2.15 shows necessary considerations for data transmission for INET and ENET, as follows:

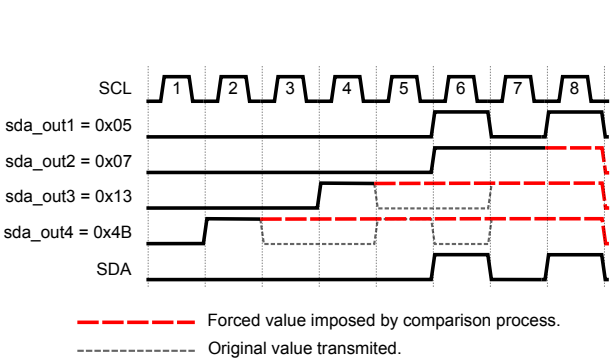
- All transactions begin with a start condition and are terminated by a stop condition.
- Start condition: a high to low transition on the SDA line while SCL is high.
- Stop condition: a low to high transition on the SDA line while SCL is high.
- Data validity: The data on the SDA line must be stable during the high period of the clock. The high or low state of the data line can only change when the clock signal on the SCL line is low. One clock pulse (SCL) is generated for each data bit transferred.
- The bus stays busy if a repeated start condition is generated instead of a stop condition. In this respect, the start and repeated start conditions are functionally identical.

2.11.3 Comparison Process

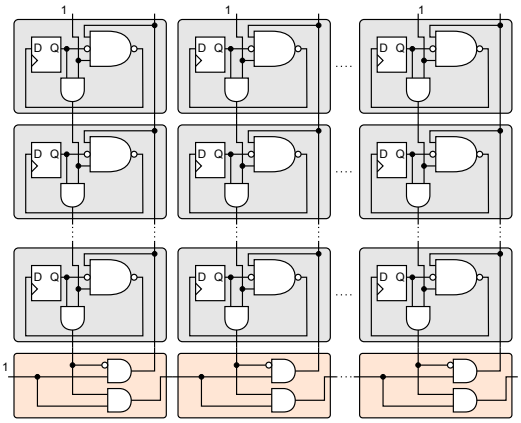
The algorithms of self-placement and self-routing require the implementation of a data comparison process from one or more cells, to identify the cell with better characteristics for a particular function with respect to another cell within the array. The physical implementation of the INET described previously permits this functionality, thus many cells can simultaneously compare the data while the transmission/reception is being executed. This is possible due to the logic AND between all *sda_out* signals, so that the lowest value will be imposed in a comparison.

An example of a comparison process is shown in Figure 2.16a. Let us assume that four cells send the following values: *sda_out1=0x05*, *sda_out2=0x07*, *sda_out3=0x13* and *sda_out4=0x4B*. These values will be compared bit to bit, starting with the most significant bit. The dominant value has to be the smallest one. As result of the logic AND, when one of the cells sends a high value but on the SDA line it appears a low value, this means that another cell has a better (lower) comparison value. This cell is self-discarded from the comparison and sets its *sda_out* to high value for the remaining bits. The cell that terminates the process without being discarded will be the winner (*sda_out1* in Figure 2.16a).

If there are two or more cells with the same value for comparison, the winner will be the leftmost uppermost cell in the array. This selection process takes place outside the INET with some additional bits, by means of the circuit shown in Figure 2.16b. The cells involved in the process put a low level in the flip flop, and after a clock pulse, only one of them must continue in low level. This one will be the "winner cell" of the comparison process.

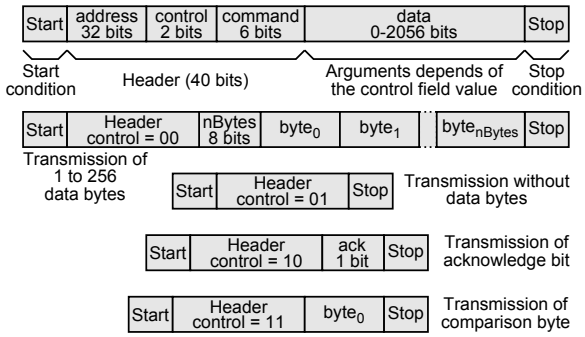


(a) Example of the comparison process.

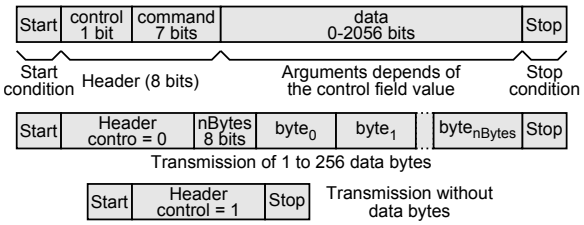


(b) Implementation of the cell selection process.

Figure 2.16: Comparison process.



(a) Communication protocol for Internal Network.



(b) Communication protocol for External Network.

Figure 2.17: Comumnication protocols.

2.12 Communication Protocol for Internal Network

The communication protocol is formed by the grouping of bits that are transmitted and received. This exchange of information for INET takes place between several CUs inside a chip, specifically the GCU, the PIMCUs and the CCUs. Figure 2.17a shows a detailed description of this protocol.

All bits belonging to the protocol are delimited by the start and stop conditions previously exposed. The frame can be divided in two sections: the header and the arguments (or data bytes). The 40-bit header is always present in a frame. The arguments sent or received later are dependent on the *control bits* and the *command* value. The header fields are described below:

1. ***address***: This 32-bit field is used to identify the cell to which the message is directed. This field includes the *id_cell* and *id_component*, each 16-bit long. The value 0xFFFFFFFF is reserved for broadcasting.

2. ***control bits***: The 2-bit *control* field specifies the number of data bits after the *command*. Thus, there may be commands that require or not any additional information, as follows:

   - ***control=00 ⇔ Data bytes:*** When the *command* field requires additional information, the 8-bit *nBytes* field is used. It indicates the number N of bytes that will be sent. These bytes may include connection tables, command arguments or information for cells that will be for program memory and configuration registers.
     $1 \leq N \leq 256, \quad nBytes = N - 1$

- ***control=01 ⇔ No data:*** No additional data bytes are required.
- ***control=10 ⇔ Acknowledge bit:*** The 1-bit *ack* field is used as an acknowledgment mechanism, indicating that the *address* field sent by a source cell matches with the *address* of a target cell.
- ***control=11 ⇔ Comparison byte:*** The $byte_0$ is the comparison value that is sent simultaneously by one or more cells. It is used when required to search the best cell location with respect to another to perform a given function (see section 2.11.3 for details).

3. ***command***: The 6-bit *command* field specifies the action that will take place inside the element(s) to which the frame is addressed. The commands correspond to the execution of necessary algorithms for implementing the self-routing and self-placement processes. The data sent or received later are dependent on the control bits and the command value. The list of commands is showed in Table 2.1. The use of this commands and its functionality is explained in chapter 4.

Command (INET)	TX→RX	Description
insert_first_cell	GCU→CCUs	Commands used for the placement of the first cell of a new component.
scan_first_cell	CCUs→CCUs	
end_first_cell	CCU→GCU	
insert_other_cell	GCU→CCUs	Commands used for the placement of other cells in a component (from the second). Additionally performs the routing of the connections between cells of the same component already placed in the array.
scan_new_cell_connections	CCUs→CCUs	
request_scan_affinity_value	CCU→CCUs	
scan_affinity_value	CCUs→CCUs	
set_target_cell_sr	CCU→CCUs	
connect_others_to_new_cell_sr	CCU→CCUs	
end_other_cell	CCU→GCU	
error_routing	CCU, PIMCU→GCU	Error when there is no routing resources available for a connection.
start_components_connection	GCU→CCUs	Commands used for the connection of components.
set_target_cell_sr_component	CCU→CCU	
end_components_connection	CCU→GCU, CCUs	
configure_chip_connection	CCU→GCU	Additional commands used for the connection of components in different chips. These commands must be executed after the placement and routing of all components.
set_target_cell_sr_chip	GCU→CCU	
search_pin_free_mb	GCU→PIMCUs	
pin_free_mb	PIMCU→GCU	
set_target_port_mb	GCU→PIMCUs	
start_cell_pin_connection	GCU→CCUs	
cell_pin_connection_conf	CCU→GCU	
start_pin_cell_connection	GCU→PIMCUs	
pin_cell_connection_conf	PIMCU→GCU	
configure_chip_connection_conf	GCU→CCUs	
write_configuration_registers	GCU→CCU	Write the Configuration Registers and Program Memory of Functional Unit processors.
write_program_memory[0,1,2,3]	GCU→CCU	
write_FU_memory_conf	CCU→GCU	
restart_processors	GCU→CCUs	Commands used for controlling the Functional Unit processors.
disable_processors	GCU→CCUs	
restart_and_disable_processors	GCU→CCUs	
enable_processors	GCU→CCUs	

Continued on next page

Command (INET)	TX→RX	Description
wait	GCU→CCUs	Commands used for controlling the Functional Unit processors including "wait" or "standby" state.
restart_processors_wait	GCU→CCUs	
enable_processors_wait	GCU→CCUs	
start_subprocess[0,1,2,3]	CCU→GCU	Commands used for execution of subprocesses, i.e., run-time self-configuration.
end_subprocess[0,1,2,3]	GCU→CCUs	
delete_component_connections_chip	CCU, GCU→CCUs	Commands used to delete a component in the cell array.
delete_component_connections_conf	CCU→GCU	
delete_component_chip	GCU→CCUs	
delete_component_conf	CCUs→GCU	
deroute_connection_other_chip	PIMCU→GCU	Commands used to deroute connections of component in other chips, which are used for deleting components.
deroute_pin_cell_connection	GCU→PIMCU	
deroute_pin_cell_connection_conf	PIMCU→GCU	
deroute_connection_other_chip_conf	GCU→PIMCU	
replicate_cells	CCU→GCU	Commands used for self-elimination and self-replication processes.
delete_cell_connections_chip	GCU, CCU→CCUs	
delete_cell_connections_conf	CCU→GCU	
eliminate_cell	GCU→CCU	
eliminate_cell_conf	CCU→GCU	

Note: conf = confirmation

Table 2.1: Commands list for Internal Network.

2.13 Communication Protocol for External Network

The comunication protocol for ENET permits to communicate the GCUs in different chips and the EC. Figure 2.17b shows a detailed description of this protocol.

All bits belonging to the protocol are delimited by the start and stop conditions previously exposed. The frame can be divided in two sections: the header and the arguments (or data bytes). The 8-bit header is always present in a frame. The arguments sent or received later are dependent on the *control bit* and the *command* value. The header fields are described below:

1. ***control bit***: The 1-bit *control* field specifies the number of data bits after the *command*. Thus, there may be commands that require or not any additional information, as follows:

   - ***control=0*** $\Leftrightarrow$ ***Data bytes:*** When the *command* field requires additional information, the 8-bit *nBytes* field is used. It indicates the number N of bytes that will be sent. These bytes may include connection tables, command arguments or information for chips that will be for program memory and configuration registers.
     $1 \leq N \leq 256, \quad nBytes = N - 1$
   - ***control=1*** $\Leftrightarrow$ ***No data:*** No additional data bytes are required.

2. ***command***: The 7-bit *command* field specifies the action that will take place inside the chip(s). The commands correspond to the execution of necessary algorithms for implementing the

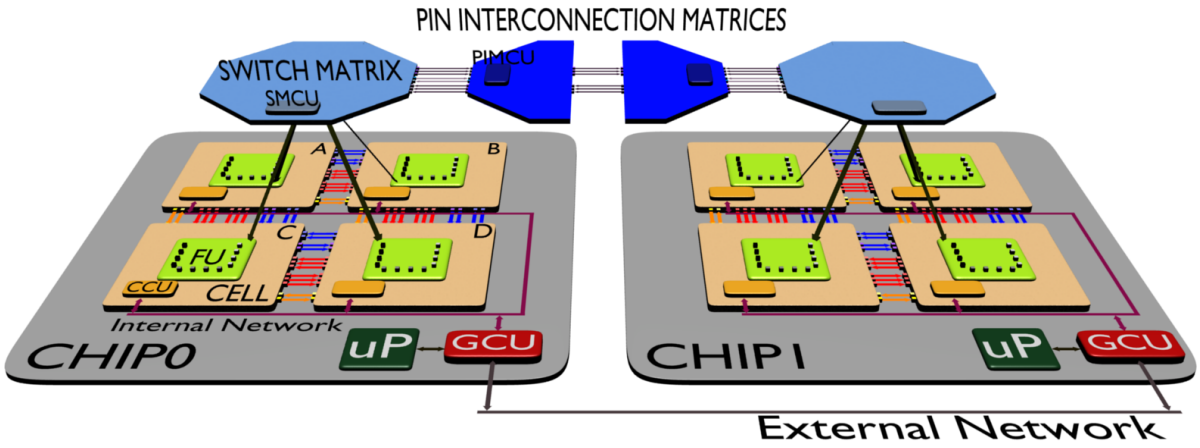


Figure 2.18: 3D representation of the prototype architecture.

self-routing and self-placement processes. The list of commands is showed in Table 2.3. The use of these commands and their functionality is explained in chapter 4.

2.14 Prototype architecture

For demonstration purposes, the original architecture previously described has been modified for the construction of a prototype, due mainly to the physical limitations in the FPGAs used for the system implementation. The prototype shown in Figure 2.18 has the following characteristics:

- The prototype has been developed in two chips, each one is a Virtex4 Xilinx FPGA (XC4VLX60), with an utilization rate close to 80% of their capacity.
- Each chip consists of a cluster that was reduced to a 2x2 cell array (this includes the SM), one PIM, one Control Microprocessor (CμP) and the GCU.
- The Internal and External networks were implemented.
- The system supports up to 32 processors working in parallel.
- The system implements all self-adaptive capabilities described in this document.
- All system components are described in VHDL. The CμP was implemented using the EDK design tool from Xilinx, which implements the MicroBlaze microprocessor; it is programmed in C language by means of the Xilinx Platform Studio (XPS) and Xilinx Software Development Kit.
- The programming and compilation of a entire project in system is manually performed using the SANE Project Developer (SPD) (See chapter 5 and appendix D).

Figure 2.19 shows the block diagram of a chip in the prototype. This design permits to have the same code description for both chips, the only difference is in the allocation of pins for each chip (files *.ucf).

The External Controller (EC) was replaced by the Control Microprocessor (CμP) inside chips. It should be noted that CμP is implemented in both chips but only one of them must assume the control for the system configuration (master chip). The CμP of the master chip is responsible

Command (ENET)	TX→RX	Description
synchronize_chips	EC→GCUs	Initial synchronization between chips
start_contest_winner_chip	EC→GCUs	Contest for the execution of a process in a chip.
cell_number_new_component	EC→GCUs	Insertion of a component in a chip.
insert_first_cell	EC→GCU	
end_first_cell	GCU→EC	
insert_other_cell	EC→GCU	
end_other_cell	GCU→EC	
error_routing	GCU→EC	There are no routing resources available for the connection of two cells in a component.
autoset_for_components_connection	EC→GCU	Commands used for connection of components.
start_components_connection	EC→GCU	
search_cell_other_chip	GCU→GCUs	
routing_cell_target_chip	GCU→GCUs	
end_routing_cell_target_chip	GCU→GCUs	
end_components_connection	GCU→EC	
error_components_connection	GCU→EC	
set_address_program_memory	EC→GCUs	Write the Configuration Registers and Program Memory of Functional Unit processors.
write_configuration_registers	EC→GCUs	
write_program_memory[0,1,2,3]	EC→GCUs	
write_FU_memory_conf	GCU→EC	
restart_processors	EC→GCUs	Commands used for controlling the Functional Unit processors.
disable_processors	EC→GCUs	
restart_and_disable_processors	EC→GCUs	
enable_processors	EC→GCUs	
wait	EC→GCUs	Commands used for controlling the Functional Unit processors including "wait" or "standby" state.
restart_processors_wait	EC→GCUs	
enable_processors_wait	EC→GCUs	
start_subprocess[0,1,2,3]	GCU→EC	Commands used for execution of subprocesses, i.e., run-time self-configuration.
end_subprocess[0,1,2,3]	EC→GCU	
autoset_for_delete_connections	EC→GCUs	Commands used to delete a component in the cell array.
delete_component_connections_chip	EC→GCU	
deroute_connection_other_chip	GCU→GCUs	
deroute_connection_other_chip_conf	GCU→GCUs	
delete_component_connections_conf	GCU→EC	
delete_component_chip	EC→GCUs	
delete_component_conf	GCU→EC	
replicate_cells	GCU→EC	Commands used for self-elimination and self-replication processes.
delete_cell_connections_chip	EC→GCUs	
delete_cell_connections_conf	GCU→EC	
eliminate_cell	EC→GCUs	
eliminate_cell_conf	GCU→EC	
cell_reinsert	EC→GCUs	

Note: conf = confirmation

Table 2.3: Commands list for External Network.

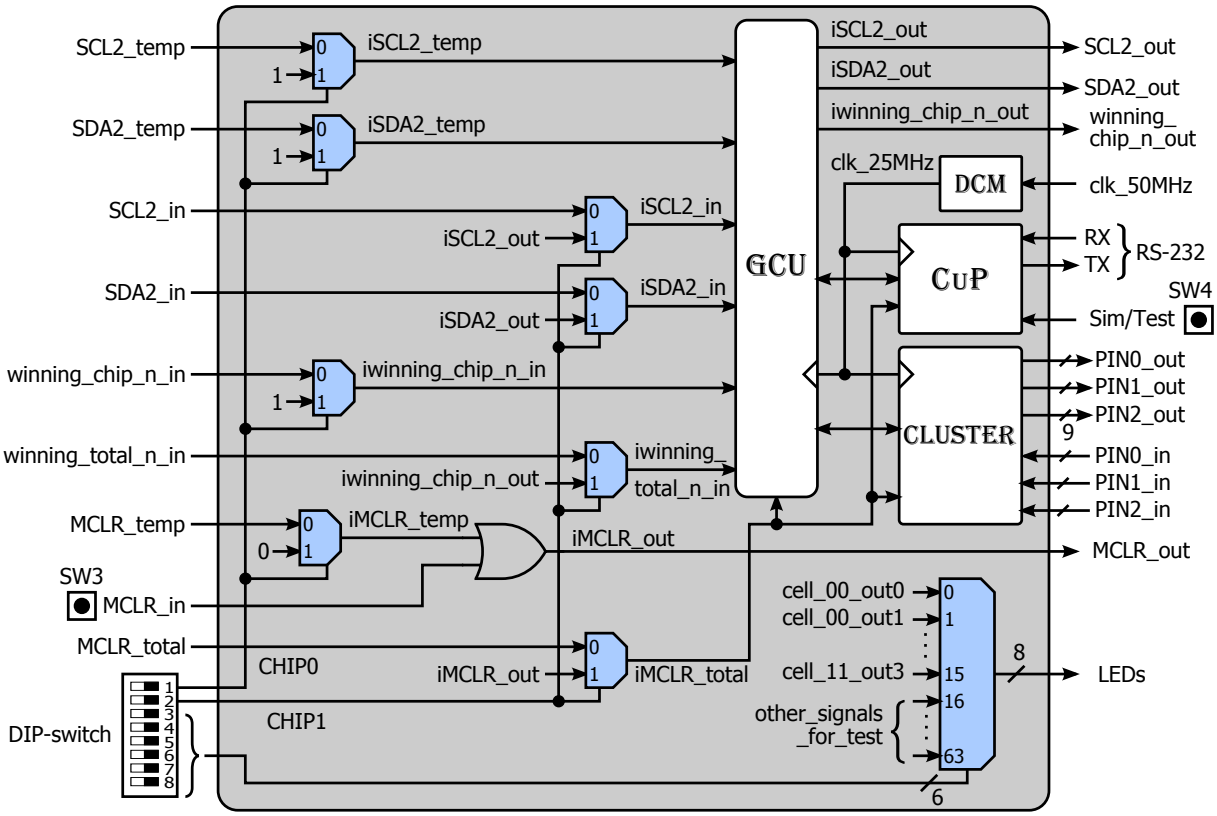
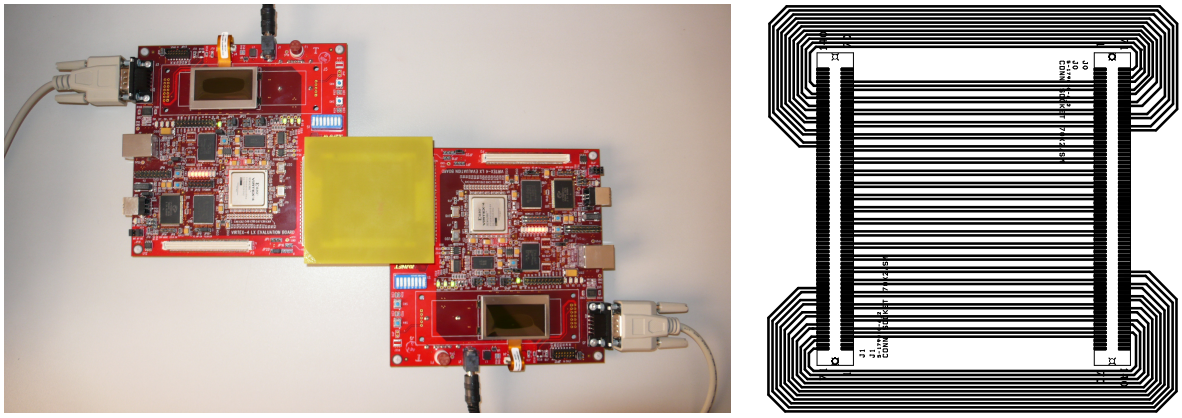


Figure 2.19: Block diagram of a chip in prototype.



(a) Prototype picture. (b) Connector layout

Figure 2.20: Prototype implementation

for implementing the main program for the configuration and execution of system functionality, even during runtime. Because of this, the dip-switch in positions 1 and 2 allows the selection of the master and slave chip.

The CμP through the GCU in both chips is responsible for controlling the self-placement, self-routing and the execution of any other self-adaptive processes.

Figure 2.20 shows a picture of the physical implementation of the prototype and the layout of the connection board.

The output of cells can be visualized in the leds included in the cards. This output is useful for the test of applications in the system. The dip-switch in positions 3 to 8 allows to select the output of cells and other internal process for debugging purposes.

2.15 Conclusions

This chapter describes the hardware components involved in the self-adaptive architecture presented in this dissertation.

The chapter starts with the definition of the four conceptual layers defined for the architecture. Afterwards, some details for the configuration of an application are detailed, specifically the description of a connection between cells, which is critical in the definition of the architecture.

The system is presented in a top-down approach, starting for a general overview of the entire system, which includes several chips interconnected with an External Controller (EC). Afterwards, the two layer architecture of the chip is detailed. This is the physical implementation of the system, and includes a cluster array interconnected with a Global Configuration Unit (GCU) and Pin Interconnection Matrices (PIMs). The cluster is composed of a 3x3 cell array and a Switch Matrix (SM). The input and output ports and the internal hardware of each component is detailed in the corresponding section.

This chapter also describes in detail the communication protocols for Internal Network (INET) and External Network (ENET). These networks participate actively in the self-adaptive processes as discussed in the following chapters. The general description of the prototype implemented for testing the architecture is presented at the end of the chapter.

Chapter 3

Functional Unit Architecture

Life would be tragic if it weren't funny.
La vida sería trágica sino fuera graciosa.
Stephen Hawking (1942)

Abstract: This section shows the functional description of the cell Functional Unit. It includes description of cores, configuration modes, and details of principal parts of processors like Data and Program Memories. Appendices A and B provide complementary information over the functionality of processors, showing details about the Instruction Set and Data Memory Registers respectively.

3.1 General Description

The Functional Unit (FU) is in charge of executing the processes scheduled to the cell or, from other point of view, the FU provides the processing capabilities to the cell. The FU can be described as a four-core configurable multicomputer [8]. The FU has twelve configuration modes that allows implementing between one to four processors, which can be configured for data processing of 8, 16, 24 or 32 bits.

The architecture of the FU could be composed of logic gates, LUTs, ALUs or any configurable digital system, but due to the hardware necessary to carry out the self-adaptive principles, in order to balance overhead, it has been decided to include a configurable digital element with greater complexity. The need for a system with general purpose computation capabilities lead to the design of the FU as a set of configurable processors with Harvard architecture.

Figure 3.1 shows a block diagram of the cell, which includes the FU and the Cell Configuration Unit (CCU). Additionally, the cell includes multiplexers that allows the interconnections of FU ports between cells. The CCU and the multiplexers provide support for the self-adaptive capabilities of cells. The FU has the following main characteristics:

- Four 9-bit input ports.
- Four 9-bit output ports.
- Four cores: each core contains the digital elements necessary for the construction of a processor.
- Output Multiplexing System (OMS): the OMS permits the cores to write data to the output ports.

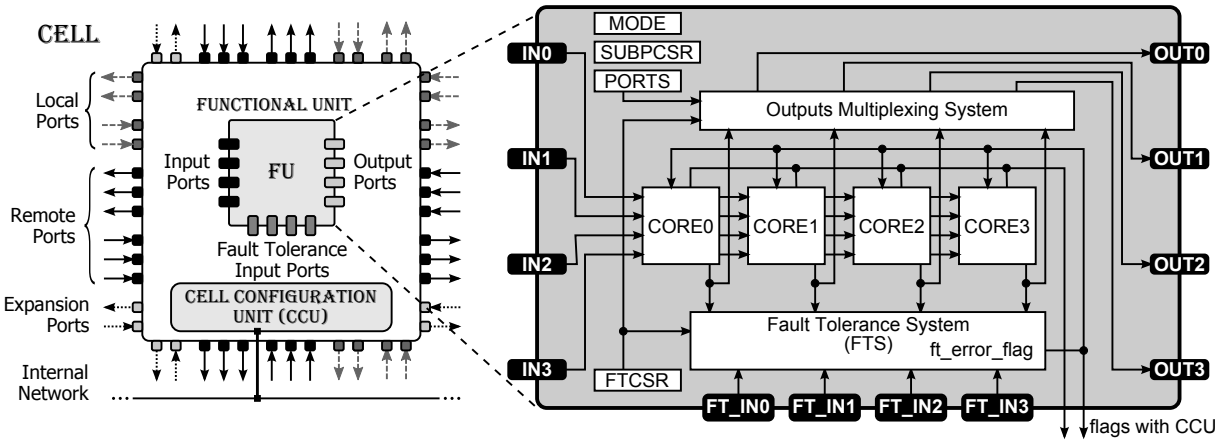


Figure 3.1: Functional Unit architecture.

- Fault Tolerance System (FTS): the FTS enables the system to continue operating properly in the event of failure of some of its processors. If the FTS detects a hardware failure in a processor, the cell or cells involved in the failure could be self-replicated in another location inside the array. Therefore the cell will be self-discarded for future self-placement processes.
- Four 9-bit fault-tolerance input ports used exclusively for FTS when enabled.
- PORTS Register: configures the OMS allowing write operations over FU output ports.
- FTCSR Register: configures the FTS and the OMS.
- MODE Register: sets the configuration mode of cell, i.e., configures how the cores are grouped to build from one to four processors in FU.
- SUBPCSR Register: interface between FU and CCU for the execution of subprocesses, i.e., it allows the execution of runtime self-configuration capability of the system.

3.2 FU Ports

The FU input ports can be interconnected with the FU output ports of two cells through the local and remote cell ports or through the Switch Matrices. This connection is performed by the self-routing process (Chapter 4).

The 9-bit FU input and output ports are composed of an 1-bit Read Enable (RE) signal and 8-bit of data (including the Fault Tolerance input ports). When a data is written to the register related to the output port, it is generated a pulse in the ninth bit of the FU port. The RE pulse is one clock-cycle.

Figure 3.2 shows an example of RE pulse when different data bytes are written in an output register. The data (ALU) and the enable signal from a core permits to write the output register associated to the FU output, and additionally permits to generate the RE pulse. When two or more output port register are written simultaneously, i.e., for 16, 24 or 32-bit processors, the RE bit in each port of FU has the same behavior.

The RE pulse of an input port is used to detect when a data has been written in the port. For this purpose, the special instruction BLMOV is used.

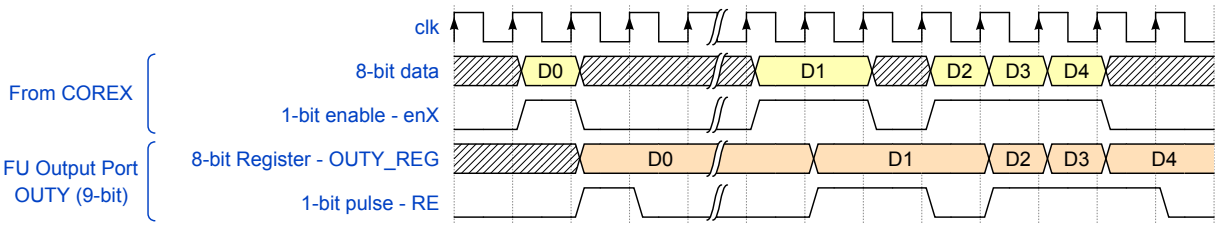


Figure 3.2: Read Enable pulse example.

3.3 Architecture of Processors

Throughout this document, the terms "cores" and "processors" are used frequently. The following sections clarify its difference from the point of view of the architecture presented.

3.3.1 Cores

The core can be described as a set of digital elements that could be used to build a processor (Figure 3.3). The FU has four cores, and each core includes the next principal components:

- General Purpose Registers (GPRs), 8 bytes, R0 to R7.
- 8-bit ALU.
- Code Condition Register (CCR).
- Program Counter.
- Control Memory.
- Program Memory (64 instructions capacity).

Most of this components have the same characteristics for all cores, except the ALU and the Program Counter. Each ALU includes specific hardware that supports the execution of instructions depending of the length of the processing data. The length of the Program Counter is different for each core: 8-bit for CORE0, 6-bit for CORE1 and CORE3, and 7-bit for CORE2, this is related to the size of the Program Memory that a core can handle in a specific configuration mode.

3.3.2 Processor

The processor is composed of the elements of one or more cores. Therefore, the FU can have between one to four processors. Figure 3.3 shows an example of the construction of three processors: the first is an 8-bit or 16-bit processor based on two cores, the second and third are 8-bit processors based in one core each. In this figure, the processor P0 shows the basic architecture of any processor in the FU. Note that there are bus lines denoted by 'N', which indicated that the data processing capability could be for 8, 16, 24 or 32 bits. The following are some consideration of processors in FU:

- The cores can be grouped in order to build a processor with more capacity. The program memories are added always in length, whilst the Data Memory can be added in length and/or width.
- The cores are grouped from left to right. The leftmost includes the most significant byte when the data processing is for 16, 24 or 32 bits.
- The Program Counter controls the sequence of the program. The processors includes conditional and unconditional instructions to modify this sequence.

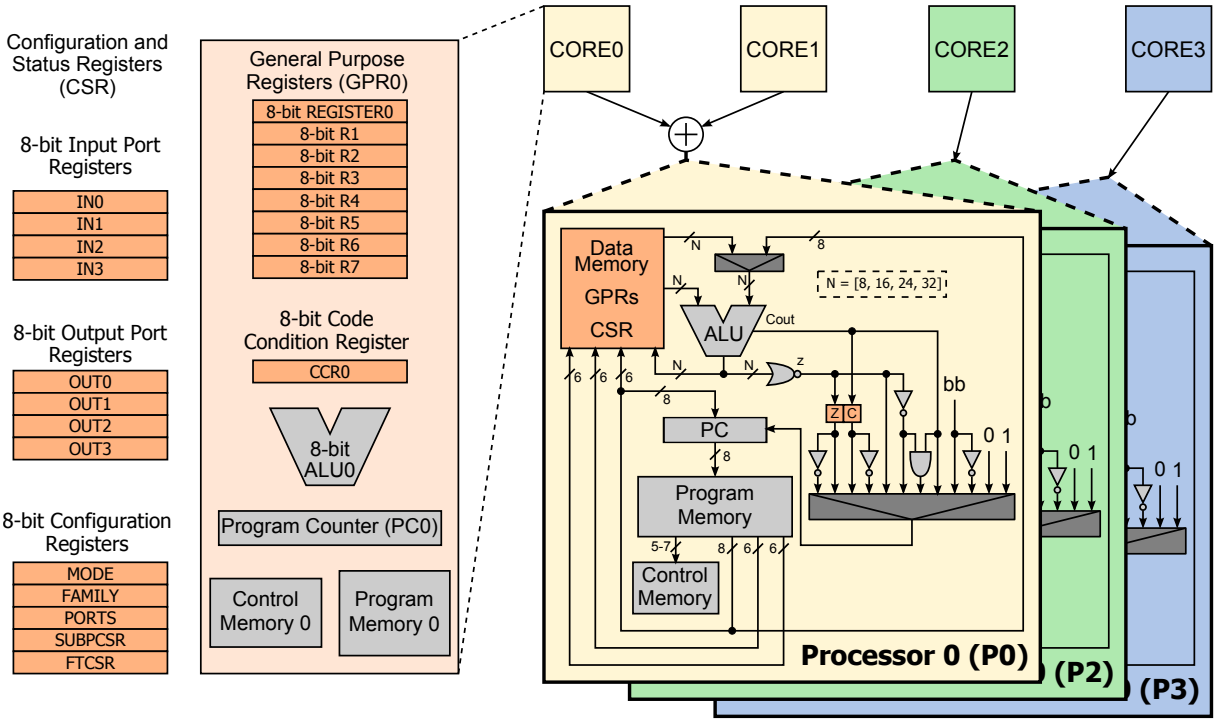


Figure 3.3: Construction of processors based on cores.

- The digital design of FU includes additional hardware that is controlled by the configuration registers (MODE, PORTS, FTCSR, SUBPCSR) and Control Memory.

The following sections describes the processing capabilities of the FU from a functional point of view.

3.3.3 Configuration modes

The configuration of the FU consists basically in grouping the digital elements of cores to build between one to four processors, where the expansion of Data and Program Memory describes the specific configuration mode. There are twelve different configuration modes, ranging between one and four processors working in parallel.

When each core implements a processor, i.e., when there are four processors in the FU, all digital components are active and the processors are identified with the same numeration of cores (CORE0 to CORE3 implements the processors P0 to P3).

The following considerations should be taken into account for processors that are composed of two or more cores:

- The core with the lower numeric order assumes the control of the processor, therefore only the Program Counter, Control Memory and CCR of this core are active for the processor. The name of the processor is related to this core. In Figure 3.3, the processors P0, P2 and P3 are controlled by CORE0, CORE2 and CORE3 respectively.
- The GPRs can be joined in length or width achieving data processing for 8, 16, 24 or 32 bits. These registers are mapped in Data Memory.
- The CSRs are mapped to the Data Memory for each processor. Only the CCR is different for each processor.
- The Program Memory of cores are added, which increases the instructions capacity of the processor.

Mode	Processors	CORE0		CORE1		CORE2		CORE3	
		CTR	ALU	CTR	ALU	CTR	ALU	CTR	ALU
0	4	✔	✔	✔	✔	✔	✔	✔	✔
		P0 [8x8 - 64]		P1 [8x8 - 64]		P2 [8x8 - 64]		P3 [8x8 - 64]	
1	3	✔	✔	✘	✘	✔	✔	✔	✔
		P0 [16x8 - 128]				P2 [8x8 - 64]		P3 [8x8 - 64]	
2	2	✔	✔	✘	✘	✔	✔	✘	✘
		P0 [16x8 - 128]				P2 [16x8 - 128]			
3	2	✔	✔	✘	✘	✘	✘	✔	✔
		P0 [24x8 - 192]						P3 [8x8 - 64]	
4	1	✔	✔	✘	✘	✘	✘	✘	✘
		P0 [32x8 - 256]							
5	3	✔	✔	✘	✔	✔	✔	✔	✔
		P0 [8x16 - 128]				P2 [8x8 - 64]		P3 [8x8 - 64]	
6	2	✔	✔	✘	✔	✔	✔	✘	✘
		P0 [8x16 - 128]				P2 [16x8 - 128]			
7	2	✔	✔	✘	✔	✔	✔	✘	✔
		P0 [8x16 - 128]				P2 [8x16 - 128]			
8	2	✔	✔	✘	✔	✘	✘	✔	✔
		P0 [8x16 - 192]						P3 [8x8 - 64]	
9	1	✔	✔	✘	✔	✘	✘	✘	✘
		P0 [16x16 - 256]							
10	2	✔	✔	✘	✔	✘	✔	✔	✔
		P0 [8x24 - 192]						P3 [8x8 - 64]	
11	1	✔	✔	✘	✔	✘	✔	✘	✔
		P0 [8x32 - 256]							

- CTR denote the control of a processor, it includes the Program Counter, Control Memory and CCR. When CTR is active for COREX, the name of the processor is defined as PX.
- Nomenclature: P0 [16x8 - 64] = Processor 0 [Data memory includes 16 words of 8 bits each - Program memory with capacity for 64 instructions].
- Data processing: x8 = 8-bit, x16 = 16-bit, x24 = 24-bit, x32 = 32-bit.

Table 3.1: Configuration modes: active components for processors and memory distribution

- For an 8-bit processor, the ALU is active only for the core that assumes the control.
- For a 16-bit, 24-bit or 32-bit processors, the ALUs are concatenated in 2, 3 or 4 cores respectively.

Table 3.1 shows a relation of the configuration modes with the number and name of processors, as well as the active components of cores for each processor. This table also shows Data and Program Memory capacity for each configuration mode. The GPRs in Data Memory can be combined in width and length, achieving combinations for data processing of 8, 16, 24 and 32 bits. The mode 8 is the only one that does not use the GPRs of CORE2. The Program Memory can only be combined in length, making possible to have programs of 64, 128, 192 or 256 instructions.

For example, in the configuration mode 0, there are four active processors (P0 to P3), all of which have 64 instructions capacity of Program Memory and 8 bytes of GPRs in Data Memory. In mode 10, two processors are built (P0 and P3). The first (P0) has a Program Memory with capacity for 192 instructions and eight 24-bit words of GPRs in Data Memory; the second (P3), 64 instructions of Program Memory and 8 bytes of GPRs in Data Memory.

3.4 Data Memory

The Data Memory is 8, 16, 24 or 32 bit and it is composed of 1, 2, 3 or 4 blocks of General-Purpose Registers (GPRs) and 14 Configuration and Status Registers (CSRs).

Data memory access is performed through three buses, two for reading and one for writing. This way, a processor could read two registers, perform the desired operation and store the result in a third register, all in a single clock pulse.

3.4.1 General Purpose Registers (GPRs)

Each core includes eight 8-bit General Purpose Registers (GPRs), which are mapped in Data Memory according to the configuration mode selected. The grouping of these blocks is included in the Data Memory map shown later in this section.

3.4.2 Configuration and Status Registers (CSRs)

Appendix B includes a detailed description of the CSRs, which can be configured properly for any application designed for the system. All CSRs are 8-bit, below is presented a functional description of these registers.

Output Port Registers (OUT0 . . . OUT3): The 8-bit OUTX register is connected directly to the 8 less significant bits of FU output port X. The ninth bit of FU corresponds to RE bit as explained in section 3.2. For a correct write operation of any OUTX register, it is necessary to be sure that register PORTS is configured properly.

Input Port Registers (IN0 . . . IN3): The 8-bit INX represent the actual data in the 8 less significant bits of FU input port X, or from other point of view, the data in the output port register of a cell connected to this port. These are read-only registers. The RE pulse in an input port is used to detect when data has been written in the port. The special instruction BLMOV is used for this purpose.

Code Condition Register (CCR): The CCR is composed of three bits: TA (Thread Active), which indicates if the execution thread has finished or not. The Z and C elements correspond to the flags that indicate when an operation is zero and when the operation has generated a carry respectively. The instruction END is the only one able to stop the execution of the thread (TA $\leftarrow$ 0).

Configuration Mode Register (MODE): The 8-bit MODE register is used for the configuration mode of the FU. Section 3.3.3 shows the twelve possible configuration modes for FU. This is a read only register, therefore a configuration mode can not be modified by a processor.

Famiy Register (FAMILY): The 8-bit FAMILY register was implemented to identify the family of the execution thread to which it belongs. The functionality of this register is reserved for future implementations, where a new programing paradigm [29] [30] [32] could be used to improve the functionality of the architecture.

Ports Configuration Register (PORTS): The 8-bit PORTS register configures the Output Multiplexing System in FU. This register configures the path between the ALU of a core and the OUTX register for a write operation over the FU output port.

Subprocesses Configuration and Status Register (SUBPCSR): The 8-bit SUBPCSR configures the execution of subprocesses. Any component in the system can start the execution of one to four subprocesses for dynamic reconfiguration in the system.

Fault-tolerance Configuration and Status Register (FTCSR): The 8-bit FTCSR configures the Fault-tolerance system that permits to detect a hardware failure in desired processors in the system (See section 3.7 for details).

3.4.3 Data Memory Map

Figure 3.4 shows the memory map for all possible processor in FU. Each memory map has a table that indicates the mode and the processor associated. For 8-bit processors, the GPRs of cores are joined lengthwise, which allows to increase the user GPRs in Data Memory.

For 16-bit processors in modes 5, 6, 7 and 8, the GPRs are joined widthwise, whereas for mode 9 the GPRs are joined lengthwise and widthwise as showed in figure. For 24-bit and 32-bit processors the GPRs are joined widthwise.

When there are more than one processor in the FU, it is important to note that all processors can read any INX register, but only one processor can write a specific OUTX register, this configuration must be performed in the PORTS register. Note that for 16, 24 and 32-bit processors, the ports IN0 and OUT0 represents the most significant byte for data processing.

3.5 Program Memory and Instructions Set

The FU includes four blocks of Program Memory (one on each core). Each block can store up to 64 instructions (64x25 bits). The Program Memory is joined depending on the configuration mode (see Table 3.1) increasing the instructions capacity for a processor.

Each 25-bit word in the Program Memory corresponds to one instruction. This word is divided in operation code (OPCODE) and the arguments. The OPCODE is 5-bit or 7-bit and the arguments could use the remaining bits depending on the instructions.

The instruction set is composed of 44 instructions, which includes arithmetic, logic, shift, branch, conditional branch and special instruction for the execution of microthreads [30].

Appendix A includes a detailed description of the instruction set for any processor in the FU.

3.6 Output Multiplexing System

Figure 3.5 shows a block diagram of Output Multiplexing System (OMS). The registers PORTS and FTCSR configure the core that is selected to perform a write operation over a specific OUT register. Note that any OUT register can be written by only one core.

When the FTS is disabled, the PORTS register allows for configuring the path between the data bus of a core and the OUT register, and also permits to configure the signal en_outX, that enables the write operation over the register. The enable signals (en_outX) are generated by the core that assumes the control of the processor. These signals are used for loading the output register and for the generation of the Read Enable (RE) pulse in the FU output ports.

When the FTS is enabled, the register FTCSR has control priority over PORTS register as shown in Table 3.2. See section 3.7 for details.

3.7 Fault Tolerance System (FTS)

The FTS enables the system to continue operating properly in the event of the failure of some of its processors. When a failure is detected by the FTS, the FU notifies the problem to the CCU, which starts the appropriate self-adaptive process for the replication of the damaged cells.

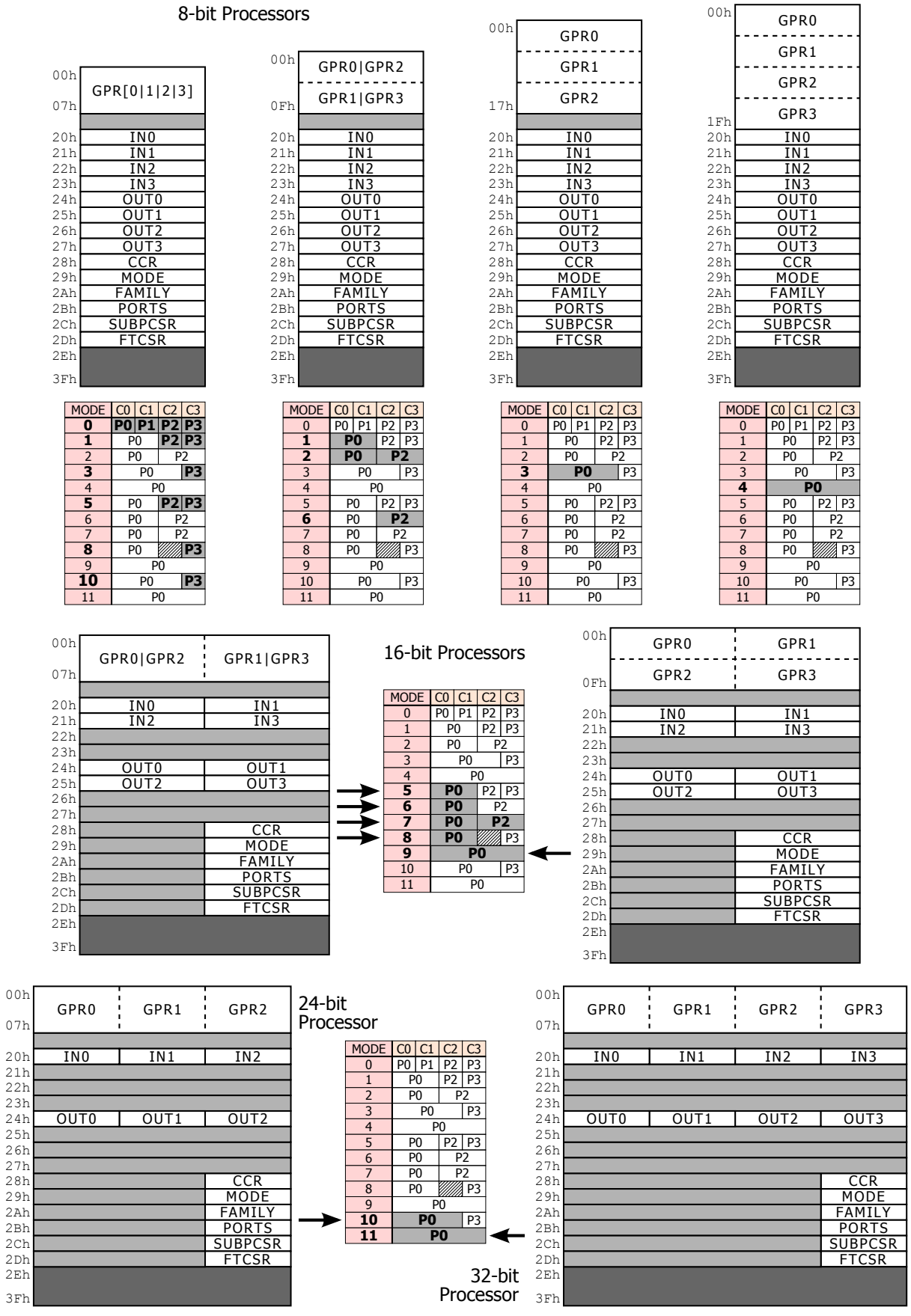


Figure 3.4: Data memory map for 8, 16, 24 and 32 bit processors.

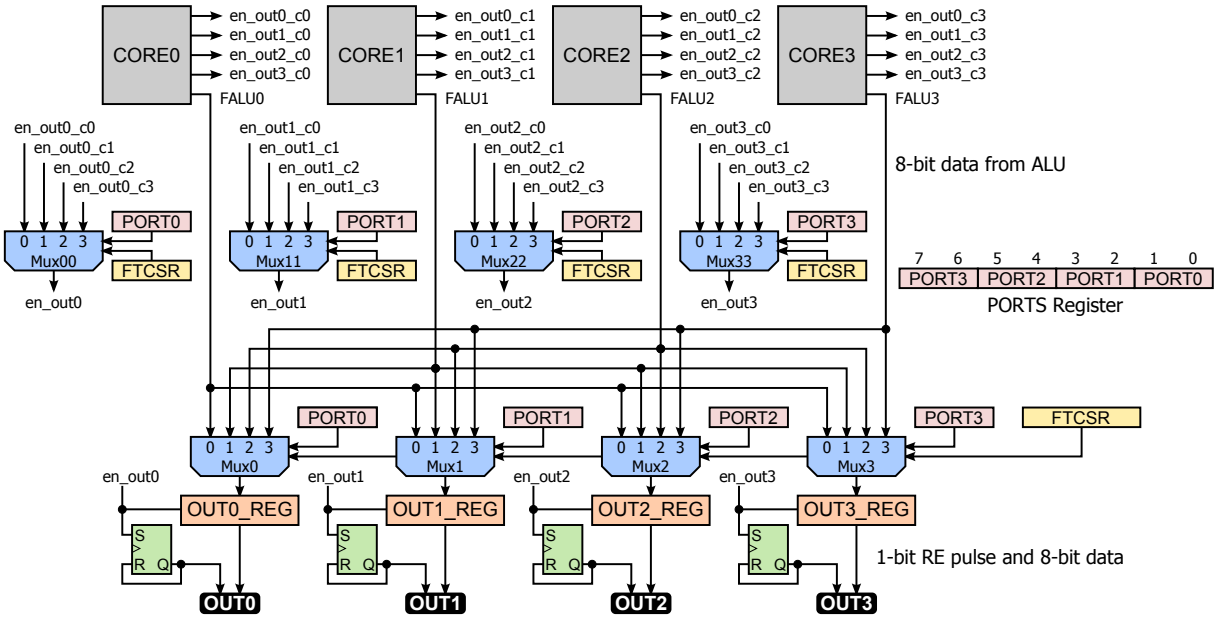


Figure 3.5: Block diagram of Output Multiplexing System.

ft_controls	en_out0 mux0	en_out1 mux1	en_out2 mux2	en_out3 mux3
1 - 1 - 5	1 FALU0	PORT1*	PORT2*	PORT3*
1 - 1 - 6	1 FALU0	1 FALU1	PORT2*	PORT3*
1 - 1 - 7	1 FALU0	1 FALU1	1 FALU2	PORT3*
1 - 1 - 8	1 FALU0	1 FALU1	1 FALU2	1 FALU3
1 - 1 - others	PORT0*	PORT1*	PORT2*	PORT3*
0 - X - X	PORT0*	PORT1*	PORT2*	PORT3*

ft_controls ↔ *ft_enable* - *ft_redundant_cell* - *ft_mode*

* The value of en_outX y muxX depends of PORTX value.

Table 3.2: Output Multiplexing System operating table.

The FTS consists of a specific hardware that allows the comparison of two identical processors each time that a instruction is executed, that means each clock cycle. This involves the comparison of 2, 4, 6 or 8 cores, depending of the configuration mode of the FTS (FT_mode).

These processors that will be compared are defined as working and redundant. They must share the same inputs, but on the other hand, the working processor takes over writing the output ports, because two outputs can not be routed to the same location. There may be one or two cells participating in the FTS. These cells are called *primary* and *redundant*. The *primary cell* is mandatory and includes working processors. This cell may or may not include redundant processors. The *redundant cell* is optional and includes only redundant processors.

Figure 3.6 shows the block diagram of FTS, which is only active when FTS is enabled and it is the *primary cell* (signals *ft_enable* and not *ft_redundant_cell*). Table 3.3 shows the data bus comparisons available for FTS. Note that FT_modes 0 to 4 only perform comparisons with data bus from the same cell, whilst FT_modes 5 to 8 perform comparison between data buses of the cell with FT_input ports, which suppose a connection between the *redundant cell* and *primary cell*.

When the *redundant cell* is used, the OMS allows to write constantly the data bus of cores to the FU output ports as described in Figure3.5 and Table 3.2.

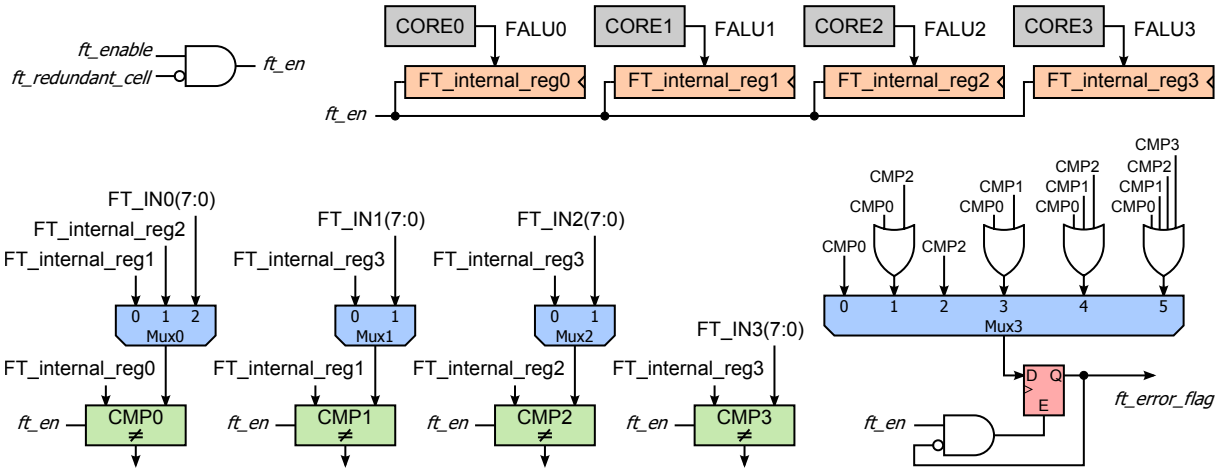


Figure 3.6: Fault Tolerance System.

FT_mode	Comparison	Mux0	Mux1	Mux2	Mux3
0	F0⇔F1	0	X	X	0
1	F0⇔F1 & F2⇔F3	0	X	0	1
2	F2⇔F3	X	X	0	2
3	F0⇔F2	1	X	X	0
4	F0⇔F2 & F1⇔F3	1	0	X	3
5	F0⇔F0*	2	X	X	0
6	F0⇔F0* & F1⇔F1*	2	1	X	3
7	F0⇔F0* & F1⇔F1* & F2⇔F2*	2	1	1	4
8	F0⇔F0* & F1⇔F1* & F2⇔F2* & F3⇔F3*	2	1	1	5

FX denotes a data bus from ALU in COREX (FALUX). ⇔ denotes a comparion between cores. & denotes a logic AND. * denotes a data bus from the redundant cell (a connection between FUs of primary and redundant cell is assumed).

Table 3.3: Multiplexers configuration for Fault Tolerance System in primary cell.

3.7.1 Fault Tolerance Input Ports

When the redundant processor is included in the same cell where the working processor is located, the FT_input ports are not used, and the FTS only performs comparison between cores of the cell. In this case the *redundant cell* is not used (FT_modes 0 to 4).

Otherwise, when the redundant processor is located in the *redundant cell*, the FTS of the *primary cell* must perform a comparison between cores of different cells, in this case the OMS of the *redundant cell* must drive the output of the data flow of the core(s) to the output of the cell (RE is set to logic 1), which in turn should be connected to the FT_inputs of the FU of the *primary cell* that includes the working processor (FT_modes 5 to 8).

When the FTS is enabled in the *redundant cell*, the register FTCSR has priority over PORTS register to control the OMS as shown in Table 3.2. For example, in FT_mode 5 the data bus of CORE0 is routed directly to the OUT0_REG (mux0=FALU0 and en_out0=1 in Figure 3.5). In FT_mode 8 all cores are routed directly to the respective OUTX register.

The FU includes four FT_input ports. These ports must be interconnected by user with the output ports of the *redundant cell* when the FTS is enabled. The number of ports depends on the configuration mode of cell and the FT_mode.

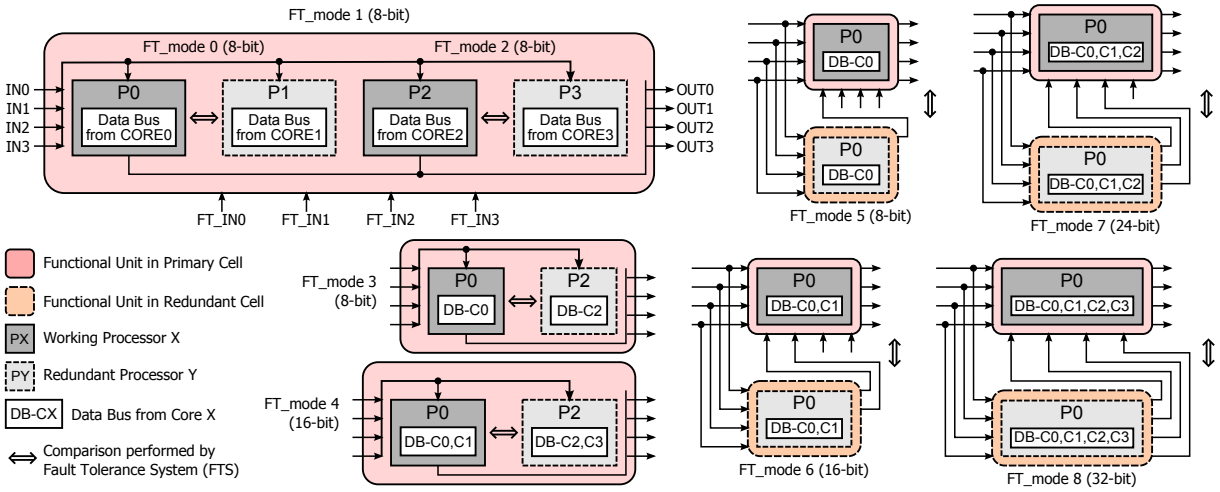


Figure 3.7: FT_modes for processors in Functional Unit.

