UNIVERSITÉ DE MONTRÉAL

IMPLEMENTATION OF DATA-DRIVEN APPLICATIONS ON TWO-LEVEL RECONFIGURABLE HARDWARE

HIMAN KHANZADI

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

GÉNIE ÉLECTRIQUE

AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

présenté par : KHANZADI Himan

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. BRAULT Jean-Jules, Ph. D., président

M. DAVID Jean-Pierre, Ph. D., membre et directeur de recherche

M. SAVARIA Yvon, Ph. D., membre et codirecteur de recherche

M. BOIS Guy, Ph. D., membre

RÉSUMÉ

Les architectures reconfigurables à large grain sont devenues un sujet important de recherche en raison de leur haut potentiel pour accélérer une large gamme d'applications. Ces architectures utilisent la nature parallèle de l'architecture matérielle pour accélérer les calculs. Les architectures reconfigurables à large grain sont en mesure de combler les lacunes existantes entre le FPGA (architecture reconfigurable à grain fin) et le processeur. Elles contrastent généralement avec les Application Specific Integrated Circuits (ASIC) en ce qui concerne la performance (moins bonnes) et la flexibilité (meilleures).

La programmation d'architectures reconfigurables est un défi qui date depuis longtemps et pose plusieurs problèmes. Les programmeurs doivent être avisés des caractéristiques du matériel sur lequel ils travaillent et connaître des langages de description matériels tels que VHDL et Verilog au lieu de langages de programmation séquentielle. L'implémentation d'un algorithme sur FPGA s'avère plus difficile que de le faire sur des CPU ou des GPU. Les implémentations à base de processeurs ont déjà leur chemin de données pré synthétisé et ont besoin uniquement d'un programme pour le contrôler. Par contre, dans un FPGA, le développeur doit créer autant le chemin de données que le contrôleur. Cependant, concevoir une nouvelle architecture pour exploiter efficacement les millions de cellules logiques et les milliers de ressources arithmétiques dédiées qui sont disponibles dans une FPGA est une tâche difficile qui requiert beaucoup de temps. Seulement les spécialistes dans le design de circuits peuvent le faire.

Ce projet est fondé sur un tissu de calcul générique contrôlé par les données qui a été proposé par le professeur J.P David et a déjà été implémenté par un étudiant à la maîtrise M. Allard. Cette architecture est principalement formée de trois composants: l'unité arithmétique et logique partagée (*Shared Arithmetic Logic Unit –SALU*-), la machine à état pour le jeton des données (*Token State Machine –TSM*-) et la banque de FIFO (FIFO Bank –FB-). Cette architecture est semblable aux architectures reconfigurables à large grain (*Coarse-Grained Reconfigurable Architecture-CGRAs*-), mais contrôlée par les données. En effet, dans cette architecture, les banques de registres sont remplacées par les FB et les contrôleurs sont les TSM. Les opérations commencent dès que les opérandes sont disponibles dans les FIFOs qui contiennent les opérandes. Les données sont déplacées de FB à FB à travers les SALU tel que programmé dans la mémoire de configuration du TSM. Les résultats finaux sont sauvegardés dans les FIFOs.

Ce projet de recherche se fonde sur les CGRAs et les Overlay Architectures (OEA), qui permettent aux concepteurs de profiter d’une architecture précompilée sur FPGA et encore fournir un moyen de configurer le système à un haut niveau. Nous proposons une méthodologie de conception pour implanter un algorithme sur un FPGA qui est préconfiguré avec un CGRA. L’algorithme de conversion nécessite un graphe de flux de données (DFG) comme entrée qui est typiquement le corps d’une boucle. Une quantité maximale d’opérations est traitée en parallèle et une nouvelle itération de la boucle est lancée le plus tôt possible (ASAP). Idéalement, ce traitement est fait avant que la boucle en exécution ne soit finie. Ceci est réalisé en utilisant des techniques de pipelinage logiciel inspirées de la technique d’ordonnancement itératif de Modulo (Iterative Modulo Scheduling). L’ordonnancement Modulo est modifié de manière à ce que les phases de placement et de routage soient intégrées. Dans l’architecture proposée, un tissu de calcul générique contrôlé par les données est connecté aux processeurs standards. En fait, la nouvelle architecture permet aux développeurs de contrôler, de recueillir et de gérer le flux de données sur les banques FIFO. Le développeur est aussi capable de répartir l’exécution de l’application entre les processeurs Microblaze et le TSM. Pour valider l’architecture et le procédé de conception proposées, nous avons développé un exemple illustratif dans lequel un processeur envoie une image en format RGB au tissu de calcul générique. Dans le tissu, l’image est transformée en format Y, Cr, Cb. Les résultats montrent que, grâce au contrôleur DMA entre la mémoire et le tissu, un gain de vitesse de 50 peut être atteint par rapport à une implémentation logicielle pure.