3.7.2 Fault Tolerance Modes

The FTS could be implemented in any processors available in the system. Figure 3.7 shows the possible comparisons of processors with the same characteristics, whose result is the creation of a specific fault tolerance configuration mode (FT_mode).

Table 3.4 shows the 9 FT_modes available and the comparison performed. This table indicates the cores that are compared in each FT_mode, which could be combined with any of the 12 configuration modes available for the system, obtaining 108 possible combinations. It is responsibility of the developer to configure an appropriate combination between configuration mode of cells (always necessary) and FT_mode (if FTS is enabled). This table also shows the suggested cell configuration modes for each FT_mode.

As example lets suppose that it is necessary to implement the FTS to four 8-bit processors. For this purpose you need two cells (cell A and cell B) configured in mode 0, i.e., four 8-bit processors each. Therefore, the fault tolerance capability could be implemented with FT_modes 1 or 8. For option with FT_mode 1 you will have in cell A and B two primary processors and two redundant processors each. For option with FT_mode 8 you will have the four primary processors in cell A and the four redundant processors in cell B.

3.7.3 Configuration of FTS

When a specific application needs to implement a processor with fault tolerance capabilities, the developer has to set the configuration mode (MODE register) and the *FT_mode* by means of the Fault Tolerance Configuration and Status Register (FTCSR) (see section B.9 for details).

For configuration of FTS you should keep in mind the following considerations:

1. Enable the FTS for primary and redundant cell (if used). If it is not enabled, the other bits of FTS are not taken into account. You must set the bit *FT_enable*.

2. Select the FT_mode for primary and redundant cell (if used).

3. Indicate if the cell is the redundant cell. The bit *FT_redundant_cell* is used for this purpose. It must be set in the redundant cell for the FT_modes 5, 6, 7 and 8 only. If a cell is used as redundant, the comparators of FTS are disabled for that cell. The data bus of cores is driven to the output of redundant cell and RE is set to 1. The user must interconnect the output ports of redundant cell with the FT_input ports of primary cell.

FT_mode	Core Comparison	Suggested Configurtion Mode	
		Primary Cell	Redundant Cell
0	C0 ⇔ C1	0	N/A
1	C0 ⇔ C1 & C2 ⇔ C3	0	N/A
2	C2 ⇔ C3	0, 1, 5	N/A
3	C0 ⇔ C2	2	N/A
4	C0-C1 ⇔ C2-C3	7	N/A
5	C0 ⇔ C0*	3, 4	3, 4
6	C0-C1 ⇔ C0*-C1*	5, 8, 9	5, 8, 9
7	C0-C1-C2 ⇔ C0*-C1*-C2*	10	10
8	C0-C1-C2-C3 ⇔ C0*-C1*-C2*-C3*	0, 11	0, 11

⇔ denotes a comparion between cores. & denotes a logic AND. * denotes a core in the redundant cell (a connection between FUs of primary and redundant cell is assumed).

Table 3.4: Fault Tolerance Modes (FT_modes)

4. The bit *FT_error_flag* in primary cell indicates when the FTS has found an error while performing a comparison between two processors, this bit stops the execution of the programs in the cores and alerts the CCU to start the self-elimination and self-replication processes of damaged cells.

3.8 Conclusions

This chapter describes the hardware architecture of Functional Unit (FU). The FU ports are composed of four 9-bits input ports, four 9-bit output ports and four 9-bit fault tolerance input ports. The FU includes an Output Multiplexing System (OMS), that is in charge of configuring the core that is enabled for write data in the FU output ports. The Fault Tolerance System (FTS) allows the detection of hardware failures, performing a comparison between two processors. If a failure is detected, the FU notifies to the CCU and damaged cells are self-replicated in system.

The FU includes four cores. Each core contains the digital elements that are used for construction of a processor: 8 bytes of General Purpose Registers (GPRs), 8-bit ALU, Code Condition Register (CCR), Program Counter, Control Memory and Program Memory (64 instructions capacity).

The processor is constituted by the elements of one or more cores. Therefore the FU can have between one to four processors working in parallel. There are twelve configuration modes, where the expansion of Data and Program Memory describes the specific configuration mode. The GPRs can be joined in length or width achieving data processing for 8, 16, 24 or 32 bits. This registers are mapped in Data Memory. The Program Memories can be joined or not depending on the configuration mode, which increases the instructions capacity for a processor allowing Program Memory sizes of 64, 128, 192 or 256 instructions. The instruction set of processors is composed of 44 instructions, which includes arithmetic, logic, shift, branch, conditional branch and special instruction for the execution of microthreads.

Chapter 4

Self-Adaptive Processes

If you want to run, run a mile. If you want to experience a different life, run a marathon.

Si quieres correr, corre una milla. Si quieres experimentar una vida diferente, corre un maratón.

Emil Zátopek (1922 – 2000)

Abstract: This chapter describes the self-adaptive capabilities included in the architecture, mainly the self-placement and self-routing, which due to its intrinsic design, enable the development of systems with runtime self-configuration, self-repair and/or fault tolerance capabilities. The self-adaptive capabilities are executed in an autonomous and distributed way by Configuration Units of Cells, Switch and Pin Interconnection Matrices.

4.1 Summary

The following is a summary of the self-adaptive processes that are supported by the architecture presented in this document, which will be explained in detail along this chapter. Note that most of the self-adaptive capabilities are based in self-placement and self-routing processes.

1. **Self-placement:** it is responsible for finding out the most suitable position in the cell array to insert the new cell of a component. This process is divided in two:
   - Self-placement for the first cell of a component.
   - Self-placement for other cells of a component.
2. **Self-routing:** it allows interconnecting the FU ports of two cells. The interconnection of cells can be at two levels:
   - Cell level, through local and remote cell ports.
   - Component level, through Switch and Pin Interconnection Matrices.

   When a connection between cells must be eliminated, the **Self-derouting** process permits to disconnect the FU ports of two cells (at cell and component level), i.e., it releases the routing resources of a connection.
3. **Self-replication:** it permits to replicate the cell of a component to an empty cell in the array. For this purpose, the process includes the execution of the following processes for a specific cell in the order listed: self-derouting, self-placement and self-routing.

4. **Self-elimination:** it permits to eliminate and discard a specific cell(s) for future self-placement process when a hardware failure is detected. Its routing resources continue available and it can participate in future self-routing processes.
5. **Self-configuration:** it includes all previous self-adaptive process listed, and could be divided in two scenarios:
   - Self-repair: when the Static Fault Tolerance mechanism is enabled and a hardware failure is detected, the processes for self-replication and self-elimination of damaged cell(s) are executed.
   - Dynamic reconfiguration: when the execution of subprocesses is enabled for a component. It has the ability of create and eliminate components, among others. Therefore, the self-placement and self-routing processes could be executed.

4.2 Previous Considerations

For the understanding of the proposed algorithms, it is important to note that the Control Microprocessor (CμP) includes the high-level instructions in system, which will be executed sequentially for the configuration of a SANE-ASM. These high-level instructions will be defined as SANE Assembler (SASM) instructions.

This is the starting point for the configuration of an application in the system. The SASM instructions will be detailed in chapter 5. In advance, for the understanding of the following sections some SASM instructions are introduced:

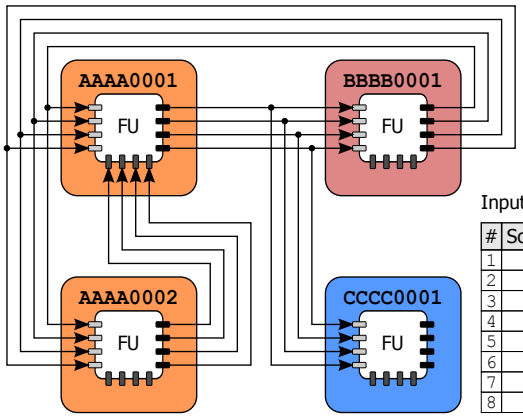
- `create_component`.
- `connect_component`.
- `delete_component`.
- `start_subprocess_X`.
- `end_subprocess_X`.
- `ft_configuration` (Fault Tolerance configuration).

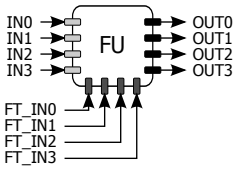
The name of each SASM instruction gives an idea of its functionality. The execution of this instructions may require a negotiation process to establish the chip that has to execute a specific process. Once the chip is defined, the CμP sends the information to the Global Configuration Unit (GCU) of the selected chip to execute the desired action. Thereby, the GCU is the start point for the execution of self-placement and self-routing algorithms inside a chip. When the processes end, the GCU receives a confirmation command and notifies to the CμP the end of the process. Thereafter, the CμP may continue with the execution of other SASM instructions. The communication between Configuration Units (CUs), GCU and CμP is made through the Internal Network (INET) and the External Network (ENET). The labels "_inet" and "_enet" will be added to the commands related with these networks along this chapter.

It is important to note, that the algorithms presented in this chapter are being executed by several CUs at the same time in distributed way. Therefore, each CU includes its proper set of variables, e.g., the result of the execution of the algorithms is independent for each cell. The flow diagrams for self-adaptive algorithms in cells and Switch Matrices (SMs) are summarized in Appendix C and will be introduced along this chapter.

4.3 Initial State, Cell Address and Connection Tables

In the initial state, all cells are free, i.e., they do not belong to any component. The cells belonging to any component have to be placed and connected for data processing and information exchange.





AAAA0001	
32-bit Address	
16-bit id_component	16-bit id_cell

Inputs Connection Table - Cell AAAA0001

#	Source Port	Target Port	Source Cell
1	OUT0	IN0	BBBB0001
2	OUT1	IN1	BBBB0001
3	OUT2	IN2	BBBB0001
4	OUT3	IN3	BBBB0001
5	OUT0	FT_IN0	AAAA0002
6	OUT1	FT_IN1	AAAA0002
7	OUT2	FT_IN2	AAAA0002
8	OUT3	FT_IN3	AAAA0002

Outputs Connection Table - Cell AAAA0001

#	Source Port	Target Port	Target Cell
1	OUT0	IN0	BBBB0001
2	OUT1	IN1	BBBB0001
3	OUT2	IN2	BBBB0001
4	OUT3	IN3	BBBB0001
5	OUT0	IN3	CCCC0001
6	OUT1	IN2	CCCC0001
7	OUT2	IN1	CCCC0001
8	OUT3	IN0	CCCC0001
9	X	X	00000000
⋮			
20	X	X	00000000

Figure 4.1: Address and Connection Tables example for cell AAAA0001.

This is a sequential process where each cell has to be placed and routed in the system. For this purpose, the cells execute in a distributed way the self-placement and self-routing algorithms.

All cells have a 32-bit unique identifier called *address*. This field is divided into two 16-bit words, called *id_component* and *id_cell*. The *id_component* is the component unique identifier, where the value FFFFh is reserved for broadcasting and the value 0000h to indicate that the cell is free and does not belong to any component (initial value). Therefore, it is possible to instantiate up to 65534 different components. The *id_cell* is the cell unique identifier in a component, so there may be up to 65536 cells in a component and a maximum close to 2^{32} cells in the system.

All cells has two connection tables as detailed below:

- The **Input Connection Table** store the connections of the eight inputs of FU (including the fault tolerance input ports). This table includes the fields source port, target port and source cell, which corresponds to a connection between the FU output port of another cell with the FU input of the cell.
- The **Output Connection Table** can store up to 20 connections, which is limited for the number of port of the cell, i.e., 8 local ports plus 12 remote ports. When the cell includes connections with other components, the number of connections within the same components is proportionally reduced. This table includes the fields source port, target port and target cell, which corresponds to a connection between the FU output port of the cell with the FU input port of another cell.

Figure 4.1 shows an example of the *address* and connection tables for the cell `AAAA0001` in a SANE with three components.

4.4 Creation of Components in a Chip

The `create_component` and `connect_component` are basic instructions that permit the system to create and interconnect new components in the cell array. These SASM instructions require the execution of two basic self-adaptive processes: self-placement and self-routing. These algorithms are executed in an autonomous and distributed way by system members (cells, SMs and PIMs). The Cell Configuration Unit (CCU) includes the algorithms for all self-adaptive processes at cell level. It includes an interface with the INET that implements the respective communication protocol (see sections 2.11 and 2.12 for details). The Switch Matrix Configuration Unit (SMCU) and Pin Interconnection Matrix Configuration Unit (PIMCU) include the algorithms for self-routing at component level.

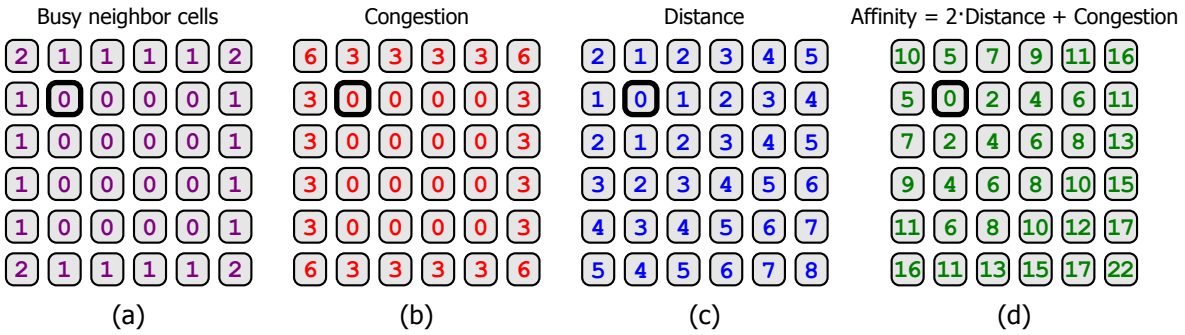


Figure 4.2: Example of *busy_neighbor_cells*, *congestion*, *distance* and *affinity*.

In addition to the communication functionality, the transmission (tx) and mainly the reception (rx) processes participate actively in the self-routing and self-placement algorithms. It is important to note that tx and rx processes can be executed simultaneously by one or more cells in the array, even the rx process can modify the data of the tx process if it is necessary, e.g., for the comparison byte and the acknowledgment bit. The flow diagram for tx and rx processes is shown in Section C.1.

4.5 Self-Placement Process

The self-placement algorithm is responsible for finding out the most suitable position in the cell array to insert the new cell of a component. For the placement of components in the array it is advisable but not essential to have the cells organized by number of connections with other cells. Therefore, the cell with more connections is first placed in a convenient place, where there is a large number of free neighboring cells. The following variables are defined for self-placement process:

- **Busy_neighbor_cells:** This value indicates the number of busy cells around a cell. The cells in the border of the array includes busy (or not implemented) cells by default.
- **Congestion:** The routing *congestion* figure (4.1) is defined as the number of remote output ports that are busy (or not available) in a cell. The cells in the border of the array include busy (or not implemented) remote ports by default.

$$congestion = remote\_ports\_used \tag{4.1}$$

- **Distance:** The *distance* between two cells is the sum of the absolute differences of their coordinates (Manhattan distance).
- **Affinity:** The figure cost *affinity* indicates the location appropriateness for the placement of the new cell of a component with respect to another cell of the same component. The *affinity* is defined in (4.2).

$$affinity = 2 \cdot distance + congestion \tag{4.2}$$

Figure 4.2 shows an example of these variables. It assumes the initial configuration, i.e., there are no routing resources used. These variables are modified dynamically, each time that a cell is placed or a connection is routed. Note that *distance* and *affinity* values are relative to a specific cell (highlighted).

4.5.1 Self-Placement of the First Cell of a Component

For the placement of the first cell of a component, a particular procedure is used, different from other cells. In this case, a good candidate position is defined as one where a free cell has low routing congestion and the largest number of free neighboring cells. The procedure includes the next steps:

1. The GCU broadcasts by means of the INET a message with the new cell identifier (*address*) and its input and output connection tables.
   - *tx_command=insert_first_cell_inet.*
   - *tx_address=0xFFFFFFFF.*
   - *tx_data* = new cell *address* (4 bytes) + connection tables.
2. Free cells send simultaneously through INET, using a comparison process, the addition result of the number of *busy_neighbor_cells* and *congestion* value ((a) + (b) in Figure 4.2).
3. The winner cell will be the leftmost uppermost cell with the lowest value. This cell stores the *address* and the connection tables. Additionally, it sends a message to the GCU indicating the end of the "self-placement of the first cell of a component" process, e.g., the highlighted cell in Figure 4.2 shows the location of the first cell of a component, when the array is empty.

Section C.2.1 presents a flow diagram of the algorithm implemented by cells for the insertion process of the first cell of a component.

4.5.2 Self-Placement of Other Cells of a Component

After inserting the component first cell, the remaining cells of the component are placed as close as possible to the cell with the largest number of connections with the new cell. For this purpose, the *affinity* is used. The procedure is as follows:

1. The GCU broadcasts to the cell array a message with the new cell identifier (*address*) and its connection tables.
   - *tx_command=insert_other_cell_inet.*
   - *tx_address=0xFFFFFFFF.*
   - *tx_data* = new cell *address* (4 bytes) + connection tables.
2. The cells belonging to that component send a message by means of the INET indicating the one's complement of the number of connections they share with the new cell, starting a comparison process. The winner cell will be the one which has the largest number of connections.
3. This cell starts a process, in which the *distance* with free cells is calculated. Then, it requests to the free cells to start a comparison process of their *affinity*.
4. The winner cell will be the leftmost uppermost cell with the lowest *affinity* value. This cell stores the *address* and connection tables, e.g., the cell below the highlighted cell in Figure 4.2 (d) corresponds to the second cell placed in the cell array.

Section C.2.2 presents the flow diagram of the algorithm that performs the self-placement of the other cells of the component (from the second). Figure 4.3 shows an example with three components in an array of two clusters (6x3 cells array). This figure shows the execution sequence of self-placement and self-routing processes for component AAAA. The final location of component BBBB and CCCC is also shown.

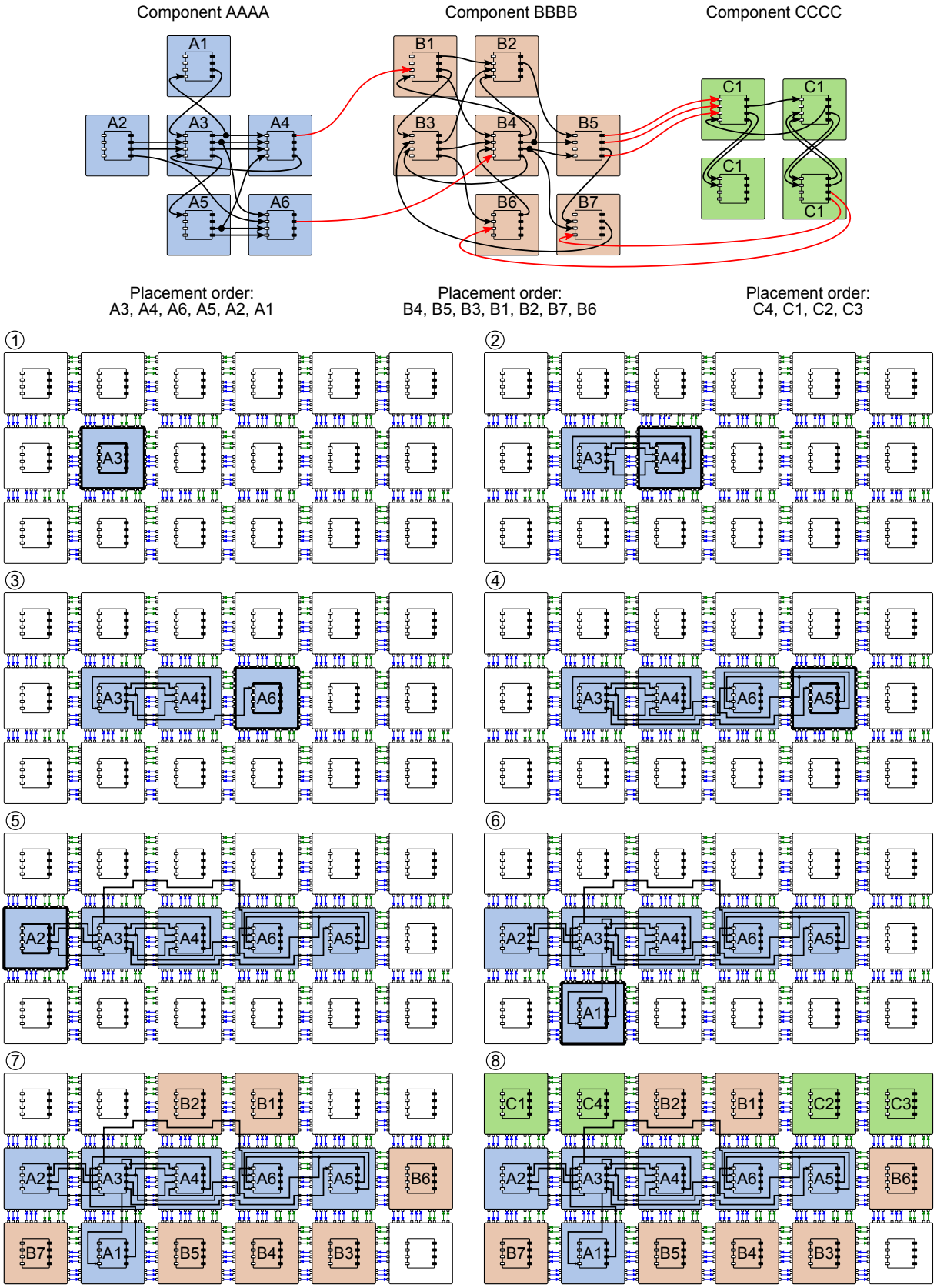


Figure 4.3: Example of the self-placement algorithm implementation for three components in an array of two clusters (6x3 cell array). The resources used by self-routing algorithm are shown only for component AAAA.

4.6 Self-Routing Process

The Self-routing allows interconnecting the FU ports of two cells. It can be at cell level, through the local and remote cell ports, or at component level, through the Switch Matrices (SMs) and Pin Interconnection Matrices (PIMs).

When a cell or a component must be deleted form the array, the self-routing process must be executed for disconnecting the FU port of cells previously connected. This process will be defined as Self-derouting. This process can be at cell level or a component level.

Section C.3 and C.4 presents the flow diagrams of the Self-Routing processes implemented in CCUs and SMCUs respectively. The following section describes these processes.

4.7 Self-Routing at Cell Level

The self-routing algorithm at cell level is executed since the insertion of the second cell of a component, each time that the self-placement process ends. The algorithm allows interconnecting the ports of the functional unit of two cells, in the same component, through the local and remote cell ports. The local input-output ports are used exclusively to interconnect the FU ports of two neighbor cells (north, east, south or west). The remote ports are used when the cells to be connected are not adjacent or when the local ports are already occupied.

This process is divided in two parts. The first consists in locating the source and the target cell inside the cell array for a specific connection. For this procedure, the INET is used. The second is an expansion process between the source and target cells. This process is executed each time that the source and target cells are configured for a new connection. The cell multiplexers are dynamically configured to interconnect the desired ports. In this process, some bits of the expansion ports are used (see section 2.10 for details).

4.7.1 Configuration of source and target cells for cell connections

The first cell that starts the self-routing process is the newly inserted cell, which activates its *source_cell_flag*.

1. The *source_cell* goes through its output connections table, and sends messages to cells of the same component.

2. If the *address* of a busy cell that receives this message matches with the *address* of the message, this cell activates its *target_cell_flag* and sets the acknowledge signal to the *source_cell*. Then, the *source_cell* starts the "Expansion Process at Cell Level" to configure the path.

3. If the *source_cell* does not receive the acknowledge signal, it will continue with its remaining connections.

4. When the *source_cell* (new inserted cell) ends with its output connections table, it broadcasts a message including its address, asking for cells in the component able to route their output connections with this cell. Then, it resets its *source_cell_flag* and activates its *target_cell_flag*.

Hitherto, the new inserted cell tried to route all its output connections, which depends of the cells already placed. The output connections that has not been routed, will be made later, when cells that are inserted later complete the whole process. The algorithm continues with the routing of the input ports of the new inserted cell.

5. The cells of the component that have at least one output connection to be routed to the *target_cell* start an elimination process, where the winner is the leftmost uppermost cell.

Signal name	Description
neighbor_out_X *neighbor_in_X*	Used in the Search Phase. These signals are used to find the *target_cell* in neighboring cells.
propagate_out_X *propagate_in_X*	Used in the Search Phase. These signals are used to find the *target_cell* in the cell array.
lock_out_X *lock_in_X*	Used in the Configuration Phase. These signals are used in the reverse process of the Search Phase, to configure the multiplexers in the cells included in the path.
local_port_out_X *local_port_in_X* *remote_port_out_X* *remote_port_in_X*	Used in both phases, the Search and Configuration. These signals are used to indicate the local or remote port that will be used in case to configure the path.
id_target_port_out *id_target_port_in_X*	Used in the Search Phase. These signals are used to indicate the connection target port of the *target_cell*.

Table 4.1: Description of expansion port signals used by the Expansion Process at Cell Level.

6. The winner cell activates its *source_cell_flag*. This new *source_cell* looks up its output connections table and starts the "Expansion Process at Cell Level" to configure all connection paths between source and target cells.

7. When the *source_cell* ends routing all possible connections with the *target_cell*, it broadcasts a message asking for other cells in the component able to make their output connections with the *target_cell*.

8. Steps 5., 6. and 7. are repeated until all connections are completed with the *target_cell*. When the process ends, the last *source_cell* sends a message to the GCU indicating the end of self-routing and self-placement process for the inserted cell.

Section C.3.1 shows the algorithm used for the selection of source and target cells. When this cells are selected, the "Expansion Process at Cell Level" can be executed. The total time of the self-placement and self-routing processes of the new inserted cell depends on some factors, like the placement order and the component interconnections that includes the number of connections between cells (inputs and outputs).

4.7.2 Expansion Process at Cell Level

Once the source and target cells of a component are ready, the expansion process starts, configuring the cell multiplexers to interconnect the FU ports between those cells. The signals that participate in this process are explained in Table 4.1.

When a cell has to be connected to another one, the system first checks if the cells are neighbors. In that case, and if a free local port exists between them, they get connected by means of the local ports, otherwise the connection is tried with a remote connection resource.

The path configuration is divided in two phases. The first one is the **Search Phase at Cell Level**. This starts from the *source_cell* and looks for the *target_cell* in the array. During the process some information is stored, in a distributed way, in every cell that has been visited. The phase ends when the *target_cell* has been found. If the first phase exhausted any possibility to find the *target_cell* without reaching it, then no path exists between the *source_cell* and the

target_cell, and there is no reason to continue with the second phase. Then, the *source_cell* sends an error message to the GCU and the self-routing process ends.

The second phase recovers the information generated by the first one for the path configuration. This is called the **Configuration Phase at Cell Level**, it starts from the *target_cell* to the *source_cell* using the propagation information stored during the first phase of the algorithm.

Sections C.3.2, C.3.3 and C.3.4 presents the flow diagrams related with the expansion process at cell level.

Search Phase at Cell Level

The Search Phase is started by the *source_cell*. This is an expansion process that propagates signals in the sides of the cell that have available routing resources, like local and/or remote free ports. The propagation process includes the configuration of signals for each side (north, east, south and west) of the cell as follows:

a Configuration of propagation signals:

- This condition is valid only for the *source_cell*: If the side of the cell has any free local port, then that side sets the *neighbor_out* signal and configures the *local_port_id_out* signal in the expansion port.
- For propagation of any cell (including the *source_cell*): If the side of the cell has any free remote port, then that side sets the *propagate_out* signal and configures the *remote_port_id_out* signal in the expansion port.

b The cell assigns to the signal *id_target_port_out* the value:

- For *source_cell*, the identification number of the Functional Unit target port for the connection with the *target_cell*. This number is read from the output connections table of the *source_cell*.
- For other cells, the signal *id_target_port_in* read from the port that receives the propagation signal.

c After the configuration of the signals previously mentioned, the cell goes to the state *lock_cell*, where it waits for the activation of any *lock_in* signal.

The propagation previously explained is executed by each cell that reads the *propagate_in* signal. When a cell receives more than one *propagate_in* signal simultaneously it follows a priority order (NORTH, EAST, SOUTH and WEST) and it serves only the highest priority signal received. The cell receiving the *propagate_in* signal stores the side from which the expanding cell accessed it by means of the *origin* register. The propagation process explores the entire cell array until finding the *target_cell*, if possible.

It is important to mention that busy cells also participate in the expansion process since the remote connections can also cross busy cells. Nevertheless, this process is completely transparent to the internal cell operation since it is executed using dedicated resources, at the placement and routing layer.

Configuration Phase at Cell Level

The Configuration Phase starts when the Search Phase finds the *target_cell*. This occurs when any propagation signal in any side of the *target_cell* is activated, then the *target_cell* proceeds in priority order as follows:

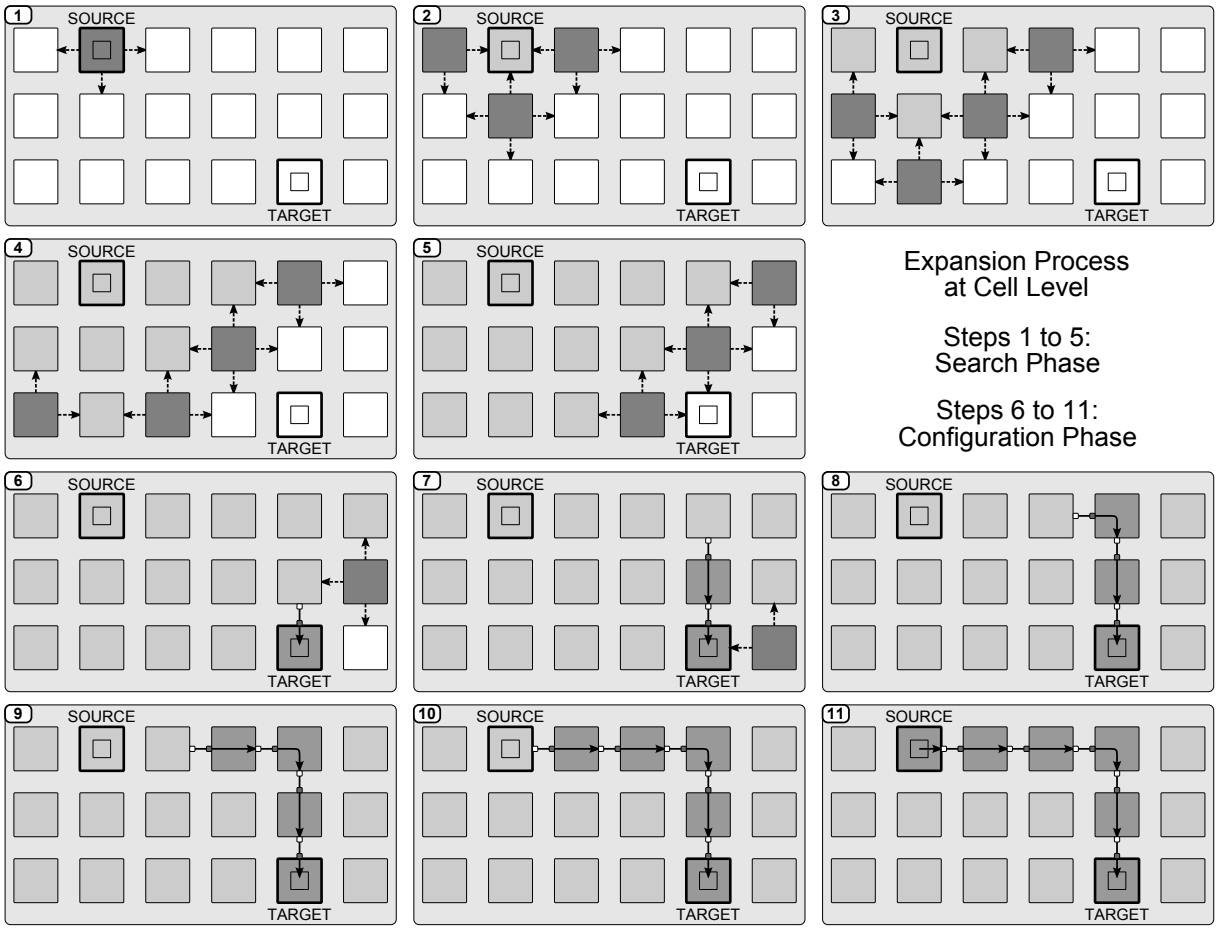


Figure 4.4: Example of Expansion Process at Cell Level.

1. If the *target_cell* detects in any of its sides the activation of a *neighbor_in* signal, it will know that the *source_cell* is its neighbor and also its position (If not, go to step 2.). At that moment, the local connection is established. The connection is configured taking into account the signals *id_target_port_in* and *local_port_id_in*. Then the cell activates the signal *lock_out* and *neighbor_out* in the side where the *neighbor_in* signal was received, and it goes to *idle* state.

2. If the target cell detects in any of its sides the activation of a *propagate_in* signal, it will know its position. At that moment, the remote connection is established. The connection is configured taking into account the signals *id_target_port_in* and *remote_port_id_in*. Then the cell activates the signal *lock_out* in the side where the *propagate_in* signal was received, and goes to *idle* state.

The Configuration Phase that was started by the *target_cell* goes backward over the path previously configured in the Search Phase until it arrives to the *source_cell*. This process configures the multiplexers of the corresponding cells to fix the path. The cells that participate in this process are in the state *lock_cell* and perform the following actions when the signal *lock_in* in the corresponding side is activated:

1. If it is the *source_cell*: the cell configures the local or remote port taking into account the source port of the Functional Unit included in the connection table and the signal *local_port_id* or *remote_port_id* previously configured. The local port is used if the signal *neighbor_in* is active. The cell activates the signal *routing_complete* (shared by all the cells) in order to reset

all the signals related to the expansion process, and the Expansion Process for a connection ends.

2. If it is not the source cell: the cell configures the remote port of the side where the *lock_in* signal was read taking into account the *origin* register and the *remote_port_id* previously configured. The cell goes to *idle* state and the signal *lock_out* is activated in the side that indicates the register *origin*.

3. The next cell that receives the signal *lock_in* starts the process again from step 1.

Figure 4.4 shows an example of the Search and Configuration Phases of the Expansion Process at Cell Level. This example assumes that routing resources of all cells are available. Step 1 to 5 represents the Search Phase, that starts from the *source_cell* propagating signals until it finds the *target_cell*, then the Configuration Phase starts (steps 6 to 11). Note that the propagation of signals continue in steps 6 and 7, since the Expansion Process is finalized by *source_cell* in step 11. Note that the distance between *source_cell* and *target_cell* is 5, the connection is established through six routing resources (or multiplexers), five in remote port of cells and one in FU input of *target_cell*, this resources are represented with the head of the arrows in step 11.

Figure 4.3 shows a sequence that illustrate the execution of self-placement and self-routing algorithms for the component AAAA (steps 1 to 6). Note that a connection can cross a free cell (step 5). Equation (4.3) represents the execution time of the expansion process implemented for the prototype. This time is proportional to the *distance* between cells and the clock period, T.

$$T_{EP} = (4 \cdot distance + 2)T \tag{4.3}$$

4.8 Self-Routing at Component Level

When the creation of all components in system is completed, the self-routing process at component level can be executed by means of the SASM instruction `connect_component`. This algorithm implements the interconnection of cells belonging to different components through SMs (and PIMs if chip-to-chip connection is necessary).

The process is divided into two parts, the first configures the source and target cells of a connection between different components, and the second is the "Expansion Process at Component Level" that is performed through the SMs, where the Switch Matrix Configuration Unit (SMCU) and eventually Pin Interconnection Matrix Configuration Unit (PIMCU) are responsible of the configuration of the multiplexers to connect the desired ports. The interconnection of components can be done in the same cluster or in different clusters. In the latter case the SM will be interconnected with one of its eight neighbors and so on, until reaching the target cell of the connection.

The self-routing process at component level is similar to the self-routing process at cell level previously described. The process is started and completed by the cells with the difference that the expansion process is performed in the Switching Matrices, using a similar algorithm that is used by cells in the expansion process. The main difference is that matrices have eight neighbors, while cells have only four. This allows interconnecting distant cells using less routing resources (multiplexers) in comparison with routing hardware resources for cells.

If in the search process of the target cell the acknowledge signal is not received, it may be because the target cell is in another chip. In this case, the pins through which the connection between source and target will be made are configured by the GCUs of the corresponding chips, initiating self-routing processes at the component level in the chips through which the connection goes through. This process will involve the PIMCU–in addition to the SMCUs– configuring the multiplexers required to perform the desired connection.

4.8.1 Configuration of Source and Target Cells for Components Connections

The procedure is as follows:

1. The GCU of a chip starts the execution of the algorithm sending to the cell array the command *start_components_connection_inet*.

2. All cells with at least one pending connection with a cell of other component participate in the elimination process, where the leftmost uppermost is the winner cell. If there is a winner cell, it activates its *source_cell_flag* (goto step 3). If there is not a winner the GCU receives the command *end_components_connection_inet* and the process ends.

3. The *source_cell* looks up its output connections table. If there is a non-routed connection with a cell belonging to other component, the *source_cell* sends a message to find the target cell by means of command *set_target_cell_sr_component_inet* (goto step 4). If there is no more connections to route goto step 2.

4. If the target cell of a connection is in the same chip, this cell activates its *target_cell_flag* and sets the acknowledge signal to the *source_cell*. Then, the *source_cell* starts the "Expansion Process at Component Level" to configure the path with the *target_cell*, the process continues in step 3. If the *source_cell* does not receive the acknowledge signal, it is possible that the *target_cell* is located in another chip. Then, the following additional steps must be executed:

   a. The chip that contains the *source_cell* is marked as *source_chip*. The *source_cell* sends to the GCU the command *configure_chip_connection_inet* for the configuration of a connection between chips.

   b. The GCU broadcasts to chips the command *search_cell_other_chip_enet* with the *address* of the target cell and the FU target port of the connection.

   c. The chips broadcast internally the information. The chip that contains the target cell is marked as *target_chip*. The *target_cell* activates its *target_cell_flag*. The *target_chip* selects a free pin of its PIM and sends the information to the *source_chip*.

   d. The *source_chip* starts the "Expansion Process at Component Level" between the *source_cell* and the *target_pin* of the PIM. When the process ends, the *source_cell* sends to the GCU the confirmation command *cell_pin_connection_confirmation_inet*. The GCU sends to the *target_chip* the command *routing_cell_target_chip_enet* requesting the routing process.

   e. The GCU of the *target_chip* broadcasts internally the command *start_pin_cell_connection_inet*. The *target_chip* performs an "Expansion Process at Component Level" between the *source_pin* of the PIM and the *target_cell*. When the process ends, the PIM sends to the GCU the confirmation command *pin_cell_connection_confirmation_inet*.

   f. The GCU of the *target_chip* broadcasts the confirmation message *end_routing_cell_target_chip_enet*, later the GCUs broadcast internally the command *configure_chip_connection_confirmation_inet*.

   g. The connection between *source_cell* and *target_cell* at component level using PIMs has been performed, the *source_cell* continues the process from step 3.

4.8.2 Expansion Process at Component Level

The expansion process at component level can be executed for three scenarios:

- Between *source_cell* and *target_cell* when the components belong to the same chip.

Signal name	Description
propagate_out_matrix (cell) *propagate_in_matrix (cell)* *propagate_out_cell_X (SM)* *propagate_in_cell_X (SM)* *propagate_out_X (SM, PIM)* *propagate_in_X (SM, PIM)*	Used in the Search Phase. These signals are used to find the *target_cell* or *target_pin* in the chip.
lock_out_matrix (cell) *lock_in_matrix (cell)* *lock_out_cell_X (SM)* *lock_in_cell_X (SM)* *lock_out_X (SM, PIM)* *lock_in_X (SM, PIM)*	Used in the Configuration Phase. These signals are used in the reverse process of the Search Phase, to configure the multiplexers in the matrices included in the path.
id_source_port_out (cell) *id_source_port_in_X (SM)*	Used in the Search Phase. These signals are used to indicate the FU source port of a connection.
mux_id_matrix (cell) *mux_id_cell_X (SM)*	Used when the Search Phase finds the *target*. These signals are used to indicate the multiplexer used by SM for configures a connection.
matrix_port_out (SM, PIM) *matrix_port_in (SM, PIM)*	Used in both phases, the Search and Configuration. These signals are used to indicate the matrix port that will be used for the path configuration.
id_target_port_out (cell, SM, PIM) *id_target_port_in_matrix (cell)* *id_target_port_in_X (SM, PIM)*	Used in the search phase of the target cell. These signals are used to indicate the FU target port of the *target_cell* or the *target_pin* of the PIM.

Table 4.2: Description of expansion port signals used by the Expansion Process at component level.

- Between *source_cell* and *target_pin* when the components belongs to different chips (partial configuration of a connection).
- Between *source_pin* and *target_cell* when the components belongs to different chips (partial configuration of a connection).

For the description of the process, the term *source* may refer to *source_cell* or *source_pin*, whilst the term *target* my refer to *target_cell* or *target_pin*. Once the *source* and *target* are ready, the expansion process starts, configuring the multiplexers of SMs (and PIMs) for a connection. The signals that participate in this process are explained in Table 4.2.

The path configuration is divided in two phases. The first one is the **Search Phase at Component Level**. This starts from the *source* and looks for the *target* in the chip. During the process some information is stored, in a distributed way, in every SM or PIM that has been visited. The phase ends when the *target* has been found. If the first phase exhausted any possibility to find the *target* without reaching it, then no path exists between the *source* and the *target*, and there is no reason to continue with the second phase. Then, the *source* sends an error message to the GCU and the self-routing process ends.

The second phase recovers the information generated by the first one for the path configuration. This is called the **Configuration Phase at Component Level**, it starts from the *target* to the *source* using the propagation information stored during the first phase of the algorithm.

Sections C.3.2, C.4, C.4.1 and C.4.2 present the flow diagrams related with the expansion process at component level. Note that the *source* (CCU for presented flow diagrams) is the start and end point of the process.

Search Phase at Component Level

This is an expansion process that propagates signals in cells and the sides of the SMs or PIMs that have available routing resources. The Search Phase is started by the *source* as follows:

a. If the *source* is a cell: the cell sets its *propagation_out_matrix* signal. The cell reads the FU source port from output connections table and assigns the value to the signal *id_source_port_out*. If the *target_cell* is in the chip, the cell reads the FU target port from output connections table and assigns the value to the signal *id_target_port_out*. If the *target_cell* is in another chip, the cell reads the *target_pin* from a previous configured value and assigns it to the signal *id_target_port_out*

b. If the *source* is a PIM: whether the side of the PIM has any free output port, then that side sets the *propagate_out* signal. The PIM configures the *id_target_port_out* that contains the FU target port from the data read in a previous configuration.

c. After the configuration of the signals previously mentioned, the cell or PIM goes to the state *lock_component*, where it waits for the activation of any *lock_in* signal.

The SM receiving the *propagate_in_cell_X* signal stores the side from which the expanding cell accessed it by means of the *origin_cell* register. Thereafter, the propagation process continues configuring the propagation signals for SMs in each side (north, northeast, east, southeast, south, southwest, west and northwest) as follows:

d. If the side of the SM has any free port, then that side sets the *propagate_out* signal.

e. The SM assigns to the signal *id_target_port_out* the value read from the cell, SM or PIM, depending of the port that receives the propagation signal.

f. After the configuration of the signals previously mentioned, the SM goes to the state *lock_components*, where it waits for the activation of any *lock_in* signal.

The propagation previously explained (from d.) is executed by each SM that reads the *propagate_in* signal. When a SM receives more than one *propagate_in* signal simultaneously it follows a priority order (NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST and NORTHWEST) and it serves only the highest priority signal received. The SM receiving the *propagate_in* signal stores the side from which the expanding SM accessed it by means of the *origin_matrix* register. The propagation process explores the entire chip until finding the *target*, if possible.

Configuration Phase at Component Level

The Configuration Phase starts when the Search Phase finds the *target*. This occurs when any propagation signal in any side of the *target* is activated, then the *target* proceeds as follows:

- If the *target* is a cell: when the *target_cell* detects the activation of the signal *propagate_in_matrix*, the connection with the SM is established. The connection is configured taking into account the signals *mux_id_matrix* and *id_target_port_in_matrix*. Then the cell activates the signal *lock_out_matrix*, and it goes to *idle* state.

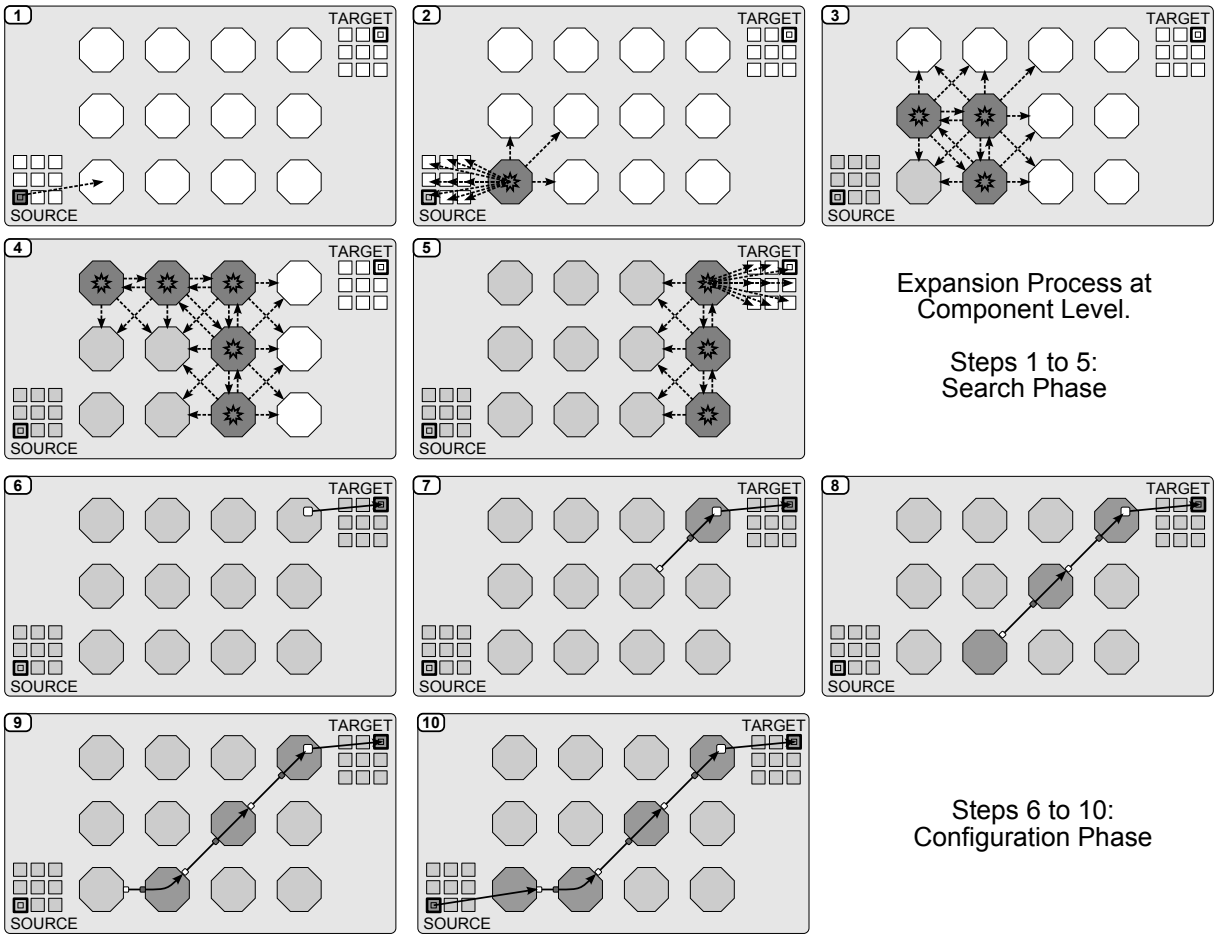


Figure 4.5: Example of Expansion Process at component level.

- If the *target* is a PIM: when the PIM detects the activation of the signal *propagate_in_X*, the connection with the SM is established. The connection is configured taking into account the signals *matrix_port_in_X* and *id_target_port_in_X*. Then the PIM activates the signal *lock_out_X*, and it goes to *idle* state.

The Configuration Phase that was started by the *target* goes backward over the path previously configured in the Search Phase until it arrives to the *source*. This process configures the multiplexers of the corresponding SMs to fix the path. The SMs that participate in this process are in the state *lock_component* and perform the following actions when the signal *lock_in* in the corresponding side is activated:

- If the *source_cell* is not in the cluster: the SM configures the port of the side where the *lock_in* signal was read taking into account the *origin_matrix* register and the *matrix_port* previously configured. The SM goes to *idle* state and the signal *lock_out* is activated in the side that indicates the register *origin_matrix*.

The next SM that receives the signal *lock_in* repeats the previous process again until the *source* is found, then the process continues as follows:

- If the *source* is a cell: when the *source_cell* is in the cluster, the SM configures the multiplexer related with the cell X (Mux Y from cell X) taking into account the *origin_cell* register and the

signals *id_source_port_in_X* and *matrix_port_in_X* previously configured. The SM activates the signal *lock_out_cell_X* and it goes to the *idle state*. Thereafter, the cell that receives the signal *lock_in_matrix* activates the signal *routing_complete* (shared by all CUs), and the "Expansion Process at Component Level" ends.

- If the *source* is a PIM (it includes the *source_pin*): the PIM configures the port taking into account the signals *id_source_port_in_X* and *matrix_port_in_X* previously configured. Thereafter, the PIM activates the signal *routing_complete* (shared by all CUs), and the "Expansion Process at Component Level" ends.

Figure 4.5 shows an example of the Search and Configuration Phases of the Expansion Process at Component Level. This example assumes that routing resources of all SMs are available. Step 1 to 5 represents the Search Phase, that starts from the *source_cell* propagating signals until finds the *target_cell*, then the Configuration phase starts (steps 6 to 10). Note that the distance between *source_cell* and *target_cell* is 19, however the connection is established through five routing resources (or multiplexers), four in SMs and one in *target_cell*, this resources are represented with the head of the arrows in step 10.

4.9 Self-Elimination and Self-Replication

The Static Fault Tolerance mechanism included in the system may require the execution of processes for elimination and replication of cells. This mechanism is a combination between the Fault Tolerance System (FTS) and the SASM instruction `ft_configuration`, which is used for configuring the cell or cells involved in the process. Note that previous to the elimination of a cell, all its connections must be derouted as detailed in section 4.11.

When a hardware failure is detected by the FTS, the damaged cell(s) request to the CμP to execute the processes for self-eliminate and self-replicate the cell (or cells) involved in the failure. The SASM instruction `ft_configuration` executes the process as detailed in section 5.3.8. The cell data registers and its processing capacity is lost (probably corrupted). However, the routing resources and some adaptive capabilities of the cell (included in the CCU) are still working. This is due to routing resources of cell are independent of processing capabilities of cell, where the failure was detected. This means that the cell continues participating in the self-placement (as busy cell) and self-routing processes. This allows the interconnection of two distant cells in a component using the routing resources of a cell with damage in its FU.

4.9.1 Elimination of a Cell inside a Chip

The elimination of a cell is executed when the GCU sends the command *eliminate_cell_inet*. Then, the cell whose *address* matches with the *address* of frame executes the following actions:

1. Deletes the input and output connection tables.
2. Disables FU processors.
3. Deletes FU Program Memories.
4. Sets the cell as busy. Therefore, the cell will not participate in future self-placement process.
5. Sets the *address* as 0xFFFF0001. This is a special reserved address used for identification of cells that have been eliminated by the Static Fault Tolerance mechanism.

Subsequently, the cell sends the confirmation command *eliminate_cell_confirmation_inet* and the process ends.

4.10 Self-Configuration by means of Subprocesses

Any SANE application can be assumed to be a self-adaptive processing system with the MIMD architecture advantages, like multiple parallel processing [7]. The runtime self-configuration capability of the system is present when a SANE application requests the execution of at least one subprocess.

Any component in the system has the ability to start up to four subprocesses in execution time. The SASM instructions `start_subprocess_X` and `end_subprocess_X` are used for this purpose. Each subprocess may includes others SASM instructions like `create_component`, `connect_component` and `delete_component` among others, i.e., a subprocess could create, interconnect or delete components between others.

The creation and interconnection of components are capabilities that has been described in previous sections. The delete component capability will be detailed in the following section. Note that before deleting a component, all its connections must be derouted as detailed in section 4.11.

4.10.1 Delete a Component inside a Chip

The runtime Self-configuration mechanism included in the system may require to start a process for deleting components. The deletion of a component is executed when the GCU sends the command *delete_component_chip_inet*. Then, the cells whose *id_component* matches with the *id_component* included in the command execute the following actions:

1. Deletes the input and output connection tables.
2. Disables FU processors.
3. Deletes FU Program Memories.
4. Sets the cell as free. Therefore, the cell can participate in future self-placement process.
5. Sets the *address* as 0x00000000. This is a special reserved address used for identification of free cells.

Subsequently, the cells send the confirmation command *delete_component_confirmation_inet* and the process ends. Note that when a component is deleted, the cells that belonged to the component can participate in future self-placement and self-routing processes. The routing resources of cells that will be deleted in this process are not altered, i.e, if a connection between two cells crosses a cell that will be deleted, the connection remains configured after cell deletion.

4.11 Self-Derouting Process

The Self-derouting process permits to release all routing resources used to interconnect cells. This process can be executed for a single cell or a entire component. For cells, the process is executed taking into account the *address* of a specific cell (*cell_X*). For components, the process requires the identification of a component (*component_X*).

The self-derouting algorithm for a single cell is executed by the Static Fault Tolerance mechanism, before starting the processes for eliminating and replicating cells. The cell that will be eliminated is the *cell_X*. The algorithm permits to release all routing resources used to interconnect the *cell_X*, i.e., it permits to disconnect the *cell_X*.

The self-derouting algorithm for a entire component is executed when a component will be deleted, this is normally used by the runtime self-configuration mechanism by means of subprocesses in the high-level configuration file. The component that will be deleted is the *component_X*. The algorithm permits to release all routing resources used to interconnect the *component_X*, i.e., it allows to disconnect the *component_X*.

This process is divided in two parts. The first consists in locating the cells that have at least one connection with the *cell_X* or with the *component_X*. For this procedure, the INET is used. The second is the Release Process between the interconnected cells, i.e., the *source* and *target* of a connection. This process is executed for each connection with the *cell_X* or the *component_X*.

Unlike the expansion processes at cell or component level, the Release Process has only one phase, that goes from the *target* to the *source* of a connection. When a connection between components uses PIMs, the *source* of a connection can be a *source_cell* or a *source_pin*, whilst the *target* can be a *target_cell* or a *target_pin*. The Release Process is the same for *target_cell* or *target_pin*.

Sections C.3.2, C.3.5 and C.4.3 present the flow diagrams related with the release processes at cell and component level. Note that the CCU in the *target_cell* and CCU in the *source_cell* (for presented flow diagrams) are the start and end point of the process respectively.

4.11.1 Cell Selection for Derouting Process of a Single Cell

The process is described below:

1. The GCU starts the self-derouting process sending the command *delete_cell_connections_chip_inet* with the *address* of *cell_X* as argument.

2. The cells look up their input connections table. The *cell_X* and the cells that have at least one input connection with the *cell_X* participate in an elimination process, where the leftmost uppermost cell will be the winner, which will be the *target* for the Release Process. If there is not a winner cell, there are no more connections with *cell_X*, the process ends and the GCU receives the command *delete_cell_connections_confirmation_inet*.

3. The winner cell (*target*) starts the "Release Process" for disconnecting all its input connections with the *cell_X*. If the winner is the *cell_X*, this starts the "Release Process" for disconnecting all its inputs connections with other cells.

4. When the winner cell ends looking up its inputs connections table, this sends the command *delete_cell_connections_chip_inet* with the *address* of *cell_X* as argument and the process is repeated again from step 2.

4.11.2 Cell Selection for Derouting Process of a Entire Component

The process is described below:

1. The GCU starts the self-derouting process sending the command *delete_components_connections_chip_inet* with the identifier of *component_X* as argument (*id_component*).

2. The cells look up their input connections table. The cells that belongs to the *component_X*, and the cells that have at least one input connection with the *component_X* participates in an elimination process, where the leftmost uppermost cell will be the winner, which will be the *target* for the "Release Process". If there is not a winner cell, there are no more connections with *component_X*, the process ends and the GCU receives the command *delete_components_connections_confirmation_inet*.

3. The winner cell (*target*) starts the "Release Process" for disconnecting all its input connections with the *component_X*. If the winner cell belongs to the *component_X*, this starts the "Release Process" for disconnecting all its inputs.

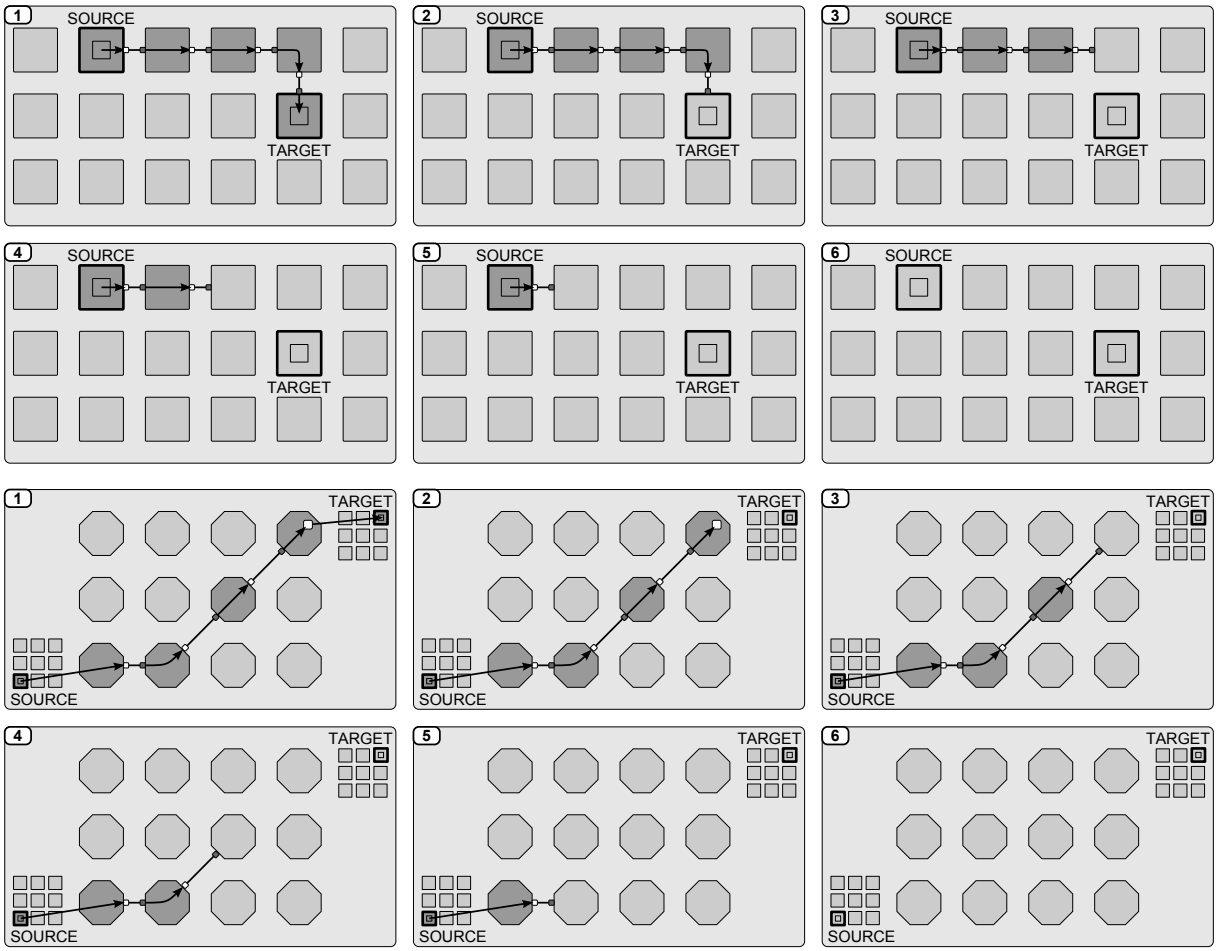


Figure 4.6: Example of Release Process at Cell and Component Level.

4. When the winner cell ends looking its input connections table, this sends the command *delete_component_connections_chip_inet* with the identifier of *component_X* as argument (*id_component*) and the process is repeated again from step 2.

4.11.3 Release Process

This process releases the routing resources used for a interconnection between cells, or between a cell and a PIM. In this process, the multiplexers of cells, SMs and PIMs are dynamically released to disconnect the desired ports. If the connection is at cell level, the multiplexers of FU inputs, local and remote cell ports are released. If the connection is at component level, the multiplexers of FU inputs, SMs and PIMs are released. Figure 4.6 shows an example of the Release Process at cell and component level.

For this process some bits of the expansion ports are used (see section 2.10 for details). The signals that participate in this process are explained in Table 4.3.

The Release Process has only one phase, that goes from the *target* to the *source* of a connection. When a connection between components use PIMs, the *source* of a connection could be a *source_cell* or a *source_pin*, whilst the *target* could be a *target_cell* or a *target_pin*. The Release Process is the same for *target_cell* or *target_pin*, so the following procedures assume the *source* and *target* as cells. Once the FU input port of the *target_cell* is selected, the release process starts, configuring the multiplexers to their initial state.

Signal name	Description
del_connection_out_X (cell, SM) *del_connection_out_cell_X* *del_connection_out_matrix* *del_connection_out_cell*	Used to indicate the port that must be released. The signal used is propagated only in one side of the cell or SM.
neighbor_out_X *neighbor_in_X*	Used to indicate that the connection is with a local port.
local_port_out_X *local_port_in_X* *remote_port_out_X* *remote_port_in_X*	Used to indicate the local or remote cell port that was used to configure the path.
matrix_port_out_X *matrix_port_in_X*	Used to indicate the SM port that was used to configure the path.
id_source_port_out *id_source_port_in_X*	Used to indicate to SM the FU input port that was used to configure the path.

Table 4.3: Description of expansion port signals used by the Release Process.

The Release Process starts from the FU input port of the *target_cell* an go backwards to the origin of a connection, i.e., the FU output port of the *source_cell*. This process reads the multiplexers configuration and propagate the signal *del_connection_out_X* in the direction where the connection was previously established by Expansion Process. The process is described bellow:

1 The *target_cell* reads the configuration of the FU input port multiplexer selected for the Release Process and performs one of the following actions:

- If the FU input port is connected to a local port, the *target_cell* propagates the signal *del_connection_out_X* in the side where the connection was established. It also propagates the signals *neighbor_out_X* and *local_port_out_X* that permits to the neighboring cell to identify the local port used for a connection.
- If the FU input port is connected to a remote port, the *target_cell* propagates the signal *del_connection_out_X* in the side where the connection was established. It also propagates the signal *remote_port_out_X* that permits to the neighboring cell to identify the remote port used for a connection.
- If the FU input port is connected to SM, the *target_cell* propagates the signal *del_connection_out_matrix*. It also propagates the signal *id_source_port_out* that permits to the matrix to identify the multiplexer used for a connection.

2 The *target_cell* releases the FU input port multiplexer, and goes to *idle* state.

3 The process continues at cell level or component level as follows:

a. At cell level: when a cell receives the signal *del_connection_in_X* proceeds as follows:

i. If the signal *neighbor_in_X* is received: it confirms that this cell is the origin of a connection, the local port multiplexer is released and the process ends.

ii. If the remote port is connected to a FU output port of the cell: it confirms that this cell is the origin of a connection, the remote port multiplexer is released and the process ends.

iii. If the remote port is connected to another remote port: the cell propagates the signal *del_connection_out_X* in the side where the connection was established. It also propagates the signal *remote_port_out_X* that permits to the neighboring cell to identify the remote port used for a connection. The process continues from a.

b. At component level: when a SM receives the signal *del_connection_in_cell_X*:

i. If the SM multiplexer is connected to a cell: it confirms that this cell is the origin of a connection. The SM propagates the signal *del_connection_out_cell_X*, the SM multiplexer is released and the process is finalized by the cell.

ii. If the SM multiplexer is connected to a SM port. The SM propagates the signal *del_connection_out_X* in the side where the connection was established. It also propagates the signal *matrix_port_out_X* that permits to the neighboring SM to identify the port used for a connection. The process continues from b.

When the process is completed, the *source* activates the signal *routing_complete* (shared by all CUs), and the "Release Process" ends. It is important to mention that busy cells also participate in the Release Process since the connections can also cross busy cells. Nevertheless, this process is completely transparent to the internal cell operation since it is executed using dedicated resources, at the routing layer.

4.12 Conclusions

This chapter describes the self-adaptive processes implemented in the system. The algorithms presented are te base for the implementation of the high-level instructions in system. These algorithms are implemented in the Configuration Units of Cells, Switch and Pin interconnection Matrices.

The self-placement algorithm is responsible for finding out the most suitable position to insert the new cell of a component. For the placement of the first cell of a component, a particular procedure is used, different from other cells. In this case, a good candidate position is defined as one where a free cell has low routing congestion and the largest number of free neighboring cells. After the insertion of the component first cell, the next cells to be inserted are placed as close as possible to the cell with the largest number of connections with the new cell.

The self-routing algorithm allows interconnecting the Functional Unit ports of two cells. This process can be executed at cell or component level. The self-routing process at cell level is executed since the insertion of the second cell of a component, each time that the self-placement process ends. The algorithm allows interconnecting the ports of the functional unit of two cells, in the same component, through the local and remote cell ports. After the insertion of all components, the self-routing process at component level can be executed. This algorithm implements the interconnection of two cells belonging to different components through Switch and Pin Interconnection Matrices.

The self-derouting process permits to release all routing resources used for interconnect cells or component. This process can be executed for a single cell or a entire component. For this purpose, the Release Process is implemented. This process releases the routing resources (multiplexers) used for a interconnection between cells.

The elimination of a single cell and the deletion of a entire component are processes used by the runtime self-configuration capabilities included in the system: (1) the Static Fault Tolerance mechanism is able to execute the self-replication and self-elimination of cells, and, (2) the dynamic reconfiguration by means of subprocess can execute the creation, connection and deletion of components, among others.

Chapter 5

Development and Implementation of Self-adaptive Applications with Parallel Processing Capabilities.

Being honest may not get you a lot of friends but it'll always get you the right ones.

Ser honesto puede que no te dé muchos amigos, pero te dará los amigos adecuados.

John Lennon (1940 – 1980)

Abstract: This chapter presents the main features for the creation of the SANE Assembler (SASM) configuration file, which is used for configuring any SANE-ASM application that could be implemented by a user. Consequently, a detailed description of format and syntax of SASM instructions is presented. Additionally, two application examples with "Dynamic" and "Static" Fault Tolerance capabilities are presented. This chapter is complemented with the appendixes D and E, which respectively present the generalities of the software tool implemented for configuring applications, and the listings of the application examples described in this chapter.

5.1 SANE ASSEMBLY Development System

One of the main inconveniences in the design phase of the architecture was the creation of applications that permit to test the system. This process consisted in the creation of a SANE ASSEMBLY (SANE-ASM), that includes a specific number of interconnected components, and interconnected cells inside each component. This functionality involved, for each cell in the application, the creation of inputs and outputs connection tables, the configuration of special registers, and the generation of hexadecimal instructions code for each processor (between 1 and 4 per cell) from programs written in the native assembler language created for the Functional Unit (FU) (Appendix A). Any modification in the application implies a lot of time rewriting the data for its configuration. Similar to any commercial general purpose device a software tool is fundamental for improving the capacity of a designer when developing applications.

The SANE Project Developer (SPD) is an Integrated Development Environment (IDE) that allows generating the memory initialization data for the Control Microprocessor (CμP) inside the prototype. The SPD allows the creation and edition of files that describe the configuration

of a SANE-ASM. This file includes high-level instructions that are defined as SANE Assembler (SASM) instructions. The file that includes these instructions is called the SASM file.

The SPD allows the insertion and edition of all related information of the SANE-ASM, this involves the configuration of the following parameters: *i*) the identification number of cells and components; *ii*) the configuration registers; *iii*) the input and output connection tables; *iv*) text descriptions of cells and components; *v*) text aliases for cells; and *vi*) creation and edition of Assembler (ASM) and SASM files. For writing the Program Memories of the processors, ASM files will be created, this permits to execute the functionality of a processor in the cell.

The SPD supports building of the hexadecimal files (SHEX and SXM) with the configuration of the SANE-ASM that will be implemented in the FPGA. This process includes the compilation of the files involved in the process according to the SASM file. The SPD generates a list of errors, warnings and infos for all files involved in the building process and guides the user to make the appropriate corrections if required. The SPD also supports the compilation of individual ASM files.

Appendix D shows a general description of the SANE Project Developer (SPD).

5.2 Overview for the Configuration of an Application

Any application scheduled to the SANE-ASM has to be organized in components, where each component is composed by one or more interconnected cells. The interconnection of cells inside of a component is made at cell level, while the physical interconnections of components are made in another layer, at the Switch Matrix (SM) level. The connections between components can be inside a chip or may span several chips. The interconnection of SANEs is just conceptual, because it takes place at the same layer of components.

Once the application has been defined and the components are ready for implementation, the main configuration file has to be created. This includes a sequence of high-level instructions and their corresponding data arguments relative to the application. The high-level instructions are called the SANE Assembler (SASM) instructions and are included in the SASM file (`*.SASM` file).

The Control Microprocessor (CμP)[1] is responsible for implementing the main configuration file that includes the SASM instructions for the configuration and execution of system functionality, even during runtime. This program is stored in a section of memory dedicated to this purpose. The CμP is able to execute the instructions shown in Table 5.1. Note that each instruction has assigned a unique identifier or Instruction Code (IC). Additionally, these instructions may or may not include other words in memory concerning to the execution of the instruction (arguments), as detailed in following sections.

The CμP executes sequentially the main program starting with the first word stored in memory, which is interpreted as a instruction. Afterwards, each word read is interpreted according to the instruction format, i.e, a word in memory could be interpreted as a SASM instruction or an argument related with an instruction. The instructions format is detailed in Section 5.3. The CμP includes a *pointer* that permits to read sequentially each word in memory. The CμP also includes a *stack* that stores temporally the value of the *pointer* when some special instructions call for it.

The algorithms for the execution of SASM instructions may require that CμP sends through the External Network (ENET) frames to Global Configuration Units (GCUs), which are responsible for controlling the self-adaptive processes inside the chips. A specific protocol is used for this purpose. Then, the chips start a negotiation process to establish the chip that has to start the

[1] The CμP replaces the External Controller (EC) in prototype.

Instructions	IC	Context
create_component connect_components delete_component	0x00 0x01 0x02	Create, interconnect or delete components in order to built dynamically a SANE that best fits the applications goals.
write_fu_memory_cr write_fu_memory_pm0 write_fu_memory_pm1 write_fu_memory_pm2 write_fu_memory_pm3	0x03 0x04 0x05 0x06 0x07	Write configuration registers and memories of processors in the Functional Unit (FU) of the Cell, in order to configure the configuration registers and the application of their processors.
restart_processors disable_processors restart_and_disable_processors enable_processors	0x08 0x09 0x0A 0x0B	Manipulation of program counters of the FU processors in order to restart, disable and enable the processors of the cells.
wait restart_processors_wait enable_processors_wait	0x0C 0x0D 0x0E	Manipulation of program counters of FU processors, and to set the system in ***wait*** mode for SANE requirements like execution of subprocesses or the Static Fault Tolerance mechanism.
end	0x0F	End of SANE-ASM configuration
start_subprocess_0 end_subprocess_0 start_subprocess_1 end_subprocess_1 start_subprocess_2 end_subprocess_2 start_subprocess_3 end_subprocess_3	0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17	Denotes the start and end of a subprocess.
ft_configuration	0x18	Configuration of original and redundant cells for static fault tolerance.

IC = Instruction Code
Note = The arguments of the instructions are not shown in this table.

Table 5.1: List of SASM or high-level instructions for initial and run-time configuration.

process to execute the command. This depends of a priority order assigned to the chips, and the utilization rate inside every chip. The selection process uses the circuit presented in Figure 2.3b.

Remember that ENET allows to interconnect the CμP with the chips by means of its GCU. Similarly, the Internal Network (INET) connects the Configuration Units (CUs) inside the chips, it includes the GCU, the Cell Configuration Units (CCUs) and the Pin Interconnection Matrix Configuration Units (PIMCUs). These networks give support to all self-adaptive process in the system. The sections 2.12 and 2.13 show details about the protocols implemented for INET and ENET respectively.

For the execution of the self-adaptive capabilities, the CμP, the GCUs and the CUs inside the chips implement low-level commands relative to the specific algorithm needed for the execution of these instructions. These commands are included in frames that go through the ENET and the INET. The description of SASM instructions in following sections include the terms *"_enet"* and *"_inet"*, which are used to discriminate the network by which the command (or process requested) is included.

5.3 Description of SASM Instructions

The SASM instructions that CμP is able to execute are shown in Table 5.1. Note that all instructions in this table have an equivalent instruction for the implementation of the main configuration file in the SPD. However, the SPD includes an additional instruction called `write_FU_memory`, which automatically includes the instructions of the table that are related with writing the FU memory (`write_FU_memory_XXX`) for a specific cell.

The following sections present a detailed description of the SASM instructions included in system. Each section presents one or more instructions depending on the context. Note that each section presents a table that shows the instruction syntax related with the SPD and the instruction format that must be included in the corresponding memory section in CμP. The **Built Project** option in SPD compiles the main configuration file included in the SASM file (**.sasm*), as well as all ASM files (**.asm*) that includes the task scheduled for processors in cells. The result is the creation of two files. The first is the SHEX file (**.shex*), which includes the instructions format shown in following sections. The other is the SXM file (**.sxm*), which is a representation of the hexadecimal value that will be downloaded to the CμP in prototype using the **Write Memory** option in the SPD.

5.3.1 Creation of Components

The SASM instruction `create_component` is used for the creation of new components in the system. Table 5.2 shows its syntax and instruction format. Note that each word of the input/output connection table is 37-bit width, therefore it is implemented in two memory positions in CμP. The eight words of the input connection table will be included in memory, whilst the number of outputs could be between 0 and 20.

SPD Syntax and arguments	`create_component`	`id_component`

Instruction format in CμP memory	`create_component_IC,`
	`id_component,`
	`comp_cells_number,`
	`num_outputs_cell_1,`
	`address_cell,1`
	`0x00,`
	`0x00000000,`
	`...`
	`0x00,`
	`0x00000000,`
	`...`
	`...`
	`num_outputs_cell_2,`
	`address_cell_2,`
	`...`
	`...`
		`//Start the creation of a component`
		`//16-bit component identifier`
		`//Zero-based num cells in component`
		`//Number of outputs cell_1 (0 to 20)`
		`//32-bit address, first cell in component`
		`//bits(36:32) connection table for IN0`
		`//bits(31:0) connection table for IN0`
		`//similar for IN1...IN3, FT_IN0...FT_IN3`
		`//bits(36:32) Output connection table #1`
		`//bits(31:0) Output connection table #1`
		`//similar for other outputs depending on`
		`//num_outputs value (0 to 20 outputs)`
		`//Number of outputs cell_2 (0 to 20)`
		`//32-bit address of second cell`
		`//input/output connection tables, cell_2`
		`//similar for other cells in component`

Table 5.2: Syntax and format for `create_component` instruction.

When the CμP executes the `create_component` instruction, it follows the next steps:

1. The CμP sends a message to the GCUs (chips) with a command and the value relative to the number of cells in the component (*cell_number_new_component_enet* + *comp_cells_number*).

2. Each GCU calculates if the component fits into the chip. The GCU knows the number of busy cells in the chip and the number of cells of the new component. If the chip has the capacity to insert the new component, it participates in the selection process.

3. The CμP starts the selection process using the command *start_contest_winner_chip_enet*. This determines which chip should make the placement of the new component. If there is not a winner chip, the CμP shows an error message and the process ends (system full).

4. The CμP sends to chips commands that start a specific process for the insertion of a new cell. Thereafter, the CμP waits for a confirmation command and continues with the next cell, as follows:

   a. First cell: The CμP sends the command and arguments: *insert_first_cell_enet* + *address* + *connection_tables*. The GCU of the winner chip translates the command *insert_first_cell_enet* to the *insert_first_cell_inet* and broadcasts the message inside the chip. The algorithm "**Self-placement of the first cell of a component**" is executed (Section 4.5.1). When the process ends the GCU receives the command *end_first_cell_inet*, which is translated and sent to the CμP as *end_first_cell_enet*. The CμP continues with the next cell.

   b. Other cells: The CμP sends the command and arguments: *insert_other_cell_enet* + *address* + *connection_tables*. The GCU of the winner chip translates the command *insert_other_cell_enet* to *insert_other_cell_inet* and broadcasts the message inside the chip. The algorithms "**Self-placement of other cells of a component**" and "**Self-routing at cell level**" are executed (Sections 4.5.2 and 4.7). When the process ends the GCU receives the command *end_other_cell_inet*, which is translated and sent to the CμP as *end_other_cell_enet*. The CμP continues with the next cell.

5. When the self-placement and self-routing processes for all cells of a component have finalized, the CμP continues with the next SASM instruction.

Each time that a cell is inserted, the GCU of the chip sets to high two flags: *components_connection_enable* and *delete_components_or_cells_enable*. These flags are used for instructions `connect_component`, `delete_component` and `ft_configuration` to indicate that a chip is able to connect components, delete components and eliminate cells respectively.

5.3.2 Connection of Components

After creation of all components in a SANE-ASM, the SASM instruction `connect_component` must be executed. This instruction permits to interconnect all components, i.e., performs the interconnection of cells at component level through Switch Matrices (SMs). This command does not include arguments. Table 5.3 shows their syntax and instruction format.

SPD Syntax and arguments	`connect_component`
Instruction format in CμP memory	`connect_component_IC, //Interconnecting cells at component level`

Table 5.3: Syntax and format for `connect_component` instruction

Each time that a cell has been inserted inside a chip using the instruction `create_component`, the GCU sets to high the flag *components_connection_enable*. This flag indicates to the chip that

it is able to execute the algorithm related with the components connection: "**Self-Routing at Component Level**" (Section 4.8)

For the understanding of the following steps, let us assume that in at least two chips in system have been inserted new components, i.e, these chips have the flag *components_connection_enable* activated.

1. The CμP sends a message to the GCUs (chips) with the command *autoset_for_components_connection_enet*. Each GCU that has active the flag *components_connection_enable* is enabled to participate in the selection process.

2. The CμP starts the selection process using the command *start_contest_winner_chip_enet*. This determines which chip will start the algorithm. If there is not a winner chip the process ends, therefore the CμP executes the next SASM instruction.

3. The CμP sends to chips the command *start_components_connection_enet*. The winner chip translates and broadcasts the command inside the chip: *start_components_connection_inet*. The winner chip executes the "**Self-Routing at Component Level**" algorithm (Section 4.8). It goes through the output connection tables of cells inside the chip and performs all possible interconnections at component level.

4. When there are no more possible connections at component level for the winner chip, the flag *components_connection_enable* is cleared. This chip will not participate in future selection process. The GCU receives the command *end_components_connection_inet*, which is traslated and sent to CμP using the command *end_components_connection_enet*. The CμP repeats the process from step 1. until all connections at component level will be performed. Note that loop is broken in step 2.

5.3.3 Delete Components

The SASM instruction `delete_component` is implemented when the deletion of a component is required. This is useful for runtime self-configuration, where the dynamic creation and deletion of components could be required by subprocesses. Table 5.4 presents their syntax and instruction format. This instruction requires the component identifier as argument (*id_component*).

SPD Syntax and arguments	`delete_component`	`id_component`
Instruction format in CμP memory	`connect_component_IC,` `id_component,`	`//Start process to delete a component` `//16-bit component identifier`

Table 5.4: Syntax and format for `delete_component` instruction

Each time that a cell has been inserted inside a chip using the instruction `create_component`, the GCU sets to high the flag *delete_component_or_cell_enable*. This flag indicates to the chip that it is able to execute the algorithms related with the component deletion: "**Self-Derouting Process**" and "**Delete a Component inside a Chip**" (Sections 4.11 and 4.10.1).

For the understanding of the following steps, lets assume that in at least two chips in system has been inserted new components, i.e, these chips has the flag *delete_component_or_cell_enable* activated. When the CμP executes this instruction, it follows the next steps.

1. The CμP sends a message to the GCUs (chips) with the command *autoset_for_delete_connections_enet*. Each GCU that has active the flag *delete_component_or_cell_enable* is enabled to participate in the selection process.

2. The CμP starts the selection process using the command *start_contest_winner_chip_enet*. This determines the chip that will execute the algorithm to disconnect the component. If there is a winner chip, goto next step. If there is not a winner chip goto step 5.

3. The CμP sends to chips the command *delete_component_connections_chip_enet*. The winner chip translates and broadcasts the command inside the chip: *delete_component_connections_chip_inet*. The winner chip executes the "**Self-Derouting Process**" (Section 4.11). It goes through the input connection tables of cells inside the chip and releases all routing resources related with the interconnection of a specific component.

4. When all connections of a component have been released for the winner chip the flag *delete_component_or_cell_enable* is cleared. In this case, The chip will not participate in future selection process. The GCU receives the command *delete_components_connection_confirmation_inet*, which is translated and sent to CμP using the command *delete_component_connections_confirmation_enet*. The CμP repeats the process from step 1. until all connections of a component are released. Note that loop is broken in step 2.

5. The CμP sends a frame with the information of the component to delete: *delete_component_chip_enet* + *id_component*. The GCUs broadcast the frame internally, the chip that includes the component executes the algorithm "**Delete a Component inside a Chip**" (Section 4.10.1). The chips that have at least one component activate the flag *delete_component_or_cell_enable* for future deletion processes. The appropriate confirmation command is sent to the GCU, which translates and transmits the command to the CμP and the process ends. Therefore, the CμP will be able to execute the next SASM instruction.

5.3.4 Write Functional Unit Program Memories and Configuration Registers

The Functional Unit (FU) of each cell includes four cores (CORE0 to CORE3), which can be configured to build between one to four processors as detailed in section 3.3. Each core includes a Program Memory that is identified with the same numeric order of the core (PM0, PM1, PM2 and PM3). Therefore, the SASM instructions `write_FU_memory_PM0`, `write_FU_memory_PM1`, `write_FU_memory_PM2` and `write_FU_memory_PM3` are used for writing the Program Memory of the correspondent core. Note that depending on the cell configuration mode, the PMX may or may nor correspond with the processor PX. For example in mode 0, there are four processors (P0 to P3) with 64 instructions capacity each; it corresponds with the PM of each core (PM0 to PM3). In mode 4, there is only one processor (P0) with 256 instructions capacity, which requires the concatenation of the PMs of all cores.

The FU includes some configuration registers (CRs), which could be written similarly to Program Memories since they are mapped in memory. These registers are: MODE, FAMILY, PORTS and FTCSR. The SASM instruction `write_FU_memory_CR` is used for this purpose. Table 5.5 shows the syntax and instructions format of these instructions, which requires the *address* of cell as argument.

It is important to note, that SPD includes an additional instruction called `write_FU_memory`, which automatically includes the instructions of the table that are related with writing the FU memory (`write_FU_memory_XXX`) for a specific cell. The SPD performs this action by reading the MODE register and the ASM files associated with the processors of cells. This instruction permits to reduce the length of the SASM file when multiple `write_FU_memory_XXX` instructions must be used.

The execution of these instructions is similar, so it will be explained jointly. However, note that each instruction is executed independently. The CμP executes these instructions as follows:

SPD Syntax and arguments	`write_FU_memory_CR`	`address`
	`write_FU_memory_PM0`	`address`
	`write_FU_memory_PM1`	`address`
	`write_FU_memory_PM2`	`address`
	`write_FU_memory_PM3`	`address`
	`write_FU_memory`	`address`[1]
Instruction format in CμP memory	`write_FU_memory_CR_IC,`	`//Write Configuration Regsters`
	`address,`	`//32-bit address of cell`
	`0x04,`	`//number of registers to write`
	`0x00,`	`//data for MODE register`
	`0x00,`	`//data for FAMILY register`
	`0x00,`	`//data for PORTS register`
	`0x00,`	`//data for FTCSR register`
	`write_FU_memory_PMX_IC,`	`//Write Program Memory [0,1,2,3]`
	`address,`	`//32-bit address of cell`
	`asm_instructions_number,`	`//Y number of asm instructions (1-64)`
	`0x00000000,`	`//asm instruction 1`
	`0x00000000,`	`//asm instruction 2`
	`...`	`//`
	`0x00000000,`	`//asm instruction Y`

[1] Instruction of the SPD that automatically includes other *write_FU_memory_XXX* instructions for a specific cell; it depends on the configuration mode and the length of the assembler code(s) for the processor(s) in the cell.

Table 5.5: Syntax and format for instruction related to writing Function Unit Program Memories and Configuration Register

1. The CμP sends a message to the GCUs (chips) with the *address* of the cell to which the data is directed: *set_address_program_memory_enet* + *address*.

2. The CμP sends a message to the GCUs (chips) with the data as follows:
   - To write PM0: *write_program_memory0_enet* + *data for PM0*.
   - To write PM1: *write_program_memory1_enet* + *data for PM1*.
   - To write PM2: *write_program_memory2_enet* + *data for PM2*.
   - To write PM3: *write_program_memory3_enet* + *data for PM3*.
   - To write CRs: *write_configuration_registers_enet* + *data for CRs*.

3. The GCU of each chip translates the command from ENET to INET and broadcasts the information internally as follows:
   - To write PM0: *write_program_memory0_inet* + *address* + *data for PM0*.
   - To write PM1: *write_program_memory1_inet* + *address* + *data for PM1*.
   - To write PM2: *write_program_memory2_inet* + *address* + *data for PM2*.
   - To write PM3: *write_program_memory3_inet* + *address* + *data for PM3*.
   - To write CRs: *write_configuration_registers_inet* + *address* + *data for CRs*.

4. The cell whose *address* matches with the *address* of the frame performs the write operation of the data frame to FU program memory or Configuration Registers. This cell sends the appropriate confirmation command to the GCU, which translates and transmits the command to the CμP and the process ends. The CμP starts the execution of the next SASM instruction.

SPD Syntax and arguments	`restart_processors`
	`disable_processors`
	`restart_and_disable_processors`
	`enable_processors`

Instruction format in CμP memory	`restart_processors_IC, //Restart all processors`
	`disable_processors_IC, //Disable all processors`
	`restart_and_disable_processors_IC,//Restart & disable all proc...`
	`enable_processors_IC, //Enable all processors`

Table 5.6: Syntax and format for instructions related to management of processors in the SASM file.

5.3.5 Restart, Enable and Disable Processors

The initial state of all processors in system is disabled. The processors of a specific cell are enabled automatically when any instruction that write the FU memory is executed (`write_FU_memory_XXX`). The SASM instructions presented in Table 5.6 permit to restart, disable or enable all processors in system.

The instruction `restart_processors` could be used to restart the processors after the configuration of a SANE-ASM. Therefore, all processors restart the execution of their scheduled process at the same time. The threads start their execution from origin `0x0`.

The instruction `disable_processors` is used to disable the execution of processors in system. This could be implemented before the execution of other SASM instructions to ensure that processors do not executes an undesirable action, e.g., if a processor executes a subprocess before the configuration of the entire SANE-ASM, or, if the user wants to disable processors whilst the execution of a subprocess.

The instruction `restart_and_disable_processors` is similar to `disable_processors`. Additionally, this instruction permits to restart the processors in system at the same time that they are disabled. The threads start their execution from origin `0x0` when they are enabled again.

The instruction `enable_processors` is used to enable the execution of threads in the FU processors. The threads start their execution from origin `0x0` or any other, depending on previous instruction used (`restart_and_disable_processors` or `disable_processors`).

The execution of these instructions is similar, therefore it will be explained jointly. However, note that each instruction is executed independently as presented bellow:

1. The CμP sends a frame to the GCUs with the corresponding command:
   - To restart the processors: *restart_processors_enet*
   - To disable the processors: *disable_processors_enet*
   - To restart and disable the processors: *restart_and_disable_processors_enet*
   - To enable the processors: *enable_processors_enet*
2. The GCU of each chip translates the command from ENET to INET and broadcasts the information internally as follows:
   - To restart the processors: *restart_processors_inet*
   - To disable the processors: *disable_processors_inet*
   - To restart and disable the processors: *restart_and_disable_processors_inet*
   - To enable the processors: *enable_processors_inet*
3. The cells perform the appropriate operation for the FU processors. These instructions do not send any confirmation command, therefore the CμP may continue with the next instruction.

5.3.6 System in *"Wait"* State for Runtime Self-configuration

Remember that runtime Self-configuration is possible when the Static Fault Tolerance mechanism or when the execution of subprocesses are included in the SASM file. Therefore, when the system requires the execution of one of these features, the CμP must be in a special state, where it waits for an event coming from any cell in the system, which requests the execution of special SASM instructions: `ft_configuration`, `start_subprocess_X` and `end_subprocess_X`. This special state is called "***wait***" state, and it is included in the instructions presented in Table 5.7. Note that some of the instructions presented include additional functionalities like the option to restart or enable the processor in system, which were described in the previous section.

SPD Syntax and arguments	`wait`
	`restart_processors_wait`
	`enable_processors_wait`

Instruction format in CμP memory	`wait_IC, //CMP goto wait state: CMP->wait`
	`restart_processors_wait_IC, //CMP->wait, processors are restarted`
	`enable_processors_wait_IC, //CMP->wait, processors are enabled`

Table 5.7: Syntax and format for instruction regarding configuration of system in "***wait***" state.

The execution of these instructions is similar, therefore it will be explained jointly. However, note that each instruction is executed independently as follows:

1. The CμP stores the position where the pointer is located (*stack* ⇐ *pointer*). Afterwards, the CμP sends a frame to the GCUs with the corresponding command:
   - Only for *wait* state: *wait_enet*.
   - For *wait* state and restart processors: *restart_processors_wait_enet*.
   - For *wait* state and enable processors: *enable_processors_wait_enet*.
2. The CμP goes to *wait* state. The CμP waits for an event (or request) coming from any cell in system (this request implies the runtime self-configuration, which could be for the execution of a subprocess using the instructions `start_subprocess_X` and `end_subprocess_X`, or for the execution of a process to replicate and eliminate cells using the instruction `ft_configuration`).
3. The GCU of each chip translates the command form ENET to INET and broadcasts the information internally as follows:
   - Only for *wait* state: *wait_inet*.
   - For *wait* state and restart processors: *restart_processors_wait_inet*.
   - For *wait* state and enable processors: *enable_processors_wait_inet*.
4. The cells in the system sets to high the bit *System Wait State*, which belongs to the register SUBPCSR (this register is closely tied with the execution of subprocesses). Each cell performs the corresponding action as follows:
   - Only for *wait* state: the cells goes to *standby* state.
   - For *wait* state and restart processors: the cells restart their processors and goes to *standby* state.
   - For *wait* state and enable processors: the cells enable their processors and goes to *standby* state.
5. The processors continue (or restart) the execution of their scheduled task.

Afterwards, any cell in the state *standby* (current) could make a request for starting any of the following process: (i) Start a dynamic reconfiguration of the system by means of subprocesses. (Section 5.3.7) (ii) Start the self-elimination and self-replication of cell(s) when a failure is detected by the Fault Tolerance System (FTS). (Section 5.3.8).

5.3.7 Runtime Self-configuration by means of Subprocesses

Table 5.5 shows the syntax and instructions format for execution of subprocesses. The instructions `start_subprocess_X` and `end_subprocess_X` permit to group other SASM instructions within them, which will be executed when any cell in system requests it. This cell must be properly configured for this purpose. Note that `start_subprocess_X` instruction requires the component identifier as argument. It is not permitted to include a subprocess between other subprocess, neither to include the configuration of a static Fault Tolerance mechanism (`ft_configuration` instruction) inside a subprocess.

SPD Syntax and arguments	`start_subprocess_X`[1] `id_component` ... `end_subprocess_X`[2]

Instruction format in CμP memory	`start_subprocess_X_IC,` `//Start subprocess X`
	`id_component,` `//16-bit id_component for subprocess X`
	... `//SASM instructions`
	`end_subprocess_X_IC,` `//end of subprocess X`

[1] `start_subprocess_0`, `start_subprocess_1`, `start_subprocess_2` or `start_subprocess_3`.

[2] `end_subprocess_0`, `end_subprocess_1`, `end_subprocess_2` or `end_subprocess_3`.

Table 5.8: Syntax and format for instruction related with execution of subprocesses.

In the initial configuration or when the system is not in *wait* state, the instructions related with a subprocess are ignored, this condition includes the instructions inside the subprocess.

The runtime self-configuration mechanism by means of subprocesses is closely tied with the execution of the following instructions: *i*) `wait`; *ii*) `restart_processors_wait`; and *iii*) `enable_processors_wait`. After the execution of any of these instructions, the CμP is in *wait* state, which means that it is waiting for a message from a cell to start a subprocess. Remember that CμP saves the location of the current instruction (*stack* ⇐ *pointer*), which corresponds with any instructions that includes the label (`wait`).

Some bits of the register SUBPCSR must be configured appropriately for the execution of subprocesses. Therefore, the CCU of a cell whose processor sets the bit *EXECUTE_SUBPROCESS (EXSP)* starts the procedure detailed bellow for the execution of a subprocess.

The following procedure is detailed for a subprocess X, where X could be any of the four instructions available for subprocesses (0, 1, 2 or 3). The process starts when the bit *EXSP* is set to high for any processor. The FU notifies to the CCU the activation of this bit, an the CCU requests to the CμP the execution of a subprocess as follows:

1. The CCU sends a frame with the corresponding information: command *start_subprocessX_inet* + *id_component*.

2. The GCU translates and broadcasts the information through ENET by means of the command *start_subprocessX_enet*.

3. The CμP searches in the code the instruction related with the subprocess requested (`start_subprocess_X`). Note that the argument of the instruction must match with the data received, which is the identifier of the component (*id_component*).

4. The CμP executes sequentially the SASM instructions inside subprocess X.

5. The subprocess ends when the CμP executes the instruction `end_subprocess_X`. The CμP sends a frame with the confirmation of the end of the subprocess X by means of the command *end_subprocessX_enet* and the *id_component* as argument, which is forwarded inside the chips by the GCU using the command *end_subprocessX_inet*. The CCUs whose *id_component* matches with the data in frame notifies it to the FU, which sets to high the bit *ENDS SUBPROCESS X* that indicated the end of the execution of the subprocess X.

6. The CμP recovers the value of the pointer (*pointer* ⇐ *stack*). The CμP executes the instruction denoted by pointer that contains a SASM instruction with the label `wait` and the process continues normally.

Note that cell processors may continue their scheduled task normally whilst subprocess X is executed. It is responsibility of the user to include the appropriate validation in the ASM program of cells, or disable and enable the processors in the subprocess X using the appropriate SASM instructions.

5.3.8 Static Fault Tolerance Configuration

The Static Fault Tolerance mechanism is a combination between the Fault Tolerance System (FTS) included in the Functional Unit (FU) of cells and the SASM instruction `ft_configuration`, which is used for indicating to CμP the cells involved in the process. Table 5.9 shows the syntax and instruction format of this instruction, which configures the primary and redundant cells for the self-elimination and self-replication of damaged cells when a hardware failure is detected. If the redundant cell is not implemented, the parameter must be set to `0x00`.

SPD Syntax and arguments	`ft_configuration`	`addressPrimaryCell,addressRedundantCell`

Instruction format in CμP memory	`ft_configuration_IC,`
	`addressPrimaryCell,`
	`addressRedundantCell,`
	`...`
	`...`
		`//Static Fault Tolerance IC`
		`//32-bit address of primary cell`
		`//32-bit address of redundant cell if`
		`//implemented, 0x00000000 if redundant`
		`//cell is not implemented`

Table 5.9: Syntax and format for instruction related to Static Fault Tolerance mechanism

In the initial configuration or when the system is not in *wait* state, the instruction `ft_configuration` is ignored, the CμP continues with the execution of other SASM instructions. The primary and redundant cells (configured as arguments) involved in the Static Fault Tolerance mechanism must be properly configured by means of register FTCSR.

The `ft_configuration` instruction is closely tied with the execution of the following instructions: *i*) `wait`; *ii*) `restart_processors_wait`; and *iii*) `enable_processors_wait`. After the execution of any of these instructions, the CμP is in *wait* state, which means that it is waiting for a message from a cell to start the elimination and replication of damaged cells. Remember that CμP saves the location of the current instruction (*stack* ⇐ *pointer*), which corresponds with any instructions that includes the label (`wait`).

The CCU of a cell where the hardware failure is detected starts the process detailed bellow:

1. The processors of primary cell are disabled, i.e., no more instructions will be executed. The CCU of the primary cell starts the process for replication of cells. This CCU sends to GCU the command *replicate_cells_inet* with the *address* of the primary cell as argument. This is forwarded to CμP with the command *replicate_cells_enet*.

2. The CμP searches in memory the instruction (`ft_configuration`) whose first argument matches with the *address* included in the message (primary cell).

3. The CμP start a process for eliminating the primary and redundant cells. The *elimination* term implies that cells will not participate in future self-placement processes. The procedure is as follows:

   a. The CμP sends a message to the GCUs (chips) with the command *autoset_for_delete_connections_enet*. Each GCU that has active the flag *delete_component_or_cell_enable* is enabled to participate in the selection process.

   b. The CμP starts the selection process using the command *start_contest_winner_chip_enet*. This determines the chip that will execute the algorithm to disconnect the primary cell. If there is a winner chip, goto step c. for disconnecting the primary cell. If there is not a winner chip goto step e. for eliminating the primary cell.

   c. The CμP sends to chips the command *delete_cell_connections_chip_enet* with the *address* of the primary cell. The winner chip translates and broadcasts the command inside the chip: *delete_cell_connections_chip_inet*. The winner chip executes the "**Self-derouting Process**" (Section 4.11). It goes through the input connection tables of primary cell inside the chip and releases all routing resources related with the interconnection of this cell.

   d. When all connections with the primary cell have been released for the winner chip the flag *delete_component_or_cell_enable* is cleared. In this case, the chip will not participate in future selection process. The GCU receives the command *delete_cell_connections_confirmation_inet*, which is translated and sent to CμP using the command *delete_cell_connections_confirmation_enet*. The CμP repeats the process from step a. until all connections with the primary cell are released. Note that loop is broken in step b.

   e. The CμP sends a frame with the information of the primary cell: *eliminate_cell_chip_enet* + *address*. The GCUs broadcast the frame internally, the cell whose *address* matches executes the algorithm "**Elimination of a Cell inside a Chip**" (Section 4.9.1), which configures the damaged cell as *busy cell* and its *address* will be fixed to `0xFFFF0001` to avoid future use.

   f. The chips that have at least one component activate the flag *delete_component_or_cell_enable* for future elimination processes. The appropriate confirmation command is sent to the GCU, which translates and transmits the command to the CμP and the process of elimination of the primary cell ends.

   g. If the *address* of the redundant cell is different to `0x00`, the process of elimination must be executed again from step a. This time for the *Redundant Cell*. When elimination of primary and redundant cells is finalized, the process continues with the insertion of primary and redundant cells in different location.

4. The CμP starts the process for inserting the primary and redundant cells (replication process).

   a. The CμP sets the *pointer* to zero and starts searching the cells in the configuration memory. That means finding the instruction `create_component` that includes the primary or redundant cells.

b. When the configuration data of the primary or redundant cell is found, one of the following processes is executed, which depends on the features of the component that includes the cell:
   - **Self-placement of First Cell of a Component** (Section 4.5.1).
   - **Self-placement of Others Cell of a Component** (Section 4.5.2).

c. The previous step is repeated for the remaining cell (primary or redundant). Afterwards, the process continues with the configuration of program memories of these cells.

5. The CμP starts the process for writing the Configuration Registers or Program Memories of primary and redundant cells (replication process).

   a. The CμP sets the *pointer* to zero and starts searching all instructions related with writing the FU memory (`write_FU_memory_XXX`).

   b. The instruction `write_FU_memory_XXX` is executed if the *address* of the primary or redundant cells matches with the *address* configured for the instruction.

   c. The process ends when the instruction `end` is found.

6. The GCU asks to the system to start the process **"Self-routing at Component Level"** to route the missing connections in the system (Section 4.8).

7. The CμP recovers the value of the pointer (*pointer* ⇐ *stack*). The CμP executes the instruction denoted by pointer that contains a SASM instruction with the label `wait` and the process continues normally.

When elimination and replication processes ends, the processors in the system are enabled again and continue working (start working for replicated cells). It is user responsibility to restart the software application scheduled to the SANE, or to recover a known starting point.

5.4 Development of Applications

Remember that any SANE application can be assumed to be a self-adaptive processing system with the MIMD architecture advantages, like multiple parallel processing. The runtime self-configuration capability of the system is present when a SANE application requests the execution of at least one subprocess. For this case, Listing 5.1 shows an example of the syntax of the high-level configuration file (or SASM file)[2]. Despite this example does not represent the solution of a problem and only shows the structure of an application with subprocesses, the following features could be abstracted from the code:

- The SPD permits to simplify the instructions for writing the configuration registers and program memories of FU. Note that lines 8 to 12 could be simplified using the instruction *write_FU_memory*, which is used along this listing.

- The subprocesses are ignored (or skipped) the first time the code is executed, i.e., the initial configuration starts from the line 49.

- From line 49 to line 56, the basic configuration of the system is performed. It includes a instruction to restart and disable the processors, creation of components, writing of configuration registers and program memories for processors, and the connection of components.

[2]The special characters semicolon (`;`) or double-slash (`//`) are used to include comments in the code.

- The execution of threads starts when the instruction of line 57 permits to enable the processors. In this moment, the CμP remains in *wait* state. The cell(s) that includes the appropriate functionality can request to the CμP to start the execution of a subprocess. For this example, the cells `AAAA0001` and `BBBB0001` could start subprocesses for the component to which the cell belongs, i.e., the components `AAAA` and `BBBB` respectively. Note that a component can request the execution up to four subprocesses, which are identified with the *id_component*.

- Note that any cell in component `AAAA` may execute a dynamic reconfiguration as follows: the subprocess_0 and subprocess_2 permits to create and delete dynamically the component `DDDD`. Similarly, subprocess_1 and subprocess_3 permits to create and delete components `EEEE`, `FFFF` and `AABB`.

- Any cell in component `BBBB` may execute the subprocess_0 in line 42 to restart the processors in system.

- The instruction `end` denotes the last instruction of the SASM file.

Listing 5.2 presents the structure of a SANE-ASM that includes other case for dynamic reconfiguration, the Static Fault Tolerance mechanism. This mechanism permits to self-repair the system when a hardware failure is detected in the Fault Tolerance System (FTS) configured for the working or the redundant processor, which are located in primary and redundant cells (`cell_A_primary` and `cell_A_redundant`) . This listing includes the following features:

- The SPD lets users make definition of variables with the reserved word `equ`. Lines 4 to 11 show the syntax of these definitions.

- From line 20 to line 30, the basic configuration of the system is performed. It includes instructions to disable the processors, creation of components, writing of configuration registers, writing of program memories for processors, connection of components and enable processors.

- The cells denoted as `cell_A_primary` and `cell_A_redundant` must enable and configure appropriately the FTS.

- The execution of threads starts when the instruction of line 31 permits to restart the processors. In this moment, the CμP remains in *wait* state. If a hardware failure occurs in the *Working_Processor* or *Redundant_Processor*, the primary cell starts the processes for self-elimination and self-replication of damaged cells. This process is executed by the CμP, which starts the execution of the elimination and replication of primary and redundant cells. For this example, the cells `AAAA0001` and `AAAA0002` (`cell_A_primary` and `cell_A_redundant`) are configured for this processes with the instruction *ft_configuration*.

When an application does not include the instructions that include the self-configuration capabilities previously explained, the system executes a general purpose application with capacity for parallel processing. The dynamic reconfiguration is enabled when the SASM file ends with a instruction that includes the option to put the system in *wait* state, i.e., when the program ends with any of the following instructions: `wait`, `restart_processors_wait` or `enable_processors_wait`.

Many applications were implemented and tested in the prototype using the SPD. However not all can be presented because of space. The following sections presents two application examples that include Dynamic and Static Fault Tolerance capabilities.

Listing 5.1: Example of a SANE-ASM with Subprocesses for dynamic reconfiguration.

```
;*****************************************************
;* Subprocesses are ignored in the initial configuration
;*****************************************************
;* Subprocesses for component 0xAAAA
;*****************************************************
start_subprocess_0      0xAAAA     //Subprocess 0 for component 0xAAAA
  create_component      0xDDDD
  write_FU_memory_CR    0xDDDD0001; These instructions
  write_FU_memory_PM0   0xDDDD0001; are equivalent
  write_FU_memory_PM1   0xDDDD0001; to the instruction
  write_FU_memory_PM2   0xDDDD0001; 'write_FU_memory  0xDDDD0001'
  write_FU_memory_PM3   0xDDDD0001; for the SANE Project Developer
  connect_component
end_subprocess_0
;*****************************************************
start_subprocess_1   0xAAAA     //Subprocess 1 for component 0xAAAA
  create_component   0xEEEE
  create_component   0xFFFF
  create_component   0xAABB
  write_FU_memory    0xEEEE0000
  write_FU_memory    0xEEEE0001
  write_FU_memory    0xEEEE0002
  write_FU_memory    0xFFFF0000
  write_FU_memory    0xFFFF0001
  write_FU_memory    0xAABB0000
  write_FU_memory    0xAABB0001
  connect_component
end_subprocess_1
;*****************************************************
start_subprocess_2   0xAAAA     //Subprocess 2 for component 0xAAAA
  delete_component   0xDDDD
end_subprocess_2
;*****************************************************
start_subprocess_3   0xAAAA     //Subprocess 3 for component 0xAAAA
  delete_component   0xEEEE
  delete_component   0xFFFF
  delete_component   0xAABB
end_subprocess_3
;*****************************************************
;* Subprocesses for component 0xBBBB
;*****************************************************
start_subprocess_0   0xBBBB     //Subprocess 0 for component 0xBBBB
  restart_processors
end_subprocess_2

;*****************************************************
;* Start of SANE-ASM configuration
;*****************************************************
restart_and_disable_processors
create_component         0xAAAA
write_FU_memory          0xAAAA0001
create_component         0xBBBB
write_FU_memory          0xBBBB0001
create_component         0xCCCC
write_FU_memory          0xCCCC0001
connect_component
enable_processors_wait   ;CMP waits a request from a Cell
                         ;for the execution of a subprocess
end                      ;End of SASM configuration file
```

Listing 5.2: Example of a SANE-ASM with Subprocesses for dynamic reconfiguration.

```
;***************
;* Definitions
;***************
component_A        equ  0xAAAA       //Component with four cells
cell_A_primary     equ  0xAAAA0001
cell_A_redundant   equ  0xAAAA0002
cell_A_function3   equ  0xAAAA0003
cell_A_function4   equ  0xAAAA0004
component_B        equ  0xBBBB       //Component with two cells
cell_B_function1   equ  0xBBBB0000
cell_B_function2   equ  0xBBBB0001
;*********************************************************************
;* Fault Tolerance configuration is ignored in the initial configuration
;* If the redundant cell is not inplemented, the argument 2 must be 0x0
;*********************************************************************
ft_configuration   cell_A_primary,cell_A_redundant
;*********************************
;* Start of SANE-ASM configuration
;*********************************
disable_processors
create_component          component_A
write_FU_memory           cell_A_primary
write_FU_memory           cell_A_redundant
write_FU_memory           cell_A_function3
write_FU_memory           cell_A_function4
create_component          component_B
write_FU_memory           cell_B_function1
write_FU_memory           cell_B_function2
connect_component
enable_processors
restart_processors_wait   ;CMP waits a request from a Cell for the
                          ;elimination and replicacion of primary
                          ;and redundat cells
end                       ;End of SASM configuration file
```

5.5 Application Example: Dynamic Fault-Tolerance Scaling

As previously described, any component in the system has the ability to start up to four subprocesses in execution time. Each subprocess is a group of any number of SASM instructions, e.g., a subprocess could create, eliminate or interconnect components between others. The Dynamic Fault-Tolerance Scaling technique uses this functionality for its purpose.

Lets us say that a component in a specific SANE is used for monitoring another component that is used for computing purposes (*Monitor* and *Compute* sections respectively). The *Monitor* could use two subprocesses to create exact copies of the *Compute* and the remaining two subprocesses to eliminate it. This way the *Monitor* section has the ability to create and eliminate redundant copies of the *Compute* section when a specific condition in execution time is achieved. This principle is used for the demonstration application explained in this section.

5.5.1 Dynamic Fault-Tolerance Structure

The SANE developed for any Dynamic Fault-Tolerance application can be divided in four parts, the *Compute*, *Monitor*, *Control* and *Interface* sections.

The *Compute* section executes the functionality scheduled to the SANE. This section is

composed of a variable number of cells that implement any general-purpose application.

The *Monitor* section is composed of a variable number of cells, which has the task of monitoring the real-time functionality of the *Compute* section, thus it has the ability to stop the operation of a SANE if a fault condition is detected. This section requests the execution of four subprocesses (SP) as follows:

- SP0: Creation of first copy of *Compute* section.
- SP1: Creation of second copy of *Compute* section.
- SP2: Kill the first copy of of *Compute* section.
- SP3: Kill the second copy of of *Compute* section.

Therefore, the *Monitor* section has the ability to request the creation/killing of exact copies of the *Compute* section (maximum two), depending of the requirements of the SANE. When there is only the original *Compute* section, it is not possible to detect a fault. When there are two *Compute* sections implemented, the *Monitor* section has the ability to stop the functionality of the SANE if the comparison between the original and the copy is different. When there are two copies of the *Compute* section implemented, the monitoring system decides the continuity of the SANE, it depends if at least two of them have the same results, otherwise the *Monitor* section ends the SANE functionality.

The *Control* and *Interface* sections are included by default in the architecture. The *Control* section executes the subprocesses. This functionality is included in all cells, but it is only enabled in one cell of the *Monitor* section, which will be responsible to start the subprocesses for creating or killing the copies of the *Compute* section. The *Interface* section corresponds to the resources available for the interconnection of cells and components. This is configured for the self-placement and self-routing algorithms.

The first and second copy of the *Compute* section are not configured (placed and routed) in the initial configuration, instead it exclusively depends on the runtime application characteristics. This additional hardware is dynamically created and eliminated by the *Monitor* section of a SANE. Even if the self-placement and self-routing processes of the new hardware are not completed, the processors of the other active SANEs (or active cells) in the system continue working in parallel without any interruption. This is possible because the self-placement and self-routing processes are executed in a parallel and distributed way by the configuration units of cells, while the processors of the cells are executing their scheduled processing work.

5.5.2 Description of the application

This demonstration application is a SANE-based subsystem. It can be described as a dynamic fault-tolerance scaling technique implemented on the self-adaptive architecture described in this document. The application should be able to improve autonomously its fault tolerance features based on its current workload.

As previously mentioned, the organization of a SANE includes four sections: *Compute*, *Monitor*, *Control* and *Interface*. For this demonstrator application the *Compute* section is implemented by a single-cell component, it is a 16-bit pseudo-random number generator, which has been chosen just to illustrate the principles proposed in the architecture.

The *Monitor* section is implemented in a two-cell component. Each cell implements the subsections called *Monitor_1* and *Monitor_2*. The *Monitor_1* determines on-line the power consumption average produced by the *Compute* section. The power consumption has been divided in high, medium and low consumption thresholds; it is calculated with the average of transitions of the random sequence generated.

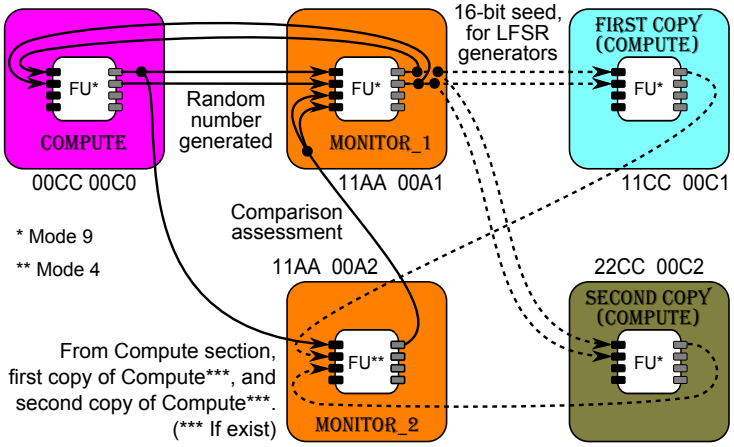


Figure 5.1: Component interconnection for the dynamic fault tolerance application.