ABSTRACT

Coarse-grained reconfigurable computing architectures have become an important research topic because of their high potential to accelerate a wide range of applications. These architectures apply the concurrent nature of hardware architecture to accelerate computations. Substantially, coarse-grained reconfigurable computing architectures can fill up existing gaps between FPGAs and processor. They typically contrast with Application Specific Integrated Circuits (ASICs) in connection with performance and flexibility.

Programming reconfigurable computing architectures is a long-standing challenge, and it is yet extremely inconvenient. Programmers must be aware of hardware features and also it is assumed that they have a good knowledge of hardware description languages such as VHDL and Verilog, instead of the sequential programming paradigm. Implementing an algorithm on FPGA is intrinsically more difficult than programming a processor or a GPU. Processor-based implementations “only” require a program to control their pre-synthesized data path, while an FPGA requires that a designer creates a new data path and a new controller for each application. Nevertheless, conceiving an architecture that best exploits the millions of logic cells and the thousands of dedicated arithmetic resources available in an FPGA is a time-consuming challenge that only talented experts in circuit design can handle.

This project is founded on the generic data-driven compute fabric proposed by Prof. J.P. David and implemented by M. Allard, a previous master student. This architecture is composed of three main individual components: the Shared Arithmetic Logic Unit (SALU), the Token State Machine (TSM) and the FIFO Bank (FB). The architecture is somewhat similar to Coarse-Grained Reconfigurable Architectures (CGRAs), but it is data-driven. Indeed, in that architecture, register banks are replaced by FBs and the controllers are TSMs. The operations start as soon as the operands are available in the FIFOs that contain the operands. Data travel from FBs to FBs through the SALU, as programmed in the configuration memory of the TSMs. Final results return in FIFOs.

The present work builds on CGRAs, and Overlay Architectures (OAs), that allow a designer to take advantage of a pre-compiled FPGA architecture and still provide a way to configure the system at a higher level. We propose a design methodology to map an algorithm on an FPGA preconfigured with a CGRA. The mapping algorithm requires a data flow graph (DFG) as input, typically the body of a loop. A maximum number of operations are processed in parallel, and a new iteration of the body loop is started as soon as possible, ideally before the completion of the current one, by using software pipelining techniques, inspired of Iterative Modulo Scheduling. Modulo scheduling is modified in a way that placement and routing phases are integrated to the procedure. In the proposed architecture, a generic data-driven compute fabric is interfaced to standard processors. In fact, the new architecture enables the user to control, collect and manage the data flow on FIFO banks. The programmer is also able to program an application split between Microblaze processors and TSMs. To validate the proposed architecture and design method, an illustrative example is developed in which a processor sends an RGB image to a processing fabric, where it is converted to Y, Cr, Cb. Results show that thanks to DMA between the memory and the fabric, a speedup of 50 are reached compared to a pure software implementation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ALU	Architecture Logic Unit
ADRES	Architecture for Dynamically Reconfigurable Embedded system
CC	Configuration Context
CU	Control Unit
CDFG	Control Dataflow Graph
CDMA	Central Direct Memory Access
CPU	Central Processing Units
CGRA	Coarse-Grained Reconfigurable Architecture
CP	Critical Path
DIL	Dataflow Intermediate Language
DFG	Data Flow Graph
EGRA	Expression-Grained Reconfigurable Array
EMS	Edge-centric Modulo Scheduling
FGRA	Fine-Grained Reconfigurable Architecture
FU	Function Unit
FB	FIFO Bank
FPGA	Field-Programmable Gate Arrays
GPU	Graphics Processing Units
HPC	High Performance Computing
II	Initiation Interval
ILP	Integer Linear Programming
LUT	Lookup Table

LLP	Loop Level Parallelism
MRRG	Modulo Routing Resource Graph
MSPR	Modulo Schedule Place & Route
MII	Minimum Initiation Interval
MS	Modulo Scheduling
OpenCL	Open Computing Language
PE	Processing Element
QEA	Quantum-inspired Evolutionary Algorithm
RAC	Reconfigurable ALU Cluster
RCA	Reconfigurable Computing Architecture
RC	Reconfigurable Cell
RaPiD	Reconfigurable Pipelined Datapath
rALU	reconfigurable ALU
rDPA	Reconfigurable Data Path Array
REMARC	Reconfigurable Multi Media Array Coprocessor
RF	Register File
RISC	Reduce Instruction Set Computing
SOC	System on Chip
SALU	Shared-ALU
TSM	Token State Machine
SA	Simulated Annealing
VLIW	Very Long Instruction Word