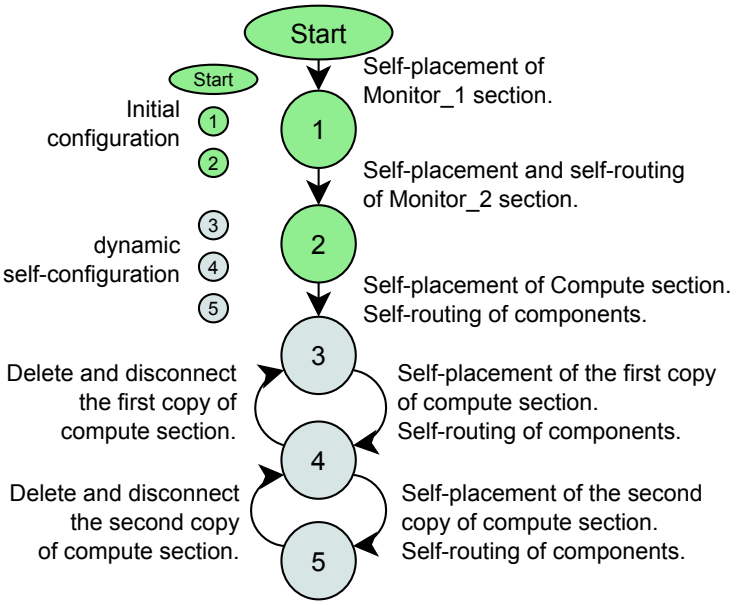


Figure 5.2: Sequence of activities. The processes executed by the system are represented by the text over the arrows. All cells are free in the "start" state.

The average for high consumption has to be larger than 11 changes. In this case, the *Monitor_1* section maintains only the original *Compute* section. In medium consumption regime, the thresholds are between 6 and 10 changes, the *Monitor_1* section maintains the first copy of the *Compute* section while the power consumption average is within this thresholds. In low consumption regime, the threshold is between 1 and 5 changes. Therefore, while the average is in low consumption the second copy of the *Compute* section is present in the system.

The *Monitor_2* compares the outputs provided by the original *Compute* section and its copies (if they exist), and sends the result of these comparisons to *Monitor_1*, which depending of the result of this comparisons takes the decision to stop or not the system.

The *Control* is implicit in the system, it is constituted by the CCU of the *Monitor_1* section. The *Interface* section includes the routing resources available for interconnections of cells, which includes the ports of cells, Switch and Pin Interconnection Matrices.

The first and the second copy of the *Compute* section are implemented in components

Address: (id_component + id_cell)	Cell Configuration Mode	Description
00CC 00C0	4	Compute section. Generator of a 16-bit pseudo-random number sequence
11AA 00A1	9	Monitor_1 section. Calculates the average consumption of the Compute section
11AA 00A2	9	Monitor_2 section. Comparisons of all compute sections.
11CC 00C1	9	First copy of Compute section.
22CC 00C2	9	Second copy of Compute section.

Table 5.10: Description of components for the example application: Dynamic Fault Tolerance Scaling.

composed of a single cell. The application for this *Compute* copies is exactly the same, the pseudo-random number generation. The components developed for this application and their interconnections are shown in Figure 5.1 and Table 5.10. The cells in mode 9 have one 16-bit processor with capacity for 256 instructions in program memory and 16x16 data memory. The cell in mode 4 has one 8-bit processor with 32-bytes data memory and 256 instructions capacity.

Figure 5.2 shows the processes executed in the SANE, where it is important to note the dynamic creation and kill processes of copies depending on the changing power consumption of the *Compute* section. Steps "start", 1 and 2 constitute the initial configuration of basic hardware required for the execution of the application, the *Monitor* and original *Compute* section.

Steps 3 to 5 are executed alternately and controlled by the *Monitor_1* section, which creates and kills copies of the *Compute* section depending on its average power consumption.

In the appendix E, Table E.1 presents a relation of the files created and generated by SPD regarding this application. Listings presented in section E.1 show these files.

5.6 Application Example: Static Fault-Tolerance

Lets suppose a 8-bit sequence of data that has to be generated by a processor with a capacity for 250 instructions, which has to include a Fault Tolerance System (FTS) to protect the reliability of the sequence. The sequence has to be generated at a low speed, much lower than the clock available, so it is necessary to implement a delay in the sequence. This could be implemented in the processor of another cell with a capacity for 50 instructions.

The problem can be solved in many ways, the figure 5.3 shows a specific solution. This SANE includes three components (`AAAA`, `BBBB` and `CCCC`). The cell identified as `AAAA0001` includes the primary processor, which generates the sequence, and the cell `BBBB0002` includes the redundant processor that generates the same sequence. These cells are configured in configuration mode 4 and in FT_mode 5, that allows the FTS of the primary cell making the comparison of 2 cores (Core 0 in primary and redundant cells). The cell `CCCC0003` contains a processor that performs the delay. This will be in mode 0, and its FTS will be disabled (only one processor is used, the other three could be used for future implementations).

When the primary and redundant cells generate one data of the sequence, the OUT0 port of the primary cell is written, producing the Read Enable (RE) flag. This is conducted to the cell `CCCC0003`, which waits this RE pulse from cell `AAAA0001` to start the generation of a delay

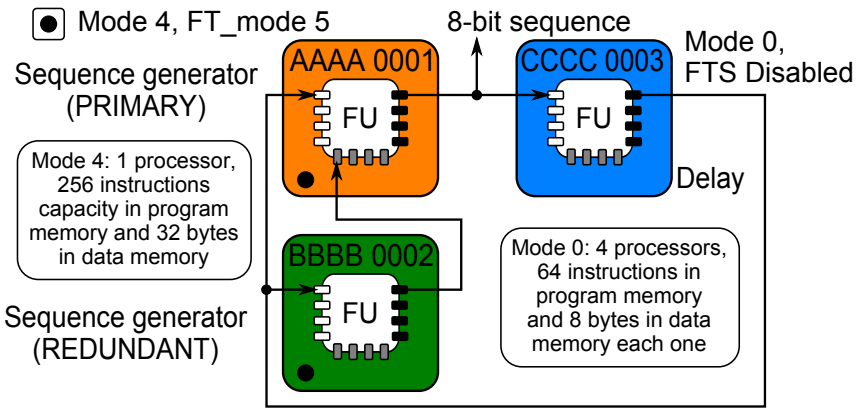


Figure 5.3: Components configuration for Static Fault Tolerance application example.

(this RE is received by means of the special instruction called BLMOV "Blocked Move", which reads the port, saves the data in a memory location and follows with the next instruction when a RE pulse is produced). The data read is not important for the cell that produces the delay (`CCCC0003`). This cell executes the delay algorithm, and then writes any data in its OUT0 port, which produces a RE pulse that has to be conducted to the inputs of the primary and redundant cells. When these cells receive the RE pulse they generate the next data of the special sequence, and so on. While the delay is performed, the primary and redundant processors could calculate the next data of the sequence in a parallel way.

If the primary or redundant processor have a hardware failure, the cells `AAAA0001` and `BBBB0002` are self-eliminated, the address `FFFF0001` will be fixed in both cells, and they will not be used in subsequent self-placement operations. Its routing resources continue available. Next, the cells `AAAA0001` and `BBBB0002` start the self-replication process, it involves the self-placement and self-routing of these cells in another location inside the cell array. Once this process ends, the sequence starts the generation again.

In the appendix E, Table E.2 presents a relation of the files created and generated by SPD regarding this application. Listings presented in section E.2 show these files.

5.7 Conclusions

The high-level configuration file for the description of a SANE ASSEMBLY (SANE-ASM) has been defined as the SANE Assembler (SASM) file. This file is composed of a sequence of SASM instructions that describes the configuration of the SANE-ASM. The syntax and instruction format for all SASM instructions has been presented. There are 25 instructions that permits the execution of the following process:

- Create, delete and interconnect components.
- Write the Functional Unit (FU) memory of cells, including the Configuration registers.
- Restart, disable or enable the processors in system.
- Configuration of system in *wait* state, which permits the execution of runtime self-configuration processes in system.
- Execution of one to four subprocess, which provides the system with dynamic reconfiguration capabilities.
- Configuration of Static Fault Tolerance mechanism, which provides the system with self-healing capability.
- Instruction to denote the end of the SASM configuration file.

Two application examples have been presented. The first is the Dynamic Fault Scaling Technique, which permits to check the dynamic reconfiguration of the system by means of subprocesses. This application is able to improve autonomously its fault tolerance features based on its current workload. Therefore, depending on a specific condition in the functional section of an application, the system dynamically either creates or kills one or two copies of its functional section. The fault tolerance subsystem compares the redundant copies of the functional section, if they exist, so as to decide the continuity of the system depending on eventual failures in the comparison.

The other mechanism of fault tolerance is a dedicated or Static Fault Tolerance mechanism. It provides redundant processing capabilities that are working continuously. In the application example, when a failure in the execution of a binary sequence in the Primary or Redundant processors is detected, the processors in Primary cell are stopped and the self-elimination and self-replication processes start for the cells involved in the failure. These cells will be self-discarded for future self-placement processes.

Chapter 6

Publications and Results

When you have the running spirit,
you look forward to life.

Cuando se tiene espíritu de atleta,
se tiene ilusión por la vida.

Max Popper (1893 – 1976)

Abstract: This chapter presents the publications performed during the elaboration of this thesis. Additionally, this chapter shows a summary of the files generated for the application , which are divided in three sections: hardware, firmware and software. The synthesis implementation results of the hardware architecture are presented at the final of the chapter.

6.1 Publications

The following is the list of publications organized from latest to oldest.

6.1.1 Neurocomputing Journal

Title:	A self-adaptive hardware architecture with fault tolerance capabilities.
Authors:	Javier Soto, Juan Manuel Moreno, Joan Cabestany.
Journal:	Neurocomputing, Volume 121, 9 December 2013, Pages 25-31. Advances in Artificial Neural Networks and Machine Learning.
Abstract:	This paper describes a Fault Tolerance System (FTS) implemented in a new self-adaptive hardware architecture. This architecture is based on an array of cells that implements in a distributed way self-adaptive capabilities. The cell includes a configurable multiprocessor, so it can have between one and four processors working in parallel, with a programmable configuration mode that allows selecting the size of program and data memories. The self-elimination and self-replication capabilities of cell(s) are performed when the FTS detects a failure in any of the processors that include it, so that this cell(s) will be self-discarded for future implementations. Other adaptive capabilities of the system are self-routing, self-placement and runtime self-configuration. Additionally, it is described as an example application and a software tool that has been implemented to facilitate the development of applications to test the system.
DOI:	http://dx.doi.org/10.1016/j.neucom.2012.10.038
ISSN:	0925-2312

6.1.2 Advances in Computational Intelligence - IWANN 2011

Title	Description of a Fault Tolerance System Implemented in a Hardware Architecture with Self-adaptive Capabilities
Authors	Javier Soto, Juan Manuel Moreno, Joan Cabestany.
Book Title	Advances in Computational Intelligence
Book Subtitle	11th International Work-Conference on Artificial Neural Networks conference, IWANN 2011, Torremolinos-Málaga, Spain, June 8-10, 2011, Proceedings, Part II.
Other	Publisher: Springer Berlin Heidelberg, 2011. Series Volume: 6692, pp 557-564.
Abstract:	This paper describes a Fault Tolerance System (FTS) implemented in a new self-adaptive hardware architecture. This architecture is based on an array of cells that implements in a distributed way self-adaptive capabilities. The cell includes a configurable multiprocessor, so it can have between one and four processors working in parallel, with a programmable configuration mode that allows selecting the size of program and data memories. The self-elimination and self-replication capabilities of cell(s) are performed when the FTS detects a failure in any of the processors that include it, so that this cell(s) will be self-discarded for future implementations. Other self-adaptive capabilities of the system are self-routing, self-placement and runtime self-configuration.
DOI	http://dx.doi.org/10.1007/978-3-642-21498-1_70
Print ISBN	978-3-642-21497-4
Online ISBN	978-3-642-21498-1

6.1.3 International Conference - Reconfig'09

Title	Implementation of a Dynamic Fault-Tolerance Scaling Technique on a Self-adaptive Hardware Architecture
Authors	Javier Soto, Juan Manuel Moreno, Jordi Madrenas, Joan Cabestany.
Publication	International Conference on Reconfigurable Computing and FPGAs, 2009. ReConFig '09, Cancun, Quintana Roo, Mexico, December 9-11, 2009. Pages 445-450.
Abstract:	The purpose of this paper is to describe a dynamic fault tolerance scaling technique that is supported by the self-adaptive features of a hardware architecture developed within the framework of the AETHER project. The architecture is composed of an array of cells that support dynamic and distributed self-routing and self-placement of components in the system. The combination of a large array of cells together with component-level routing ultimately constitutes a SANE (self-adaptive networked entity). The dynamic fault tolerance scaling technique proposed in this paper permits a given subsystem to modify autonomously its structure in order to achieve fault detection and fault recovery. The decision to modify or not its organization is based on the actual power consumption of the system.
DOI	http://dx.doi.org/10.1109/ReConFig.2009.45
Print ISBN	978-1-4244-5293-4
E-ISBN	978-0-7695-3917-1

6.1.4 International Conference - DCIS 2008

Title	Design of a Configurable Multiprocessor for a Self-Adaptive Hardware Architecture
Authors	Javier Soto, Juan Manuel Moreno, Jordi Madrenas, Joan Cabestany.
Publication	XXIII Conference on Design of Circuits and Integrated Systems (DCIS 2008). ISBN: 978-2-84813-124-5. Grenoble, France. November 12-14, 2008.
Abstract:	The purpose of this paper is to describe the design of a configurable multiprocessor used as a functional unit of a cell that is able to perform self-placement and self-routing. An array of such cells ultimately constitutes a SANE (Self-Adaptive Networked Entity). The Functional Unit includes one to four processors working in parallel, with a programmable configuration mode that allows selecting the size of program and data memories. Data processing can be done in modes of 8, 16, 24 and 32 bits. The architecture includes special instructions and specific hardware designed to be compatible with a model of computation based on microthreads.

6.1.5 International Conference - JCRA 08

Title	Diseño de un Multiprocesador Configurable y de la Interfaz de Comunicaciones para una Arquitectura de Hardware Auto-Adaptable (Design of a Configurable Multiprocessor and the Communication Interface for a Self- Adaptive Hardware Architecture)
Authors	Javier Soto, Juan Manuel Moreno, Jordi Madrenas, Joan Cabestany.
Publication	VIII Jornadas de Computación Reconfigurables y aplicaciones (JCRA 08) (Universidad Rey Juan Carlos – URJC). ISBN: 978-84-612-5635-8. Pages 295-304. Madrid, Spain, September 18-19, 2008.
Abstract:	El propósito de este artículo es describir el diseño de un multiprocesador configurable usado como unidad funcional de una célula que es capaz de realizar procesos de auto-enrutamiento y auto-emplazamiento. La unidad funcional incluye de uno a cuatro procesadores trabajando en paralelo, cuyo modo de configuración programable permite seleccionar el tamaño de las memorias de datos (8, 16, 24 o 32 bits) y programa. Este procesador incluye hardware e instrucciones específicas para ser compatible con un modelo de computación basado en microthreads. Se describe también el diseño de la interfaz de comunicaciones necesaria para la ejecución de los procesos de auto-emplazamiento y auto-enrutamiento.

6.1.6 International Conference - ReCoSoC'08

Title	Communication Infrastructure for a Self-Adaptive Hardware Architecture
Authors	Javier Soto, Juan Manuel Moreno, Jordi Madrenas, Joan Cabestany.
Publication	Proceedings of the Reconfigurable Communication-centric Systems-on-Chip Workshop (ReCoSoC'08), ISBN: 978-84-691-3603-4, pp. 175-180, Barcelona, Spain, July 9-11, 2008.
Abstract:	The purpose of this paper is to describe the design of a communication interface between a cell array and a global configuration unit to support self-placement and self-routing capabilities. The interface is based on the I2C bus specification. The combination of a large array of such cells together with component-level routing ultimately constitutes a SANE (Self-Adaptive Networked Entity).

6.2 Code Generated

This section presents a relation of the code generated for the implementation of the self-adaptive architecture presented in this document. This code represents only the final prototype implemented. The system developed includes different types of technologies divided in three scenarios: hardware, firmware and software, which are detailed in following sections.

6.2.1 Hardware

The hardware section represents the self-adaptive hardware architecture with parallel processing capabilities presented along this document. This section was developed in two Virtex4 Xilinx FPGAs (XC4VLX60), each one includes the same configuration and hardware description written in VHDL code. Some differences has been implemented in the design for differentiating the master and slave chips in prototype. The FPGA pins configuration is different for these chips, therefore two projects were built for the implementation of the prototype. Table 6.1 presents a relation of files, code lines and a brief description of each file generated for prototype. Note that the hardware definition of the Control Microprocessor (CμP) is not included in the list, since it was generated with the Xilinx Platform Studio.

It is important to note that these files only represent the final prototype, which is a reduced approach of the architecture presented in this dissertation. This is due mainly to the physical limitations in the FPGAs used for the system implementation. It is worth noting the following consideration: in prototype the cell (without CCU) and the SM have 1558 and 1912 code lines respectively. A previous version of the prototype without FU and with two clusters (3x3 cell array each) was implemented for testing of self-routing and self-placement algorithms. For this case the cell (without CCU) and the SM have 2300 and 11900 code lines respectively. The comparison of the code lines of a cluster with 2x2 cell array (prototype) and the cluster with 3x3 cell array is a reference of the hardware complexity for these approaches, and the main reason for the implementation of the prototype described in section 2.14.

File Name	Lines	Description
CCU.vhd	1945	Cell Configuration Unit. Implements all self-adaptive algorithms for a cell, mainly for self-placement and self-routing processes.
cell_NE.vhd[1]	1558	Routing resources for cell North-East.
cell_NW.vhd[1]	1558	Routing resources for cell North-West.

Continued on next page

File Name	Lines	Description
cell_SE.vhd[1]	1558	Routing resources for cell South-East.
cell_SW.vhd[1]	1558	Routing resources for cell South-West.
Cluster_array.vhd	833	Instantiation of GCU, cells, SM and PIM.
Commands.vhd	207	Library for global definition of special addresses, commands for INET and ENET, and other definitions.
dcm1.vhd	107	Clock divider for obtaining 25 MHz clock.
GCU.vhd	1960	Global Configuration Unit. Interface between CµP and CUs inside chips. Controls the self-adaptive processes inside a chip.
Matrix.vhd[2]	1912	Switch Matrix. It includes the routing resources of SM and the SMCU, which implements the self-adaptive processes inside the SM.
Matrix_border.vhd	1172	Pin Interconnection Matrix. It includes the routing resources of PIM and the PIMCU, which implements the self-adaptive processes inside the PIM.
System_chip.vhd	281	Top module. Instantiation of CµP, cluster_array and clock_divider (dcm1).
table.vhd	62	Input/output connection tables.
winning_column_box.vhd	50	Part of the circuit for cell selection process: leftmost uppermost cell.
Mux15x1.vhd	61	Multiplexer 15x1 for 9-bit data bus. FU inputs.
Mux19x1.vhd	66	Multiplexer 19x1 for 9-bit data bus. Connects the SM port with the FU inputs in cell.
Mux20x1.vhd	67	Multiplexer 20x1 for 9-bit data bus. SM ports
Mux4x1.vhd	47	Multiplexer 4x1 for 9-bit data bus. Chip ports (PIM ports).
Mux5x1.vhd	50	Multiplexer 5x1 for 9-bit data bus. Cell local ports.
Mux7x1.vhd	52	Multiplexer 7x1 for 9-bit data bus. PIM ports.
Mux8x1.vhd	52	Multiplexer 8x1 for 9-bit data bus. Cell remote ports.
alu0.vhd	390	ALU for CORE0.
alu1.vhd	391	ALU for CORE1.
alu2.vhd	391	ALU for CORE2.
alu3.vhd	385	ALU for CORE3.
CM.vhd	115	Control Memory. Same for all cores.
DM.vhd	102	General purpose registers. Same for all cores.
DM4.vhd	316	Configuration and status registers mapped in Data Memory: IN0...IN3, OUT0...OUT3.
DM5.vhd	385	Configuration and status registers mapped in Data Memory: MODE FAMILY, PORTS, SUBPCSR and FTCSR.
FTS.vhd	143	Description of Fault Tolerance System.
FU.vhd	1211	Functional Unit top module. Instantiation and logic for all modules that belongs to the FU.
PC0.vhd	70	Program Counter for CORE0.
PC1.vhd	67	Program Counter for CORE1.

Continued on next page

File Name	Lines	Description
PC2.vhd	70	Program Counter for CORE2.
PC3.vhd	67	Program Counter for CORE3.
PM.vhd	121	Program memory. Same for all cores.
sum_com.vhd	45	Full Adder for design of ALUs.
Total	19425	Total number of VHDL code lines for prototype.

[1] Cell with routing resources for two sides (1560 lines approximately). For comparative purposes, a generic cell with routing resources for four sides may have 2300 lines approximately.

[2] Switch Matrix with routing resources for two sides and four cells (1900 lines approximately). For comparative purposes, a SM with routing resources for eight sides and nine cells may have 11900 lines approximately.

Table 6.1: List of VHDL files for hardware implementation of prototype.

6.3 Firmware

The firmware section of the application is represented by the control program implemented in the Control Microprocessor (CμP), which was implemented in the same FPGAs used for the hardware prototype. This firmware was implemented using the microprocessor MicroBlaze, which was generated by Xilinx Platform Studio (XPS). It is programmed in C language by means of the Xilinx Software Development Kit.

Table 6.2 presents a relation of files, code lines and a brief description of the firmware developed for the system.

File name	Code Lines	Description
main.c	1898	Main program implemented in CμP.
print.h	69	Functions for writing data in UART. This information is presented in the communication tab of SPD.
uart.h	602	Uart configuration and algorithms for coomunication with SPD by means of XMODEM-based protocol.
commands.h	217	Definitions of commands and generic values.
Total	2786	Total number of code lines implemented for CμP in prototype.

Table 6.2: List of C files for firmware section of prototype (Control Microprocessor).

6.4 Software

The software section of the system represents the SANE Project Developer (SPD), which is a software tool developed for the creation and edition of projects that can be downloaded to prototype (See appendix D for details). This software was developed in C Sharp (C#) using the Microsoft Visual C# 2008.

Table 6.3 presents the files, code lines and a brief description of the classes created for the implementation of the SPD. Note that the list only presents the code created for the application; it is not included the code generated by the developed tool, i.e., it is not included the code that contains the configuration of the forms, which is normally created in the files FormX.Designer.cs.

File Name	Lines	Description
AppData/ApplicationData.cs	586	Object for serialization of application data.
AppData/FormAppData.cs	736	Form events management for edition of application data.
AppData/FormSaneApps.cs	68	Form events management for configuration of default location of projects.
AppData/SerializableColor.cs	59	Color object with XML serialization.
AppData/SerializableFont.cs	77	Font object with XML serialization.
Compiler/AsmCodeLine.cs	138	Object used for discrimination of instructions, arguments and others when a ASM file is compiled.
Compiler/AsmInstructions.cs	84	Object used for definition of ASM instructions.
Compiler/Compile.cs	1493	Object used for compiling ASM files. Generation of HEX files.
Compiler/Output.cs	69	Object used for generation of output results for compiling and building processes.
Compiler/Build.cs	1717	Object used for building the project. Generation of SHEX and SXM files.
Compiler/SasmCodeLine.cs	162	Object used for discrimination of instructions, arguments and others when a SASM files is built.
Compiler/SasmInstructions.cs	74	Object used for definition of SASM instructions.
Figure/Figure.cs	1086	Object for generation of a Figure of prototype.
Form/Form1.cs	1213	Management of events for main Form.
Form/Form2.Tree.cs	435	Partial class Form1. Management of events left tree.
Form/Form3.ProjectTab.cs	1850	Partial class Form1. Management of events for Project Tab.
Form/MenuCommunication.cs	156	Partial class Form1. Management of events for menu communications.
Form/MenuFile.cs	1488	Partial class Form1. Managements of events for File menu.
Form/MenuProject.cs	394	Partial class Form1. Management of events for Project menu.
Form/MenuTools.cs	129	Partial class Form1. Management of events for Tools menu.
Form/Tab.cs	93	Object for management of Tab pages.
Other/FormAbout.cs	47	Management of About form for credits in SPD.
Project/Cell.cs	165	Object with the parameters of a cell.
Project/CellsArray.cs	1895	Object for management of the cell array.
Project/Project.cs	985	Object for management of the project.
Comm.cs	1372	Partial class Form1. Management of communications ports and threads.
Common.cs	231	Definition of constant values and static methods common for the application.
FindAndReplaceForm.cs	475	Management of events and methods for find and replace action in text editor.

Continued on next page

File Name	Lines	Description
FormNewFileTemplate.cs	530	Management of events and methods for creation of a new file with template wizard.
FormNewProject.cs	140	Management of events and methods for creation of new projects.
FormRenameCell.cs	48	Management of events for edition of cells.
FormSelFileProgFPGA.cs	137	Management of events for selection of file when Write process is executed.
Globals.cs	27	Static class with global values for text editor management.
Program.cs	29	Main program. Start the execution of SPD, call for main Form - Form1.cs.
SerialPortFixer	207	Solve compatibility problems for communication ports when used alternately with hyperterminal and others.
Total	18395	Total number of C# code lines for SANE Project Developer (SPD). This number does not includes the code generated automatically for configuration of Forms.

Note: Automatic files created for configuration of Forms are not included in this table.

Table 6.3: List of C# files developed for implementation of SANE Project Developer.

6.5 Synthesis Process for Prototype

The results of the synthesis process that shows the usage rate for elements of the architecture are detailed in Table 6.4 (hardware section). The Xilinx Synthesis Technology (XST) was used for the system implementation in the device selected. This table shows the usage rate for the FU, a cell, a cluster and a complete chip, this permits to have an idea of the system granularity. The program memory of the FU and the connection table have been implemented by means of the RAM blocks available in the FPGA used. After generation of bitstream the total utilization rate was 80%.

6.6 Conclusions

The publications developed during the elaboration of this thesis project have been presented. They encompass one article in the Journal *Neurocomputing* and five international conferences, two of which are referenced in electronic publications in *Springer Berlin Heidelberg* and *IEEE Xplore Digital Library.*

The code generated for the final prototype includes approximately 40.000 code lines distributed in hardware, firmware and software, this code has been developed respectively in the following languages: VHDL, C and C#.

The results of the synthesis processes are presented for the main components of the hardware architecture, this gives an idea of the granularity of the hardware prototype implemented.

Part (Description)	Slices	Slice Flip Flops	4 input LUTs	RAMB16
FPGA: chip resources, for comparison	26624 100%	53248 100%	53248 100%	160 100%
Functional Unit: Four-core configurable multicomputer	1927 7%	402 1%	3715 6%	4 2%
cell: the routing resources of two sides were eliminated	3537 13%	992 1%	6771 12%	6 3%
cluster: 2x2 cell array + switch matrix	15920 59%	4086 7%	30315 56%	24 15%
chip: cluster + GCU + μP of control + pin interconnection matrix	19560 73%	9638 18%	36418 68%	56 35%

Note: the total utilization rate was 80% after generation of bitstream.

Table 6.4: Results of the synthesis process for the proposed prototype.

Chapter 7

Conclusions and Future Work

Somewhere, something incredible is waiting to be known.
En algún sitio algo increíble espera ser descubierto.
Carl Sagan (1934 – 1996)

Abstract: This chapter presents general conclusions related with the architecture developed. Additionally, some research lines are suggested as future work in this research area.

7.1 Conclusions

A novel self-adaptive hardware architecture with parallel processing capability has been developed. Basically, this is an unconventional MIMD hardware architecture with self-adaptive capabilities including self-placement and self-routing, which due to its intrinsic design, enable the development of systems with runtime self-configuration, self-repair and/or fault tolerance capabilities.

The self-adaptive capabilities of the architecture are executed autonomously and in a distributed way by cells. One of the main features of this architecture is its high degree of parallelism. The major drawback is the configuration of complex applications, where many processors have to be programmed and synchronized in order to accomplish a specific task. A new high-level programming paradigm has to be implemented, with the purpose of obtaining the maximum performance of the architecture.

The architecture presented includes a dedicated or static Fault Tolerance mechanism. It provides redundant processing capabilities that are working continuously. When a failure in the execution of a program is detected, the processors of the cell are stopped and the self-elimination and self-replication processes starts for the cell (or cells) involved in the failure. This cell(s) will be self-discarded for future self-placement processes.

The runtime self-configuration capability of the system is possible with the execution of subprocesses, which can be started by any cell in the system. This dynamic reconfiguration capability permits the implementation of a Dynamic Fault Tolerance Scaling Technique, which permits a given subsystem to modify autonomously its structure in order to achieve fault detection and fault recovery. It has the ability to create and eliminate the redundant copies of the functional section of a specific application.

A software tool and a hardware prototype that checks the functionality of the system have been developed. The SANE Project Developer (SPD) is an Integrated Development Environment that permits in a friendly way the management of projects that will be implemented in the hardware

prototype. The applications developed provides parallel processing capabilities, and may include Fault Tolerance mechanisms and runtime self-configuration.

7.1.1 About System Architecture

The proposed architecture consists of four conceptual layers:

- **First Layer - Cells:** the cells implement the self-adaptive capabilities and provide the computing capacity of the system.
- **Second Layer - Components:** the components are composed of interconnected cells.
- **Third Layer - SANE:** The Self-Adaptive Networked Entity (SANE) layer consists of a group of interconnected components. The SANE is the basic self-adaptive computing system; it has the ability of monitoring its local environment and its internal computation process.
- **Fourth Layer - SANE-ASM:** The top layer, the SANE ASSEMBLY (SANE-ASM) is composed of a group of interconnected SANEs.

The proposed architecture is composed of one or more chips and an External Controller (EC), which are interconnected by means of an External Network (ENET). Each chip includes a two-layers implementation with interconnected cells in the first level and interconnected SMs in the second level. The SANE and SANE-ASM are just conceptual and are implemented in the same layer of components. The chip includes a cluster array, a Global Configuration Unit (GCU) and Pin Interconnection Matrices (PIMs). The main features of these components are described bellow:

- **Cluster:** The cluster is composed of a 3x3 cell array and a Switch Matrix (SM).
  - **Cell:** The cell is the basic element of the proposed self-adaptive architecture. The cell consists of the Functional Unit (FU), the Cell Configuration Unit (CCU) and additional hardware that allows the interconnection between FU ports of two cells.
    - **Functional Unit:** The FU is in charge of executing the processes scheduled to the cell, i.e, it includes the processing capabilities of the cell. The FU includes four cores. Each core contains the digital elements that are used for the construction of a processor. The processor is constituted by the elements of one or more cores. Therefore the FU can have between one to four processors working in parallel. There are twelve configuration modes, where the expansion of Data and Program Memory describes the specific configuration mode. Therefore, the data processing of FU could be configured for 8, 16, 24 or 32 bits and the Program Memories can be joined or not depending on the configuration mode, allowing Program Memory sizes of 64, 128, 192 or 256 instructions. The instruction set of processors is composed of 44 instructions, which includes arithmetic, logic, shift, branch, conditional branch and special instruction for the execution of microthreads.
    - **Cell Configuration Unit:** the CCUs are responsible for the execution of the required algorithms for the implementation of the system self-adaptive capabilities, mainly the self-placement and self-routing algorithms.
  - **Switch Matrix:** The SMs permit to connect cells from two different components. The SM is connected to its eight adjacent neighbors; it is also connected to the nine cells belonging to the cluster. The Switch Matrix Configuration Unit (SMCU) participates in the component self-routing process. In this process, the FU output port of a cell in a given component is connected to the FU input port of a cell in a different component.

- **Global Configuration Unit:** The GCU is in charge of controlling the self-adaptive processes inside the chip. The GCU is interconnected with the EC through the ENET and with the internal components of the chip through the INET. The GCU is the interface between the EC and CUs inside the chip. The GCU receives and translates information related to self-adaptive processes or configuration data.

- **Pin Interconnection Matrix:** The PIMs are used exclusively for the port interconnection between cells of two components in different chips. The PIM is connected to its three adjacent clusters. The Pin Interconnection Matrix Configuration Unit (PIMCU) participates in the component self-routing process, to allow for the interconnection of the FU port of a cell with a pin of the chip. Previously to this configuration, the GCU undertook a negotiation process with other chips, with the aim of assigning a pin of the chip to connect these components.

The INET, ENET and Configuration Units (CUs) participate actively in all self-adaptive processes implemented in the architecture.

7.1.2 About the Self-Adaptive Processes

All self-adaptive processes described in this dissertation have been implemented and tested in the hardware prototype. The algorithms presented are te base for the implementation of the high-level instructions in the system. These algorithms are executed by the Configuration Units and are presented bellow:

- **The self-placement algorithm:** it is responsible for finding out the most suitable position to insert the new cell of a component. For the placement of the first cell of a component, a particular procedure is used, different from other cells. In this case, a good candidate position is defined as one where a free cell has low routing congestion and the largest number of free neighboring cells. After the insertion of the component first cell, the next cells to be inserted are placed as close as possible to the cell with the largest number of connections with the new cell.

- **The self-routing algorithm:** it permits to connect the Functional Unit ports of two cells. This process can be executed at cell or component level. The self-routing process at cell level is executed since the insertion of the second cell of a component, each time that the self-placement process ends. The algorithm allows interconnecting the ports of the functional unit of two cells, in the same component, through the local and remote cell ports. After the insertion of all components, the self-routing process at component level can be executed. This algorithm implements the interconnection of two cells belonging to different components through Switch and Pin Interconnection Matrices.

- **The self-derouting algorithm:** it permits to release all routing resources used to interconnect cells or components. This process can be executed for a single cell or a entire component. For this purpose, the Release Process is implemented. This process releases the routing resources (multiplexers) used for an interconnection between cells.

- **Self-configuration:** The elimination of a single cell and the deletion of a entire component are processes used by the runtime self-configuration capabilities included in the system: (1) the Static Fault Tolerance mechanism is able to execute the self-replication and self-elimination of cells, and, (2) the dynamic reconfiguration by means of subprocess can execute the creation, connection and deletion of components, among others.

7.1.3 About Integrated Development System

The proposed hardware architecture has been implemented and tested in a hardware prototype developed in VHDL for two chips (boards with Virtex 4 Xilinx FPGAs). Additionally, an Integrated Development Environment has been developed. This software tool called SANE Project Developer (SPD) permits the implementation of general purpose applications that include all capabilities presented along the document.

The hardware and software tools constitute the Integrated Development System, which permits to create projects that implement the self-adaptive and parallel processing capabilities presented in this dissertation. The following are some features supported by the system:

- The high-level configuration file for the description of a SANE ASSEMBLY (SANE-ASM) has been defined as the SANE Assembler (SASM) file.
- The SASM file is composed of a sequence of SASM instructions that describe the configuration of the SANE-ASM. The syntax and instruction format for all SASM instructions have been presented. There are 25 instructions that permit the execution of the following processes:
  - Create, delete and interconnect components.
  - Write the Functional Unit (FU) memory of cells, including the Configuration registers.
  - Restart, disable or enable the processors in system.
  - Configuration of system in a special state called *wait*, which permits the execution of runtime self-configuration processes in system.
  - Execution of one to four subprocesses, which provides the system with dynamic reconfiguration capabilities.
  - Configuration of Static Fault Tolerance mechanism, which provides the system with self-healing capability.
  - Instruction to denote the end of the SASM configuration file.

Two application examples have been presented. These applications has been selected because they include all self-adaptive, parallel processing and fault tolerance capabilities developed for the architecture.

- The first is the Dynamic Fault Tolerance Scaling Technique, which permits to check the dynamic reconfiguration of the system by means of subprocesses. This application is able to improve autonomously its fault tolerance features based on its current workload. Therefore, depending on a specific condition in the functional section of an application, the system dynamically either creates or kills one or two copies of its functional section. The fault tolerance subsystem compares the redundant copies of the functional section, if they exist, so as to decide the continuity of the system depending on eventual failures in the comparison.
- The other mechanism of fault tolerance is a dedicated or Static Fault Tolerance mechanism. It provides redundant processing capabilities that are working continuously. In the application example, when a failure in the execution of a binary sequence in the Primary or Redundant processors is detected, the processors in Primary cell are stopped and the self-elimination and self-replication processes start for the cells involved in the failure. These cells will be self-discarded for future self-placement processes.

7.2 Future Work

Some research topics that can be studied in the future are discussed as follows:

- Integration or development of a new high-level programming paradigm that permits the development of complex application with the features of the architecture presented in this dissertation. The purpose of this is obtaining the maximum performance of the architecture. References [29] [30] [31] [32] present relevant information about this point.

- A software model for self-adaptive architectures has been developed for the AETHER project [30]. The SANE Virtual Processor (SVP) is a thread-based model of concurrent program composition as a basis for designing and programming many-core chips. The model is based in microthreads, which are small code fragments that can be run concurrently to gain increased performance in the proposed self-adaptive hardware architecture. As mentioned previously, the integration of a new high-level software model like this and the proposed hardware architecture constitutes an important advance for the development of complex applications.

- Depending of the programming paradigm exposed in the previous items, it must be considered the implementation of a C compiler, which involves the creation of a back-end for the Functional Units processors in the cell.

- Implementation of a larger array of elements in both scenarios: chips and cell. If a prototype with more cells is implemented inside a chip, the architecture and self-adaptive algorithm should not be modified unless mew capabilities will be implemented. If more than two chips are included in a prototype, some negotiation processes must be redesigned to achieve the negotiation between chips, which is mainly used for interconnection of components in different chips.

- If a larger array of clusters inside a chip is achieved, the interface between chips currently implemented with Pin Interconnections Matrices must be redefined. Similarly, an external interface must be defined for chips, with the purpose of implementing general purpose applications that control external devices, or in general terms that permit to communicate the system with another digital device.

Appendix A

Instructions Set for Functional Unit Processors

The integrity of men is to be measured by their conduct, not by their professions.

La integridad del hombre se mide por su conducta, no por sus profesiones.

Décimo Junio Juvenal (60 – 128)

Abstract: This section shows all relevant information related with the instruction set for Functional Unit processor(s). The instructions are summarized in a table and described in alphabetical order for easy reference.

A.1 Instructions Format

Each instruction is a 25-bit word divided into an opcode and operands. The opcode represents the unique identifier of the instruction. The remaining bits of the instructions represents the operands that specifies the operation of the instruction. The instruction formats are presented in Figure A.1, while their fields are summarized in Table A.1.

The FU instruction formats are divided in the next categories:

- **Operations with literal and registers:** W and F represent data memory registers and 'k' represents a 8-bit literal or constant value. For processors with data length greater than 8 bits, the other bits of 'k' must be assumed as 0's. The destination selector 'd' is useful to assign a byte in a specific position of the entire word without modifying the other bytes, e.g., a 32-bit constant value could be assigned using four instructions MOVLF.
- **Operations with three registers:** W and Y represent data source resisters, while F represents the operation destination register. It is important to note that there is not restriction in the use of source and destination registers i.e. the source and destination registers could be the same, any combination is possible.
- **Operations with two registers:** W and F represent source and destination register respectively. These registers could be the same, therefore the source register could be modified in the same operation. INP represents a register located between address 0x20 and 0x23, wich represents different combination of input ports which depends on configuration mode and processor used.

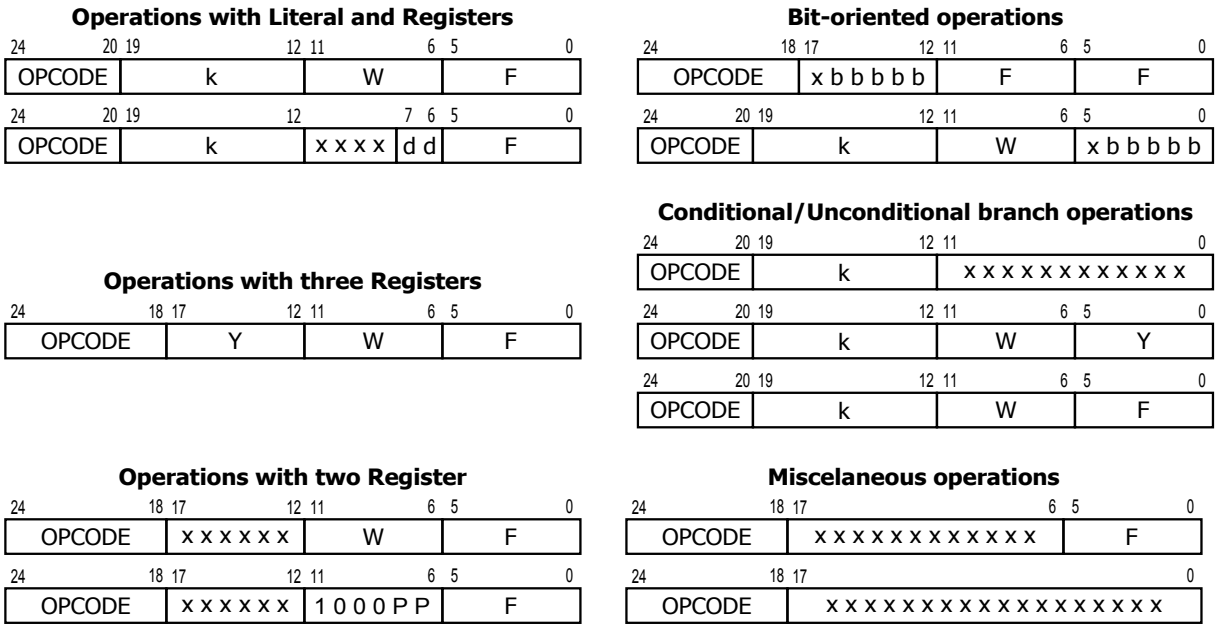


Figure A.1: Instructions format.

- **Bit-oriented operations:** F represents the register that will be affected in a set or clear bit operation, i.e., F is the source and destination register at the same time. W represents the source register used for a conditional branch to address k that depends on a bit value. 'b' represents a bit field designator which selects the bit affected by the operation. Some of these operations include conditional operations.
- **Operations with conditional and unconditional branches:** W and Y represent data source resisters, these registers will not be affected during comparison operations. F represents the operation destination register. 'k' represents the absolute destination address for this operations.
- **Miscellaneous operations:** In these operations F represents the destination register.

A.2 Instructions Set

The nomenclature shown in Table A.2 is used in the instruction descriptions throughout this section. Table A.3 shows the assembler instructions for the FU processors. Thereafter, all instructions are organized alphabetically and detailed individually. Each instruction is executed in one single clock. The oscillator frequency for processors in the prototype is 25 MHz, therefore each instruction is executed in 40 ns.

Field	Description
OPCODE	Operation code (5 or 7 bits)
W	6-bit Source register address ($0 \leq W \leq 63$)
Y	6-bit Source register address ($0 \leq Y \leq 63$)
F	6-bit Destination register address ($0 \leq F \leq 63$)
(W) (Y) (F)	Contents of W, Y and F could be 8, 16, 24 or 32 bits width, depending of configuration mode and processor selected
k	8-bit Inmediate value (operand or literal) or 8-bit Absolute address for branch (denoted as label in assembler code) ($0 \leq k \leq 255$)
b	5-bit Bit address ($0 \leq b \leq 31$). The two most significant bits could be omitted depending of the processor data width (8, 16, 24 or 32 file register).
d	2-bit Destination select ($0 \leq d \leq 3$). d=X for 8-bit processors. d=0: store result in first byte (LSB for 16, 24 or 32 bit processors) d=1: store result in second byte (MSB for 16 bit processor) d=2: store result in third byte (MSB for 24 bit processor) d=3: store result in fourth byte (MSB for 32 bit processor)
P - INP	2-bit Port(s) selection. Port(s) selected depends of the processor data width. ($0 \leq p \leq 3$) P=0: move input ports(s) of address 0x20 P=1: move input ports(s) of address 0x21 P=2: move input ports(s) of address 0x22 P=3: move input ports(s) of address 0x23

Table A.1: Instructions field description

Operators	Description
()	Content of register or memory location shown inside parentheses
$\leftarrow$	Is loaded with
$\wedge$	Boolean AND
$\vee$	Boolean OR
$\oplus$	Boolean exclusive-OR
$\neg$	One's complement or Boolean NOT
$\Rightarrow$	Then
&	Concatenate
+	Add
-	Subtract (two's complement)
<x>	Bit x (or set of bits x) of memory location used in operation
$<B_x>$	Byte x of the memory location used in operation (B_0 is the least significant byte)
:	Denote a range of bits or bytes

Table A.2: Nomenclature for processor operations

Mnemonic, Operands		Description	25-bit Instruction Code	CCR Afected
LITERAL AND REGISTERS ORIENTED OPERATIONS				
ADDLW	W,k,F	Add Literal and W	00000 kkkkkkkk wwwwww ffffff	C,Z
SUBLW	W,k,F	Subtract Literal and W	00001 kkkkkkkk wwwwww ffffff	C,Z
ANDLW	W,k,F	AND Literal with W	00010 kkkkkkkk wwwwww ffffff	Z
IORLW	W,k,F	Inclusive OR Literal with W	00011 kkkkkkkk wwwwww ffffff	Z
XORLW	W,k,F	Exclusive OR Literal with W	00100 kkkkkkkk wwwwww ffffff	Z
MOVLF	k,F,d	Move Literal to F	00101 kkkkkkkk xxxxdd ffffff	-
THREE REGISTERS ORIENTED OPERATIONS				
ADDWY	W,Y,F	Add W and Y	00110 00yyyyyy wwwwww ffffff	C,Z
SUBWY	W,Y,F	Subtract W and Y	00110 01yyyyyy wwwwww ffffff	C,Z
ANDWY	W,Y,F	AND W with Y	00110 10yyyyyy wwwwww ffffff	Z
IORWY	W,Y,F	Inclusive OR W with Y	00110 11yyyyyy wwwwww ffffff	Z
XORWY	W,Y,F	Exclusive OR W with Y	00111 00yyyyyy wwwwww ffffff	Z
TWO REGISTER ORIENTED OPERATIONS				
MOVW	W,F	Move W to F	00111 01xxxxxx wwwwww ffffff	Z
BLMOV	INP,F	Blocked Move of INP to F	00111 10xxxxxx 1000pp ffffff	Z
COMW	W,F	One's complement of W	00111 11xxxxxx wwwwww ffffff	Z
NEGW	W,F	Two's complement of W	01000 00xxxxxx wwwwww ffffff	C,Z
INCW	W,F	Increment W	01000 01xxxxxx wwwwww ffffff	Z
DECW	W,F	Decrement W	01000 10xxxxxx wwwwww ffffff	Z
SWAPW	W,F	Swap halves in W	01000 11xxxxxx wwwwww ffffff	-
RLW	W,F	Rotate left through Carry	01001 00xxxxxx wwwwww ffffff	C,Z
RRW	W,F	Rotate right through Carry	01001 01xxxxxx wwwwww ffffff	C,Z
LSL	W,F	Logical shift left (Same as ASL)	01001 10xxxxxx wwwwww ffffff	C,Z
LSR	W,F	Logical shift right	01001 11xxxxxx wwwwww ffffff	C,Z
ASL	W,F	Arithmetic shift left (Same as LSL)	01001 10xxxxxx wwwwww ffffff	C,Z
ASR	W,F	Arithmetic shift right	01010 00xxxxxx wwwwww ffffff	C,Z
MISCELANEOUS OPERATIONS				
CLRF	F	Clear F	01010 01xxxxxx xxxxxx ffffff	Z
CLC		Clear carry bit	01010 10xxxxxx xxxxxx xxxxxx	C
SEC		Set carry bit	01010 11xxxxxx xxxxxx xxxxxx	C
END		End of execution	01011 00xxxxxx xxxxxx xxxxxx	TA
NOP		No operation	01011 01xxxxxx xxxxxx xxxxxx	-
BIT-ORIENTED (CONDITIONAL) OPERATIONS				
BCLR	F,b	Clear bit b in F	01011 10-bbbbb ffffff ffffff	-
BSET	F,b	Set bit b in F	01011 11-bbbbb ffffff ffffff	-
BRCLR	W,b,k	Branch if bit b in W clear	01100 kkkkkkkk wwwwww -bbbbb	-
BRSET	W,b,k	Branch if bit b in W set	01101 kkkkkkkk wwwwww -bbbbb	-
CONDITIONAL AND UNCONDITIONAL BRANCH OPERATIONS				
GOTO	k	Go to address	01110 kkkkkkkk xxxxxx xxxxxx	-
BZ	k	Branch if Z bit set	01111 kkkkkkkk xxxxxx xxxxxx	-
BNZ	k	Branch if Z bit clear	10000 kkkkkkkk xxxxxx xxxxxx	-
BC	k	Branch if Carry bit set	10001 kkkkkkkk xxxxxx xxxxxx	-
BNC	k	Branch if Carry bit clear	10010 kkkkkkkk xxxxxx xxxxxx	-
CBEQ	W,Y,k	Compare and Branch if Equal	10011 kkkkkkkk wwwwww yyyyyy	-
CBGE	W,Y,k	Compare and Branch if greater than or equal to	10100 kkkkkkkk wwwwww yyyyyy	-
CBGT	W,Y,k	Compare and Branch if greater than	10101 kkkkkkkk wwwwww yyyyyy	-
CBNE	W,Y,k	Compare and Branch if not Equal	10110 kkkkkkkk wwwwww yyyyyy	-
DBNZ	W,F,k	Decrement and Branch if Not Zero	10111 kkkkkkkk wwwwww ffffff	-
IBNZ	W,F,k	Increment and Branch if Not Zero	11000 kkkkkkkk wwwwww ffffff	-

Table A.3: Instructions set summary

ADDLW — Add Literal and register W

Syntax: ADDLW W,k,F

Operation: (F) ← (W) + k

Description: The contents of the register W are added to the eight-bit literal 'k' and the result is placed in the register F. For *modes* of 16, 24 and 32 bits, the most significant bits of 'k' must be assumed as 0's.

ADDWY — Add registers W and Y

Syntax: ADDWY W,Y,F

Operation: (F) ← (W) + (Y)

Description: The contents of the register W are added to the contents of register Y and the result is placed in the register F.

ANDLW — AND Literal with register W

Syntax: ANDLW W,k,F

Operation: (F) ← (W) ∧ k

Description: Performs the logical AND between the contents of register W and literal 'k' and places the result in register F. For *modes* of 16, 24 and 32 bits, the most significant bits of 'k' must be assumed as 0's.

ANDWY — AND register W with register Y

Syntax: ANDWY W,Y,F

Operation: (F) ← (W) ∧ (Y)

Description: Performs the logical AND between the contents of register W and register Y and places the result in register F.

ASL — Arithmetic shift left

Syntax: ASL W,F

Operation: 8-bit: C ← (W)<7>, (F) ← (W)<6:0> & 0
16-bit: C ← (W)<15>, (F) ← (W)<14:0> & 0
24-bit: C ← (W)<23>, (F) ← (W)<22:0> & 0
32-bit: C ← (W)<31>, (F) ← (W)<30:0> & 0

Description: The contents of register W are shifted one bit to the left, bit 0 is loaded with a 0, the result in placed in register F. The C bit in the CCR is loaded from the most significant bit of W. This is mathematically equivalent to multiplication by two.

ASR — Arithmetic shift right

Syntax: ASR W,F

Operation: 8-bit: (F) ← (W)<7> & (W)<7:1>, C ← (W)<0>
16-bit: (F) ← (W)<15> & (W)<15:1>, C ← (W)<0>
24-bit: (F) ← (W)<23> & (W)<23:1>, C ← (W)<0>
32-bit: (F) ← (W)<31> & (W)<31:1>, C ← (W)<0>

Description: The contents of register W are shifted one bit to the right, most significant bit held constant, the result in placed in register F. The C bit in the CCR is loaded from bit 0 of W. This operation effectively divides a signed value by 2 without changing its sign. The carry bit can be used to round the result.

BC	**Branch if Carry bit set**
Syntax:	BC k
Operation:	if (C) = 1 ⇒ PC ← k
Description:	Tests the state of the C bit in the CCR and causes a branch if C is set. BC can be used after any instructions that affect the C bit in register CCR.

BCLR	**Clear bit b in F**
Syntax:	BCLR F,b
Operation:	(F)<b> ← 0
Description:	Bit 'b' in register F is cleared. Bits b<2:0> are used for 8-bit processors. Bits b<3:0> are used for 16-bit processors. All bits of b are used for 24-bit and 32-bit processors. If the value used in 'b' is out of appropriate range of values for 8, 16 and 24 bit processors, the result might not be as expected.

BLMOV	**Blocked Move of INP to F**
Syntax:	BLMOV INP,F
Operation:	if (RE) = 1 ⇒ (F) ← (0x20 + INP), (PC)++ else ⇒ (PC) ← (PC)
Description:	Moves INP to register F when Read Enable pulse(s) is(are) active. INP=[0,1,2,3] represents any address between 0x20 and 0x23 respectively, each address can contain one or more input ports, depending on the data processing width. RE is set during a clock pulse when a processor writes any output port(s) (address 0x24 to 0x27). Examples: 8-bit: for INP=3, (F)←(0x23) when RE pulse at IN3; for INP ∈ [0,1,2,3] 16-bit: for INP=1, (F)←(0x21) when RE pulses at IN2_IN3; for INP ∈ [0,1] 24-bit: for INP=0, (F)←(0x20) when RE pulses at IN0_IN1_IN2; INP ∈ [0] 32-bit: for INP=0, (F)←(0x20) when RE pulses at IN0_IN1_IN2_IN3; INP∈[0]

BNC	**Branch if Carry bit clear**
Syntax:	BNC k
Operation:	if (C) = 0 ⇒ PC ← k
Description:	Tests the state of the C bit in the CCR and causes a branch if C is clear. BC can be used after any instructions that affects the C bit in register CCR.

BNZ	**Branch if Z bit clear**
Syntax:	BNZ k
Operation:	if (Z) = 0 ⇒ PC ← k
Description:	Tests the state of the Z bit in the CCR and causes a branch if Z is clear. BZ can be used after any instructions that affects the Z bit in register CCR.

BRCLR	**Branch if bit b in W clear**
Syntax:	BRCLR W,b,k
Operation:	if (W)<b> = 0 ⇒ PC ← k
Description:	Test bit n of register W and branches if the bit is clear. Bits b<2:0> are used for 8-bit processors. Bits b<3:0> are used for 16-bit processors. All bits of b are used for 24-bit and 32-bit processors. If the value used in 'b' is out of appropriate range of values for 8, 16 and 24 bit processors, the result might not be as expected.

BRSET	**Branch if bit b in W set**
Syntax:	BRSET W,b,k
Operation:	if (W)<b> = 1 ⇒ PC ← k
Description:	Test bit n of register W and branches if the bit is set. Bits b<2:0> are used for 8-bit processors. Bits b<3:0> are used for 16-bit processors. All bits of b are used for 24-bit and 32-bit processors. If the value used in 'b' is out of appropriate range of values for 8, 16 and 24 bit processors, the result might not be as expected.

BSET	**Set bit b in F**
Syntax:	BSET F,b
Operation:	(F)<b> ← 1
Description:	Bit 'b' in register F is set. Bits b<2:0> are used for 8-bit processors. Bits b<3:0> are used for 16-bit processors. All bits of b are used for 24-bit and 32-bit processors. If the value used in 'b' is out of appropriate range of values for 8, 16 and 24 bit processors, the result might not be as expected.

BZ	**Branch if Z bit set**
Syntax:	BZ k
Operation:	if (Z) = 1 ⇒ PC ← k
Description:	Tests the state of the Z bit in the CCR and causes a branch if Z is set. BZ can be used after any instructions that affects the Z bit in register CCR.

CBEQ	**Compare and branch if equal**
Syntax:	CBEQ W,Y,k
Operation:	if (W) = (Y) ⇒ PC ← k
Description:	Compares the contents of register W against the contents of register Y and causes a branch if the contents are equal.

CBGE	**Compare and branch if greater than or equal to**
Syntax:	CBGE W,Y,k
Operation:	if (W) ≥ (Y) ⇒ PC ← k
Description:	The instruction causes a branch if the contents of register W is greater than or equal to the contents of register Y. The instruction assumes unsigned values in the registers.

CBGT	**Compare and branch if greater than**
Syntax:	CBGT W,Y,k
Operation:	if (W) > (Y) ⇒ PC ← k
Description:	The instruction causes a branch if the contents of register W is greater than the contents of register Y. The instruction suppose unsigned values in the registers.