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Having billions of transistors on a single chip, the best way to design modern computing chips is to make it more parallel and configurable. General purpose processors have evolved to multicore chips, where each core is independent of the other cores, but shares memory resources. For example, the Xeon PHI 7120A processor has 61 embedded cores running up to 244 threads in parallel at 1.2GHz[1].

Graphics Processing Units (GPUs) offer thousands of cores running in parallel. However, to be more efficient, the same instruction must be applied to multiple data, i.e. single instruction, multiple data (SIMD) architecture. For example, the Nvidia K40 chip has 2880 cores leading to a peak performance of 4.3 TFLOPS for single precision arithmetic[2].

Both general purpose processors and GPUs are highly configurable devices, since they are founded based on the Von Neuman model. They benefit from more than 50 years of research and development in programming languages, libraries, and design tools, enabling computer scientists to rapidly design and prototype complex applications.

However, mainstream processors are not necessarily the best targets for algorithms with high data dependencies and/or low latency constraints, since the applications must be transformed to fit the hardware. In such context, the best performances are achieved when the hardware is tailored to the algorithm, as with Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). It is known that designing a complex chip is a long task reserved for experts in the field. Despite decades of research and development, which today enable some high-level circuit design, the need for experts and the long development times are presently the biggest obstacles to the use of FPGAs as mainstream processing devices.

Coarse Grain Reconfigurable Architectures (CGRA) are intermediate solutions between mainstream processors and FPGAs. They are consisted of several units for the processing and exchanging of data with their neighbor units, which are typically organized as a mesh at very low level. Each unit can be programmed to implement a part of the application and route the data to other units. Thus, it is possible to tailor the hardware to the application, without the help of a hardware design specialist. CGRAs are good candidates for repetitive computations with high data dependency. They are employed as coprocessors to accelerate loops and to let the main processor calculate control-dominant parts of an application.

Although the architecture proposed by Allard et al. [3] (refer to pages 35-38) is similar to CGRAs, it is based on a data-driven mode of operation, i.e., the register banks are replaced with FIFO Banks (FBs) and the controllers are Token State Machines (TSMs). Thus, the operations start as soon as the operands are available in the FIFOs that contain the operands. The main advantage of this architecture [3] is that it results in a fast and simple implementation of a user-defined design implemented over a reconfigurable computing architecture.

On the other hand, the available hardware resources increase with each new generation of complex hardware designs and designers should have access to these resources. To this end, designers require a strong knowledge of the hardware design.

In fact, Allard's architecture aims at effectively exploiting increasingly abundant resources to increase the performance of hardware. The improved performance is achieved by using dedicated hardware resources with the simplicity and flexibility of software development. Also, Allard et. al[3] introduced the concept of two different configuration levels. At the lowest level, a hardware design specialist assembles a dedicated CGRA, which is composed of building blocks, such as token-based ALUs, FIFOs, and sequences, i.e., the control path of a token machine. The circuit is then synthesized, placed and routed on an FPGA. At the highest level, computer scientists may program the token machines to implement an application. Thus, the architecture may offer many advantages, including reconfigurability, evolution, high (and low) level programming, and low-level parallelism exploitation.

Figure 1-1 shows the architecture proposed by [3]. As can be seen in this figure, the architecture includes three components: Shared-ALU (SALU), FBs and TSM. The architecture can be extended in both dimensions. All the computing and routing capabilities of the fabric are concentrated in the SALU, which is consisted of eight independent ALUs, their associated decoder, and one central router network. The decoders are used to establish connections between the FBs and ALUs to handle the token production-consumption. In addition, they send data tokens to the ALUs and get back the results. Each decoder has two internal buffers (local accumulators) to temporarily store operands.

Each ALU is controlled by its programmable TSM. TSMs contain the instructions, which include the address of the operands (local accumulator or FB), the type of operation (16 different arithmetic and logic operations are supported) and the address of the result that could be located in any of the FBs connected to the SALU or in a local accumulator.