CBNE	**Compare and branch if not equal**
Syntax:	CBNE W,Y,k
Operation:	if (W) ≠ (Y) ⇒ PC ← k
Description:	Compares the contents of register W against the contents of register Y and causes a branch if the contents are not equal..

CLC	**Clear Carry bit**
Syntax:	CLC
Operation:	C ← 0
Description:	Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit. The C bit can also be used to pass status information between subroutines.

CLRF	**Increment W**
Syntax:	CLRF F
Operation:	(F) ← 0
Description:	The contents of register F is cleared, the Z bit in CCR is set.

COMW	**One's complement of W**
Syntax:	COMW W,F
Operation:	(F) ← ¬(W)
Description:	Performs the one's complement of the contents of register W and places the result in register F.

DBNZ	**Decrement and branch if not Zero**
Syntax:	DBNZ W,F,k
Operation:	(F) ← (W) - 1, if (Z) = 0 ⇒ PC ← k
Description:	The contents of register W is decremented, the result in placed in register F. The instruction causes a branch if the result is not 0.

DECW	**Decrement W**
Syntax:	DECW W,F
Operation:	(F) ← (W) - 1
Description:	The contents of register W is decremented, the result in placed in register F.

END	**End of execution**
Syntax:	END
Operation:	PC ← PC, TA ← 0
Description:	Ends the execution of instructions by the processor. Clears the TA bit in the CCR.

GOTO	**Go to address**
Syntax:	GOTO k
Operation:	PC ← k
Description:	GOTO is an unconditional branch. The 8-bit immediate value is loaded into PC bits.

IBNZ	**Increment and branch if not Zero**
Syntax:	IBNZ W,F,k
Operation:	(F) ← (W) + 1, if (Z) = 0 ⇒ PC ← k
Description:	The contents of register W is incremented, the result in placed in register F. The instruction causes a branch if the result is not 0.

INCW **Increment W**

Syntax: INCW W,F
Operation: (F) ← (W) + 1
Description: The contents of register W is incremented, the result in placed in register F.

IORLW **Inclusive OR Literal with register W**

Syntax: IORLW W,k,F
Operation: (F) ← (W) ∨ k
Description: Performs the logical inclusive-OR between the contents of register W and literal 'k' and places the result in register F. For *modes* of 16, 24 and 32 bits, the most significant bits of 'k' must be assumed as 0's.

IORWY **Inclusive OR register W with register Y**

Syntax: IORWY W,Y,F
Operation: (F) ← (W) ∨ (Y)
Description: Performs the logical inclusive-OR between the contents of register W and register Y and places the result in register F.

LSL **Logical shift left**

Syntax: LSL W,F
Operation: 8-bit: C ← (W)<7>, (F) ← (W)<6:0> & 0
16-bit: C ← (W)<15>, (F) ← (W)<14:0> & 0
24-bit: C ← (W)<23>, (F) ← (W)<22:0> & 0
32-bit: C ← (W)<31>, (F) ← (W)<30:0> & 0
Description: The contents of register W are shifted one bit to the left, bit 0 is loaded with a 0, the result in placed in register F. The C bit in the CCR is loaded from the most significant bit of W.

LSR **Logical shift right**

Syntax: LSR W,F
Operation: 8-bit: (F) ← 0 & (W)<7:1>, C ← (W)<0>
16-bit: (F) ← 0 & (W)<15:1>, C ← (W)<0>
24-bit: (F) ← 0 & (W)<23:1>, C ← (W)<0>
32-bit: (F) ← 0 & (W)<31:1>, C ← (W)<0>
Description: The contents of register W are shifted one bit to the right, most significant bit is loaded with a 0, the result in placed in register F. The C bit in the CCR is loaded from the bit 0 of W.

MOVLF **Move Literal to F**

Syntax: MOVLF k,F,d
Operation: (F)<d> ← k
Description: Moves the literal 'k' to any byte of register F, depending of destination 'd'.
8-bit: (F)←k for any value of 'd'.
16-bit: for d=1, (F)<B_1 >←k, (F)<B_0>←(F)<B_0>. If d≠[0,1], (F)←(F).
24-bit: for d=2, (F)<B_2>←k, (F)<B_1:B_0>←(F)<B_1:B_0>. If d≠[0,1,2], F←F.
32-bit: for d=3, (F)<B_3>←k, (F)<B_2:B_0>←(F)<B_2:B_0>

MOVW	**Move W to F**
Syntax:	MOVW W,F
Operation:	(F) ← (W)
Description:	Moves the contents of register W to register F.

NEGW	**Two's complement of W**
Syntax:	NEGW W,F
Operation:	(F) ← ¬(W) + 1
Description:	Performs the two's complement of the contents of register W and places the result in register F.

NOP	**No operation**
Syntax:	NOP
Operation:	None (PC ← PC + 1)
Description:	This is an instruction that does nothing except to consume one CPU clock cycle while the program counter is advanced to the next instruction. No register or memory contents are affected by this instruction.

RLW	**Rotate Left through Carry**
Syntax:	RLW W,F
Operation:	8-bit: C ← (W)<7>, (F) ← (W)<6:0> & C 16-bit: C ← (W)<15>, (F) ← (W)<14:0> & C 24-bit: C ← (W)<23>, (F) ← (W)<22:0> & C 32-bit: C ← (W)<31>, (F) ← (W)<30:0> & C
Description:	The contents of register W are rotated one bit to the left through the Carry, the result in placed in register F.

RRW	**Rotate Right through Carry**
Syntax:	RRW W,F
Operation:	8-bit: (F) ← C & (W)<7:1>, C ← (W)<0> 16-bit: (F) ← C & (W)<15:1>, C ← (W)<0> 24-bit: (F) ← C & (W)<23:1>, C ← (W)<0> 32-bit: (F) ← C & (W)<31:1>, C ← (W)<0>
Description:	The contents of register W are rotated one bit to the right through the Carry, the result in placed in register F.

SEC	**Set Carry bit**
Syntax:	SEC
Operation:	C ← 1
Description:	Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit. The C bit can also be used to pass status information between subroutines.

SUBLW	**Subtract Literal from register W**
Syntax:	SUBLW W,k,F
Operation:	(F) ← (W) - k
Description:	Subtracts the contents of literal 'k' from register W and places the result in register F (2's complement method). For *modes* of 16, 24 and 32 bits, the most significant bits of 'k' must be assumed as 0's.

SUBWY **Subtracts registers W and Y**

Syntax: SUBWY W,Y,F

Operation: (F) ← (W) - (Y)

Description: Subtracts the contents of register Y from register W and places the result in register F (2's complement method).

SWAPW **Swap halves of W**

Syntax: SWAPW W,F

Operation: 8-bit: (F)<7:4> ← (W)<3:0>, (F)<3:0> ← (W)<7:4>
16-bit: (F)<15:8> ← (W)<7:0>, (F)<7:0> ← (W)<15:8>
24-bit: (F)<23:12> ← (W)<11:0>, (F)<11:0> ← (W)<23:12>
32-bit: (F)<31:16> ← (W)<15:0>, (F)<15:0> ← (W)<31:16>

Description: Swaps upper and lower halves of the contents of register W, the result in placed in register F.

XORLW **Exclusive OR Literal with register W**

Syntax: XORLW W,k,F

Operation: (F) ← (W) ⊕ k

Description: Performs the logical exclusive-OR between the contents of register W and literal 'k' and places the result in register F. For *modes* of 16, 24 and 32 bits, the most significant bits of 'k' must be assumed as 0's.

XORWY **Exclusive OR register W with register Y**

Syntax: XORWY W,Y,F

Operation: (F) ← (W) ⊕ (Y)

Description: Performs the logical exclusive-OR between the contents of register W and register Y and places the result in register F.

Appendix B

Data Memory Registers of Functional Unit Processors

Don't walk in front of me; I may not follow.
Don't walk behind me; I may not lead.
Just walk beside me and be my friend.

No camines delante de mí, puede que no te siga.
No camines detrás de mí, puede que no te guíe.
Camina junto a mí y sé mi amigo.

Albert Camus (1913 – 1960)

Abstract: This section presents a detailed description of Data Memory registers of FU processors for all configuration modes and data sizes. The registers are shown in order according to their address and separated by pages for easy reference.

B.1 Abbreviations

Table B.1 shows the abbreviations used for description of bit registers throughout this section.

Abbreviations			
r = Readable bit	w = Writable bit	u = Unimplemented bit, read as 0	
-n = Value at POR/R	1 = Bit is set	0 = Bit is cleared	x = Bit is unknown
R = Readable byte	W = Writable byte	U = Unimplemented byte, read as 00h	
-N=Value at POR/R	00-FFh = 8-bit value	X = Byte is unknown	

POR/R = Power-On Reset or manual reset with push-button.

Table B.1: Abbreviations for bits of Data Memory registers

B.2 Input Ports Registers

The input ports registers are between addresses 20h and 23h. Its configuration is showed below. Note that input ports in a specific address depends of the processors data size.

8-bit Processors
Input Ports - IN0, IN1, IN2 and IN3 (ADDRESS 20h, 21h, 22h and 23h)

bit 7							bit 0
INX							
r-x	r-x	r-x	r-x	r-x	r-x	r-x	r-x

bit 7:0 Value in the input port X (read only registers).
IN0 when address is 20h
IN1 when address is 21h
IN2 when address is 22h
IN3 when address is 23h

16-bit Processors
Input Ports - IN0_IN1, IN2_IN3 (ADDRESS 20h, 21h)

bit 15	bit 8	bit 7	bit 0
INX		INY	
R-X		R-X	

bit 15:0 Value in the concatenation of two 8-bits input ports INX_INY(read only registers).
IN0_IN1 when address is 20h
IN2_IN3 when address is 21h
Read 0x0000 for address 22h and 23h

24-bit Processors
Input Ports - IN0_IN1_IN2 (ADDRESS 20h)

bit 23	bit 16	bit 15	bit 8	bit 7	bit 0
IN0		IN1		IN2	
R-X		R-X		R-X	

bit 23:0 Value in the concatenation of three 8-bits input ports IN0_IN1_IN2 (read only registers). Read 0x000000 for address 21h, 22h and 23h

32-bit Processors
Input Ports - IN0_IN1_IN2_IN3 (ADDRESS 20h)

bit 31	bit 24	bit 23	bit 16	bit 15	bit 8	bit 7	bit 0
IN0		IN1		IN2		IN3	
R-X		R-X		R-X		R-X	

bit 31:0 Value of the concatenation of four 8-bits input ports IN0_IN1_IN2_IN3 (read only registers). Read 0x00000000 for address 21h, 22h and 23h

B.3 Output Ports Registers

The output ports registers are between addresses 24h and 27h. Its configuration is shown below. Note that output ports in a specific address depend on the processors data size.

It is important to note that an output port can be written only by one CORE, which depends of the value of the PORTS register. Therefore if the PORTS register is not configured properly, a write operation over an address related with an output port might not affect the port. Any write operation in an output port generates a one-cycle pulse in the Read Enable (RE) bit of the corresponding port (RE is the ninth bit of the port.)

8-bit Processors - Output Ports
OUT0, OUT1, OUT2 and OUT3 (ADDRESS 24h, 25h, 26h and 27h)

bit 7							bit 0
OUTX							
w-0	w-0	w-0	w-0	w-0	w-0	w-0	w-0

bit 7:0 Value in the output port X (write only register).
OUT0 when address is 24h
OUT1 when address is 25h
OUT2 when address is 26h
OUT3 when address is 27h

16-bit Processors - Output Ports
OUT0_OUT1, OUT2_OUT3 (ADDRESS 24h, 25h)

bit 15	bit 8	bit 7	bit 0
OUTX		OUTY	
W-00h		W-00h	

bit 15:0 Value in the concatenation of two 8-bits output ports OUTX_OUTY.
OUT0_OUT1 when address is 24h
OUT2_OUT3 when address is 25h
No output is modified for addresses 26h and 27h

24-bit Processors - Output Ports
OUT0_OUT1_OUT2 (ADDRESS 24h)

bit 23	bit 16	bit 15	bit 8	bit 7	bit 0
OUT0		OUT1		OUT2	
W-00h		W-00h		W-00h	

bit 23:0 Value in the concatenation of three 8-bits output ports OUT0_OUT1_OUT2 (write only registers). No output is modified for addresses 25h, 26h and 27h.

32-bit Processors - Output Ports
OUT0_OUT1_OUT2_OUT3 (ADDRESS 24h)

bit 31 bit 24	bit 23 bit 16	bit 15 bit 8	bit 7 bit 0
OUT0	OUT1	OUT2	OUT3
W-00h	W-00h	W-00h	W-00h

bit 31:0 Value in the concatenation of four 8-bits output ports OUT0_OUT1_OUT2_OUT3 (write only registers). No output is modified for addresses 25h, 26h and 27h. PORTS register must be set to E4h.

B.4 Code Condition Register

The Code Condition Register (CCR) contains the arithmetic status of the ALU and the status of the thread executed. The CCR register can be read by any instruction, as with any other register.

The Z and C elements correspond to the flags that indicate when an operation is zero and when the operation has a carry respectively. These bits indicate the results of the instruction just executed. The Carry bit can be modified using the instructions CLC and SEC.

The bit TA (Thread Active) indicates if the execution thread has finished or not. The instruction END is the only one able to stop the execution of the thread (TA $\leftarrow$ 0).

For 16, 24 and 32 bit processors, the most significant bits of CCR (from the bit 8) must be assumed as unimplemented, a read operation will return 0's.

Condition Code Register - CCR (ADDRESS 28h)

bit 7							bit 0
-	-	-	-	-	TA	Z	C
u-0	u-0	u-0	u-0	u-0	r-0	r-0	r-0

bit 7:3 Unimplemented.

bit 2 **TA - Thread Active**
1 = Thread is active.
0 = Thread not active. The END instruction is the only that can stop a thread (TA $\leftarrow$ 0).

bit 1 **Z - Zero bit**
1 = The result of a logic or arithmetic operation is zero.
0 = The result of a logic or arithmetic operation is not zero.

bit 0 **C - Carry bit**
1 = The result of a shift, logic or arithmetic operation generates a carry bit.
0 = No carry.

B.5 Mode Register

The Mode register configures the operation mode of the FU. In other words, it configures the number of processors in the cell and specifies the configuration of the Data and Program Memories.

The FU may have 1 to 4 processors. The FU can be configured for data processing of 8, 16, 24 and 32 bits, and the thread capacity can be for 64, 128, 192 or 256 instructions.

This register is loaded for the CμP in prototype when a new cell is inserted. For 16, 24 and 32 bit processors, the most significant bits of MODE (from the bit 8) must be assumed as unimplemented, a read operation will return 0's.

Mode Register - MODE (ADDRESS 29h)

bit 7							bit 0
-	-	-	-	mode			
u-0	u-0	u-0	u-0	r-0	r-0	r-0	r-0

bit 7:4 Unimplemented.

bit 3:0 **mode:** Specifies the Configuration Mode (CM) of cores in Functional Unit.

0000 = CM 0: four processors [P0: 8x8, 64] [P1: 8x8, 64] [P2: 8x8, 64] [P3: 8x8, 64]

0001 = CM 1: three processors [P0: 16x8, 128] [P2: 8x8, 64] [P3: 8x8, 64]

0010 = CM 2: two processors [P0: 16x8, 128] [P2: 16x8, 128]

0011 = CM 3: two processors [P0: 24x8, 192] [P3: 8x8, 64]

0100 = CM 4: one processors [P0: 32x8, 256]

0101 = CM 5: three processors [P0: 8x16, 128] [P2: 8x8, 64] [P3: 8x8, 64]

0110 = CM 6: two processors [P0: 8x16, 128] [P2: 16x8, 128]

0111 = CM 7: two processors [P0: 8x16, 128] [P2: 8x16, 128]

1000 = CM 8: two processors [P0: 8x16, 192] [P2: 16x8, 64]

1001 = CM 9: one processors [P0: 16x16, 256]

1010 = CM 10: two processors [P0: 8x24, 192] [P3: 8x8, 64]

1011 = CM 11: one processors [P0: 8x32, 256]

others = Configuration mode not implemented

Note: [P0: 16x8, 64] = [Processor 0: General Purpose Registers with 16 words of 8 bits each mapped in Data Memory, Program Memory with capacity for 64 instructions]

B.6 Family Register

The Family register is reserved for future implementations. This register has been reserved for supporting the implementation of microthreads.

This register is loaded for the CμP in prototype when a new cell is inserted.

Family Register- FAMILY (ADDRESS 2Ah)

bit 7							bit 0
FAMILY							
r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0

bit 7:0 Family ID of thread.

B.7 Output Ports Configuration Register (PORTS)

This register configures the CORE that may perform a write operation over any output port of the FU i.e. select the ALU of a CORE that writes the result of any operation to the OUTX register.

For 16, 24 and 32 bit processors, the most significant bits of PORTS (from the bit 8) must be assumed as unimplemented, a read operation will return 0's.

The value of some PORTX field could be disabled when the FTS is enabled in the redundant cell (see FTCSR in section B.9).

This register is loaded for the CμP in prototype when a new cell is inserted.

Output Ports Configuration Register - PORTS (ADDRESS 2Bh)

bit 7 bit 0

PORT3		PORT2		PORT1		PORT0	
r-1	r-1	r-1	r-0	r-0	r-1	r-0	r-0

bit 7:6 **PORT3:** Sets the core that can performs write operations in the OUT3.
00 = OUT3 can be written from CORE0
01 = OUT3 can be written from CORE1
10 = OUT3 can be written from CORE2
11 = OUT3 can be written from CORE3

bit 5:4 **PORT2:** Sets the core that can performs write operations in the OUT2.
00 = OUT2 can be written from CORE0
01 = OUT2 can be written from CORE1
10 = OUT2 can be written from CORE2
11 = OUT2 can be written from CORE3

bit 3:2 **PORT1:** Sets the core that can performs write operations in the OUT1.
00 = OUT1 can be written from CORE0
01 = OUT1 can be written from CORE1
10 = OUT1 can be written from CORE2
11 = OUT1 can be written from CORE3

bit 1:0 **PORT0:** Sets the core that can performs write operations in the OUT0.
00 = OUT0 can be written from CORE0
01 = OUT0 can be written from CORE1
10 = OUT0 can be written from CORE2
11 = OUT0 can be written from CORE3

The next are sample values for PORTS register, each value depends of configuration mode, e.g., if MODE=00h and PORTS=E4h, the processor 0 (P0) can write any value to OUT0, P1 to OUT1, P2 to OUT2 and P3 to OUT3. In this case, if P0 performs a write operation over OUT1, OUT2 or OUT3, these registers will not be affected.

MODE	**PORTS**	**P0**	**P1**	**P2**	**P3**
00h	E4h	OUT0	OUT1	OUT2	OUT3
01h	E0h	OUT0, OUT1	-	OUT2	OUT3
02h	A0h	OUT0, OUT1	-	OUT2, OUT3	-
03h	C0h	OUT0, OUT1, OUT2	-	-	OUT3
04h	00h	OUT0, OUT1, OUT2, OUT4	-	-	-
05h	E4h	OUT0_OUT1	-	OUT2	OUT3
06h	A4h	OUT0_OUT1	-	OUT2, OUT3	-
07h	E4h	OUT0_OUT1	-	OUT2_OUT3	-
08h	F4h	OUT0_OUT1	-	-	OUT2, OUT3
09h	44h	OUT0_OUT1, OUT2_OUT3	-	-	-
0Ah	E4h	OUT0_OUT1_OUT2	-	-	OUT3
0Bh	E4h	OUT0_OUT1_OUT2_OUT3	-	-	

- Denotes that the processor is unimplemented.

, Denotes output registers in different addresses.

_ Denotes concatenation of two or more output registers.

B.8 Subprocess Configuration and Status Register (SUBPCSR)

This register controls the execution of subprocesses in the system.

For 16, 24 and 32 bit processors, the most significant bits of SUBPCSR (from the bit 8) must be assumed as unimplemented, a read operation will return 0's.

Subprocess Configuration and Status Register
SUBPCSR (ADDRESS 2Ch)

bit 7							bit 0
SWS	ESP3	ESP2	ESP1	ESP0	SUBPID		EXSP
r-0	w/r-0	w/r-0	w/r-0	w/r-0	w/r-0	w/r-0	w/r-0

bit 7 **SWS - SYSTEM WAIT STATE:** The SYSTEM is in wait state. The GCU of the master chip could receive subprocess instructions. This bit is controlled by CCU.
1 = SYSTEM in wait state. This bit is set when the CCU receives a frame that contains a command related to the state *wait*.
0 = SYSTEM is not in wait state. When a processor sets EXSP bit to high level or when a cell receives any data from INET, this bit is set to zero automatically.

bit 6 **ESP3 - ENDS SUBPROCESS 3:** The execution of the subprocess 3 has ended. The CCU sets this bit, but must be cleared by software.
1 = Creation of subprocess 3 end.
0 = No creation.

bit 5 **ESP2 - ENDS SUBPROCESS 2:** The execution of the subprocess 2 has ended. The CCU sets this bit, but must be cleared by software.
1 = Creation of subprocess 2 end. CCU sets this bit. It must be cleared by software.
0 = No creation.

bit 4 **ESP1 - ENDS SUBPROCESS 1:** The execution of the subprocess 1 has ended. The CCU sets this bit, but must be cleared by software.
1 = Creation of subprocess 1 end.
0 = No creation.

bit 3 **ESP0 - ENDS SUBPROCESS 0:** The execution of the subprocess 0 has ended. The CCU sets this bit, but must be cleared by software.
1 = Creation of subprocess 0 end.
0 = No creation.

bit 2:1 **SUBPID - SUBPROCESS ID:** ID of the subprocess to execute.
00 = Subprocess 0.
01 = Subprocess 1.
10 = Subprocess 2.
11 = Subprocess 3.

bit 0 **EXSP - EXECUTE SUBPROCESS:** Executes a subprocess for a specific component.
1 = The FU sends a command indicating to execute a subprocess of the component to which the cell belongs.
0 =No execute.

Note: when using the instruction MOVLF for writing this register, be sure to configure appropriately the destination (d=0), otherwise you may modify the register value.

B.9 Fault Tolerance Configuration and Status Register (FTCSR)

This register configures the Fault Tolerance System (FTS).

For 16, 24 and 32 bit processors, the most significant bits of FTCSR (from the bit 8) must be assumed as unimplemented, a read operation will return 0's.

This register is loaded for the CμP in prototype when a new cell is inserted.

Fault Tolerance Configuration and Status Register
FTCSR (ADDRESS 2Dh)

bit 7							bit 0
FTEF	FTE	-	FTRC	ft_mode			
r-0	r-0	u-0	r-0	r-0	r-0	r-0	r-0

bit 7 **FTEF - FAULT TOLERANCE ERROR FLAG:** The FT error flag indicates when the FTS found an error while performing the comparison between two processors.
1 = A FT error was detected, self-elimination and self-replication processes start.
0 = no FT error.

bit 6 **FTE - FAULT TOLERANCE ENABLE:** Enables or disables the FTS.
1 = FTS enabled.
0 = FTS disabled.

bit 5 Unimplemented.

bit 4 **FTRC - FAULT TOLERANCE REDUNDANT CELL:** Indicates if the cell is the redundant cell, this bit should be set in ft_modes = 5, 6, 7 and 8. The data bus of a core is connected to output ports directly i.e. some bits of PORTS register are disabled, RE is set to '1'.
1 = Redundant cell. It automatically disables the comparators of FTS for this cell.
FT_mode 5: The CORE0 writes permanently the OUT0 (C0⇒OUT0).
PORT0 disabled (P0-D).
FT_mode 6: C0⇒OUT0, C1⇒OUT1. [P0,P1]-D.
FT_mode 7: C0⇒OUT0, C1⇒OUT1, C2⇒OUT2. [P0,P1,P2]-D.
FT_mode 8: C0⇒OUT0, C1⇒OUT1, C2⇒OUT2, C3⇒OUT3. [P0,P1,P2,P3]-D.
0 = Primary cell.

bit 3:0 **FT_MODE:** Specifies the fault tolerance configuration mode.
0000 = FT_mode 0: [C0 ⇔ C1]
0001 = FT_mode 1: [C0 ⇔ C1] & [C2 ⇔ C3]
0010 = FT_mode 2: [C2 ⇔ C3]
0011 = FT_mode 3: [C0 ⇔ C2]
0100 = FT_mode 4: [C0 ⇔ C2] & [C1 ⇔ C3]
0101 = FT_mode 5: [C0 ⇔ C0*]
0110 = FT_mode 6: [C0 ⇔ C0*] & [C1 ⇔ C1*]
0111 = FT_mode 7: [C0 ⇔ C0*] & [C1 ⇔ C1*] & [C2 ⇔ C2*]
1000 = FT_mode 8: [C0 ⇔ C0*] & [C1 ⇔ C1*] & [C2 ⇔ C2*] & [C3 ⇔ C3*]
others = FT configuration mode not implemented

Note: Symbol ⇔ indicates comparison between CORES (C). *Denote a CORE in the redundant cell. The connections must be implemented by user.

Appendix C

Flow Diagrams for Self-adaptive Processes in System

Sports do not build character. They reveal it.
El deporte no construye el carácter. Lo revela.
Haywood Hale Broun (1918 – 2001)

Abstract: This chapter presents the flow diagrams of self-adaptive algorithms included in the system. These algorithms are implemented in the Configuration Units of Cell and Switch Matrix. The flow diagrams presented include the algorithms for self-placement, self-routing and self-derouting processes.

Along this chapter, the labels "_inet" and "_enet" will be added to differentiate frames that includes commands related with the Internal Network (INET) and External Network (ENET) respectively.

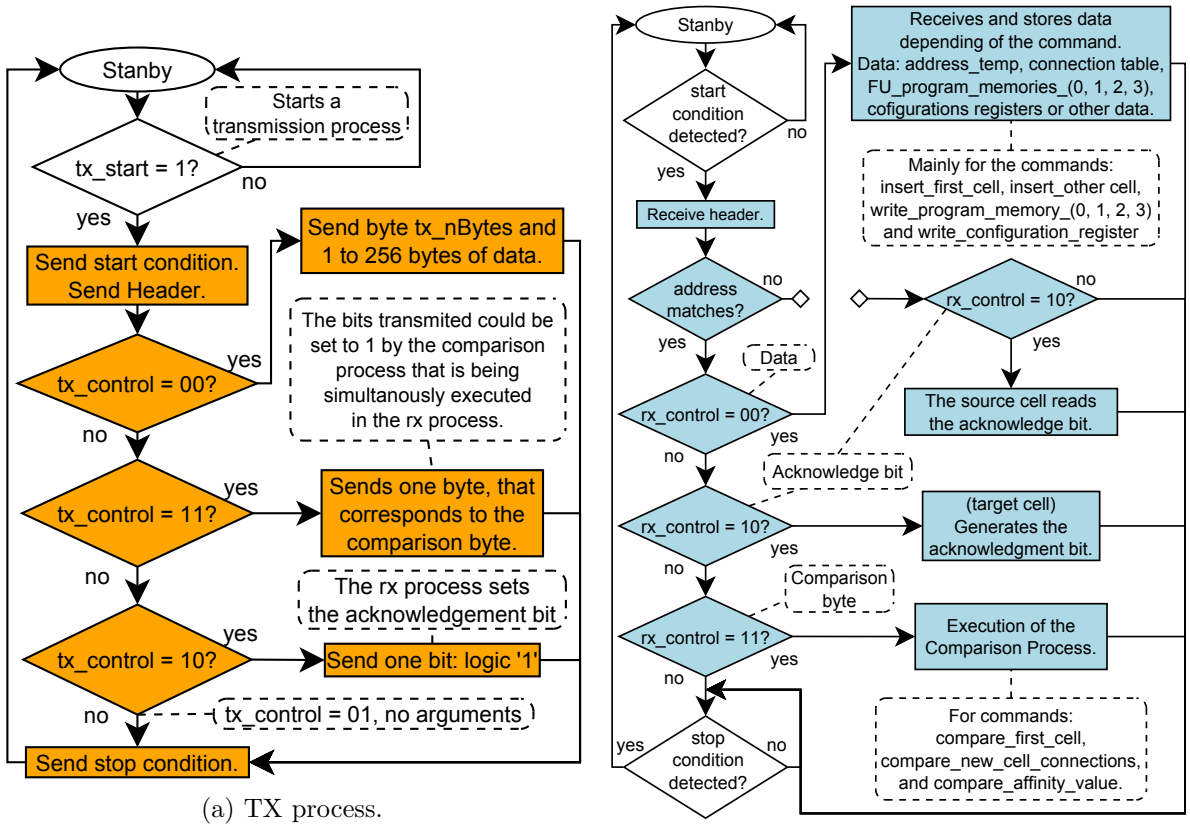
C.1 Transmission and Reception in Cell

The transmission and reception processes used for the Cell Configuration Unit (CCU) are shown in Figure C.1. The CCU implements these algorithms for the communication with the Global Configuration Unit (GCU) thought INET. The communication protocol and interface communication are explained in detail in sections 2.11 and 2.12. These processes participate actively in the self-routing and self-placement algorithms.

The data in frames is divided in four groups. The first group (tx_control = 00) sends between 1 and 256 bytes of information, using the nBytes field for specifying the number of bytes. The second group (tx_control = 01) only sends the command, it does not send arguments. The third group are the commands that use the acknowledge bit (tx_control = 10) and the last group (tx_control = 11) use the transmission and reception to perform a comparison process of data (8-bits).

When the CCU requires the acknowledge bit, the transmission process always sends a logic '1' in the time space of the acknowledge bit, and the reception process performs simultaneously the read and write of the acknowledge bit. It is important to note that all the cells in the array participate in this process, but only one can write the acknowledge bit with a logic '0', the cell which *address* match with the *address* field of the frame.

It is possible that several CCUs require the comparison of a special number to take any action, in this case the CCUs send the control bits (11), the associated command (scan_...) and



(a) TX process.

(b) RX process

Figure C.1: Transmission and reception processes in Cell Configuration Unit.

the comparison byte. The reception process performs the simultaneous comparison of the data. The winner is the cell that has the lowest comparison value, in case of a tie, the winner will be the leftmost uppermost cell in the array.

C.2 Self-Placement Processes in CCU

C.2.1 Flow Diagram for Insertion of First Cell of a Component

Figure C.2 shows a detailed description of the algorithm implemented by cells for the insertion process of the first cell of a component. The process is started by GCU who sends the command *insert_first_cell_inet* with the *address* and connection tables of the new cell as arguments. The process ends when the GCU receives the command *end_first_cell_inet*.

C.2.2 Flow Diagram for Insertion of Other Cells of a Component

The process is started by GCU who sends the command *insert_other_cell_inet* with the *address* and connection tables of the new cell as arguments. The flow diagram of the algorithm that performs the self-placement of the other cells of the component (from the second) is shown in Figure C.3.

It is important to note, that after the placement process, the self-routing process at cell level starts. The self-placement and self-routing processes ends when the GCU receives the command *end_other_cell_inet* (Figure C.4).

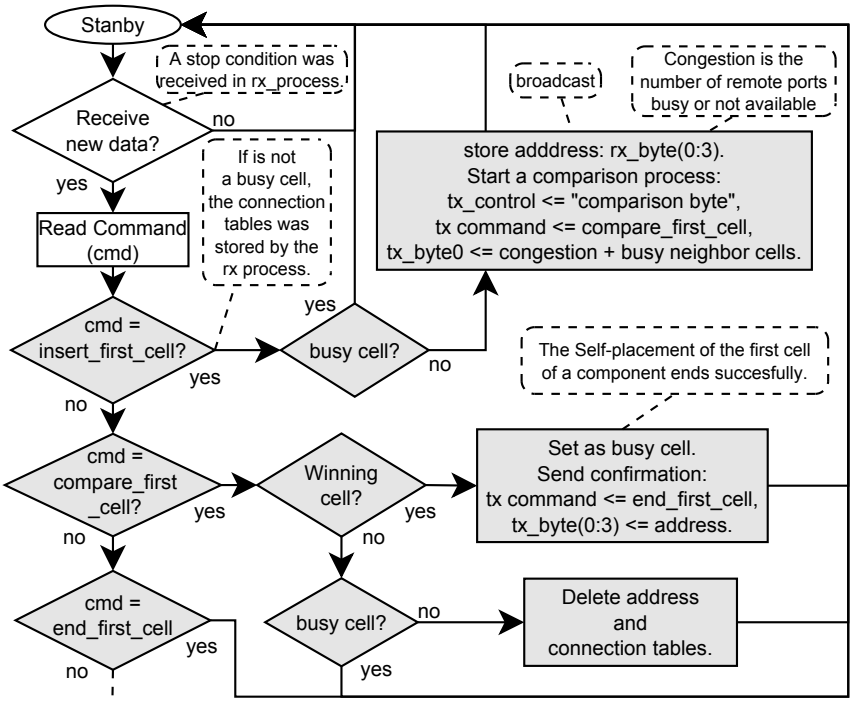


Figure C.2: Self-placement algorithm for the insertion of the first cell of a component.

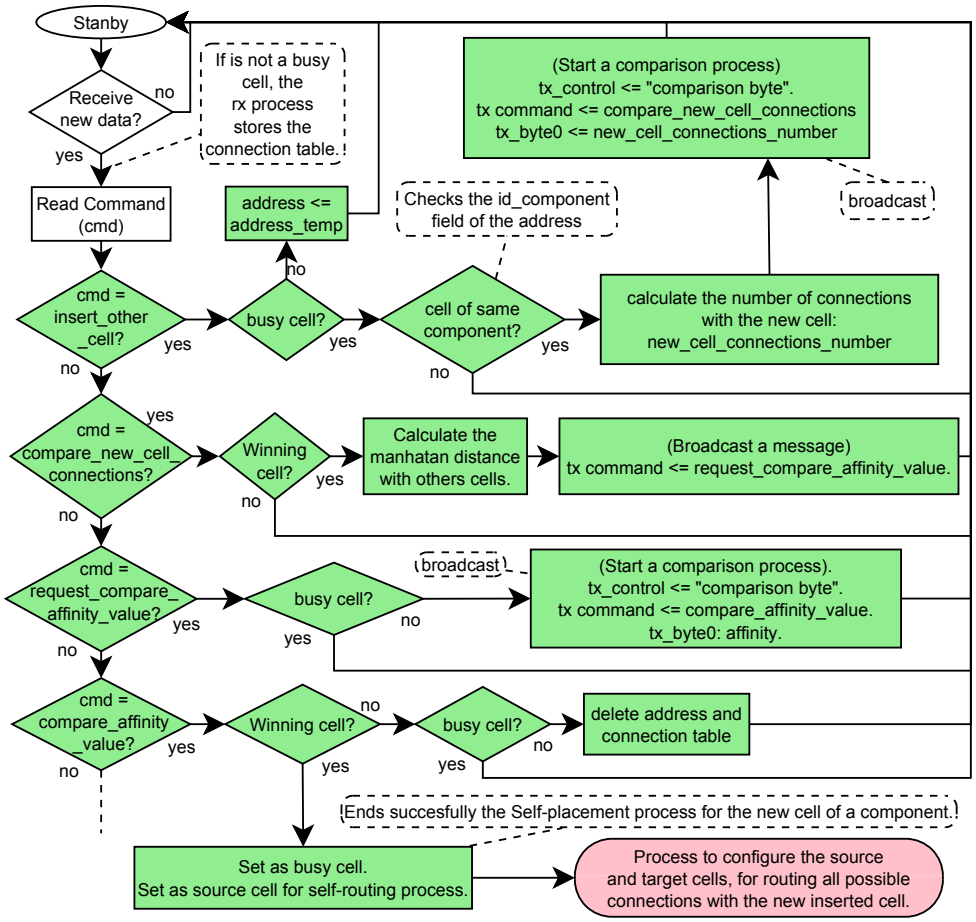


Figure C.3: Self-placement algorithm for other cells of a component (from the second).

C.3 Self-Routing Processes in CCU

C.3.1 Flow Diagram to select the Source and Target cells before the Expansion Process at Cell Level

The algorithm presented in Figure C.4 is executed after the insertion of the remaining cells of a component (from the second). Note that the process starts when the self-placement algorithm presented in Figure C.3 ends.

When the *source_cell* and *target_cell* are ready to configure a connection, the *source_cell* starts the "Expansion Process at Cell Level". For this purpose the flag *Starts Cell Propagation* is used.

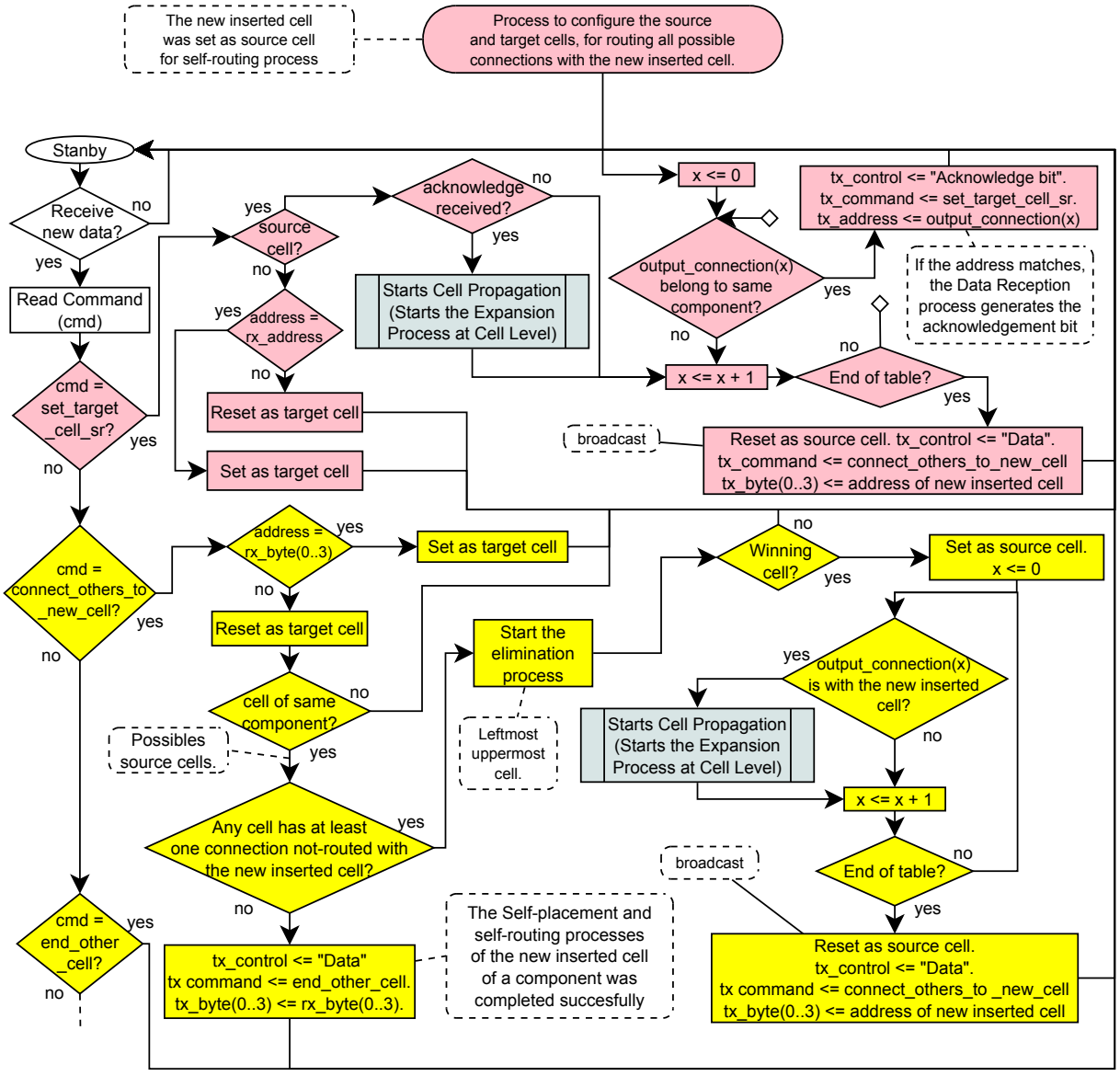


Figure C.4: Configuration of source and target cell for execution of Expansion Process at Cell Level.

C.3.2 Main Flow Diagram in CCU

Figure C.5 shows the flow diagram for the Expansion and Release processes in Cell Configuration Unit (CCU). This is the start point for the execution of any process in system that requires the propagation of signals using the expansion ports of cells, SMs and PIMs. It includes the Expansion at Cell or Component level and the Release processes.

The flow diagram remains in *standby* state until the CCU of a cell in the array starts one of the following process:

1. Expansion Process at Cell Level (Search Phase): this process is started by the *source_cell* when the signal *start_cell_propagation* is set.

2. Expansion Process at Component Level (Search Phase): this process is started by the *source_cell* when the signal *start_component_propagation* is set.

3. Expansion Process at Component Level (Search Phase): this process is started by the *source_cell* when the signal *start_component_propagation_pin* is set.

4. Release Process: this process is started by the *target_cell* when the signal *start_delete_cell_connection* is set.

The cells on the array that receive any propagation signals can execute any of the process denoted with a box in the figure, as follows:

1. Neighbor [NORTH, EAST, SOUTH or WEST]: The process is executed when a *neighbor_in_X* signal is received. These boxes represent the transition between the Search and Configuration Phases of the Expansion Process at Cell Level when the *target_cell* is reached in a neighbor cell.

2. Propagate [NORTH, EAST, SOUTH or WEST]: The process is executed when a *propagate_in_X* signal is received. These boxes represent the transition between the Search and Configuration Phases of the Expansion Process at Cell Level when the *target_cell* is reached.

3. Propagate Matrix: The process is executed when a *propagate_in_matrix* signal is received. This box represents the transition between the Search and Configuration Phases of the Expansion Process at Component Level when the *target_cell* is reached.

4. Delete [NORTH, EAST, SOUTH or WEST]: The process is executed when a *del_connection_in_X* signal is received. These boxes represent a step in the Release Process at Cell Level.

5. Delete Matrix: The process is executed when a *del_connection_in_matrix* signal is received. This box represents the end of the Release Process at Component Level.

6. Expansion Process at Cell Level - Configuration Phase [NORTH, EAST, SOUTH or WEST]: The process is executed when the cell is in *lock_cell* state and a *lock_in_X* signal is received. These boxes represent the Configuration Phase of the Expansion Process at Cell Level.

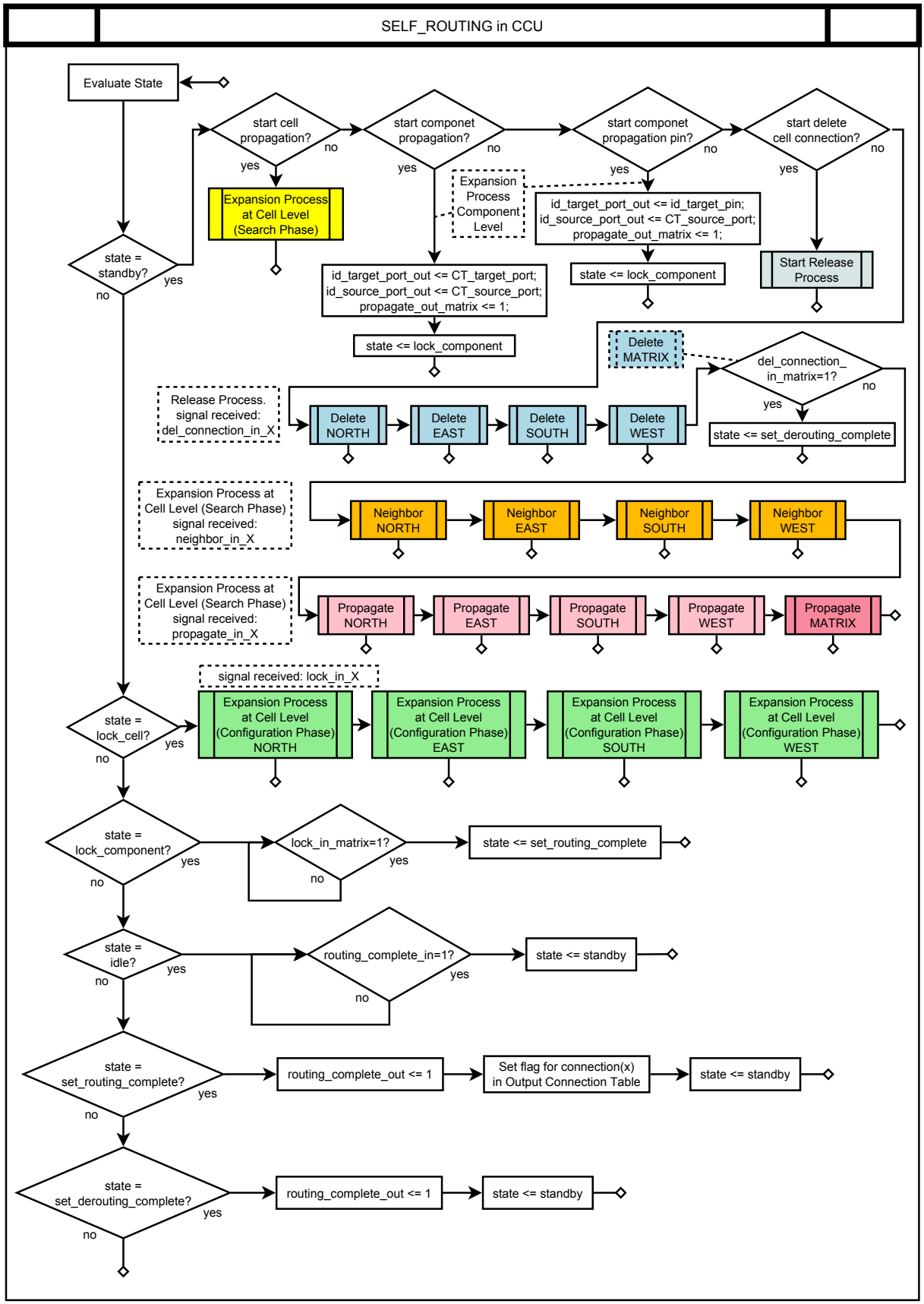


Figure C.5: Main flow diagram for Expansion and Release processes in Cell Configuration Unit.

C.3.3 Expansion Process at Cell Level - Search Phase

The search phase is started by the *source_cell*. This is an expansion process that propagates signals in the sides of the cell that have available routing resources, like local and/or remote free ports. The propagation process includes the configuration of signals for each side (north, east, south and west) of the cell as shown in Figure C.6.

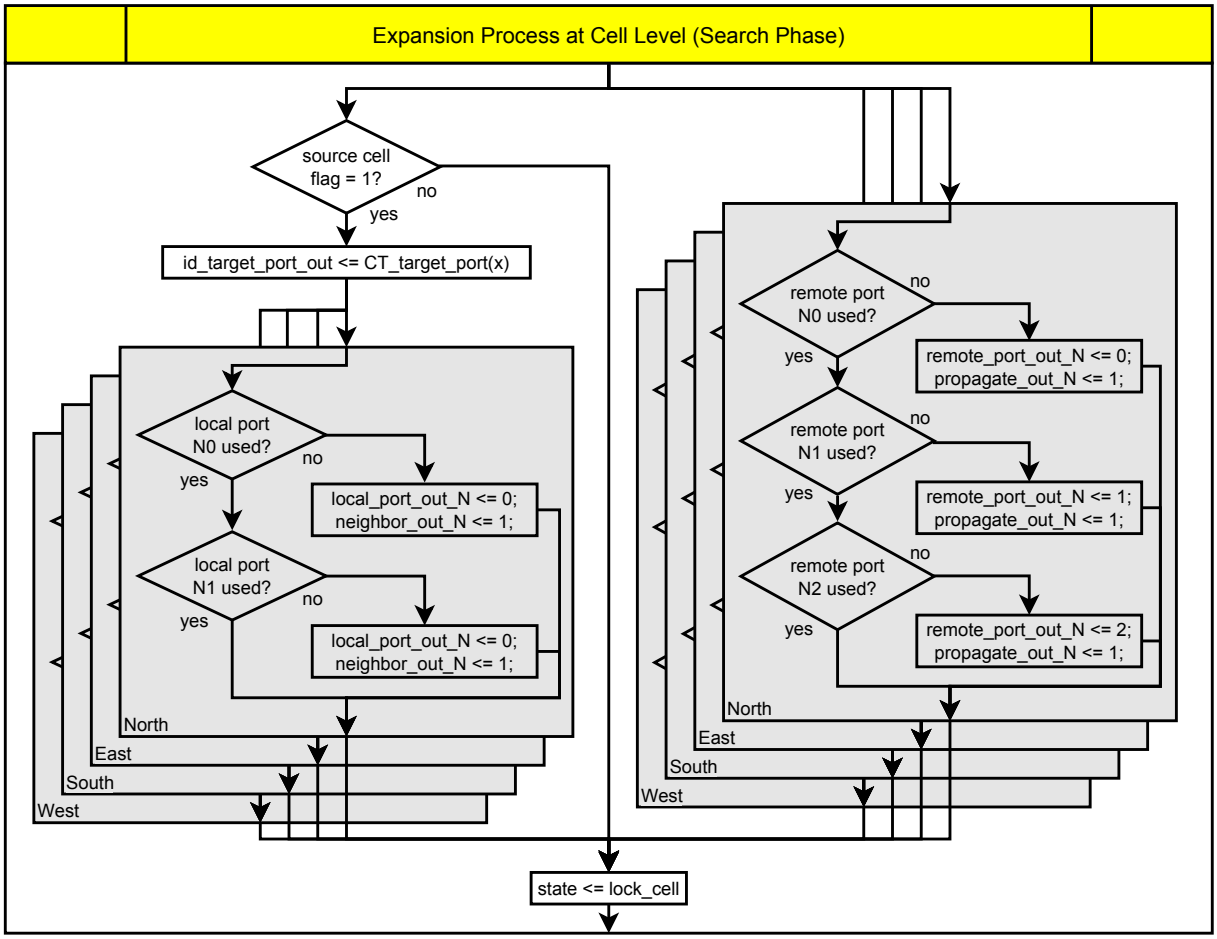
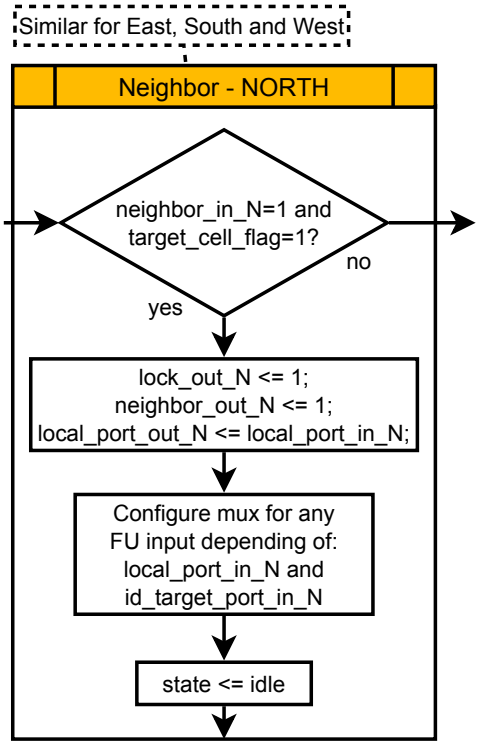


Figure C.6: Flow diagram for the propagation of Signals in the Search Phase of the Expansion Process at Cell Level.

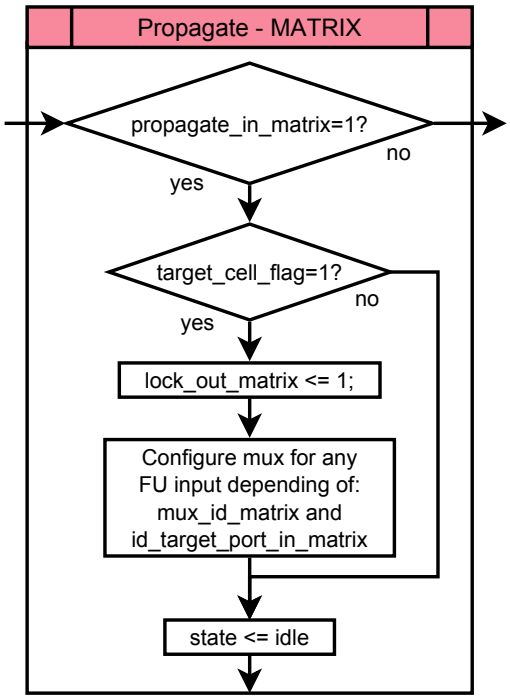
Note that propagation of *neighbor_out_X* signal is only available for *source_cell* when local ports are available. The *source_cell* and the other cells that participate in the Search Phase use the *propagate_out_X* signal when remote ports are available.

When the *target_cell* is a neighbor cell, the Search Phase ends and the Configuration Phase starts as shown in Figures C.7a and C.7c. Note that these figures show the case for a propagation signal received from NORTH. Similar functionality must be assumed for EAST, SOUTH and WEST.

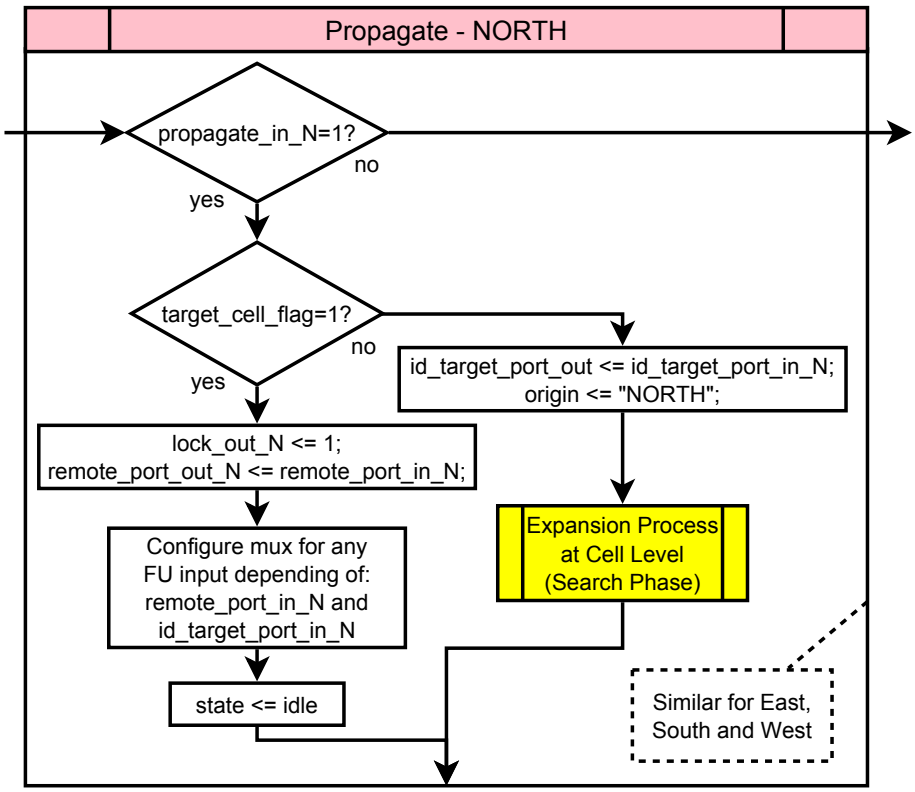
When the *target_cell* is not a neighbor cell, the propagation (C.6) is executed by each cell that reads the *propagate_in_X* signal as shown in Figure C.7c. The priority order for propagation signals is NORTH, EAST, SOUTH and WEST. The *origin* register stores the side from which the propagation signal was received. The propagation process explores the entire cell array until finding the *target_cell*, if possible. The Configuration Phase starts when the Search Phase finds the *target_cell*.



(a) Propagation signal at cell level for a neighbor cell.



(b) Propagation signal at component level.



(c) Propagation signal at cell level.

Figure C.7: Flow diagrams for Expansion Process when propagation input signals is received in Cell Configuration Unit.

C.3.4 Expansion Process at Cell Level - Configuration Phase

The Configuration Phase at Cell Level starts when the Search Phase finds the *target_cell* (Figures C.7a and C.7c).

The Configuration Phase that was started by the *target_cell* goes backward over the path previously configured in the Search Phase until it arrives to the *source_cell*. This process configures the multiplexers of the corresponding cells to fix the path. The cells that participate in this process are in the state *lock_cell* and perform the actions described in Figure C.8 when the signal *lock_in_X* in the corresponding side is activated. Similar functionality must be assumed for EAST, SOUTH and WEST.

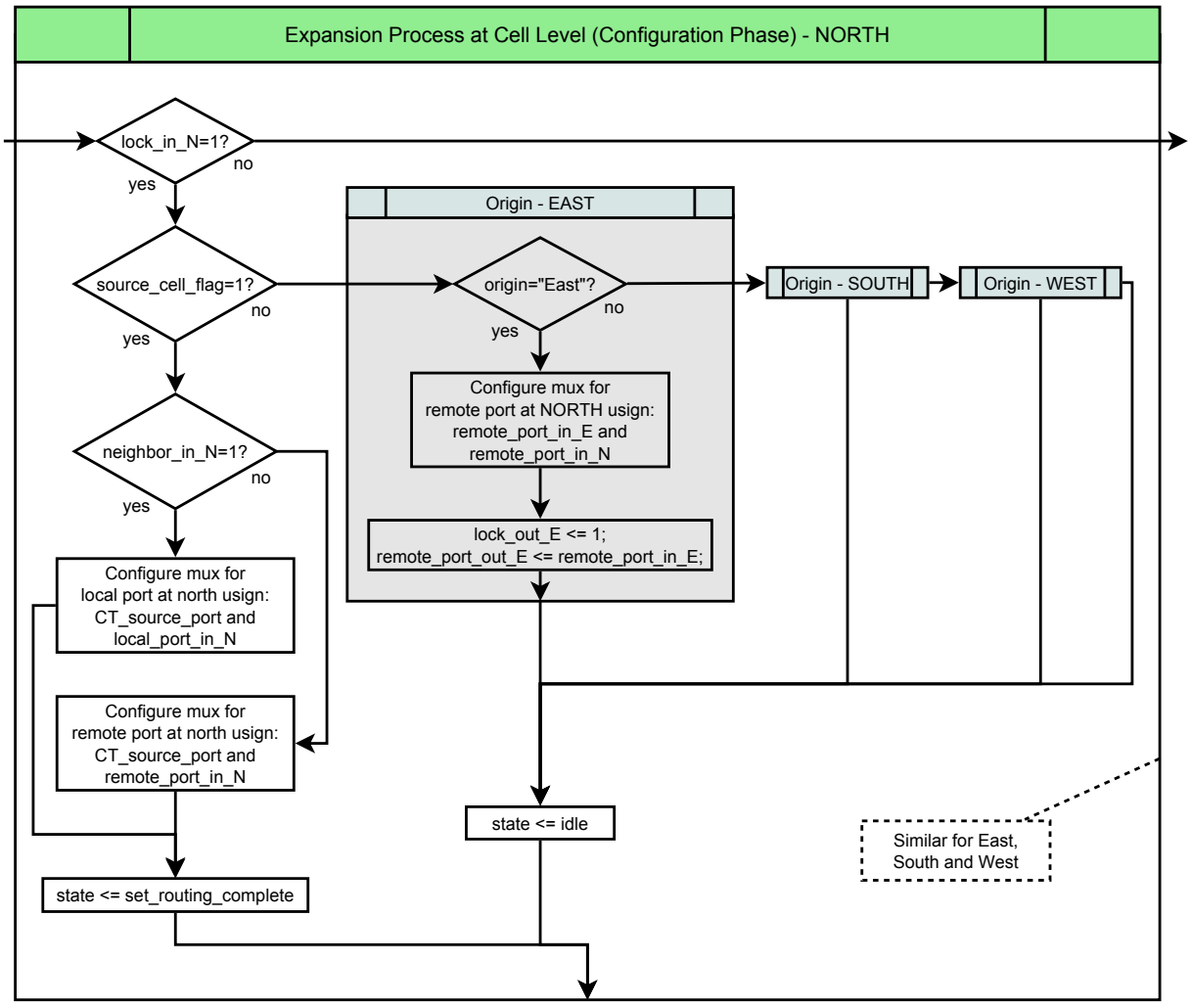


Figure C.8: Flow diagram for Configuration Phase of the Expansion Process at Cell Level.

C.3.5 Release Process at Cell Level

This process releases the routing resources used for a interconnection between cells. The Release Process goes from the *target* to the *source* of a connection.

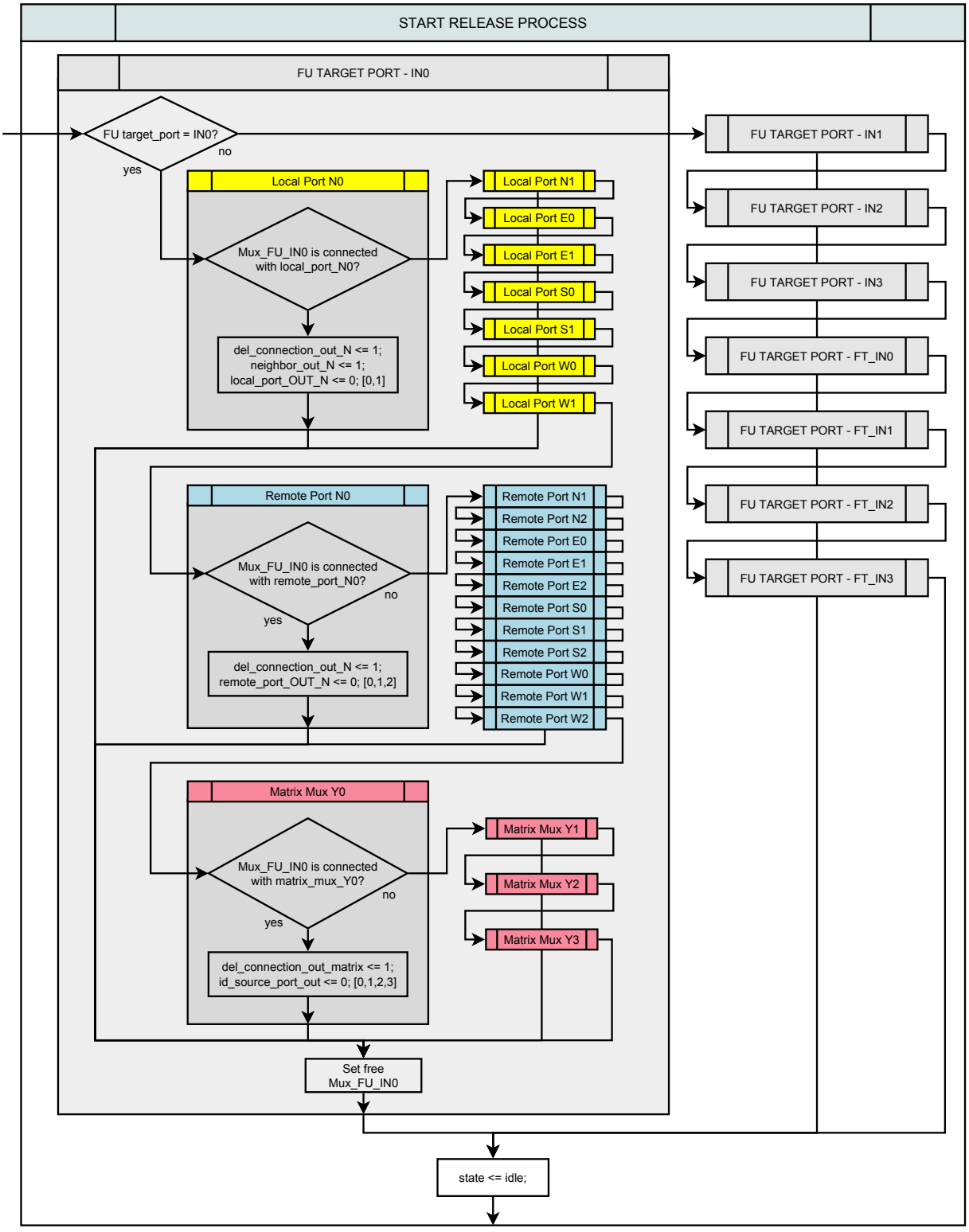


Figure C.9: Flow digram of the start point of Release Process at Cell and Component Level.

If the connection is at cell level, the multiplexers of FU inputs, local and remote cell ports are released. In this case, the Release Process starts from the FU input port of the *target_cell* an go backwards to the FU output port of the *source_cell*. This process reads the multiplexers configuration and propagate the signal *del_connection_out_X* in the direction where the connection was previously established by the self-routing process.

The *target_cell* starts the process presented in Figure C.5 when the flag *start_delete_cell_connection* is set. Then, the *target_cell* executes the algorithm presented in Figure C.9. Thereafter, each cell that receives the signal *del_connection_in_X* executes the algorithm presented in Figure C.10 until it finds the *source_cell*, which goes to the state *set_derouting_complete* and the process ends.

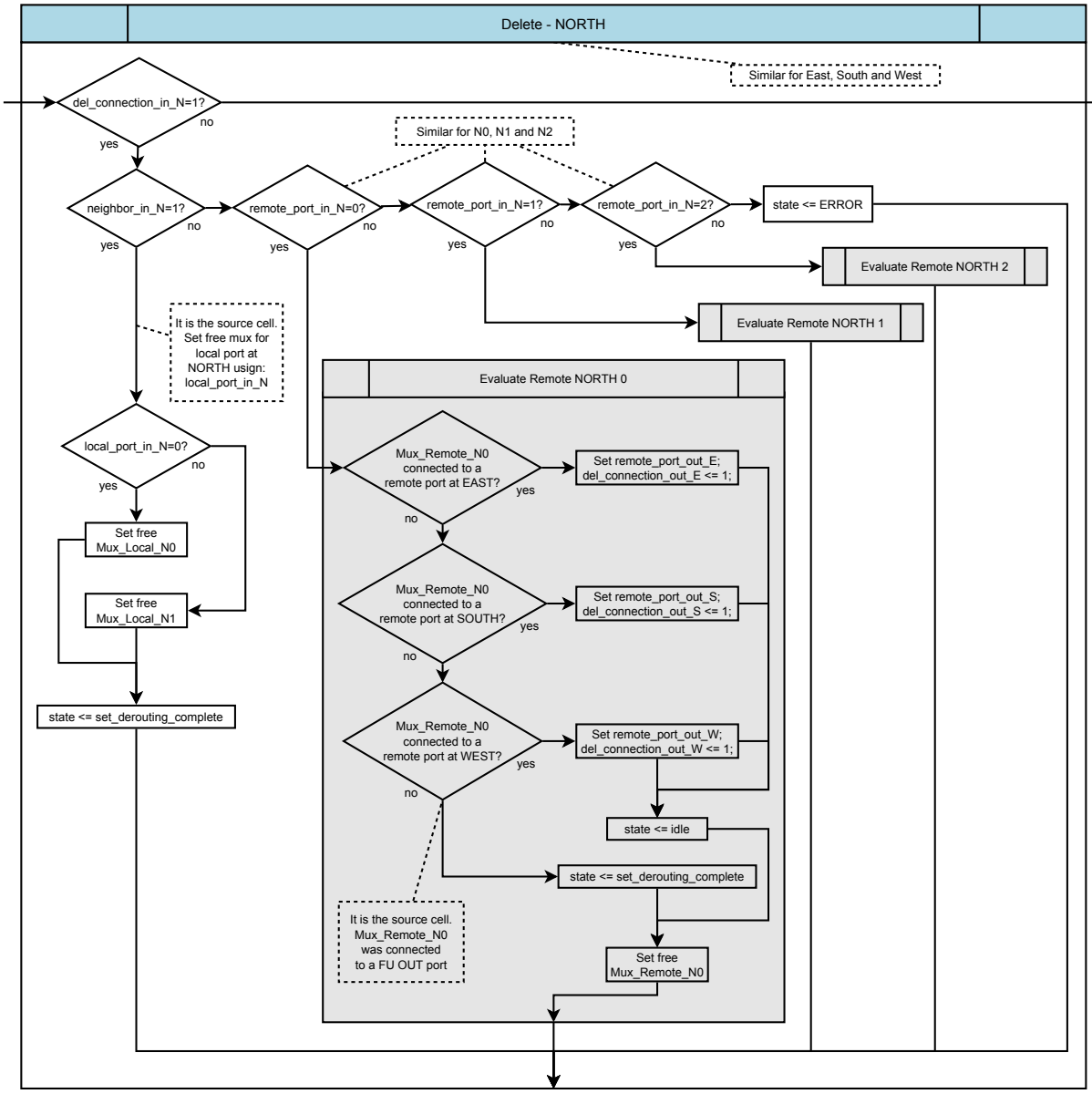


Figure C.10: Flow digram of Release Process at Cell Level.

C.4 Self-Routing Processes in SMCU

When the Expansion or Release processes are at component level, the propagation of signals is made between cells and SMs. Note that the process must be started and finalized by cells (Figure C.5) that propagate signals to the SM that belongs to the cluster, which in turn propagates signals to the SMs of neighboring clusters until it finds the destination cell of a process.

Figure C.11 shows the flow diagram for the Expansion and Release processes in Switch Matrix Configuration Unit (SMCU). It includes the Expansion at Cell or Component level and the Release processes.

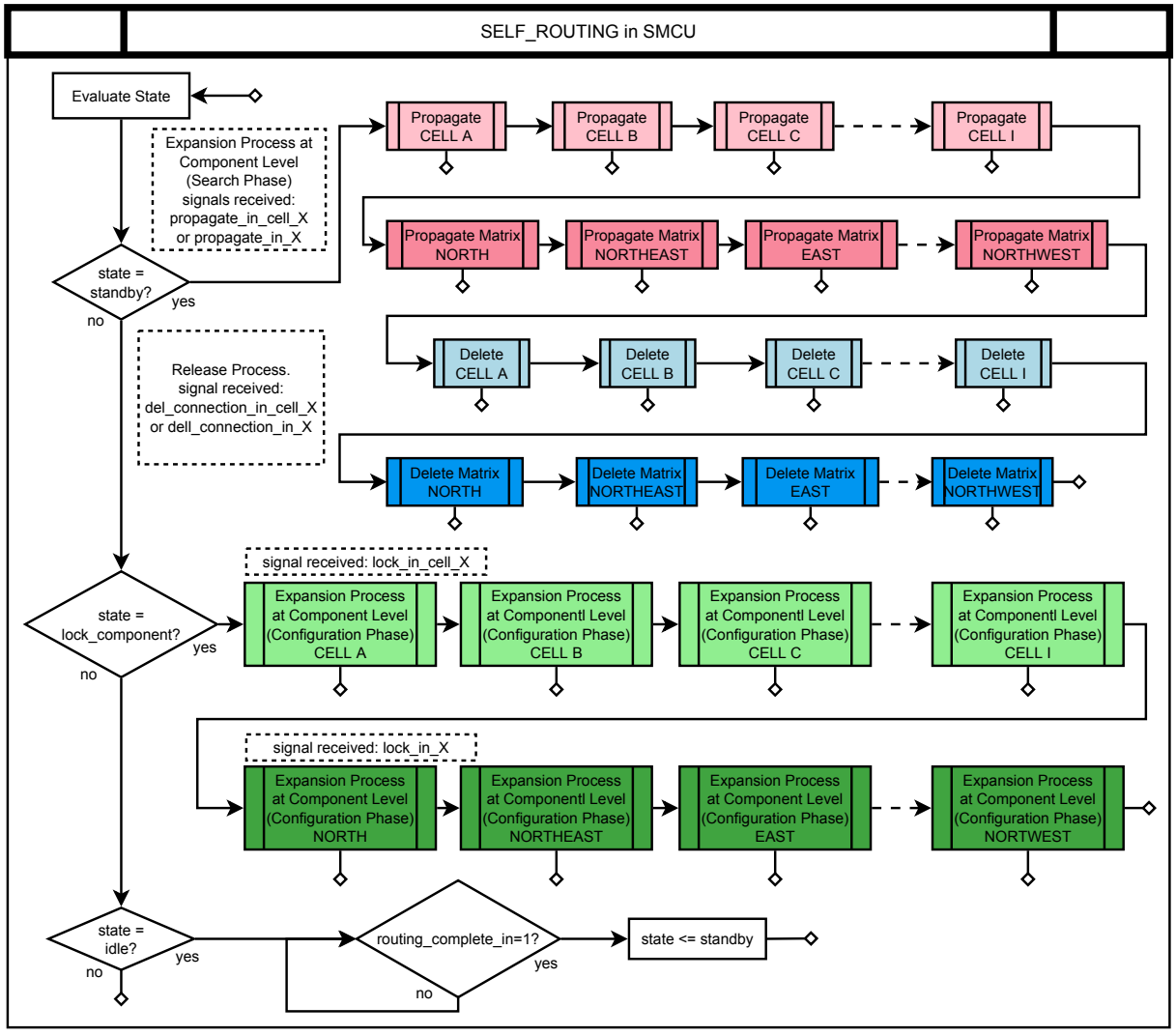


Figure C.11: Main flow diagram for Expansion and Release processes in Switch Matrix Configuration Unit.

The SMCU remains in *standby* state until the CCU of a cell in the array starts one of the following process:

1. Expansion Process at Component Level: this process is started by the *source_cell* when the signal *start_component_propagation* is set.
2. Expansion Process at Component Level: this process is started by the *source_cell* when the signal *start_component_propagation_pin* is set.

3. Release Process at Component Level: this process is started by the *target_cell* when the signal *start_delete_cell_connection* is set.

The SM that receives any propagation signals can execute any of the process denoted with a box in the figure, as follows:

1. Propagate CELL [A, B, C, D, E, F, G, H or I]: The process is executed when a *propagate_in_cell_X* signal is received. These processes correspond with the Search Phase of the Expansion Process at Component Level.

2. Propagate Matrix [NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST or NORTHWEST]: The process is executed when a *propagate_in_X* signal is received. These processes correspond with the Search Phase of the Expansion Process at Component Level.

3. Delete Cell [A, B, C, D, E, F, G, H or I]: The process is executed when a *del_connection_in_cell_X* signal is received. These boxes correspond with the Release Process at component level.

4. Delete Matrix [NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST or NORTHWEST]: The process is executed when a *del_connection_in_X* signal is received. These boxes correspond with the Release Process at component level.

5. Expansion Process at Component Level - Configuration Phase [A, B, C, D, E, F, G, H or I]: The process is executed when the SM is in *lock_component* state and a *lock_in_cell_X* signal is received.

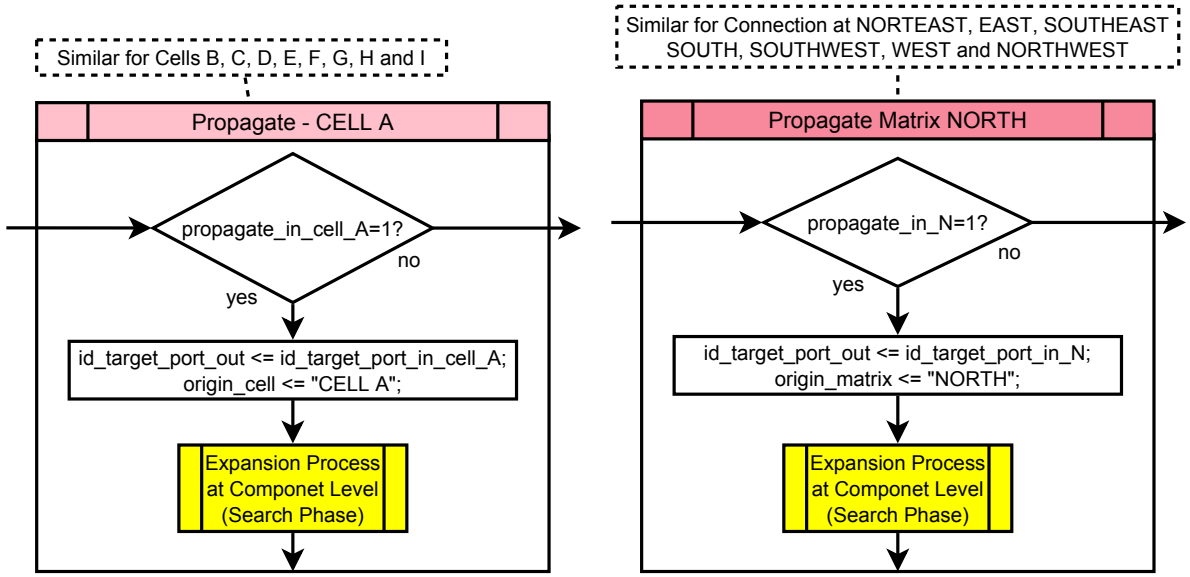
6. Expansion Process at Component Level - Configuration Phase [NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST or NORTHWEST]: The process is executed when the SM is in *lock_component* state and a *lock_in_X* signal is received.

C.4.1 Expansion Process at Component Level - Search Phase

The search phase is started by the *source_cell* (Figure C.5) with the signal *propagate_out_matrix*. The *source_cell* goes to the *lock_component* state waiting for the Configuration Phase. The process continues at component level when the SM reads the signal *propagate_in_cell_X* as shown in Figure C.12a, then the SM propagate signals until it finds the *target_cell*. If the *target_cell* is not in the same cluster, the SM continues the Search Phase when it receives the propagation signal *propagate_in_X* as shown in Figure C.12b. Note that these figures show the case for a propagation signal received from Cell A and SM at NORTH. Similar functionality must be assumed for other cells and other sides of SM.

The expansion process propagates signals in the sides of the SM that has available routing resources. The propagation process includes the configuration of signals for each side (north, northeast, east, southeast, south, southwest, west and northwest) of the SM and for each cell of the cluster (Cell A, Cell B, ... Cell I) as shown in Figure C.13.

The priority order for propagation signals is NORTH, NORTEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST and NORTHWEST. The *origin_cell* and *origin_matrix* register stores the cell or the side from which the propagation signal was received. The propagation process explores the entire cell array until finding the *target_cell*, if possible. The Configuration Phase starts when the Search Phase finds the *target_cell*.



(a) Propagation signal from a Cell. (b) Propagation signal from a Switch Matrix.

Figure C.12: Flow diagrams for propagation input signals at component level in Switch Matrix Configuration Unit.

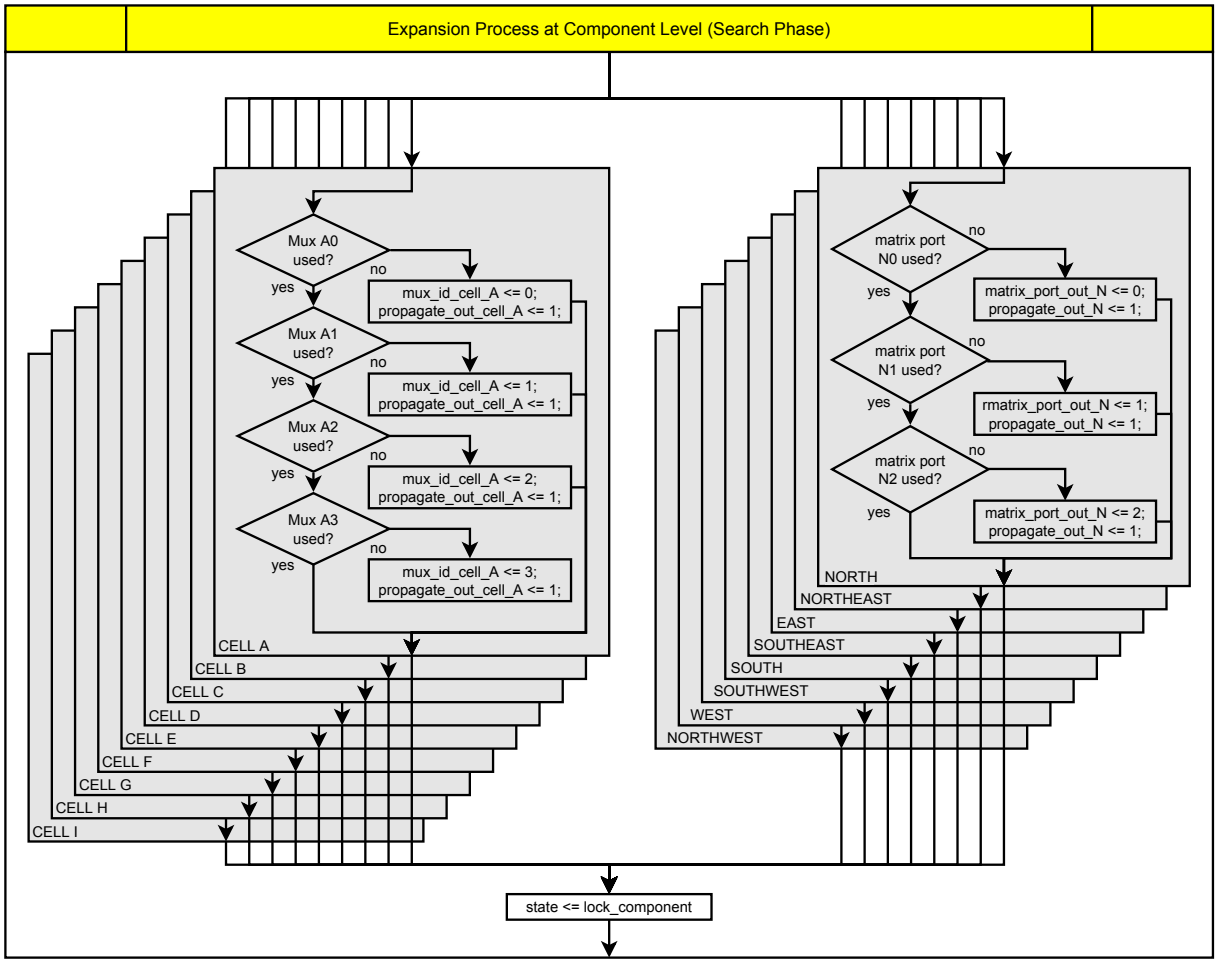


Figure C.13: Flow diagram for the propagation of Signals in the Search Phase of the Expansion Process at Component Level.

C.4.2 Expansion Process at Component Level - Configuration Phase

The Configuration Phase at Component Level starts when the Search Phase finds the *target_cell* (Figure C.5), in the process denoted as Propagate Matrix (Figure C.7b), then the multiplexer of the appropriate FU input port is configured (FU *target_port*).

The Configuration Phase that was started by the *target_cell* goes backward over the path previously configured in the Search Phase until it arrives to the *source_cell*. This process configures the multiplexers of the corresponding SMs to fix the path.

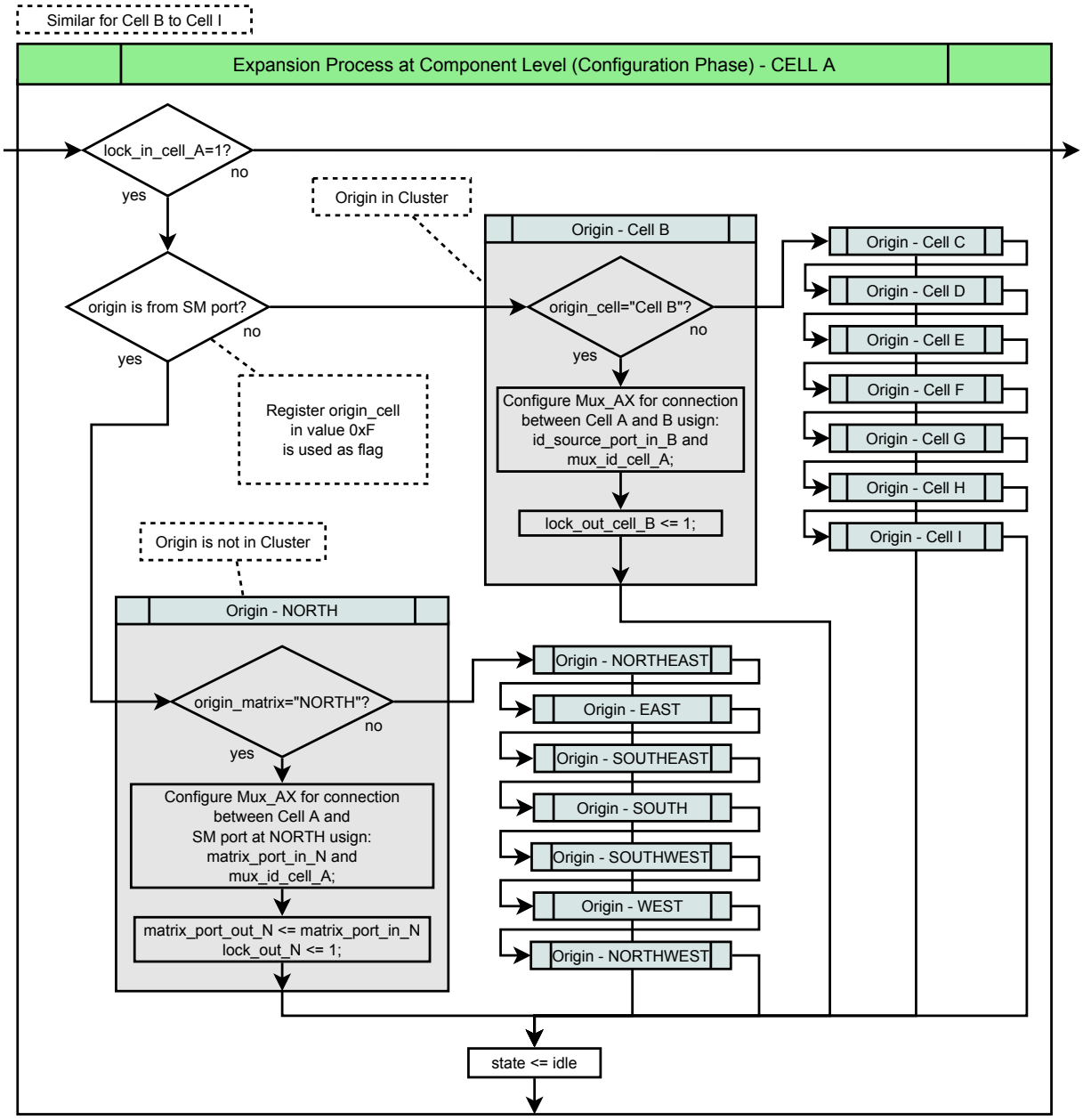


Figure C.14: Flow diagram for Configuration Phase of the Expansion Process at Component Level when the *lock_in* signal comes from a Cell.

The process continues as follows:

1. The *target_cell* sets the signal *lock_out_matrix*. Therefore, the SM that receives the signal *lock_in_cell_X* performs the action described in Figure C.14. Similar functionality must be assumed for other cells in cluster. If the *source_cell* belongs to the same cluster the process ends, otherwise the process continues in step 2.

2. The SMs that participate in this process are in the state *lock_component*. The SMs in this state perform the actions described in Figure C.15 when the signal *lock_in_X* in the corresponding side is activated. Similar functionality must be assumed for other sides of SM. This process is repeated until it finds the *source_cell*, which ends the process by setting the global signal *routing_complete*.

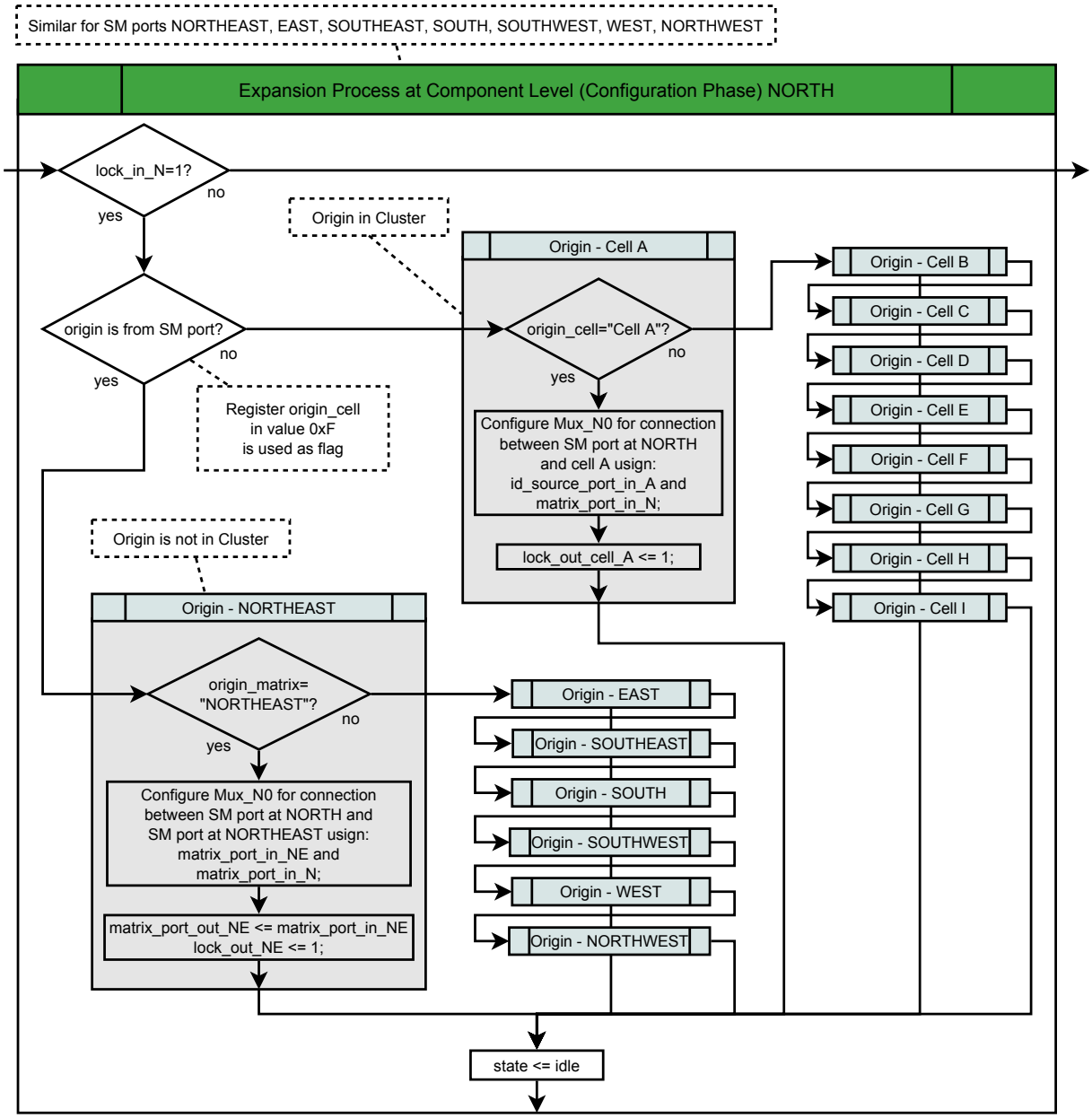


Figure C.15: Flow diagram for Configuration Phase of the Expansion Process at Component Level when the *lock_in* signal comes from a Switch Matrix.

C.4.3 Release Process at Component Level

This process releases the routing resources used for a interconnection between cells, or between a cell and a PIM. The Release Process goes from the *target* to the *source* of a connection.

If the connection is at component level, the multiplexers of FU inputs and SM ports are released. In this case, the Release Process starts from the FU input port of the *target_cell* an go backwards at component level to the FU output port of the *source_cell*. This process reads the multiplexers configuration and propagate the signals *del_connection_out_cell_X* and *del_connection_out_X* in the direction where the connection was previously established by self-routing process.

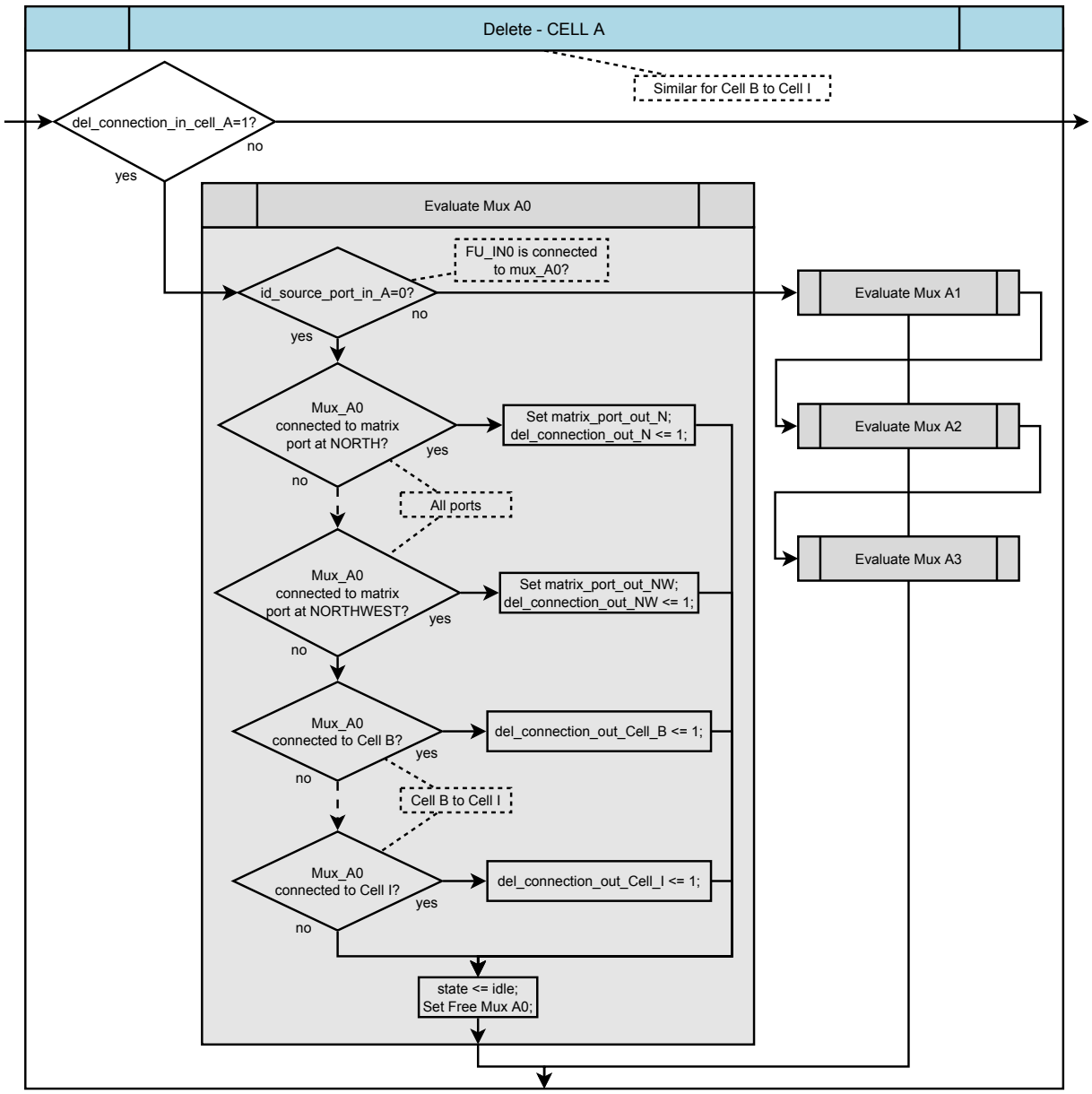


Figure C.16: Flow digram of Release Process at Component Level when *del_connection* signal comes from a cell.

The *target_cell* starts the process presented in Figure C.5 when the flag *start_delete_cell_connection* is set. Then, the *target_cell* executes the flow diagram presented in Figure C.9. Thereafter, the SM that receives the signal *del_connection_in_cell_X* executes the algorithm presented in Figure C.16.

The process continues at component level. The SMs propagate the appropriate *del_connection* signal. If the *source_cell* does not belong to the cluster, the SM propagates the signal *del_connection_out_X* and the process continues in other cluster as shown in Figure C.17.

The process continues until it reaches the *source_cell*, which ends the process setting the global signal *routing_complete* as shown in Figure C.5 in the box denoted as "Delete Matrix".

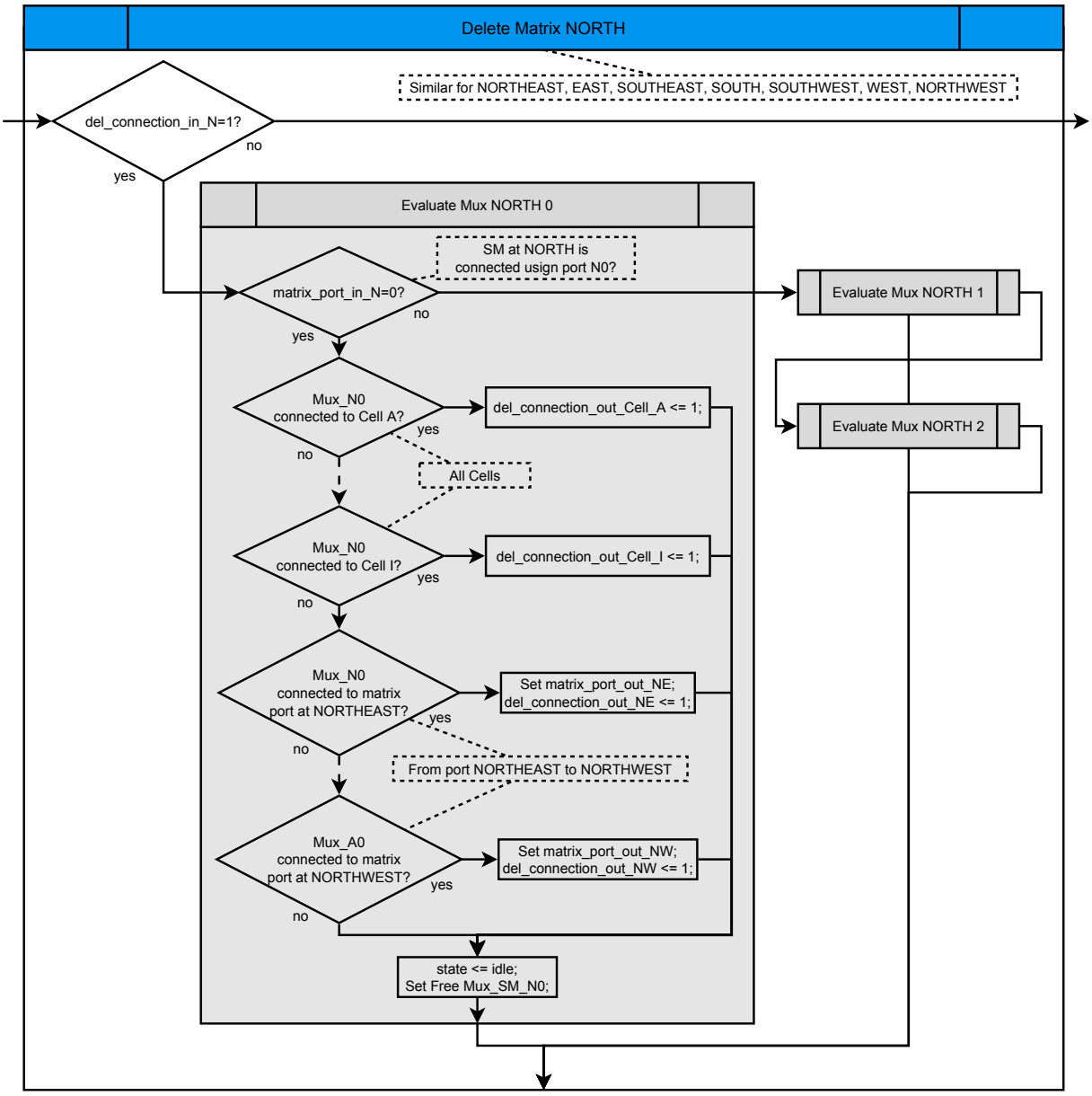


Figure C.17: Flow digram of Release Process at Component Level when *del_connection* signal comes from a Switch Matrix.

C.5 Conclusions

This section presents the flow diagrams of self-adaptive algorithms implemented in the Configuration Units of Cells and Switch Matrices.

The Cell Configuration Unit (CCU) is the start point for the execution of any process in system that requires the propagation of signals using the expansion ports of cells, SMs and PIMs. The CCU includes the following algorithms:

- The self-placement algorithm is divided in two. The first is the algorithm for the insertion of the first cell of a component, and the second is the algorithm for the insertion of other cells of a component (from the second).
- Since the insertion of the second cell of a component the self-routing at cell level algorithm is executed. This is implemented with the following process: first, the selection of source and target cells for the connections to be routed, and later the Expansion Process at Cell Level, which is divided in Search and Configuration Phases. This process is repeated for all connections that can be routed for the new inserted cell.
- The flow diagrams for the Release Process at Cell Level are presented. These algorithms permits to release the routing resources at cell level used for the interconnection of the FU ports of two cells. These algorithms are used for self-derouting process, before eliminating a single cell or deleting an entire component.

The Switch Matrix Configuration Unit (SMCU) includes the following algorithms:

- The Expansion Process at Component Level, which is divided in Search and Configuration Phases. This process is repeated for all connections that can be routed for the components.
- The flow diagrams for the Release Process at Component Level are presented. These algorithms permit to release the routing resources at component level used for the interconnection of the FU ports of two cells belonging to different components. These algorithms are used for self-derouting process, before eliminating a single cell or deleting an entire component.

Appendix D

SANE Project Developer (SPD)

You only live once, but if you do it right,
once is enough.

Sólo se vive una vez, pero si lo haces bien,
una vez puede ser suficiente

Mae West (1893 – 1980)

Abstract: This section describes the main features of the software tool SANE Project Developer (SPD). It presents all functions included in the software. Additionally, it is presented the basic syntax consideration for most common files that will be edited in SPD. The protocol for downloading files to prototype is shown at the end of this chapter.

D.1 Description

In the design phase of the architecture, one of the the main inconveniences was the creation of applications that permit to test the system. This process consisted in the creation of a SANE ASSEMBLY (SANE-ASM), that includes a specific number of interconnected components, and interconnected cells inside each component. Additionally, multiple processors had to be manually programmed and compiled. Any modification in the application implies a lot of time rewriting the data for its configuration. Similar to any commercial general purpose device a software tool is fundamental for improving the capacity of a designer when developing applications.

The SANE Project Developer (SPD) is an Integrated Development Environment (IDE) developed in C Sharp with Microsoft Visual C# 2008. The SPD permits the user –in a friendly way– the creation and edition of projects that describe any SANE-ASM application designed. The SPD includes an adaptation of the project ICSharpCode.TextEditor[37], which is a feature-rich text editor control that provides the SPD with a powerful tool for edition of files in a project[1]. The text editor for SPD includes a specific configuration for syntax highlight of files related with the project, as well as other basic features for an advanced edition of files.

The SPD allows generating and downloading the memory initialization data for the control microprocessor inside the prototype. The SPD allows the creation of files that describe the configuration of a SANE-ASM. This files are called SASM files, which includes the high-level (or SASM) instructions described in Chapter 5. For writing the Program Memories of the processors,

[1] ICSharpCode.TextEditor is licensed under The MIT License.

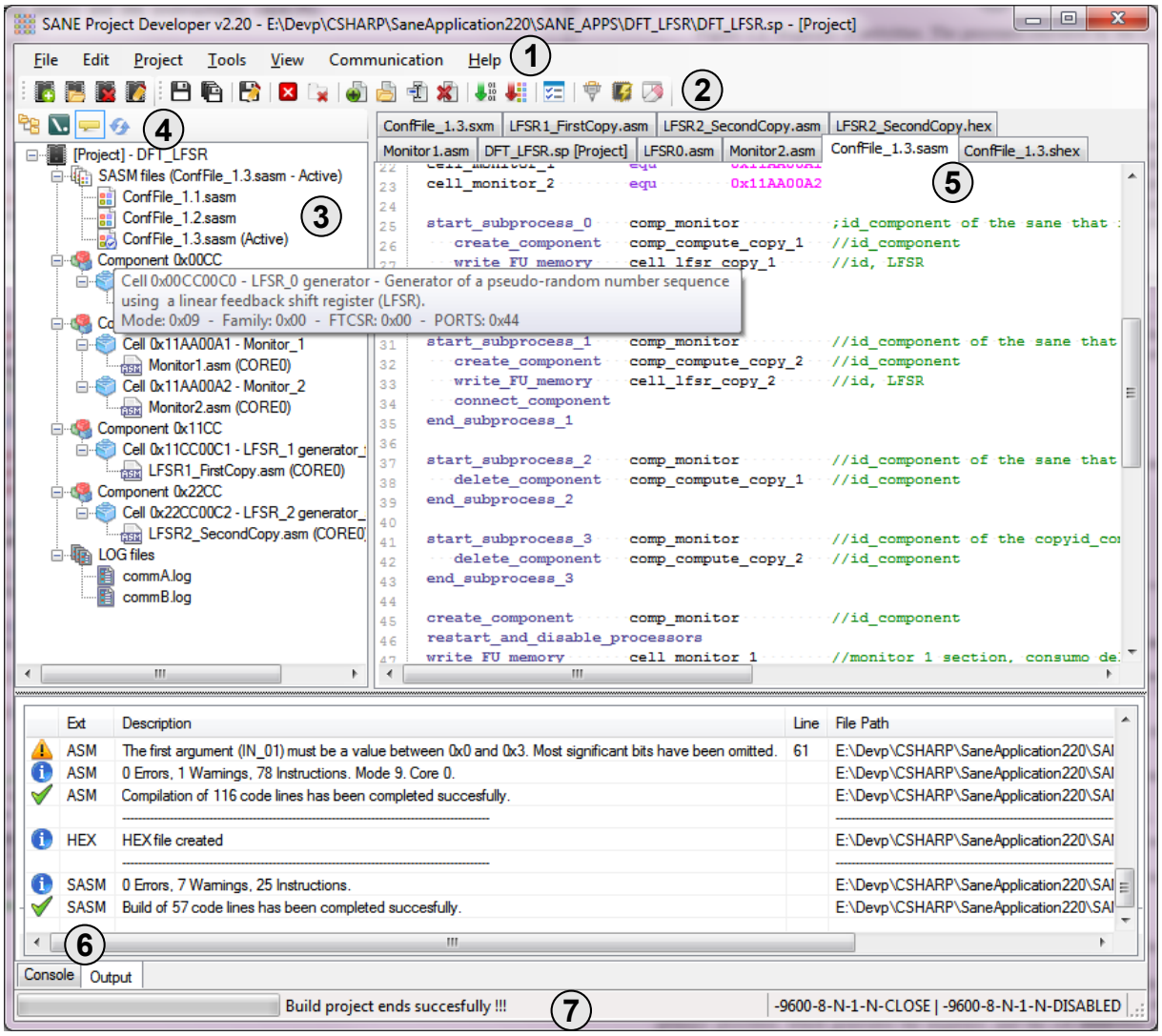


Figure D.1: Screen capture of SANE Project Developer.

Assembler (ASM) files will be created, this permits to execute the functionality of a processor in the cell (between 1 and 4 per cell depending of operation mode).

The SPD supports building of the final hexadecimal file with the configuration of the SANE-ASM that will be implemented in the FPGA. This process includes the compilation of the files involved in the process according to the SASM file. The SPD generates a list of errors, warnings and infos for all files involved in the building process and guides the user to make the appropriate corrections if required. The SPD also supports the compilation of individual ASM files.

Figure D.1 shows a screen capture of the SPD, which shows the SASM file for the Dynamic Fault Tolerance Application presented in Section 5.5. It is also shown the structure of the solution in the left tree and the output of the result of building process in the bottom section. Below are the features of the main form presented in the figure by means of numeric labels:

(1) **Menu Strip:** it includes the following menus: File, Edit, Project, Tools, View, Communication, Help and Admin (only available for developer). All actions that SPD is able to execute are included in these menus, which will be detailed later in this chapter.

(2) **Top Tool Strip:** it includes buttons for easy access to most common functions in SPD. All

this functions are included in the Menu Strip.

3. **Left panel:** it includes a Tree View object that contains the hierarchical organization of the project. This tree could be shown in Cell View or Folders View. The figure shows the tree for a Cell View, which shows the project organized in components and cells. Cell View includes all files associated with the project, whilst Folders View includes all files in project folder.

4. **Buttons for Left Panel:** these buttons permit the configuration the Tree View object in Left Panel. It includes the option to alternate between Cell View and Folder View, visualization of full path of files, visualization of tooltip text and option to force a Refresh over the tree.

5. **Main Tab Control:** it includes the Tab Pages that could be edited by SPD. These tabs are divided in three categories:

   (a) **Project Tab:** This tab permits to configure specific options for a project created by SPD. The configuration of components and cells in a project will be detailed later in this chapter.

   (b) **Communication Tab:** This tab represents the data that is sent and received from prototype by means of serial port connections (RS-232). The SPD permits to configure between one and two communication panels. This tab is used for two purposes: the visualization of data downloaded to prototype using an adaptation of XMODEM protocol, and the visualization of the execution of a SANE-ASM in the prototype for debugging purposes (for two chips in prototype).

   (c) **Text Editor**. The tabs that includes any file could be edited using the text editor adapted for the application. This editor includes a specific configuration for syntax highlight of files with the following extensions: `*.ASM`, `*.HEX`, `*.INC`, `*.SASM` and `*.SHEX`.

6. **Output Tab Control:** it includes two tab pages:

   (a) **Console Tab:** this tab shows information of all functions that are being executed in the SPD. This tab s mainly used for debugging purposes.

   (b) **Output Tab:** this tab is updated each time that a ASM file is compiled or the project is built. This tab shows Warnings, Informations, Errors and the result of the process executed. It permits to navigate in a friendly way to the file associated with the message, and presents the appropriate information related with the message.

7. **Status Bar:** This bar shows the status of some features of SPD as follows:

   (a) Status and configuration of communication ports.

   (b) Result of a single file compilation, or built of a project.

   (c) Progress of a process that spends time, like downloading of data to the prototype and others.

D.2 Files Edition

Table D.1 shows a description of files that SPD is able to edit. The ASM and SASM files are the most frequent files that will be edited in a project using the text editor. The SPD is not case sensitive for reserved words.

File Extension	Description
*.SP	**SANE Assembly Project.** This file is automatic configured for the SPD. It is advisable does not edit manually this file because the project might be corrupted.
*.SASM ✔	**SANE Assembler file.** It corresponds to the high-level configuration file, which includes the SASM instructions.
*.SHEX ✔	**SANE Hexadecimal File.** This file is generated when the project is built. The file is generated taking into account the active *.sasm file.
*.SXM	**SANE XMODEM File.** This file is generated when the project is built. The file is generated taking into account the active *.sasm file. It is used for downloading the project to prototype memory.
*.ASM ✔	**Assembler file.** It corresponds to the code implemented in processors of cells.
*.INC ✔	**Include file.** It corresponds to libraries for *.asm files.
*.HEX ✔	**Hexadecimal File.** This file is generated when an individual *.asm file is compiled.
*.LOG	**Communication LOG Files.** It corresponds to the communication activities related with the RS-232 ports.
*.TXT	**Text Files.** It corresponds to plain text for general purpose.
.	**Any Files.** The SPD allows the creation of any editable file in plain text.

✔ The files with this extension includes syntax highlight.

Table D.1: Relation of files for SANE Project Developer.

D.2.1 Assembler Files

The Assembler (ASM) files correspond to the code implemented in the processors of cells. Depending on configuration mode, it is possible to have between one to four ASM files per cell. The SPD includes the following features for ASM files:

► **Reserved words for instructions:**

```
ADDLW, SUBLW, ANDLW, IORLW, XORLW, MOVLF, ADDWY, SUBWY,
ANDWY, IORWY, XORWY, MOVW, BLMOV, COMW, NEGW, INCW, DECW,
SWAPW, RLW, RRW, LSL, LSR, ASL, ASR, CLRF, CLC, SEC, END,
NOP, BCLR, BSET, BRCLR, BRSET, GOTO, BZ, BNZ, BC, BNC,
CBEQ, CBGE, CBGT, CBNE, DBNZ, IBNZ.
```

► **Reserved words for directives:**

```
EQU, ORG, MODE_CORE, #INCLUDE
```

► **Numbers format:**

- Hexadecimal (default): `0xe9, 0XE9, e9, h'E9', H'e9'`
- Decimal: `d'233', D'233', .233`
- Octal: `o'351', O'351'`
- Binary: `b'11101001', B'11101001'`

- ASCII: `a'G', A'k'` (one alphanumeric character, from ASCII 20h to 7Eh).

▶ **Comments and labels:** The labels must start with a letter or underscore (`a-z,A-Z,_`), later may include any other alphanumeric character. The comments can be included with any sequence of text after semicolon or double-slash:

```
label1                          ;This is a label
    movlf    .255,bin8,0        ;Example 1 of a comment
cycle_38                        //This is another label
    BLMOV    BL_IN0,0x3F        //Example 2 of a comment
```

D.2.2 SANE Assembler Files

The SANE Assembler (SASM) files corresponds to the high-level configuration file for the implementation of a SANE-ASM in the system. The SPD includes the following features for SASM files:

▶ **Reserved words for instructions:**

```
CREATE_COMPONENT, CONNECT_COMPONENT, DELETE_COMPONENT,
WRITE_FU_MEMORY, WRITE_FU_MEMORY_CR, WRITE_FU_MEMORY_PM0,
WRITE_FU_MEMORY_PM1, WRITE_FU_MEMORY_PM2,
WRITE_FU_MEMORY_PM3, RESTART_PROCESSORS,
DISABLE_PROCESSORS, RESTART_AND_DISABLE_PROCESSORS,
ENABLE_PROCESSORS, WAIT, RESTART_PROCESSORS_WAIT,
ENABLE_PROCESSORS_WAIT, END, START_SUBPROCESS_0,
END_SUBPROCESS_0, START_SUBPROCESS_1, END_SUBPROCESS_1,
START_SUBPROCESS_2, END_SUBPROCESS_2, START_SUBPROCESS_3,
END_SUBPROCESS_3, FT_CONFIGURATION.
```

▶ **Reserved words for directives:**

```
    EQU
```

▶ **Numbers format:**

- Hexadecimal (default): `0xe9, 0XE9, e9, h'E9', H'e9'`
- Decimal: `d'233', D'233', .233`
- Octal: `o'351', O'351'`
- Binary: `b'11101001', B'11101001'`
- ASCII: `a'G', A'k'` (one alphanumeric character, form ASCII 20h to 7Eh).

▶ **Comments:** The comments can be included with any sequence of text after semicolon or double-slash:

```
primary_cell      equ  0xAAAA0001   ;cell definition - hexa
redundant_cell    equ  d'8613'      //cell definition - decimal
comp_A            equ  0xAAAA       //component definition hexa

ft_configuration  primary_cell,redundant_cell ;Comment style 1
create_component  comp_A                      //Comment style 2
```

D.3 Functions

The SPD includes all necessary tools for developing and executing applications in the prototype. Any SANE-ASM application is implemented by means of a Project, which includes all files related with the application. The main characteristics of the SPD are included in the Main Menu, which permits the user to execute the corresponding action as shown bellow. In many cases the name of the option selected denotes the action that will be executed and does not require additional explanation.

D.3.1 File Menu

It includes tools for management of projects and files. The options included in this menu are presented bellow:

1. **New Project.** Starts a template for creation of a new project.
2. **Open Project.**
3. **Close Project.**
4. **Save Project As...**
5. **Save File As...**
6. **Save '*.*'.** Save current file or tab selected in Main Panel (Ctlr+S).
7. **Save All.** Save all files (Ctlr+Shift+S).
8. **Close '*.*'.** Close current file or tab selected in Main Panel.
9. **Close all except selected tab.** Close all files except the selected tab in Main Panel.
10. **New File.** Start a template wizard for creation of a new file. The new file could be created with a basic configuration template for files `*.asm`, `*.sasm` and `*.txt`.
11. **Open File.**
12. **Rename File.**
13. **Delete File.**
14. **Recent Projects.** List of recent projects for quick access.
15. **Recent Files.** List of recent files for quick access.
16. **Exit.**

D.3.2 Edit Menu

This menu includes basic and advanced tools for edition of files. The options included in this menu are presented bellow:

1. **Split text area.** This options permits visualization of a file in two different segments of code.
2. **Cut** (Ctlr+X).
3. **Copy** (Ctlr+C).
4. **Paste** (Ctlr+V).
5. **Delete.**
6. **Select All** (Ctrl+A).
7. **Find ...** (Ctrl+F).
8. **Find and replace ...** (Ctrl+H).
9. **Find again** (F3).
10. **Find again reverse** (May+F3)
11. **Display.** This is a sub-menu that permits the following actions over text editor:
    - **Show line numbers.**
    - **Show end of line markers.**
    - **Highlight current line.**

- **Show spaces and tabs.**
- **Highlight matching brackets when cursor is after.**
- **Allow cursor past end of line.**

12. **Bookmark.** This is a sub-menu that permits the following action over text editor:
    - **Toogle bookmark.** (Ctrl+F2).
    - **Goto next bookmark.** (F2).
    - **Go to previous bookmark.** (May+F2).
    - **Clear all bookmark.**
13. **Uppercase/Lowercase.** This is a sub-menu that permits the following actions over text editor:
    - **Convert to uppercase.** (Ctrl+Shift+U).
    - **Convert to lowercase.** (Ctrl+U).
14. **Advanced.** This is a sub-menu that permits the following actions over text editor:
    - **Toggle line comments.** (Alt+Q).
    - **Convert spaces to tabs**.
    - **Convert tabs to spaces.**

D.3.3 Project Menu

This menu includes tools for the current project opened as follows:

1. **Component Editor (F5).** Tool that permits to edit the components configuration in project. Figure D.2 shows the tool implemented fo this purpose. Note that there are two different views, one for addition and the other for edition. This tool permits the user configure the components and cells for a specific project.

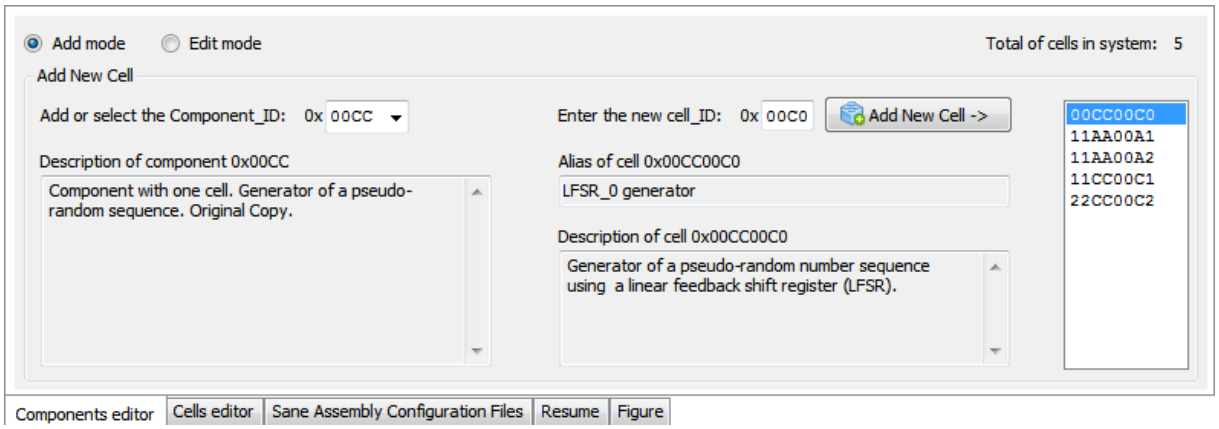
2. **Cell Editor (F6).** Tool that permits to edit the cells configuration in project. Figure D.3 shows the tool implemented for this purpose. This tool permits the user to configure the cells previously included in the project. Thus, the user is able to edit the Configuration Registers, Connection Tables and the ASM files associated to each CORE.

3. **SASM Configuration Files (F7).** Tool that permits to add and select the SASM files to project. Figure D.4 shows the tool implemented for this purpose. This tool permits to add and/or activate the SASM file that will be used when "Built Project" tool is executed. Note that SPD permits to have many different SASM files, but only one can be activated.

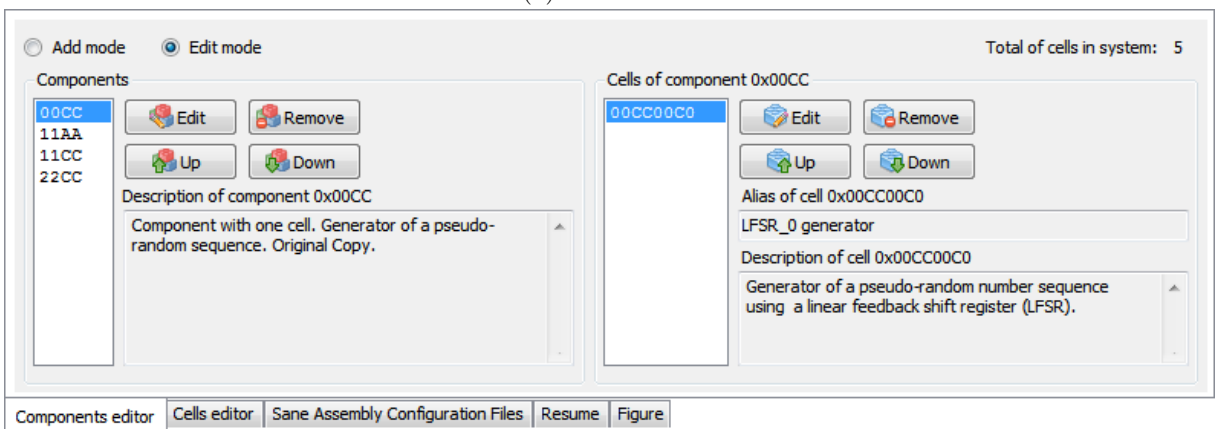
4. **Compile File (F9).** Compile the selected ASM file. This tool generates a `*.HEX` file with the same name of the correspondent `*.ASM` file. This function could be executed previously to built the project for detecting error or warnings in ASM files. In this case, the software guides the user to find the source of the correspondent message.

5. **Built Project (F10).** Bulit the project taking into account the active SASM file. This tool generate the files `*.SASM` and `*.SXM` with the same name of the active `*.SASM` file. This is the last function that must be executed before downloading the configuration file (SXM) to CμP memory in prototype. If there are errors or warnings in this process, the software guides the user to find the source of the correspondent message.

6. **Add copy of source.** This tool permits to make a copy of any file to the project folder.



(a) Add mode.



(b) Edit mode.

Figure D.2: Component editor tool.

D.3.4 Tool Menu

This menu includes the following options:

1. **Export.** This is a sub-menu that permits the user perform the following actions:
   - **Figure.** Exports a figure that represents the architecture implemented in the prototype.
   - **Screen Capture.** Exports a screen capture of the current view of SPD (formats `*.png`, `*.jpg`, `*.gif` or `*.bmp`).
   - **Console.** Exports the console as a `*.txt` file.

2. **Clear Console.**

3. **Options.** This is a form that permits the user to configure several option for SPD. This form includes five tabs as follows:
   (a) **General:** configuration of general options for SPD.
   (b) **Text Editor:** configuration of text editor.
   (c) **Compiler:** configuration for compilation tools.
   (d) **Tree:** configuration of project visualization tree in Left Panel.
   (e) **Communications:** configuration of serial ports (RS-232) connected with prototype.
   (f) **Figure:** configuration of figure that represents the system architecture.

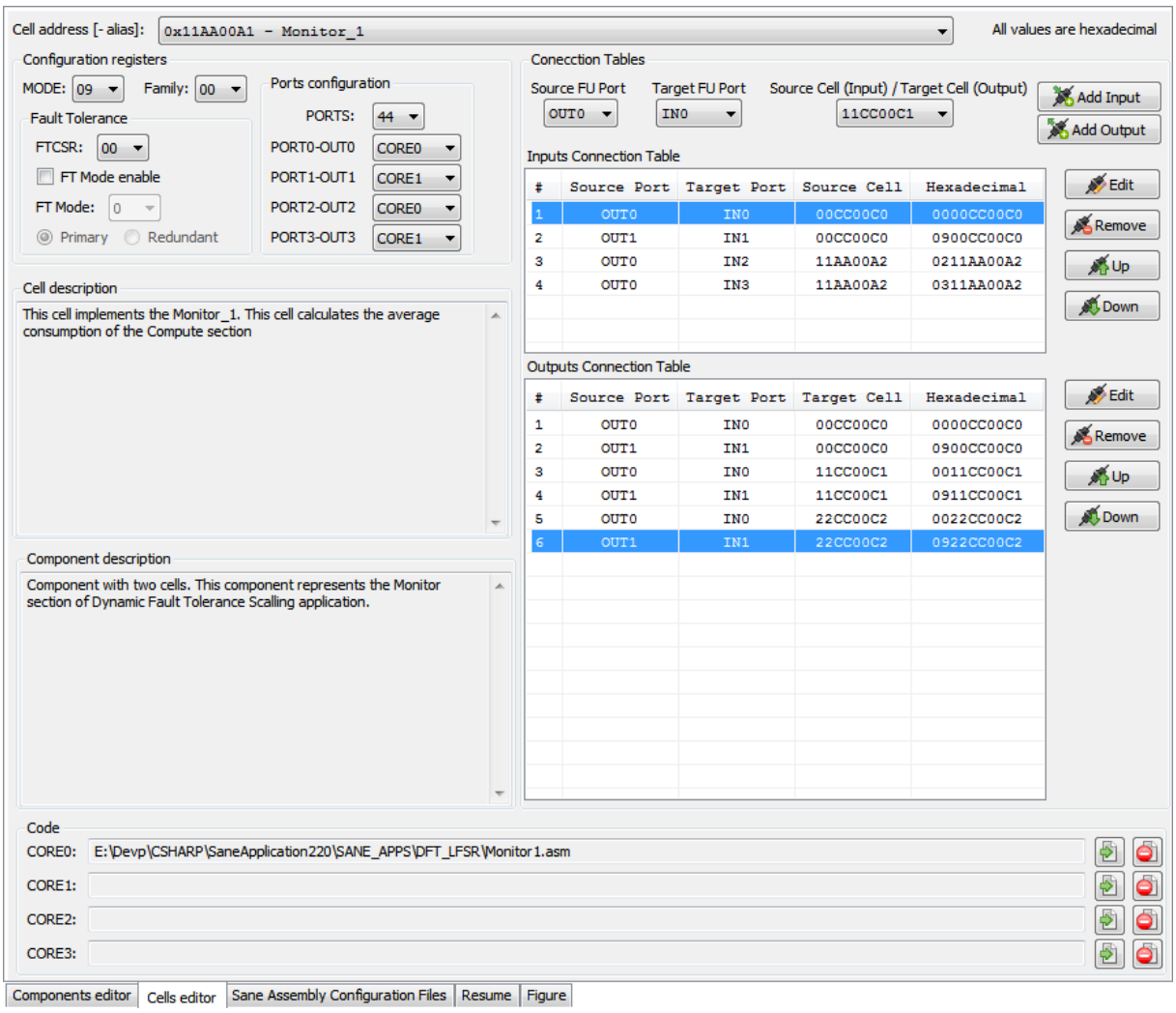


Figure D.3: Cell editor tool.

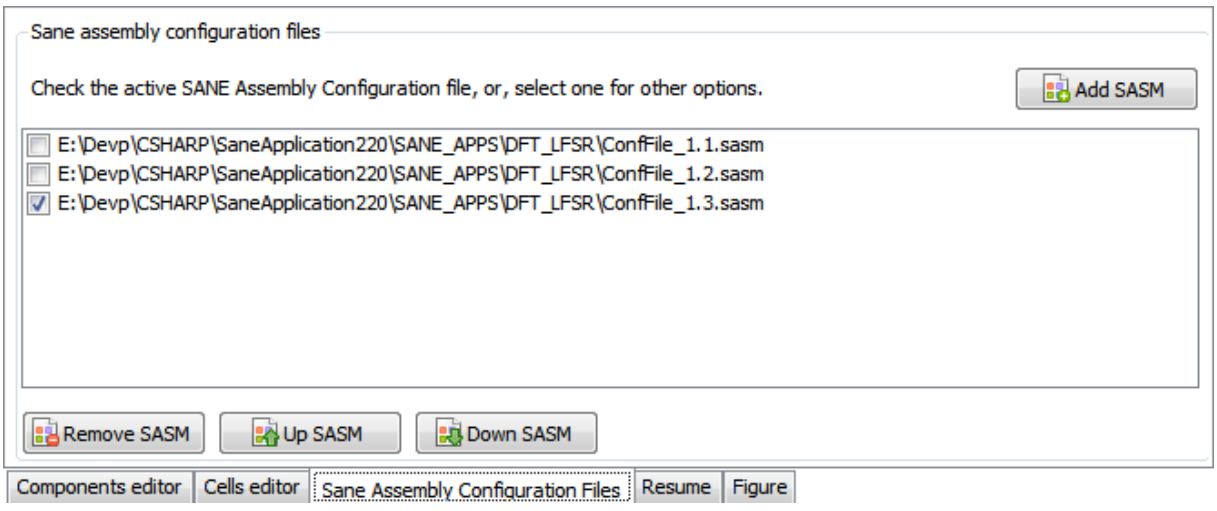


Figure D.4: Tool for addition/activation of SANE assembler files.

D.3.5 View Menu

This menu is used for accessing some special Page Tabs object. Note that project and communication tabs could be closed in the same way of a text editor tab. It includes the following options:

1. **Project Tab.** Permits to open the project tab for edition of components, cells and addition/activation of SASM files.

2. **Communication Tab.** Permits to open the communication tab for visualization of processes executed in serial ports.

3. **Console Tab.** Permits to open the console tab for debugging purposes.

D.3.6 Communication Menu

This menu includes the tools related with the serial port communication, which are used to send/receive information to FPGAs in prototype. Some of the functions listed bellow are detailed in section D.4. This menu includes the following options:

1. **Open/close Communication Ports.** This tool opens or closes the serial ports configured for communication with FPGAs in prototype. Additionally, this tool starts or stops threads that permits the correct management and visualization of data in communication panels.

2. **Clear Communication Panles.**

3. **Save copy of communication logs.**

4. **Communication Test.** Execute a communication test between the SPD and the CμP in the prototype. It permits to check the correct configuration of serial ports for the execution of other functions over CμP memory.

5. **Clear Memory.** Execute a process to clear all section of memory in CμP dedicated for the SASM configuration file.

6. **Write Memory.** Execute a process to write the SXM file that was previously generated with the "Built Project" function. This file is written in section of memory dedicated for the high-level configuration file.

7. **Read Memory.** Execute a process to read the entire section of memory dedicated for the high-level configuration file.

8. **Cancel FPGA communication process.** This function permits to cancel in a safe way any process that has been started for a communication between the SPD and the CμP.

D.3.7 Help and Admin Menus

The Help menu presents the credits of the SPD (About). Note that this chapter can be used as a User Manual for the software implemented. The Admin menu is only available for debugging or developer options. It is activated executing the SPD with the argument: `/admin`.

Field	Name	Description
<CB>	Control Byte	Define the process that will be executed: <CB> = 0x01: Communication test. <CB> = 0x02: Clear Memory. <CB> = 0x03: Write Memory. <CB> = 0x04: Read Memory.
<DB>	Data Bytes	Denote the zero-based number of bytes send or received in the frame (0x00 to 0xFF represents 1 to 256 bytes)
<AAA1>	Address 1	Initial address for write/read operations (2 bytes).
<AAA2>	Address 2	Final address for read operations (2 bytes).
<D0 D1···D255>	Data Bytes	Bytes sent or received in frame. The amount of data bytes depends on the Data Bytes field.
<CKS>	Checksum	Two's complement of the arithmetic sum of the bytes in frame except the <SOH> and <CRC>. Final carry is discarded.

Table D.2: Description of fields for XMODEM based protocol.

D.4 Downloading Project to Prototype

The SPD and the CμP implements an adaptation of the XMODEM protocol for downloading projects to prototype. XMODEM, like most file transfer protocols, breaks up the original data into a series of "packet" or "frames" that are sent to the receiver, along with additional information allowing the receiver to determine whether that packet was correctly received. Xmodem is a half-duplex communication protocol. The following bytes are defined for the protocol implemented:

- <SOH> = 0x01: Start of Heading.
- <EOT> = 0x04: End of Transmission.
- <ACK> = 0x06: Acknowledge.
- <NACK>= 0x15: Negative Acknowledge.
- <CAN> = 0x18: Cancel.

The frames sent or received have the structure presented below. Note that additional fields are only used when the frame starts with the byte <SOH>. The description of additional fields is shown in Table D.2.

```
<SOH> [<CB>] [<DB>] [<AAA1>] [<AAA2>] [<D0 D1 ... D255>] <CKS>
```

There are four basic functions implemented in SPD, which corresponds to the four values of the field "Control Byte". These functions are described in Sections D.4.1 to D.4.4. Note that the receiver, after receiving a packet, will either acknowledge (ACK) or not acknowledge (NACK) the packet. If a NACK is received five times, the sender cancel the process sending the correspondent byte (CAN). These sections present data flow examples including error recovery, which in normal conditions is not present in frames. The normal data flow is interrupted when an unexpected event is produced as follows:

1. A <NACK> is sent when there is a checksum error. On any <NACK>, the sender will re-transmit the last packet.
2. A <CAN< is sent when there is a format error or five <NACK> has been received.
3. When a <CAN> is received, the process is finalized and no more bytes are sent.

D.4.1 Communication Test

This function permits the user to ensure that communication with the prototype is configured properly. Table D.3 shows the data flow executed for communication test (<CB>=0x01).

SANE Project Developer	↔	Prototype
<SOH><01><CKS>	→	
	←	<NACK>[1]
<SOH><01><CKS>	→	
	←	<ACK>
<EOT>	→	
	←	<ACK>

[1] Framing error on any byte.

Table D.3: Example of data flow for Communication Test with prototype.

D.4.2 Clear Memory

This function permits to clear the entire section of memory dedicated to the configuration file in the prototype. Table D.4 shows the data flow executed for clear memory (<CB>=0x02).

SANE Project Developer	↔	Prototype
<SOH><02><CKS>	→	
	←	<NACK>[1]
<SOH><02><CKS>	→	
	←	<ACK>
<EOT>	→	
	←	<ACK>

[1] Framing error on any byte.

Table D.4: Example of data flow for Clear Memory in prototype.

D.4.3 Write Memory

This function permits to write a portion of the memory in prototype, which corresponds to the high-level configuration file (SASM file). The format of the first frame sent by SPD is shown bellow:

```
<SOH> <CB=0x03> <DB> <AAA1> <D0 D1 ... D255> <CKS>
```

Table D.5 shows an example of the data flow executed for write five 32-bit words in memory, i.e., from address 0x0000 to 0x0004 (<CB>=0x03).

Note that words in prototype memory are 32-bit width. Therefore, the Data Bytes must be a multiple of 4 in zero-based range, e.g., 0x03, 0x07, 0x0B. This implies that address is incremented each four data bytes as shown in the example. If the SPD receives NACK five times, the SPD sends the byte <CAN> and the process ends.

SANE Project Developer	↔	Prototype
<SOH><03><07><0000><00 01 02 03 04 05 06 07><CKS>	→	
	←	<ACK>
<SOH><03><07><0002><08 09 0A 0B 0C 0D 0E 0F><CKS>	→	
	←	<NACK>[1]
<SOH><03><07><0002><08 09 0A 0B 0C 0D 0E 0F><CKS>	→	
	←	<ACK>
<SOH><03><03><0004><10 11 12 13><CKS>	→	
	←	<ACK>
<EOT>	→	
	←	<ACK>

[1] Framing error on any byte.

Table D.5: Example of data flow for Write Memory in prototype.

D.4.4 Read Memory

This function permits to read a portion of the memory in prototype, which corresponds to the high-level configuration file (SASM file). The format of the first frame sent by SPD is shown bellow:

```
<SOH> <CB=0x04> <DB> <AAA1> <AAA2> <CKS>
```

The format of the frame with data bytes sent by CμP is shown bellow:

```
<SOH> <DB> <AAA1>  <D0 D1 ... D255> <CKS>
```

Table D.6 shows an example of the data flow executed for read five 32-bit words in memory, i.e., form address 0x0000 to 0x0004 (<CB>=0x04).

Note that words in prototype memory are 32-bit width. Therefore, the Data Bytes must be a multiple of 4 in zero-based range, e.g., 0x03, 0x07, 0x0B. This implies that address is incremented each four data bytes as shown in the example. If the SPD receives NACK five times, the SPD sends the byte <CAN> and the process ends.

The size of the data sent by CμP depends on the Data Bytes argument received the first time. Only the last Data Bytes field is modified by CμP, when the number of bytes does not correspond with the remaining bytes that will be sent.

SANE Project Developer	↔	Prototype
<SOH><04><07><0000><0004><CKS>	→	
	←	<ACK>
<ACK>	→	
	←	<SOH><07><0000><00 01 02 03 04 05 06 07><CKS>
<NACK>[1]	→	
	←	<SOH><07><0000><00 01 02 03 04 05 06 07><CKS>
<ACK>	→	
	←	<SOH><07><0002><08 09 0A 0B 0C 0D 0E 0F><CKS>
<ACK>	→	
	←	<SOH><03><0004><10 11 12 13><CKS>
<ACK>	→	
	←	<EOT>
<ACK>	→	

[1] Framing error on any byte.

Table D.6: Example of data flow for Read Memory from prototype.

D.5 Conclusions

The software tool SANE Project Developer (SPD) has been developed as an integrated development environment for creation and edition of projects that will be downloaded to prototype. The SPD permits the creation and edition in a friendly way of all relevant files for the generation of SANE-ASM based applications. It includes tools for edition of components and cells belonging to a SANE-ASM. The SPD includes an adaptation of the ICSharpCode.TextEditor, which is a text editor control with syntax-highlighting and other functionalities, which permits the appropriate edition of files related with an application project.

The SPD provide tools to compile individual assembler files, or for building a entire SANE-ASM project, which is closely tied with a SANE Assembler (SASM) file. In case of errors (or warnings) in the execution of any of these functions, the software provides the appropriate information for helping the user to solve theses issues.

The SPD provides all necessary tool for configuring a communications system based in serial ports (RS-232), which is used for interaction with prototype. This system implements a XMODEM-based protocol that permits the user to perform basic functions over the section of memory dedicated to the execution of the SANE-ASM application. Therefore, the SPD together with the Control Microprocessor (CμP) are able to execute the following functions: Communication Test, Clear Memory, Write Memory and Read Memory. After downloading an application to prototype, the communication system permits to read the debugging frames in both cards of prototype, which shows step by step the execution of an application.

Appendix E

Listings of Example Applications

Just as iron rusts from disuse,
even so does inaction spoil the intellect.

Así como el hierro se oxida por falta de uso,
también la inactividad destruye el intelecto.

Leonardo Da Vinci (1452 – 1519)

Abstract: This chapter presents a set of listings related with application examples developed for testing the self-adaptive hardware architecture described in this document. These listings include `*.SASM` and `*.ASM` files, which represents the high-level configuration file for the configuration of an application, and the threads scheduled for processors in cells.

The listings shown in this chapter are related with two projects with example applications:

- The first is a Dynamic Fault Tolerance Scaling application, which demonstrates the correct execution of runtime self-configuration by means of subprocesses (Section 5.5).
- The second is related with the Static Fault Tolerance mechanism, which demonstrates the self-repair ability of the system by means of self-elimination and self-replications of damaged cells when a hardware failure is detected (Section 5.6).

Tables E.1 and E.2 show a relation of listings presented in this section. Each listing is explained by means comments in code, therefore no additional explanation will be added along this chapter.

References	Listing	Description
Application description in Section 5.5. Listings in Section E.1	E.1	SASM file with high-level configuration of application.
	E.2	ASM file with code for Monitor_1 Section.
	E.3	ASM file with code for Monitor_2 Section.
	E.4	ASM file with code for original copy of Compute Section (copy0). The code for first copy (copy1) and second copy (copy2) of Compute Section is the same.
	E.5	SHEX file generated after building project. This listing shows the instructions code implemented in section of memory of CμP dedicated to SANE-ASM configuration.
	E.6	SXM file generated after building project. For visualization purposes, the SXM file was generated with eight 32-bit word per line. However, the SPD could be configured to write between 1 to 32 words per line (default value is 16).

Table E.1: Listings for Dynamic Fault Tolerance Scaling application example.

References	Listing	Description
Application description in Section 5.6. Listings in Section E.2	E.7	SASM file with high-level configuration of application.
	E.8	ASM file with code for *Working Processor* in *Primary cell*. The code for *Redundant Processor* in *Redunadnt Cell* is the same. A modification in the sequence must be performed manually in one of this files to test the Static Fault Tolerance mechanism as shown in listing.
	E.9	ASM file with delay function for visualization of sequence in leds.
	E.10	SHEX file generated after building project. This listing shows the instructions code implemented in section of memory of CμP dedicated to SANE-ASM configuration.
	E.11	SXM file generated after building project. For visualization purposes, the SXM file was generated with eight 32-bit word per line. However, the SPD could be configured to write between 1 to 32 words per line (default value is 16).

Table E.2: Listings for Static Fault Tolerance application example.

E.1 Listings for Dynamic Fault Tolerance Scaling Application Example

Listing E.1: SASM file for configuration of Dynamic Fault Tolerance Scaling application.

```
;--------------------------------------------------------------------------------
;-- SANE assembly file template for Project DFT_APP_TESIS
;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
;-- Engineer: JSV
;--
;-- Create Date:  22/02/2013 17:12:19
;-- Project Name: DFT_APP_TESIS
;-- Filename:     ConfFile_1.3.sasm
;-- Description: SASM file for Dynamic Fault Tolerace example application.
;--
;-- Revision 0.01 - File created.
;-- Additional Comments:
;--
;--------------------------------------------------------------------------------
;*********************************
;Final configuration after execution
;*********************************
;CHIP0 |-----| |-----|  CHIP1 |-----| |-----|
;      | C00 | | C10 |        | C00 | | C10 |
;      |-----| |-----|        |-----| |-----|
;
;      |-----| |-----|        |-----| |-----|
;      | C01 | | C11 |        | C01 | | C11 |
;      |-----| |-----|        |-----| |-----|

;c00_CHIP0: address=0x00CC00C0, MODE=0x09, PORTS=0x44, FTCSR=0x00.
;           Pseudo-random number generator (PRNG) using a linear
;           feedback shift register (LFSR). Compute Section  (PRNG_0).
;c01_CHIP0: address=0x11AA00A1, MODE=0x09, PORTS=0x44, FTCSR=0x00.
;           Monitor_1: monitor.
;c10_CHIP0: address=0x11AA00A2, MODE=0x04, PORTS=0x00, FTCSR=0x00.
;           Monitor_2: comparator.
;c11_CHIP0: address=0x11CC00C1, MODE=0x09, PORTS=0x44, FTCSR=0x00.
;           First copy of Compute section (PRNG_1).
;           This component (1 cell) is dinamically created and deleted.
;c00_CHIP1: address=0x22CC00C2, MODE=0x09, PORTS=0x44, FTCSR=0x00.
;           Second copy of Compute section (PRNG_2).
;           This component (1 cell) is dinamically created and deleted.
;others   : empty.

;*********************************
;Difinitions
;*********************************
comp_compute          equ   0x00CC
comp_compute_copy_1   equ   0x11CC
comp_compute_copy_2   equ   0x22CC
comp_monitor          equ   0x11AA
cell_lfsr             equ   0x00CC00C0
cell_lfsr_copy_1      equ   0x11CC00C1
cell_lfsr_copy_2      equ   0x22CC00C2
cell_monitor_1        equ   0x11AA00A1
cell_monitor_2        equ   0x11AA00A2
;*********************************
;subprocess 0 for Monitor component
;*********************************
start_subprocess_0  comp_monitor
  create_component  comp_compute_copy_1 ;Create Compute section, copy 1
  write_FU_memory   cell_lfsr_copy_1    ;Write FU memories for cell
  connect_component                     ;connect component
end_subprocess_0
;*********************************
;subprocess 1 for Monitor component
;*********************************
```

```
start_subprocess_1   comp_monitor
  create_component   comp_compute_copy_2 ;Create Compute section, copy 2
  write_FU_memory    cell_lfsr_copy_2    ;Write FU memories for cell
  connect_component                      ;connect component
end_subprocess_1
;*********************************
;subprocess 2 for Monitor component
;*********************************
start_subprocess_2   comp_monitor
  delete_component   comp_compute_copy_1 ;delete copy 1 of Compute section
end_subprocess_2
;*********************************
;subprocess 3 for Monitor component
;*********************************
start_subprocess_3   comp_monitor
  delete_component   comp_compute_copy_2 ;delete copy 2 of Compute section
end_subprocess_3

;************************
;* Initial Configurtion *
;************************
create_component    comp_monitor     ;Create Monitor component
restart_and_disable_processors
write_FU_memory     cell_monitor_1   ;write FU program memories for Monitor_1
write_FU_memory     cell_monitor_2   ;write FU program memories for Monitor_2
create_component    comp_compute     ;Create Compute component
write_FU_memory     cell_lfsr        ;Write FU program memories for cell
connect_component                    ;Connect component
enable_processors_wait               ;Enable processors and wait for
                                     ;an event to start a subprocess.
end                                  ;End of SASM configuration file
```

Listing E.2: ASM code for Monitor_1 section.

```
;-------------------------------------------------------------------------
;-- Assembler template for Processor Core 0 in Mode 9
;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
;-- Engineer: Javier Soto
;--
;-- Date Created: 22/02/2013 17:12:03
;-- Project Name: DFT_APP_TESIS
;-- Filename:     Monitor_1.asm
;-- Description: One 16-bit processor (P0) with 16x16 data memory and 256
;--              instructions. Monitor_1 section, measure the power
;--              consumpsion of PRNGs, and start the creation/killing of
;--              copies of compute sections.
;-- Revision 0.01 - File created.
;-- Additional Comments: Cell address: 0x11AA00A1. MODE = 0x09
;-------------------------------------------------------------------------
#include  <pMode9Core0.inc>  ;Definition of Configuration and Status
                             ;Registers including inputs and outputs.
;16-bit General purpose Registers
;Definitions for meassuring the compute section.
new_data       equ   0x0
old_data       equ   0x1
threshold      equ   0x2
average        equ   0x3
m1_status      equ   0x4
cont_changes   equ   0x5
comparison     equ   0x6
;FIFO for average of last eigth values
FIFO_0         equ   0x8
FIFO_1         equ   0x9
FIFO_2         equ   0xA
FIFO_3         equ   0xB
FIFO_4         equ   0xC
FIFO_5         equ   0xD
FIFO_6         equ   0xE
FIFO_7         equ   0xF
;************************************************************
```

```
37 ;* Description of register with status of MONITOR_1: m1_status
38 ;************************************************************
39 ;m1_status<0> : 1 -> first copy active,  0-> no active
40 ;m1_status<1> : 1 -> second copy active, 0-> no active
41 ;m1_status<2> : 1 -> creation first copy in progress,  0-> no started
42 ;m1_status<3> : 1 -> creation second copy in progress, 0-> no started
43 ;m1_status<4> : 1 -> killing first copy in progress,   0-> no started
44 ;m1_status<5> : 1 -> killing seconf copy in progress,  0-> no started
45 ;***********************************
46 ;Description of register: comparison
47 ;***********************************
48 ;comparison<0> : 1 -> PRNG_copy_0_original == PRNG_copy1
49 ;                0 -> PRNG_copy_0_original /= PRNG_copy1
50 ;comparison<1> : 1 -> PRNG_copy_1 == PRNG_copy2
51 ;                0 -> PRNG_copy_1 /= PRNG_copy2
52 ;***********************************
53 ;Directives
54 ;***********************************
55 MODE_CORE    0x9,0  ;Directive for MODE and CORE
56 ORG          0x0    ;Origin of first instruction
57 ;***********************************
58 ;Start of program - initialization
59 ;***********************************
60 start
61   MOVLF  0x05,old_data,1    ;Seed for LFSR 0x05F3
62   MOVLF  0xF3,old_data,0
63   MOVLF  0x00,FIFO_0,1      ;Set FIFO registers to 0x000F
64   MOVLF  0x0F,FIFO_0,0      ;initial value for high consumption regime,
65   MOVW   FIFO_0,FIFO_1      ;there are not COMPUTE sections copies
66   MOVW   FIFO_0,FIFO_2
67   MOVW   FIFO_0,FIFO_3
68   MOVW   FIFO_0,FIFO_4
69   MOVW   FIFO_0,FIFO_5
70   MOVW   FIFO_0,FIFO_6
71   MOVW   FIFO_0,FIFO_7
72   CLRF   threshold             ;Set threshold to 0x0000
73   CLRF   m1_status             ;Set m1_status to 0x0000
74 cycle
75   MOVW   old_data,OUT_01    ;Move old_data to OUT_01 Port two times
76   MOVW   old_data,OUT_01    ;for metaestability issues with two chips.
77   BLMOV  IN_01,new_data     ;Wait pseudo-random data from COMPUTE section
78   BLMOV  IN_23,comparison   ;Wait comparison value from MONITOR_2 section
79   GOTO   calculate_average  ;Calculate average
80 ret_calc_average
81   MOVW   new_data,old_data  ;old_data <= new_data
82   MOVW   average,OUT_23     ;Move average to OUT_23 Port - average in leds
83 review_progress
84   BRCLR  m1_status,2,next1  ;next1 if creation copy_1 is not in progess
85   BRCLR  SUBPCSR,3,nextLast ;nextLast if ends creation of copy_1
86                             ;(nextlast if ends execution of subprocess0)
87   BCLR   m1_status,2        ;clear flag creation_first_copy_in_progess
88   BCLR   SUBPCSR,3          ;clear subprocess0 ends flag
89   BSET   m1_status,0        ;set flag copy_1 of COMPUTE section active
90   GOTO   nextLast
91 next1
92   BRCLR  m1_status,3,nextLast ;nextLast if create copy_2 is not in progess
93   BRCLR  SUBPCSR,4,nextLast   ;nextLast if ends creation of copy_2
94                               ;(nextlast if ends execution of subprocess1)
95   BCLR   m1_status,3          ;clear flag creation_second_copy_in_progess
96   BCLR   SUBPCSR,4            ;clear subprocess1 ends flag
97   BSET   m1_status,1          ;set flag copy_2 of COMPUTE section active
98   GOTO   nextLast
99 next2
100  BRCLR  m1_status,4,next3    ;next3 if killing copy_1 is not in progess
101  BRCLR  SUBPCSR,5,nextLast   ;nextLast if ends killing of copy_1
102                              ;(nextlast if ends execution of subprocess2)
103  BCLR   m1_status,4          ;clear flag kill_first_copy_in_progess
104  BCLR   SUBPCSR,5            ;clear subprocess2 ends flag
105  GOTO   nextLast
106 next3
```

```
107   BRCLR  m1_status,5,nextLast ;nextLast if killing copy_2 is not in progess
108   BRCLR  SUBPCSR,6,nextLast   ;nextLast if ends killing of copy_2
109                               ;(nextlast if ends execution of subprocess3)
110   BCLR   m1_status,5          ;clear flag kill_second_copy_in_progess
111   BCLR   SUBPCSR,6            ;clear subprocess3 ends flag
112 nextLast
113   BRSET  m1_status,1,second_copy_active  ;process when copy_2 is active
114   BRSET  m1_status,0,first_copy_active   ;process when copy_1 is active
115 none_copy_active                         ;process when none copy is active
116   MOVLF  0x0B,threshold,0         ;for high consumption regime
117   CBGE   average,threshold,cycle  ;goto cycle if average >= 11
118   MOVLF  0x01,threshold,0         ;for low consumption regime (copy_1)
119   CBGE   average,threshold,pre_create_first_copy ;jump if average >= 1
120   MOVLF  0xF1,OUT_23,1            ;set error code 1 (leds)
121   END                             ;End with error
122 pre_create_first_copy
123   BRCLR  SUBPCSR,7,cycle   ;goto cycle if system is not in wait state
124 create_first_copy
125   BSET   m1_status,2       ;set flag creation_first_copy_in_progress
126   BCLR   SUBPCSR,2         ;configure creation of first copy
127   BCLR   SUBPCSR,1         ;subprocess0, SUBPCSR<2:1> <= 00
128   BSET   SUBPCSR,0         ;start creation of first copy
129   BCLR   SUBPCSR,0         ;(start execution of subprocess_0)
130   GOTO   cycle
131
132   org   0x40 ;Optional, for locating code in PM1
133 ;***********************************
134 ;Process when first copy is active
135 ;***********************************
136 first_copy_active
137   BRSET  comparison,0,$+3   ;jump 3 positions if copy_original == copy_1
138   MOVLF  0xF2,OUT_23,1      ;set error code 2 (leds)
139   END                       ;End with error
140   MOVLF  0x0B,threshold,0         ;for high consumption regime
141   CBGE   average,threshold,pre_delete_first_copy  ;jump if average >= 11
142   MOVLF  0x06,threshold,0         ;for medium consumption regime
143   CBGE   average,threshold,cycle  ;jump if average >= 6
144   MOVLF  0x01,threshold,0         ;for low consumption regime
145   CBGE   average,threshold,pre_create_second_copy ;jump if average >= 1
146   MOVLF  0xF3,OUT_23,1            ;set error code 3 (leds)
147   END                             ;End with error
148 pre_delete_first_copy
149   BRCLR  SUBPCSR,7,cycle  ;goto cycle if system is not in wait state
150 delete_first_copy
151   BCLR   m1_status,0      ;clear flag first_copy_active
152   BSET   m1_status,4      ;set flag kill_first_copy_in_progress
153   BSET   SUBPCSR,2        ;configure subprocess for killing first copy
154   BCLR   SUBPCSR,1        ;subprocess2, SUBPCSR<2:1> <= 10
155   BSET   SUBPCSR,0        ;start killing of copy_1
156   BCLR   SUBPCSR,0        ;(start execution of subprocess_2)
157   GOTO   cycle
158 pre_create_second_copy
159   BRCLR  SUBPCSR,7,cycle  ;goto cycle if system is not in wait state
160 create_second_copy
161   BSET   m1_status,3      ;set flag creation_second_copy_in_progress
162   BCLR   SUBPCSR,2        ;configure subprocess for create second copy
163   BSET   SUBPCSR,1        ;subprocess_1, SUBPCSR<2:1> <= 01
164   BSET   SUBPCSR,0        ;start creation of copy_2
165   BCLR   SUBPCSR,0        ;(start execution of subprocess_1)
166   GOTO   cycle
167 ;***********************************
168 ;Process when second copy is active
169 ;***********************************
170 second_copy_active
171   BRSET  comparison,1,$+3   ;jump 3 positions if copy_1 == copy_2
172   MOVLF  0xF4,OUT_23,1      ;set error code 4 (leds)
173   END                       ;End with error
174   MOVLF  0x06,threshold,0         ;for medium consumption regime
175   CBGE   average,threshold,pre_delete_second_copy ;jump if average >= 6
176   MOVLF  0x01,threshold,0         ;for low consumption regime
```

```
177   CBGE    average,threshold,cycle   ;jump if average >= 1
178   MOVLF   0xF5,OUT_23,1             ;set error code 5 (leds)
179   END                               ;End with error
180 pre_delete_second_copy
181   BRCLR   SUBPCSR,7,cycle   ;goto cycle if system is not in wait state
182 delete_second_copy
183   BCLR    m1_status,1       ;clear flag second_copy_active
184   BSET    m1_status,5       ;set flag kill_second_copy_in_progress
185   BSET    SUBPCSR,2         ;configure subprocess for killing second copy
186   BSET    SUBPCSR,1         ;subprocess_3, SUBPCSR<2:1> <= 11
187   BSET    SUBPCSR,0         ;start killing of copy_2
188   BCLR    SUBPCSR,0         ;(start execution of subprocess_3)
189   GOTO    cycle
190
191 org 0x80
192 ;*********************************
193 ;Calculate average of changes
194 ;*********************************
195 calculate_average
196   CLRF    cont_changes                ;counter of changes between old_data
197 bit0                                  ;and new_data
198   XORWY   new_data,old_data,old_data  ;commparison of old_data and new_data
199   BRCLR   old_data,0,$+2              ;Increment counter when a transition
200   INCW    cont_changes,cont_changes   ;ocurrs
201   BRCLR   old_data,.1,$+2             ;Same for all bits
202   INCW    cont_changes,cont_changes
203   BRCLR   old_data,.2,$+2
204   INCW    cont_changes,cont_changes
205   BRCLR   old_data,.3,$+2
206   INCW    cont_changes,cont_changes
207   BRCLR   old_data,.4,$+2
208   INCW    cont_changes,cont_changes
209   BRCLR   old_data,.5,$+2
210   INCW    cont_changes,cont_changes
211   BRCLR   old_data,.6,$+2
212   INCW    cont_changes,cont_changes
213   BRCLR   old_data,.7,$+2
214   INCW    cont_changes,cont_changes
215   BRCLR   old_data,.8,$+2
216   INCW    cont_changes,cont_changes
217   BRCLR   old_data,.9,$+2
218   INCW    cont_changes,cont_changes
219   BRCLR   old_data,.10,$+2
220   INCW    cont_changes,cont_changes
221   BRCLR   old_data,.11,$+2
222   INCW    cont_changes,cont_changes
223   BRCLR   old_data,.12,$+2
224   INCW    cont_changes,cont_changes
225   BRCLR   old_data,.13,$+2
226   INCW    cont_changes,cont_changes
227   BRCLR   old_data,.14,$+2
228   INCW    cont_changes,cont_changes
229   BRCLR   old_data,.15,$+2
230   INCW    cont_changes,cont_changes
231   MOVW    FIFO_6,FIFO_7               ;Shift FIFO registers one position
232   MOVW    FIFO_5,FIFO_6
233   MOVW    FIFO_4,FIFO_5
234   MOVW    FIFO_3,FIFO_4
235   MOVW    FIFO_2,FIFO_3
236   MOVW    FIFO_1,FIFO_2
237   MOVW    FIFO_0,FIFO_1
238   MOVW    cont_changes,FIFO_0         ;Set new value of changes
239   MOVW    FIFO_0,average              ;to FIFO_0 register and
240   ADDWY   FIFO_1,average,average      ;calculate average
241   ADDWY   FIFO_2,average,average
242   ADDWY   FIFO_3,average,average
243   ADDWY   FIFO_4,average,average
244   ADDWY   FIFO_5,average,average
245   ADDWY   FIFO_6,average,average
246   ADDWY   FIFO_7,average,average
```

```
247   LSR     average,average
248   LSR     average,average
249   LSR     average,average
250   GOTO    ret_calc_average
```

Listing E.3: ASM code for Monitor_2 section.

```
1  ;-------------------------------------------------------------------------
2  ;-- Assembler template for Processor Core 0 in Mode 4
3  ;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
4  ;-- Engineer: JSV
5  ;--
6  ;-- Date Created:  22/02/2013 17:12:47
7  ;-- Project Name:  DFT_APP_TESIS
8  ;-- Filename:      Monitor_2.asm
9  ;-- Description: Monitor_2 section: comparison of PRNGs
10 ;--
11 ;-- Revision 0.01 - File created.
12 ;-- Additional Comments: address: 11AA00A2
13 ;--
14 ;-------------------------------------------------------------------------
15 #include        <pMode4Core0.inc>
16
17 ;8-Bit General Purpose Registers
18 data0_L      equ    0x0  ;LSB read from original COMPUTE section (copy0)
19 data0_H      equ    0x1  ;MSB read from original COMPUTE section (copy0)
20 data1_L      equ    0x2  ;LSB read from COMPUTE section (copy1)
21 data1_H      equ    0x3  ;MSB read from COMPUTE section (copy1)
22 data2_L      equ    0x4  ;LSB read from COMPUTE section (copy2)
23 data2_H      equ    0x5  ;MSB read from COMPUTE section (copy2)
24 comparison   equ    0x6
25 ;comparison<0> : 1 -> PRNG_copy_0_original == PRNG_copy1
26 ;                0 -> PRNG_copy_0_original /= PRNG_copy1
27 ;comparison<1> : 1 -> PRNG_copy_1 == PRNG_copy2
28 ;                0 -> PRNG_copy_1 /= PRNG_copy2
29 ;*************************************
30 ;*Directives
31 ;*************************************
32   MODE_CORE   0x4,0
33   ORG         0x0
34 ;*************************************
35 ;*Start of program
36 ;*************************************
37 start
38   BLMOV   IN0,data0_L  ;Wait for a new pseudo-random number from
39   NOP                  ;COMPUTE section (original or copy_0).
40   NOP                  ;Read low byte first.
41   NOP                  ;Delay due metaestability between chips
42   NOP
43   NOP
44   MOVW    IN1,data1_L  ;Read low byte from COMPUTE section(copy_1)
45   MOVW    IN2,data2_L  ;Read low byte from COMPUTE section(copy_2)
46   BLMOV   IN0,data0_H  ;Read high byte from COMPUTE section
47   NOP                  ;Delay because metaestability between chips
48   NOP
49   NOP
50   NOP
51   NOP
52   MOVW    IN1,data1_H  ;Read high byte from COMPUTE section(copy_1)
53   MOVW    IN2,data2_H  ;Read high byte from COMPUTE section(copy_2)
54   CLRF    comparison   ;Clear comparison register
55 compare_1_2
56   CBNE    data0_L,data1_L,write_port  ;Jump if copy_0 /= copy_1 (low bytes)
57   CBNE    data0_H,data1_H,write_port  ;Jump if copy_0 /= copy_1 (high bytes)
58   BSET    comparison,0                ;Set flag copy_0 == copy_1
59 compare_2_3
60   CBNE    data1_L,data2_L,write_port  ;Jump if copy_1 /= copy_2 (low bytes)
61   CBNE    data1_H,data2_H,write_port  ;Jump if copy_1 /= copy_2 (high bytes)
62   BSET    comparison,1                ;Set flag copy_1 == copy_2
63 write_port
```

```
64   MOVW    comparison,OUT0             ;Write comparison register to port OT0
65   GOTO    start
```

Listing E.4: ASM code for Compute sections (original, first and second copy).

```
1  ;-------------------------------------------------------------------------
2  ;-- Assembler template for Processor Core 0 in Mode 9
3  ;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
4  ;-- Engineer: JSV
5  ;--
6  ;-- Date Created:  22/02/2013 17:12:55
7  ;-- Project Name:  DFT_APP_TESIS
8  ;-- Filename:      PRNG0_OriginalCompute.asm
9  ;-- Description: Pseudo-random number generator, 16-bits,
10 ;--               output port OUT0_OUT01 (original compute section)
11 ;--
12 ;-- Revision 0.01 - File created.
13 ;-- Additional Comments: Mode 9, address: 00CC00C0, 16-bit processor
14 ;--                      with 16x16 bits in memory data, 256 instructions
15 ;--                      (Linear feedback shift register) PRNG_0 generator
16 ;-------------------------------------------------------------------------
17 #include  <pMode9Core0.inc>
18 ;*****************************************************************/
19 ;Polynomial for LFSR 16-bit: x11 + x13 + x14 + x16 + 1
20 ;
21 ;  -> x1 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 ->
22 ;  |                                    |       |   |       |
23 ;  |                                    |       |  XOR<-----
24 ;  |                                    |       |   |
25 ;  |                                    |      XOR<-
26 ;  |                                    |       |
27 ;  |<-----------------------------------XOR<------
28
29 ;16-Bit General Purpose Registers
30 data        equ   0x0  ;pseudo-random data generted
31 seed        equ   0x1  ;seed for calculate data
32 cont_taps   equ   0x2  ;counter of taps
33 none        equ   0x3F ;empty address
34 ;*****************************************
35 ;* Directives
36 ;*****************************************
37   MODE_CORE   0x9,0
38   ORG         0x0
39 ;*****************************************
40 ;* Start of program
41 ;*****************************************
42 start
43   BLMOV   IN_01,seed    ;read 16-bit seed from Monitor_1
44   NOP
45   MOVW    IN_01,seed    ;read again for LFSR_2 (metaestability)
46   GOTO    next_data     ;calculate next data
47 ret_next_data
48   MOVW    seed,OUT_23   ;move seed to output ports OUT2 and OUT3 (leds)
49   MOVW    data,OUT_01   ;move 16-bit pseudo-random data to Monitor_1
50   NOP                   ;Monitor_2 read low byte.
51   NOP                   ;Delay implemented to Monitor_2, which needs time
52   NOP                   ;to read data from Compute sections:
53   NOP                   ;PRNG_0, PRNG_1 (if exist) and PRNG_2 (if_exist)
54   NOP
55   NOP
56   NOP
57   NOP
58   NOP
59   NOP
60   NOP
61   NOP
62   NOP
63   NOP
64   NOP
65   NOP
```

```
66    NOP
67    NOP
68    SWAPW   data,OUT_01   ;SWAP and write data to port OUT0-OUT1,
69    GOTO    start         ;Monitor_2 read high byte.
70  ;*****************************************
71  ;* Calculation of pseudo-random data
72  ;*****************************************
73    org 0x40  ;used for allocate code in Program memory 1 (optional).
74  next_data
75    CLRF    cont_taps             ;clear taps counter
76    BRCLR   seed,0,jump1          ;evaluate x16
77    INCW    cont_taps,cont_taps   ;increment taps counter
78  jump1
79    BRCLR   seed,2,jump2          ;evaluate x14
80    INCW    cont_taps,cont_taps   ;increment taps counter
81  jump2
82    BRCLR   seed,3,jump3          ;evaluate x13
83    INCW    cont_taps,cont_taps   ;increment taps counter
84  jump3
85    BRCLR   seed,5,jump4          ;evaluate x11
86    INCW    cont_taps,cont_taps   ;increment taps counter
87  jump4
88    SEC                           ;Set carry
89    BRSET   cont_taps,0,$+2       ;Jump 2 positions if cont_taps is odd (XOR=1)
90    CLC                           ;Clear carry
91    RRW     seed,data             ;Rotate right through carry for
92    GOTO    ret_next_data         ;generation of new pseudo-random data.
```

Listing E.5: SHEX file generated by SANE Project developer after execute Build Project option.

```
1  unsigned int memoria[] =
2  {
3  start_subprocess0_IC,  //0x10
4  0x11AA,                //id_comp
5  create_component_IC,   //0x00
6  0x11CC,                //id_comp
7  0,                     //Zero-based num cells in component
8  1,                     //num of outputs
9  0x11CC00C1,            //address
10 0x00,
11 0x11AA00A1,            //Input 0
12 0x09,
13 0x11AA00A1,            //Input 1
14 0x00,
15 0x00000000,            //Input 2
16 0x00,
17 0x00000000,            //Input 3
18 0x00,
19 0x00000000,            //FT Input 0
20 0x00,
21 0x00000000,            //FT Input 1
22 0x00,
23 0x00000000,            //FT Input 2
24 0x00,
25 0x00000000,            //FT Input 3
26 0x01,
27 0x11AA00A2,            //Output 0
28 write_FU_memory_CR_IC, //0x03
29 0x11CC00C1,            //address
30 4,                     //register number
31 0x09,                  //MODE
32 0x00,                  //FAMILY
33 0x44,                  //PORTS
34 0x00,                  //FTCSR
35 write_FU_memory_PM0_IC,//0x04
36 0x11CC00C1,            //address
37 26,                    //register number
38 0x0780801,             //0       BLMOV   IN_01,seed,
39 0x0B40000,             //1       NOP     ,,
40 0x0740801,             //2       MOVW    IN_01,seed,
```

```
0x0E40000,              //3       GOTO    next_data,,
0x0740065,              //4       MOVW    seed,OUT_23,
0x0740024,              //5       MOVW    data,OUT_01,
0x0B40000,              //6       NOP     ,,
0x0B40000,              //7       NOP     ,,
0x0B40000,              //8       NOP     ,,
0x0B40000,              //9       NOP     ,,
0x0B40000,              //10      NOP     ,,
0x0B40000,              //11      NOP     ,,
0x0B40000,              //12      NOP     ,,
0x0B40000,              //13      NOP     ,,
0x0B40000,              //14      NOP     ,,
0x0B40000,              //15      NOP     ,,
0x0B40000,              //16      NOP     ,,
0x0B40000,              //17      NOP     ,,
0x0B40000,              //18      NOP     ,,
0x0B40000,              //19      NOP     ,,
0x0B40000,              //20      NOP     ,,
0x0B40000,              //21      NOP     ,,
0x0B40000,              //22      NOP     ,,
0x0B40000,              //23      NOP     ,,
0x08C0024,              //24      SWAPW   data,OUT_01,
0x0E00000,              //25      GOTO    start,,
write_FU_memory_PM1_IC,//0x05
0x11CC00C1,             //address
14,                     //register number
0x0A40002,              //64      CLRF    cont_taps,,
0x0C43040,              //65      BRCLR   seed,0,jump1
0x0840082,              //66      INCW    cont_taps,cont_taps,
0x0C45042,              //67      BRCLR   seed,2,jump2
0x0840082,              //68      INCW    cont_taps,cont_taps,
0x0C47043,              //69      BRCLR   seed,3,jump3
0x0840082,              //70      INCW    cont_taps,cont_taps,
0x0C49045,              //71      BRCLR   seed,5,jump4
0x0840082,              //72      INCW    cont_taps,cont_taps,
0x0AC0000,              //73      SEC     ,,
0x0D4C080,              //74      BRSET   cont_taps,0,$+2
0x0A80000,              //75      CLC     ,,
0x0940040,              //76      RRW     seed,data,
0x0E04000,              //77      GOTO    ret_next_data,,
connect_component_IC,   //0x01
end_subprocess0_IC,     //0x11
start_subprocess1_IC,   //0x12
0x11AA,                 //id_comp
create_component_IC,    //0x00
0x22CC,                 //id_comp
0,                      //Zero-based num cells in component
1,                      //num of outputs
0x22CC00C2,             //address
0x00,
0x11AA00A1,             //Input 0
0x09,
0x11AA00A1,             //Input 1
0x00,
0x00000000,             //Input 2
0x00,
0x00000000,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x02,
0x11AA00A2,             //Output 0
write_FU_memory_CR_IC, //0x03
0x22CC00C2,             //address
4,                      //register number
```

```
0x09,                   //MODE
0x00,                   //FAMILY
0x44,                   //PORTS
0x00,                   //FTCSR
write_FU_memory_PM0_IC,//0x04
0x22CC00C2,             //address
26,                     //register number
0x0780801,              //0       BLMOV   IN_01,seed,
0x0B40000,              //1       NOP     ,,
0x0740801,              //2       MOVW    IN_01,seed,
0x0E40000,              //3       GOTO    next_data,,
0x0740065,              //4       MOVW    seed,OUT_23,
0x0740024,              //5       MOVW    data,OUT_01,
0x0B40000,              //6       NOP     ,,
0x0B40000,              //7       NOP     ,,
0x0B40000,              //8       NOP     ,,
0x0B40000,              //9       NOP     ,,
0x0B40000,              //10      NOP     ,,
0x0B40000,              //11      NOP     ,,
0x0B40000,              //12      NOP     ,,
0x0B40000,              //13      NOP     ,,
0x0B40000,              //14      NOP     ,,
0x0B40000,              //15      NOP     ,,
0x0B40000,              //16      NOP     ,,
0x0B40000,              //17      NOP     ,,
0x0B40000,              //18      NOP     ,,
0x0B40000,              //19      NOP     ,,
0x0B40000,              //20      NOP     ,,
0x0B40000,              //21      NOP     ,,
0x0B40000,              //22      NOP     ,,
0x0B40000,              //23      NOP     ,,
0x08C0024,              //24      SWAPW   data,OUT_01,
0x0E00000,              //25      GOTO    start,,
write_FU_memory_PM1_IC,//0x05
0x22CC00C2,             //address
14,                     //register number
0x0A40002,              //64      CLRF    cont_taps,,
0x0C43040,              //65      BRCLR   seed,0,jump1
0x0840082,              //66      INCW    cont_taps,cont_taps,
0x0C45042,              //67      BRCLR   seed,2,jump2
0x0840082,              //68      INCW    cont_taps,cont_taps,
0x0C47043,              //69      BRCLR   seed,3,jump3
0x0840082,              //70      INCW    cont_taps,cont_taps,
0x0C49045,              //71      BRCLR   seed,5,jump4
0x0840082,              //72      INCW    cont_taps,cont_taps,
0x0AC0000,              //73      SEC     ,,
0x0D4C080,              //74      BRSET   cont_taps,0,$+2
0x0A80000,              //75      CLC     ,,
0x0940040,              //76      RRW     seed,data,
0x0E04000,              //77      GOTO    ret_next_data,,
connect_component_IC,   //0x01
end_subprocess1_IC,     //0x13
start_subprocess2_IC,   //0x14
0x11AA,                 //id_comp
delete_component_IC,    //0x02
0x11CC,                 //id_comp
end_subprocess2_IC,     //0x15
start_subprocess3_IC,   //0x16
0x11AA,                 //id_comp
delete_component_IC,    //0x02
0x22CC,                 //id_comp
end_subprocess3_IC,     //0x17
create_component_IC,    //0x00
0x11AA,                 //id_comp
1,                      //Zero-based num cells in component
6,                      //num of outputs
0x11AA00A1,             //address
0x00,
0x00CC00C0,             //Input 0
0x09,
```

```
0x00CC00C0,             //Input 1
0x02,
0x11AA00A2,             //Input 2
0x03,
0x11AA00A2,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x00,
0x00CC00C0,             //Output 0
0x09,
0x00CC00C0,             //Output 1
0x00,
0x11CC00C1,             //Output 2
0x09,
0x11CC00C1,             //Output 3
0x00,
0x22CC00C2,             //Output 4
0x09,
0x22CC00C2,             //Output 5
2,                      //num of outputs
0x11AA00A2,             //address
0x00,
0x00CC00C0,             //Input 0
0x01,
0x11CC00C1,             //Input 1
0x02,
0x22CC00C2,             //Input 2
0x00,
0x00000000,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x02,
0x11AA00A1,             //Output 0
0x03,
0x11AA00A1,             //Output 1
restart_and_disable_processors_IC,//0x0A
write_FU_memory_CR_IC, //0x03
0x11AA00A1,             //address
4,                      //register number
0x09,                   //MODE
0x00,                   //FAMILY
0x44,                   //PORTS
0x00,                   //FTCSR
write_FU_memory_PM0_IC,//0x04
0x11AA00A1,             //address
56,                     //register number
0x0505041,              //0       MOVLF   0x05,old_data,1
0x05F3001,              //1       MOVLF   0xF3,old_data,0
0x0500048,              //2       MOVLF   0x00,FIFO_0,1
0x050F008,              //3       MOVLF   0x0F,FIFO_0,0
0x0740209,              //4       MOVW    FIFO_0,FIFO_1,
0x074020A,              //5       MOVW    FIFO_0,FIFO_2,
0x074020B,              //6       MOVW    FIFO_0,FIFO_3,
0x074020C,              //7       MOVW    FIFO_0,FIFO_4,
0x074020D,              //8       MOVW    FIFO_0,FIFO_5,
0x074020E,              //9       MOVW    FIFO_0,FIFO_6,
0x074020F,              //10      MOVW    FIFO_0,FIFO_7,
0x0A40002,              //11      CLRF    threshold,,
```

```
0x0A40004,              //12     CLRF    m1_status,,
0x0740064,              //13     MOVW    old_data,OUT_01,
0x0740064,              //14     MOVW    old_data,OUT_01,
0x0780800,              //15     BLMOV   IN_01,new_data,
0x0780846,              //16     BLMOV   IN_23,comparison,
0x0E80000,              //17     GOTO    calculate_average,,
0x0740001,              //18     MOVW    new_data,old_data,
0x07400E5,              //19     MOVW    average,OUT_23,
0x0C1A102,              //20     BRCLR   m1_status,2,next1
0x0C29B03,              //21     BRCLR   SUBPCSR,3,nextLast
0x0B82104,              //22     BCLR    m1_status,2,
0x0B83B2C,              //23     BCLR    SUBPCSR,3,
0x0BC0104,              //24     BSET    m1_status,0,
0x0E29000,              //25     GOTO    nextLast,,
0x0C29103,              //26     BRCLR   m1_status,3,nextLast
0x0C29B04,              //27     BRCLR   SUBPCSR,4,nextLast
0x0B83104,              //28     BCLR    m1_status,3,
0x0B84B2C,              //29     BCLR    SUBPCSR,4,
0x0BC1104,              //30     BSET    m1_status,1,
0x0E29000,              //31     GOTO    nextLast,,
0x0C25104,              //32     BRCLR   m1_status,4,next3
0x0C29B05,              //33     BRCLR   SUBPCSR,5,nextLast
0x0B84104,              //34     BCLR    m1_status,4,
0x0B85B2C,              //35     BCLR    SUBPCSR,5,
0x0E29000,              //36     GOTO    nextLast,,
0x0C29105,              //37     BRCLR   m1_status,5,nextLast
0x0C29B06,              //38     BRCLR   SUBPCSR,6,nextLast
0x0B85104,              //39     BCLR    m1_status,5,
0x0B86B2C,              //40     BCLR    SUBPCSR,6,
0x0D5A101,              //41     BRSET   m1_status,1,second_copy_active
0x0D40100,              //42     BRSET   m1_status,0,first_copy_active
0x050B002,              //43     MOVLF   0x0B,threshold,0
0x140D0C2,              //44     CBGE    average,threshold,cycle
0x0501002,              //45     MOVLF   0x01,threshold,0
0x14310C2,              //46     CBGE    average,threshold,pre_create_first_copy
0x05F1065,              //47     MOVLF   0xF1,OUT_23,1
0x0B00000,              //48     END     ,,
0x0C0DB07,              //49     BRCLR   SUBPCSR,7,cycle
0x0BC2104,              //50     BSET    m1_status,2,
0x0B82B2C,              //51     BCLR    SUBPCSR,2,
0x0B81B2C,              //52     BCLR    SUBPCSR,1,
0x0BC0B2C,              //53     BSET    SUBPCSR,0,
0x0B80B2C,              //54     BCLR    SUBPCSR,0,
0x0E0D000,              //55     GOTO    cycle,,
write_FU_memory_PM1_IC,//0x05
0x11AA00A1,             //address
43,                     //register number
0x0D43180,              //64     BRSET   comparison,0,$+3
0x05F2065,              //65     MOVLF   0xF2,OUT_23,1
0x0B00000,              //66     END     ,,
0x050B002,              //67     MOVLF   0x0B,threshold,0
0x144B0C2,              //68     CBGE    average,threshold,pre_delete_first_copy
0x0506002,              //69     MOVLF   0x06,threshold,0
0x140D0C2,              //70     CBGE    average,threshold,cycle
0x0501002,              //71     MOVLF   0x01,threshold,0
0x14530C2,              //72     CBGE    average,threshold,pre_create_second_copy
0x05F3065,              //73     MOVLF   0xF3,OUT_23,1
0x0B00000,              //74     END     ,,
0x0C0DB07,              //75     BRCLR   SUBPCSR,7,cycle
0x0B80104,              //76     BCLR    m1_status,0,
0x0BC4104,              //77     BSET    m1_status,4,
0x0BC2B2C,              //78     BSET    SUBPCSR,2,
0x0B81B2C,              //79     BCLR    SUBPCSR,1,
0x0BC0B2C,              //80     BSET    SUBPCSR,0,
0x0B80B2C,              //81     BCLR    SUBPCSR,0,
0x0E0D000,              //82     GOTO    cycle,,
0x0C0DB07,              //83     BRCLR   SUBPCSR,7,cycle
0x0BC3104,              //84     BSET    m1_status,3,
0x0B82B2C,              //85     BCLR    SUBPCSR,2,
0x0BC1B2C,              //86     BSET    SUBPCSR,1,
```

```
0x0BC0B2C,              //87     BSET    SUBPCSR,0,
0x0B80B2C,              //88     BCLR    SUBPCSR,0,
0x0E0D000,              //89     GOTO    cycle,,
0x0D5D181,              //90     BRSET   comparison,1,$+3
0x05F4065,              //91     MOVLF   0xF4,OUT_23,1
0x0B00000,              //92     END     ,,
0x0506002,              //93     MOVLF   0x06,threshold,0
0x14630C2,              //94     CBGE    average,threshold,pre_delete_second_copy
0x0501002,              //95     MOVLF   0x01,threshold,0
0x140D0C2,              //96     CBGE    average,threshold,cycle
0x05F5065,              //97     MOVLF   0xF5,OUT_23,1
0x0B00000,              //98     END     ,,
0x0C0DB07,              //99     BRCLR   SUBPCSR,7,cycle
0x0B81104,              //100    BCLR    m1_status,1,
0x0BC5104,              //101    BSET    m1_status,5,
0x0BC2B2C,              //102    BSET    SUBPCSR,2,
0x0BC1B2C,              //103    BSET    SUBPCSR,1,
0x0BC0B2C,              //104    BSET    SUBPCSR,0,
0x0B80B2C,              //105    BCLR    SUBPCSR,0,
0x0E0D000,              //106    GOTO    cycle,,
write_FU_memory_PM2_IC,//0x06
0x11AA00A1,             //address
54,                     //register number
0x0A40005,              //128    CLRF    cont_changes,,
0x0701001,              //129    XORWY   new_data,old_data,old_data
0x0C84040,              //130    BRCLR   old_data,0,$+2
0x0840145,              //131    INCW    cont_changes,cont_changes,
0x0C86041,              //132    BRCLR   old_data,.1,$+2
0x0840145,              //133    INCW    cont_changes,cont_changes,
0x0C88042,              //134    BRCLR   old_data,.2,$+2
0x0840145,              //135    INCW    cont_changes,cont_changes,
0x0C8A043,              //136    BRCLR   old_data,.3,$+2
0x0840145,              //137    INCW    cont_changes,cont_changes,
0x0C8C044,              //138    BRCLR   old_data,.4,$+2
0x0840145,              //139    INCW    cont_changes,cont_changes,
0x0C8E045,              //140    BRCLR   old_data,.5,$+2
0x0840145,              //141    INCW    cont_changes,cont_changes,
0x0C90046,              //142    BRCLR   old_data,.6,$+2
0x0840145,              //143    INCW    cont_changes,cont_changes,
0x0C92047,              //144    BRCLR   old_data,.7,$+2
0x0840145,              //145    INCW    cont_changes,cont_changes,
0x0C94048,              //146    BRCLR   old_data,.8,$+2
0x0840145,              //147    INCW    cont_changes,cont_changes,
0x0C96049,              //148    BRCLR   old_data,.9,$+2
0x0840145,              //149    INCW    cont_changes,cont_changes,
0x0C9804A,              //150    BRCLR   old_data,.10,$+2
0x0840145,              //151    INCW    cont_changes,cont_changes,
0x0C9A04B,              //152    BRCLR   old_data,.11,$+2
0x0840145,              //153    INCW    cont_changes,cont_changes,
0x0C9C04C,              //154    BRCLR   old_data,.12,$+2
0x0840145,              //155    INCW    cont_changes,cont_changes,
0x0C9E04D,              //156    BRCLR   old_data,.13,$+2
0x0840145,              //157    INCW    cont_changes,cont_changes,
0x0CA004E,              //158    BRCLR   old_data,.14,$+2
0x0840145,              //159    INCW    cont_changes,cont_changes,
0x0CA204F,              //160    BRCLR   old_data,.15,$+2
0x0840145,              //161    INCW    cont_changes,cont_changes,
0x074038F,              //162    MOVW    FIFO_6,FIFO_7,
0x074034E,              //163    MOVW    FIFO_5,FIFO_6,
0x074030D,              //164    MOVW    FIFO_4,FIFO_5,
0x07402CC,              //165    MOVW    FIFO_3,FIFO_4,
0x074028B,              //166    MOVW    FIFO_2,FIFO_3,
0x074024A,              //167    MOVW    FIFO_1,FIFO_2,
0x0740209,              //168    MOVW    FIFO_0,FIFO_1,
0x0740148,              //169    MOVW    cont_changes,FIFO_0,
0x0740203,              //170    MOVW    FIFO_0,average,
0x0603243,              //171    ADDWY   FIFO_1,average,average
0x0603283,              //172    ADDWY   FIFO_2,average,average
0x06032C3,              //173    ADDWY   FIFO_3,average,average
0x0603303,              //174    ADDWY   FIFO_4,average,average
```

```
0x0603343,              //175    ADDWY   FIFO_5,average,average
0x0603383,              //176    ADDWY   FIFO_6,average,average
0x06033C3,              //177    ADDWY   FIFO_7,average,average
0x09C00C3,              //178    LSR     average,average,
0x09C00C3,              //179    LSR     average,average,
0x09C00C3,              //180    LSR     average,average,
0x0E12000,              //181    GOTO    ret_calc_average,,
write_FU_memory_CR_IC, //0x03
0x11AA00A2,             //address
4,                      //register number
0x04,                   //MODE
0x00,                   //FAMILY
0x00,                   //PORTS
0x00,                   //FTCSR
write_FU_memory_PM0_IC,//0x04
0x11AA00A2,             //address
25,                     //register number
0x0780800,              //0      BLMOV   IN0,data0_L,
0x0B40000,              //1      NOP     ,,
0x0B40000,              //2      NOP     ,,
0x0B40000,              //3      NOP     ,,
0x0B40000,              //4      NOP     ,,
0x0B40000,              //5      NOP     ,,
0x0740842,              //6      MOVW    IN1,data1_L,
0x0740884,              //7      MOVW    IN2,data2_L,
0x0780801,              //8      BLMOV   IN0,data0_H,
0x0B40000,              //9      NOP     ,,
0x0B40000,              //10     NOP     ,,
0x0B40000,              //11     NOP     ,,
0x0B40000,              //12     NOP     ,,
0x0B40000,              //13     NOP     ,,
0x0740843,              //14     MOVW    IN1,data1_H,
0x0740885,              //15     MOVW    IN2,data2_H,
0x0A40006,              //16     CLRF    comparison,,
0x1617002,              //17     CBNE    data0_L,data1_L,write_port
0x1617043,              //18     CBNE    data0_H,data1_H,write_port
0x0BC0186,              //19     BSET    comparison,0,
0x1617084,              //20     CBNE    data1_L,data2_L,write_port
0x16170C5,              //21     CBNE    data1_H,data2_H,write_port
0x0BC1186,              //22     BSET    comparison,1,
0x07401A4,              //23     MOVW    comparison,OUT0,
0x0E00000,              //24     GOTO    start,,
create_component_IC,    //0x00
0x00CC,                 //id_comp
0,                      //Zero-based num cells in component
3,                      //num of outputs
0x00CC00C0,             //address
0x00,
0x11AA00A1,             //Input 0
0x09,
0x11AA00A1,             //Input 1
0x00,
0x00000000,             //Input 2
0x00,
0x00000000,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x00,
0x11AA00A1,             //Output 0
0x09,
0x11AA00A1,             //Output 1
0x00,
0x11AA00A2,             //Output 2
write_FU_memory_CR_IC, //0x03
```

```
0x00CC00C0,             //address
4,                      //register number
0x09,                   //MODE
0x00,                   //FAMILY
0x44,                   //PORTS
0x00,                   //FTCSR
write_FU_memory_PM0_IC,//0x04
0x00CC00C0,             //address
26,                     //register number
0x0780801,              //0       BLMOV   IN_01,seed,
0x0B40000,              //1       NOP     ,,
0x0740801,              //2       MOVW    IN_01,seed,
0x0E40000,              //3       GOTO    next_data,,
0x0740065,              //4       MOVW    seed,OUT_23,
0x0740024,              //5       MOVW    data,OUT_01,
0x0B40000,              //6       NOP     ,,
0x0B40000,              //7       NOP     ,,
0x0B40000,              //8       NOP     ,,
0x0B40000,              //9       NOP     ,,
0x0B40000,              //10      NOP     ,,
0x0B40000,              //11      NOP     ,,
0x0B40000,              //12      NOP     ,,
0x0B40000,              //13      NOP     ,,
0x0B40000,              //14      NOP     ,,
0x0B40000,              //15      NOP     ,,
0x0B40000,              //16      NOP     ,,
0x0B40000,              //17      NOP     ,,
0x0B40000,              //18      NOP     ,,
0x0B40000,              //19      NOP     ,,
0x0B40000,              //20      NOP     ,,
0x0B40000,              //21      NOP     ,,
0x0B40000,              //22      NOP     ,,
0x0B40000,              //23      NOP     ,,
0x08C0024,              //24      SWAPW   data,OUT_01,
0x0E00000,              //25      GOTO    start,,
write_FU_memory_PM1_IC,//0x05
0x00CC00C0,             //address
14,                     //register number
0x0A40002,              //64      CLRF    cont_taps,,
0x0C43040,              //65      BRCLR   seed,0,jump1
0x0840082,              //66      INCW    cont_taps,cont_taps,
0x0C45042,              //67      BRCLR   seed,2,jump2
0x0840082,              //68      INCW    cont_taps,cont_taps,
0x0C47043,              //69      BRCLR   seed,3,jump3
0x0840082,              //70      INCW    cont_taps,cont_taps,
0x0C49045,              //71      BRCLR   seed,5,jump4
0x0840082,              //72      INCW    cont_taps,cont_taps,
0x0AC0000,              //73      SEC     ,,
0x0D4C080,              //74      BRSET   cont_taps,0,$+2
0x0A80000,              //75      CLC     ,,
0x0940040,              //76      RRW     seed,data,
0x0E04000,              //77      GOTO    ret_next_data,,
connect_component_IC,  //0x01
enable_processors_wait_IC,//0x0E
end_IC                  //0x0F
};
```

Listing E.6: SXM file generated by SANE Project developer after execute Build Project option.

```
01031F000000000010000011AA00000000000011CC000000000000000111CC00C10000000097
01031F000811AA00A10000000911AA00A1000000000000000000000000000000000000000015
01031F001000000000000000000000000000000000000000000000000000000000000001CD
01031F001811AA00A20000000311CC00C1000000040000000900000000000000440000000077
01031F00200000000411CC00C10000001A0078080100B4000000740801 00E400000074006593
01031F00280074002400B4000000B4000000B4000000B4000000B4000000B4000000B4000032
01031F003000B4000000B4000000B4000000B4000000B4000000B4000000B4000000B400000E
01031F003800B4000000B4000000B40000008C002400E000000000000511CC00C10000000E49
01031F004000A4000200C4304000840082 00C4504200840082 00C4704300840082 00C490454C
01031F00480084008200AC000000D4C08000A8000000940040 00E040000000001 0000001122
01031F00500000000120000 11AA00000000000022CC000000000000000122CC00C20000000022
```

```
01031F005811AA00A10000000911AA00A1000000000000000000000000000000000000000C5
01031F006000000000000000000000000000000000000000000000000000000000000027C
01031F006811AA00A20000000322CC00C2000000040000009000000000000000440000000015
01031F007000000000422CC00C20000001A007808010 0B4000000 74080100E400000074006531
01031F00780074002400B4000000B4000000B4000000B4000000B4000000B4000000B40000E2
01031F008000B4000000B4000000B4000000B4000000B4000000B4000000B4000000B40000BE
01031F008800B4000000B4000000B40000008C002400E00000000000522CC00C20000000EE7
01031F009000A4000200C43040008400820 0C45042008400820 0C47043008400820 0C49045FC
01031F00980084008200AC000000D4C08000A8000000940040 00E040000000000100000013D0
01031F00A0000000140000 11AA0000000 2000011CC0000001500000016000011AA00000002A8
01031F00A8000022CC0000001700000000000011AA0000000100000006 11AA00A10000000013
01031F00B000CC00C00000000900CC00C00000000211AA00A20000000311AA00A2000000004E
01031F00B800000000000000000000000000000000000000000000000000000000000000026
01031F00C000CC00C00000000900CC00C00000000011CC00C10000000911CC00C100000000B8
01031F00C822CC00C20000000922CC00C20000000211AA00A20000000000CC00C000000001C1
01031F00D011CC00C10000000222CC00C2000000000000000000000000000000000000000BE
01031F00D800000000000000000000000000000000000000000000000211AA00A100000003A5
01031F00E011AA00A10000000A0000000311AA00A1000000040000000900000000000000044E8
01031F00E800000000000000411AA00A1000000380050504100 5F300100500048005 0F0080D
01031F00F000740209007402 0A0074020B0074020C0074020D0074020E0074020F00A40002BA
01031F00F800A400040074006400740064007808000078084600E8000000740001007400E592
01031F010000C1A10200C29B0300B8210400B83B2C00BC010400E2900000C2910300C29B0433
01031F010800B8310400B84B2C00BC110400E2900000C2510400C29B0500B8410400B85B2CC1
01031F011000E2900000C2910500C29B0600B8510400B86B2C00D5A10100D401000050B002F6
01031F0118014 0D0C200501002014310C2005F106500B0000000C0DB0700BC210400B82B2C64
01031F012000B81B2C00BC0B2C00B80B2C00E0D00000000000511AA00A10000002B00D431801B
01031F0128005F206500B000000050B0020144B0C2005060020140D0C200501002014530C249
01031F0130005F306500B0000000C0DB0700B8010400BC410400BC2B2C00B81B2C00BC0B2CA4
01031F013800B80B2C00E0D00000C0DB0700BC310400B82B2C00BC1B2C00BC0B2C00B80B2C7F
01031F014000E0D00000D5D181005F406500B0000000506002014630C2005010020140D0C2F2
01031F0148005F506500B0000000C0DB0700B8110400BC510400BC2B2C00BC1B2C00BC0B2C48
01031F015000B80B2C00E0D0000000000611AA00A10000003600A4000500701001 00C84040E4
01031F0158008401450 0C86041008401450 0C88042008401450 0C8A043008401450 0C8C044F3
01031F0160008401450 0C8E045008401450 0C90046008401450 0C92047008401450 0C94048D8
01031F0168008401450 0C96049008401450 0C9804A008401450 0C9A04B008401450 0C9C04CBF
01031F0170008401450 0C9E04D008401450 0CA004E008401450 0CA204F008401450074038FF8
01031F01780074034E0074030D007402CC0074028B0074024A0074020900740148007402 0364
01031F0180006032430060328300603 2C300603303006033430060338300603 3C3009C00C3E7
01031F0188009C00C3009C00C300E120000000000311AA00A2000000040000000400000002E
01031F0190000000000000000000000000411AA00A2000000190078080000B4000000B40000EB
01031F019800B4000000B4000000B4000000740842007408840078080100B4000000B4000082
01031F01A000B4000000B4000000B4000000740843007408850 0A4000601617002016170 43CE
01031F01A800BC0186016170840161 70C500BC1186007401A400E000000000000000000CCED
01031F01B000000000000000300CC00C00000000011AA00A10000000911AA00A100000000DD
01031F01B800000000000000000000000000000000000000000000000000000000000000025
01031F01C000000000000000000000000000000011AA00A10000000911AA00A1000000005C
01031F01C811AA00A20000000300CC00C0000000040000000900000000000000044000000000D8
01031F01D00000000400CC00C00000001A007808010 0B4000000 74080100E4000000740065F4
01031F01D80074002400B4000000B4000000B4000000B4000000B4000000B4000000B4000081
01031F01E000B4000000B4000000B4000000B4000000B4000000B4000000B4000000B400005D
01031F01E800B4000000B4000000B40000008C002400E00000000000500CC00C00000000EAA
01031F01F000A4000200C43040008400820 0C45042008400820 0C47043008400820 0C490459B
01031F01F80084008200AC000000D4C08000A8000000940040 00E040000000001000000E74
0103030200000000 0FE9
```

E.2 Listings for Static Fault Tolerance Application Example

Listing E.7: SASM file for Static Fault Tolerace example application.

```
;-------------------------------------------------------------------------------
;-- SANE assembly file template for Project ftMode5_8BitsSequence_3Components
;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
;-- Engineer: JSV
;--
;-- Date Created:  07/04/2014 6:40:09
;-- Project Name:  ftMode5_8BitsSequence_3Components
;-- Filename:      Program3.sasm
;-- Description: SASM file for Static Fault Tolerace example application.
;--
;-- Revision 0.01 - File created.
;-- Additional Comments:
;--
;-------------------------------------------------------------------------------
;**********************************
;Initial configuration after execution
;**********************************
;CHIP0 |-----| |-----|  CHIP1 |-----| |-----|
;      | C00 | | C10 |        | C00 | | C10 |
;      |-----| |-----|        |-----| |-----|
;
;      |-----| |-----|        |-----| |-----|
;      | C01 | | C11 |        | C01 | | C11 |
;      |-----| |-----|        |-----| |-----|

;c00_CHIP0: address=0xAAAA0001, MODE=0x04, PORTS=0x00, FTCSR=0x45.
;           Primary_Cell with one 8-bit Working_processor.
;           Generator of a 8-bit binary sequence as follows:
;           00000000
;           00000001
;           00000011
;           00000111
;           00001111
;           00011111
;           00111111
;           01111111
;           11111111
;c01_CHIP0: address=0xBBBB0002, MODE=0x04, PORTS=0x00, FTCSR=0x55.
;           Redundant_Cell with one 8-bit Redundant_processor.
;           Same sequence of Working_Processor.
;c10_CHIP0: address=0xCCCC0003, MODE=0x00, PORTS=0x00, FTCSR=0x00.
;           Delay for Working and Redundant Processors.
;others   : Empty in the initial configuration, when the FTS detect the
;           failure, thprimary and redundant will be located in this cells.
;note: Normally the Working_processor and the redundant_processor
;      must execute the same thread (or sequence for the current example).
;      For testing purposes, one of the sequences must be altered to
;      check the Fault Tolerance functionallity presented here.
;**********************************
;* Declarations
;**********************************
cont_primary     equ   0xAAAA0001
cont_redundant   equ   0xBBBB0002
retardo          equ   0xCCCC0003
;**********************************
;* Main configuration program
;**********************************
ft_configuration  cont_primary,cont_redundant ;original and redundant cells
create_component  0xAAAA         ;create component with primary cell
create_component  0xBBBB         ;create component with redundant cell
create_component  0xCCCC         ;create component with delay process
restart_and_disable_processors   ;
write_FU_memory   cont_primary   ;write FU memories for primary cell
write_FU_memory   cont_redundant ;write FU memories for redundant cell
write_FU_memory   retardo        ;write FU memories for delay cell
```

```
connect_component                ;Connect components
enable_processors_wait           ;Start execution and wait for an event.
                                 ;When a hardware failure is detected the
                                 ;instruction ft_configuration  is executed
end                              ;End of SASM configuration file
```

Listing E.8: ASM code for Working and Redundant Processors in Primary and Redundant Cells.

```
;-------------------------------------------------------------------------------
;-- Assembler template for Processor Core 0 in Mode 4
;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
;-- Engineer: JSV
;--
;-- Date Created:  07/04/2014 6:45:03
;-- Project Name:  ftMode5_8BitsSequence_3Components
;-- Filename:      Primary_3bitsBinaryCounter.asm
;-- Description: Primary_Cell with the Working_processor. 8-bits Binary sequence,
;--              for test Fault Tolerance System in ft_mode=5
;--              Secuence; 0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF, ...
;-- Revision 0.01 - File created.
;-- Additional Comments: address AAAA0001 - one 8-bit processor. Mode 4.
;--
;-------------------------------------------------------------------------------
#include        <pMode4Core0.inc>

;8-Bit General Purpose Registers
bin0      equ   0x0  ;Constant definitions for binary sequence
bin1      equ   0x1
bin2      equ   0x2
bin3      equ   0x3
bin4      equ   0x4
bin5      equ   0x5
bin6      equ   0x6
bin7      equ   0x7
bin8      equ   0x8
H3F       equ   0x3F
BL_IN0    equ   0x0  ;input port definition for BLMOV instruction
;************************************
;* Directives
;************************************
MODE_CORE   0x4,0
ORG         0x0
;************************************
;* Start of programm
;************************************
start
  movlf  0x00, bin0, 0  ;move constants values to binX registers
  movlf  0x01, bin1, 0  ;for sequence generation
  movlf  0x03, bin2, 0
  movlf  0x07, bin3, 0
  movlf  0x0F, bin4, 0
  movlf  0x1F, bin5, 0
  movlf  0x3F, bin6, 0
  movlf  0x7F, bin7, 0
  movlf  0xFF, bin8, 0
cycle
  movw   bin0, OUT0     ;move binX data to output port
  blmov  BL_IN0, H3F    ;wait for an input data from Working_processor
  movw   bin1, OUT0     ;Continue with the sequence ...
  blmov  BL_IN0, H3F
  movw   bin2, OUT0
  blmov  BL_IN0, H3F
  movw   bin3, OUT0
  blmov  BL_IN0, H3F
  movw   bin4, OUT0
  blmov  BL_IN0, H3F
  movw   bin5, OUT0
  blmov  BL_IN0, H3F
  movw   bin7, OUT0    ;THIS LINE INCLUDE THE INDUCED SOFTWARE ERROR
  ;movw   bin6, OUT0    ;THE CORRECT SEQUENCE MUST INCLUDE THIS LINE
```

```
63   blmov   BL_IN0, H3F
64   movw    bin7, OUT0
65   blmov   BL_IN0, H3F
66   movw    bin8, OUT0
67   blmov   BL_IN0, H3F
68   goto    cycle
```

Listing E.9: ASM code for delay of binary sequence.

```
1  ;-------------------------------------------------------------------------------
2  ;-- Assembler template for Processor Core 0 in Mode 0
3  ;-- Created by: SANE Project Developer - v 2.20 - Javier Soto Vargas
4  ;-- Engineer: JSV
5  ;--
6  ;-- Date Created:        07/04/2014 6:51:35
7  ;-- Project Name:        ftMode5_8BitsSequence_3Components
8  ;-- Filename:            retardo.asm
9  ;-- Description: Delay for sequence
10 ;--
11 ;-- Revision 0.01 - File created.
12 ;-- Additional Comments:
13 ;--
14 ;-------------------------------------------------------------------------------
15 #include        <pMode0Core0.inc>
16
17 ;8-Bit General Purpose Registers
18 cont1       equ     0x1
19 cont2       equ     0x2
20 cont3       equ     0x3
21 BL_IN0      equ     0x0
22 ;************************************
23 ;* Directives
24 ;************************************
25   MODE_CORE  0x0,0 ;mode core directive
26   ORG        0x0   ;origin of first instruction
27 ;************************************
28 ;* Start of programm
29 ;************************************
30 start
31   BLMOV   BL_IN0,0x3F   ;Read an indicator from Working_processor
32 delay                   ;to start the delay process
33   MOVLF   .127,cont3,0  ;constant cont3 to generate delay
34 del3
35   MOVLF   0,cont2,0     ;constant cont2 to generate delay
36 del2
37   MOVLF   0,cont1,0     ;constant cont1 to generate delay
38 del1
39   nop
40   DBNZ    cont1,cont1,del1  ;loop using cont1
41   DBNZ    cont2,cont2,del2  ;loop using cont2
42   DBNZ    cont3,cont3,del3  ;loop using cont3
43   MOVLF   0x55,OUT0,0       ;move any data to OUT0 for comunicate the end
44   GOTO    start             ;of the process to other cells
```

Listing E.10: SHEX file generated by SANE Project developer after execute Build Project option.

```
1  unsigned int memoria[] =
2  {
3  ft_configuration_IC,    //0x18
4  0xAAAA0001,             //id_primary
5  0xBBBB0002,             //id_redundant
6  create_component_IC,    //0x00
7  0xAAAA,                 //id_comp
8  0,                      //Zero-based num cells in component
9  1,                      //num of outputs
10 0xAAAA0001,             //address
11 0x00,
12 0xCCCC0003,             //Input 0
13 0x00,
```

```
0x00000000,             //Input 1
0x00,
0x00000000,             //Input 2
0x00,
0x00000000,             //Input 3
0x04,
0xBBBB0002,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x00,
0xCCCC0003,             //Output 0
create_component_IC,    //0x00
0xBBBB,                 //id_comp
0,                      //Zero-based num cells in component
1,                      //num of outputs
0xBBBB0002,             //address
0x00,
0xCCCC0003,             //Input 0
0x00,
0x00000000,             //Input 1
0x00,
0x00000000,             //Input 2
0x00,
0x00000000,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x04,
0xAAAA0001,             //Output 0
create_component_IC,    //0x00
0xCCCC,                 //id_comp
0,                      //Zero-based num cells in component
2,                      //num of outputs
0xCCCC0003,             //address
0x00,
0xAAAA0001,             //Input 0
0x00,
0x00000000,             //Input 1
0x00,
0x00000000,             //Input 2
0x00,
0x00000000,             //Input 3
0x00,
0x00000000,             //FT Input 0
0x00,
0x00000000,             //FT Input 1
0x00,
0x00000000,             //FT Input 2
0x00,
0x00000000,             //FT Input 3
0x00,
0xAAAA0001,             //Output 0
0x00,
0xBBBB0002,             //Output 1
restart_and_disable_processors_IC,//0x0A
write_FU_memory_CR_IC, //0x03
0xAAAA0001,             //address
4,                      //register number
0x04,                   //MODE
0x00,                   //FAMILY
0x00,                   //PORTS
```

```
84  0x45,                   //FTCSR
85  write_FU_memory_PM0_IC,//0x04
86  0xAAAA0001,             //address
87  28,                     //register number
88  0x0500000,              //0       movlf   0x00,bin0,0
89  0x0501001,              //1       movlf   0x01,bin1,0
90  0x0503002,              //2       movlf   0x03,bin2,0
91  0x0507003,              //3       movlf   0x07,bin3,0
92  0x050F004,              //4       movlf   0x0F,bin4,0
93  0x051F005,              //5       movlf   0x1F,bin5,0
94  0x053F006,              //6       movlf   0x3F,bin6,0
95  0x057F007,              //7       movlf   0x7F,bin7,0
96  0x05FF008,              //8       movlf   0xFF,bin8,0
97  0x0740024,              //9       movw    bin0,OUT0,
98  0x078083F,              //10      blmov   BL_IN0,H3F,
99  0x0740064,              //11      movw    bin1,OUT0,
100 0x078083F,              //12      blmov   BL_IN0,H3F,
101 0x07400A4,              //13      movw    bin2,OUT0,
102 0x078083F,              //14      blmov   BL_IN0,H3F,
103 0x07400E4,              //15      movw    bin3,OUT0,
104 0x078083F,              //16      blmov   BL_IN0,H3F,
105 0x0740124,              //17      movw    bin4,OUT0,
106 0x078083F,              //18      blmov   BL_IN0,H3F,
107 0x0740164,              //19      movw    bin5,OUT0,
108 0x078083F,              //20      blmov   BL_IN0,H3F,
109 0x07401E4,              //21      movw    bin7,OUT0,
110 0x078083F,              //22      blmov   BL_IN0,H3F,
111 0x07401E4,              //23      movw    bin7,OUT0,
112 0x078083F,              //24      blmov   BL_IN0,H3F,
113 0x0740224,              //25      movw    bin8,OUT0,
114 0x078083F,              //26      blmov   BL_IN0,H3F,
115 0x0E09000,              //27      goto    cycle,,
116 write_FU_memory_CR_IC, //0x03
117 0xBBBB0002,             //address
118 4,                      //register number
119 0x04,                   //MODE
120 0x00,                   //FAMILY
121 0x00,                   //PORTS
122 0x55,                   //FTCSR
123 write_FU_memory_PM0_IC,//0x04
124 0xBBBB0002,             //address
125 28,                     //register number
126 0x0500000,              //0       movlf   0x00,bin0,0
127 0x0501001,              //1       movlf   0x01,bin1,0
128 0x0503002,              //2       movlf   0x03,bin2,0
129 0x0507003,              //3       movlf   0x07,bin3,0
130 0x050F004,              //4       movlf   0x0F,bin4,0
131 0x051F005,              //5       movlf   0x1F,bin5,0
132 0x053F006,              //6       movlf   0x3F,bin6,0
133 0x057F007,              //7       movlf   0x7F,bin7,0
134 0x05FF008,              //8       movlf   0xFF,bin8,0
135 0x0740024,              //9       movw    bin0,OUT0,
136 0x078083F,              //10      blmov   BL_IN0,H3F,
137 0x0740064,              //11      movw    bin1,OUT0,
138 0x078083F,              //12      blmov   BL_IN0,H3F,
139 0x07400A4,              //13      movw    bin2,OUT0,
140 0x078083F,              //14      blmov   BL_IN0,H3F,
141 0x07400E4,              //15      movw    bin3,OUT0,
142 0x078083F,              //16      blmov   BL_IN0,H3F,
143 0x0740124,              //17      movw    bin4,OUT0,
144 0x078083F,              //18      blmov   BL_IN0,H3F,
145 0x0740164,              //19      movw    bin5,OUT0,
146 0x078083F,              //20      blmov   BL_IN0,H3F,
147 0x07401A4,              //21      movw    bin6,OUT0,
148 0x078083F,              //22      blmov   BL_IN0,H3F,
149 0x07401E4,              //23      movw    bin7,OUT0,
150 0x078083F,              //24      blmov   BL_IN0,H3F,
151 0x0740224,              //25      movw    bin8,OUT0,
152 0x078083F,              //26      blmov   BL_IN0,H3F,
153 0x0E09000,              //27      goto    cycle,,
```

```
write_FU_memory_CR_IC, //0x03
0xCCCC0003,            //address
4,                     //register number
0x00,                  //MODE
0x00,                  //FAMILY
0x00,                  //PORTS
0x00,                  //FTCSR
write_FU_memory_PM0_IC,//0x04
0xCCCC0003,            //address
10,                    //register number
0x078083F,             //0       BLMOV   BL_IN0,0x3F,
0x057F003,             //1       MOVLF   .127,cont3,0
0x0500002,             //2       MOVLF   0,cont2,0
0x0500001,             //3       MOVLF   0,cont1,0
0x0B40000,             //4       nop     ,,
0x1704041,             //5       DBNZ    cont1,cont1,del1
0x1703082,             //6       DBNZ    cont2,cont2,del2
0x17020C3,             //7       DBNZ    cont3,cont3,del3
0x0555024,             //8       MOVLF   0x55,OUT0,0
0x0E00000,             //9       GOTO    start,,
connect_component_IC,  //0x01
enable_processors_wait_IC,//0x0E
end_IC                 //0x0F
};
```

Listing E.11: SXM file generated by SANE Project developer after execute Build Project option.

```
01031F000000000018AAAA0001BBBB000200000000000AAAA000000000000000001AAAA00014F
01031F0008000000000CCCC000300000000000000000000000000000000000000000000000003B
01031F001000000004BBBB000200000000000000000000000000000000000000000000000052
01031F0018000000000CCCC000300000000000BBBB000000000000000001BBBB00020000000003C
01031F0020CCCC000300000000000000000000000000000000000000000000000000000000023
01031F00280000000000000000000000000000000000000000000000000000000000000004B2
01031F0030AAAA000100000000000CCCC000000000000002CCCC000300000000AAAA0001CF
01031F00380000000000000000000000000000000000000000000000000000000000000000A6
01031F004000000000000000000000000000000000000000000000000000000000AAAA000149
01031F004800000000BBBB00020000000A00000003AAAA000100000004000000040000000B4
01031F005000000000000004500000004AAAA000100000001C005000000050100100503002A1
01031F00580050700300 50F0040051F0050053F0060057F007005FF008007400240078083FF4
01031F006000740064007808 3F007400A40078083F007400E40078083F007401240078083FA1
01031F006800740164007808 3F007401E40078083F007401E40078083F007402240078083F55
01031F007000E0900000000003BBBB000200000004000000040000000000000000000005526
01031F007800000004BBBB00020000001C005000000050100100503002005070030050F00494
01031F00800051F0050053F0060057F007005FF008007400240078083F007400640078083F3C
01031F0088007400A40078083F007400E40078083F007401240078083F007401640078083F78
01031F0090007401A40078083F007401E40078083F007402240078083F00E090000000000392
01031F0098CCCC000300000004000000000000000000000000000000000000000004CCCC000308
01031F00A00000000A0078083F0057F003005000020050000100B400000170404101703082BF
01031700A8017020C30055502400E000000000001000000 0E0000000F23
```

E.3 Conclusions

The listings for the Dynamic Fault Tolerance Scaling application and for the Static Fault Tolerance mechanism has been presented. These applications has been described in section 5.5 and 5.6 respectively.

The listing presented includes SANE Assembler (SASM) and Assembler (ASM) files. The listings of SASM files presents the syntax and structure of the high-level configuration file, which represents the sequence of instructions that the Control Microprocessor (CμP) executes for the configuration of a SANE ASSEMBLY (SANE-ASM). The listings of ASM files represents the tasks scheduled to cell processors.

The SASM and ASM listings presented in this chapter shows the syntax and structure of their correspondent languages. These listings has been created and edited using the SANE Project

Developer (SPD). The listings for SHEX and SXM files has been generated automatically by SPD after execution of the "Build Project" process.

The SHEX files represents the byte-code of the SASM instructions that CμP read when execute the configuration of a SANE-ASM. This information is stored in CμP memory and includes the Instruction Code and their correspondent arguments of SASM instructions.

The SXM files presents the frames that will be downloaded to CμP memory when the option "Write Memory" is executed in the SPD. The format of this file is an adaptation of the XMODEM protocol, which includes the same data generated for SHEX files when the project was built.

Glossary

A

ASM (Assembler) Assembly language instructions that are executed by processors. This term can be also associated with the file that includes the ASM instructions, p. 66.

C

CCR (Code Condition Register) Contains the status of bits: Carry, Zero-bit and Thread Active, p. 33.

CCU (Cell Configuration Unit) Part of the cell that executes the self-adaptive algorithms and configure the internal multiplexers, p. xi.

CSR (Configuration and Status Register) Register mapped in Data Memory used for system configuration, p. 34.

CμP (Control Microprocessor) It is responsible for implementing the main program for the configuration and execution of system functionality. In the prototype, the CμP is implemented inside the chip and replaces the External Controller, p. xii.

CU (Configuration Unit) It refers to any configuration unit in a chip, i.e., GCU, CCUs, SMCUs or PIMCUs, p. 20.

E

EC (External Controller) Connected to the External Network, it controls the execution of all processes in the system. In the prototype, the External Controller was replaced by the Control Microprocessor (CμP) inside the chip, p. 13.

ENET (External Network) Network based in the I2C protocol that allows interconnect several chips, p. 13.

F

FTS (Fault Tolerance System) Enables the system to continue operating properly in the event of the failure of some of its processors, p. 17.

FU (Functional Unit) Part of the cell with processing capabilities, p. xi.

G

GCU (Global Configuration Unit) Part of the chip that controls the execution of all self-adaptive processes, p. 14.

GPR (General Purpose Register) Register mapped in Data Memory used for data processing, p. 33.

H

HEX (Hexadecimal File) Hexadecimal file with the compilation result of ASM files.

I

INET (Internal Network) Network based in the I2C protocol that allows interconnect several configuration units inside the chip, p. 14.

M

MIMD (Multiple Instruction Multiple Data) Technique employed to achieve parallelism. Different processors may be executing different instructions on different pieces of data.

MISD (Multiple Instruction Single Data) Many functional units perform different operations on the same data.

O

OMS (Output Multiplexing System) Configures a path between CORES and output registers. It allows write operations over FU output ports., p. 31.

P

PIM (Pin Interconnection Matrix) Part of the chip that allows the interconnection between two chips, p. 14.

PIMCU (Pin Interconnection Matrix Configuration Unit) Part of the pin interconecction matrix that executes the self-adaptive algortihms and configures the internal multiplexers, p. 18.

R

RE (Read Enable) Ninth bit of any FU port. Pulse of one clock-cycle when an FU output port is written, p. 17.

S

SANE (Self-Adaptive Networked Entity) The SANE is the basic self-adaptive computing system; it has the ability of monitoring its local environment and its internal computation process, p. xi.

SANE-ASM (SANE ASSEMBLY) Composed of a group of interconnected SANEs, p. xi.

SASM (SANE Assembler) High-level instructions that are executed by the Control Microprocessor for the implementation of a SANE ASSEMBLY. This term can be also associated with the file that includes the SASM instructions, p. 44.

SCL (Serial Clock Line) Bus line for Internal and External Netorks, p. 20.

SDA (Serial Data Line) Bus line for Internal and External Netorks, p. 20.

SHEX (SANE Hexadecimal File) Hexadecimal file with the configuration of the SANE-ASM that will be implemented in the FPGA prototype, p. 66.

SIMD (Single Instruction Multiple Data) Multiple processing elements that perform the same operation on multiple data points simultaneously.

SISD (Single Instruction Single Data) A single processor executes a single instruction stream, to operate on data stored in a single memory.

SM (Switch Matrix) Part of the cluster that allows the interconnection of components, p. 11.

SMCU (Switch Matrix Configuration Unit) Part of the Swicth Matrix that executes the self-adaptive algortihms and configures the internal multiplexers, p. 18.

SPD (SANE Project Developer) Integrated development environment (IDE) used to develop complete SANE applications, p. xii.

SXM (SANE X-Modem File) Hexadecimal file with the final configuration of the SANE-ASM that will be downloaded in the FPGA prototype, p. 66.

V

VLIW (Very Long Instruction Word) , p. 5.

References

[1] AETHER CONSORTIUM *AETHER Project Home, Self-Adaptative Embedded Technologies for Pervasive Computing Architectures*. URL: http://www.aether-ist.org/ (see pp. 1, 7, 8)

[2] K. WALDSCHMIDT. "Adaptive system architectures" in: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. 2004. 147– DOI: 10.1109/IPDPS.2004.1303130 (see pp. 1, 2)

[3] J.A. CASAS, J.M. MORENO, J. MADRENAS, and J. CABESTANY. "A Novel Hardware Architecture for Self-adaptive Systems" in: *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. 2007. 592–599 DOI: 10.1109/AHS.2007.11 (see pp. 2, 12)

[4] N.J. MACIAS and L.J.K. DURBECK. "Self-assembling circuits with autonomous fault handling" in: *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*. 2002. 46–55 DOI: 10.1109/EH.2002.1029864 (see pp. 2, 14)

[5] J.MANUEL MORENO, YANN THOMA, and EDUARDO SANCHEZ. "POEtic: A Prototyping Platform for Bio-inspired Hardware" in: *Evolvable Systems: From Biology to Hardware* ed. by J.MANUEL MORENO, JORDI MADRENAS, and JORDI COSP. vol. 3637 Lecture Notes in Computer Science Springer Berlin Heidelberg, 2005. 177–187 DOI: 10.1007/11549703_17 (see pp. 2, 3, 14)

[6] J.M. MORENO AROSTEGUI, E. SANCHEZ, and J. CABESTANY. "An in-system routing strategy for evolvable hardware programmable platforms" in: *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*. 2001. 157–166 DOI: 10.1109/EH.2001.937957 (see pp. 2, 14)

[7] M. FLYNN. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, **C-21**: 948–960, 1972. DOI: 10.1109/TC.1972.5009071 (see pp. 2, 8, 59)

[8] WILLIAM STALLINGS. *Computer organization and architecture - designing for performance*. Prentice Hall, 1996. 597–602 (see pp. 3, 17, 31)

[9] MIN-YOU WU and WEI SHU. "MIMD programs on SIMD architectures" in: *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers '96., Sixth Symposium on the*. 1996. 162–170 DOI: 10.1109/FMPC.1996.558073 (see p. 3)

[10] ANDY M. TYRRELL, EDUARDO SANCHEZ, DARIO FLOREANO, GIANLUCA TEMPESTI, DANIEL MANGE, JUAN-MANUEL MORENO, JAY ROSENBERG, and ALESSANDRO E. P. VILLA. "POEtic tissue: an integrated architecture for bio-inspired hardware" in: *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*. ICES'03 Trondheim, Norway: Springer-Verlag, 2003. 129–140 (see p. 3)

[11] Eduardo Sanchez, Daniel Mange, Moshe Sipper, Marco Tomassini, Andres Perez-Uribe, and André Stauffer. "Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware" in: *Evolvable Systems: From Biology to Hardware* ed. by Tetsuya Higuchi, Masaya Iwata, and Weixin Liu. vol. 1259 Lecture Notes in Computer Science Springer Berlin Heidelberg, 1997. 33–54 doi: 10.1007/3-540-63173-9_37 (see p. 3)

[12] E. Sanchez, A. Perez-Uribe, A. Upegui, Y. Thoma, J.M. Moreno, Andrzej Napieralski, A. Villa, G. Sassatelli, H. Volken, and E. Lavarec. "PERPLEXUS: Pervasive Computing Framework for Modeling Complex Virtually-Unbounded Systems" in: *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. 2007. 587–591 doi: 10.1109/AHS.2007.84 (see p. 4)

[13] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J.-M. Moreno, and J. Madrenas. "The Perplexus bio-inspired reconfigurable circuit" in: *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. 2007. 600–605 doi: 10.1109/AHS.2007.105 (see p. 4)

[14] P.-A. Mudry, F. Vannel, G. Tempesti, and D. Mange. "CONFETTI: A reconfigurable hardware platform for prototyping cellular architectures" in: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. 2007. 1–8 doi: 10.1109/IPDPS.2007.370376 (see p. 4)

[15] P.-A. Mudry and G. Tempesti. "Self-Scaling Stream Processing: A Bio-Inspired Approach to Resource Allocation through Dynamic Task Replication" in: *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. 2009. 353–360 doi: 10.1109/AHS.2009.25 (see p. 4)

[16] M.R. Boesen and J. Madsen. "eDNA: A Bio-Inspired Reconfigurable Hardware Cell Architecture Supporting Self-organisation and Self-healing" in: *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. 2009. 147–154 doi: 10.1109/AHS.2009.22 (see p. 4)

[17] Xiaoxuan She and Mark Zwolinski. "A novel self-routing reconfigurable fault-tolerant cell array" in: *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*. Washington, DC, USA: IEEE Computer Society, 2007. 725–731 doi: http://dx.doi.org/10.1109/AHS.2007.13 (see p. 4)

[18] C. Azar, S. Chevobbe, Y. Lhuillier, and J-P Diguet. "Dynamic routing strategy for embedded distributed architectures" in: *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*. 2011. 653–656 doi: 10.1109/ICECS.2011.6122359 (see p. 5)

[19] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Commun. ACM*, **43**: 74–82, 2000. doi: http://doi.acm.org/10.1145/332833.332842 (see p. 5)

[20] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, **41**: 179–196, 2006. doi: 10.1109/JSSC.2005.859896 (see p. 5)

[21] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo Man, and Rudy Lauwereins. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix" in: *Field Programmable Logic and Application* ed. by Peter Cheung and GeorgeA. Constantinides. vol. 2778 Lecture Notes in Computer Science Springer Berlin Heidelberg, 2003. 61–70 doi: 10.1007/978-3-540-45234-8_7 (see p. 5)

[22] Bingfeng Mei, S. Vernalde, D. Verkest, and R. Lauwereins. "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study" in: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings.* vol. 2 2004. 1224–1229 Vol.2 doi: 10.1109/DATE.2004.1269063 (see p. 5)

[23] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, FadiJ. Kurdahi, EliseuM.C. Filho, and VladimirCastro Alves. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *Journal of VLSI signal processing systems for signal, image and video technology*, **24**: 147–164, 2000. doi: 10.1023/A:1008189221436 (see p. 5)

[24] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, **49**: 465–481, 2000. doi: 10.1109/12.859540 (see p. 5)

[25] Guangming Lu, H. Singh, Ming-Hau Lee, N. Bagherzadeh, F.J. Kurdahi, E.M.C. Filho, and V. Alves. "The MorphoSys dynamically reconfigurable system-on-chip" in: *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on.* 1999. 152–160 doi: 10.1109/EH.1999.785447 (see p. 5)

[26] Takashi Miyamori and Kunle Olukotun. "REMARC: Reconfigurable Multimedia Array Coprocessor" in: *IEICE Transactions on Information and Systems E82-D.* 1998. 389–397 (see p. 6)

[27] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *J. Supercomput.*, **26**: 167–184, 2003. doi: 10.1023/A:1024499601571 (see p. 6)

[28] João M. P. Cardoso and Markus Weinhardt. "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture" in: *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications.* FPL '02 London, UK, UK: Springer-Verlag, 2002. 864–874 (see p. 6)

[29] Chris Jesshope *SVP and uTC A dynamic model of concurrency and its implementation as a compiler target* . 2006 (see pp. 7, 8, 36, 101)

[30] Thuy Duong Vu and Chris Jesshope. "Formalizing SANE virtual processor in thread algebra" in: *Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering.* ICFEM'07 Boca Raton, FL, USA: Springer-Verlag, 2007. 345–365 (see pp. 7, 36, 37, 101)

[31] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. "Enabling a Uniform Programming Model Across the Software/Hardware Boundary" in: *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on.* 2006. 89–98 doi: 10.1109/FCCM.2006.40 (see pp. 7, 101)

[32] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews. "Hthreads: A Computational Model for Reconfigurable Devices" in: *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*. 2006. 1–4 doi: 10.1109/FPL.2006.311336 (see pp. 7, 36, 101)

[33] Javier Soto, Juan Manuel Moreno, and Joan Cabestany. "Description of a fault tolerance system implemented in a hardware architecture with self-adaptive capabilities" in: *Proceedings of the 11th international conference on Artificial neural networks conference on Advances in computational intelligence - Volume Part II*. IWANN'11 Torremolinos-Málaga, Spain: Springer-Verlag, 2011. 557–564 (see pp. 8, 17)

[34] Javier Soto, Juan Manuel Moreno, and Joan Cabestany. A self-adaptive hardware architecture with fault tolerance capabilities. *Neurocomputing*, **121**: ¡ce:title¿Advances in Artificial Neural Networks and Machine Learning¡/ce:title¿ ¡ce:subtitle¿Selected papers from the 2011 International Work Conference on Artificial Neural Networks (IWANN 2011)¡/ce:subtitle¿, 25 –31, 2013. doi: http://dx.doi.org/10.1016/j.neucom.2012.10.038 (see p. 8)

[35] J. Soto-Vargas, J.M. Moreno, J. Madrenas, and J. Cabestany. "Implementation of a Dynamic Fault-Tolerance Scaling Technique on a Self-Adaptive Hardware Architecture" in: *International Conference on Reconfigurable Computing and FPGAs, 2009. ReConFig '09*. 2009. 445–450 doi: 10.1109/ReConFig.2009.45 (see p. 8)

[36] NXP Semiconductors *I2C-bus specification and user manual* Rev. 4 2012 (see p. 20)

[37] Qwertie. Software Developer Trapeze Software. Inc. Canada. *ICSharpCode.TextEditor* url: http://www.codeproject.com/Articles/30936/Using-ICSharpCode-TextEditor (see p. 147)