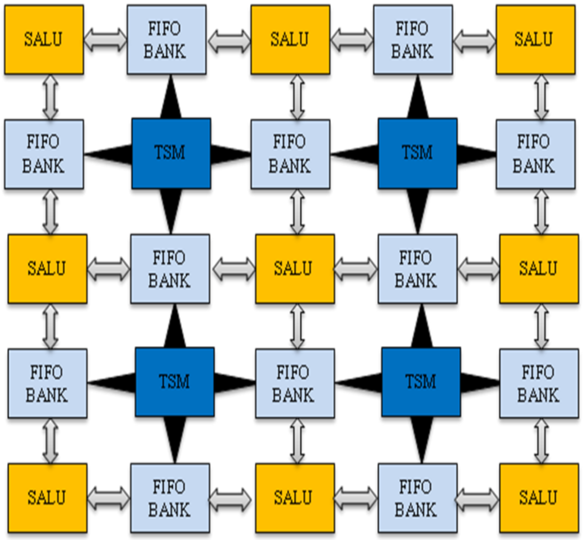


Figure 1-1- Reconfigurable architecture proposed by Allard et al. [3]© 2010 IEEE.

However, the proposed CGRA has been elaborated with fixed applications and was not directly capable of supporting advanced features, such as running the application with new data at runtime. It is to be noted that the FIFO’s contents are charged once the architecture is synthesized. If the FIFOs need to be filled up with a new data, the CGRA should be re-synthesized. Given this, the application of the proposed CGRA is restricted to executing a few input data in parallel. In addition, at the highest level, this architecture still requires an auxiliary method to take advantage of the simplicity and flexibility of software development. In fact, mapping an application manually with thousand operations on the CGRA was essentially an intractable process.

Due to the abovementioned challenges, this work intends to address the problem of mapping complex applications in the Allard et al. architecture.

At a high level, mapping a complex data flow graph with hundreds or thousands of nodes onto the Allard et al. architecture is a complex, tedious and error-prone task. This work proposes an algorithm to automate such mapping. Furthermore, the CGRA is enhanced regarding supporting runtime executing applications. The proposed architecture is consisted of Allard et al. architecture (CGRA-based) that is tightly coupled with a processor-based IP (Microblaze).

In the proposed architecture, CGRA is defined as a custom peripheral Intellectual Properties (IP). This IP is attached to the Processor Local Bus (PLB), where the Microblaze is defined as a microprocessor. To run the applications with a high data throughput, a Central Direct Memory Access (CDMA) is employed. Using the CDMA, the CGRA's FIFO banks could dynamically be recharged with new data tokens. Then, it leads to support runtime execution applications through CDMA by recharging embedded FIFO banks inside the CGRA.

The new architecture makes it possible to analyze the application code to separate the non-critical from sequential or controller application code. The users of this architecture will be able to determine what section of the application should be executed on the CGRA and the Microblaze. The loops are executed on the CGRA as computation-intensive kernels while the Microblaze can execute the sequential code.

In fact, the new architecture enables the user to control, assemble and manage the data flow on FIFO banks. The programmer is also able to program both Microblaze processors and Token State Machines. The proposed architecture provides a simple and fast method for programmers enabling the runtime execution of the applications on the hardware at high abstraction level implemented through software.

The contributions of this work can be summarized as:

- Introducing an automated mapping of the application on proposed CGRA to facilitate the implementation of algorithms that are executing over that CGRA.
- Proposing a reconfigurable architecture model to manage, control and collect data tokens set to the CGRA through a high-level language (C/C++) supported by software (Software Development Kit (SDK)). These modifications on data tokens executing over the hardware

are possible at run time without a need to repeat synthesizing, placing and routing. Ease-of-use and flexibility of the proposed architecture provide an opportunity to support applications requiring dynamic adaptation.

This work is organized as follows: in Chapter 2, a literature review is done to clearly identify the subject and describe exiting mapping algorithms in reconfigurable architectures based on CGRA as well as describing the architecture proposed by Allard et al. In Chapter 3, a solution to automate the mapping of applications on the CGRA is proposed in detail . In addition, a new architecture is explained in detail to support runtime applications on existing CGRA.

In Chapter 4, experimental results are presented. Fast Fourier Transformation (FFT) and matrix multiplication applications showed that the proposed automated methodology could lead to high throughput and/or low latency within a reasonable design time. The proposed architecture was elaborated, synthesized, placed and routed on a Xilinx Virtex-5 FPGA using suitable tools. In addition, the runtime matrix multiplication result showed that the proposed architecture could lead to a high throughput.

CHAPTER 2 STATE-OF-THE-ART

2.1 Introduction

The demand for high-speed accelerator devices or computing architectures to perform a computation has significantly risen regarding. High-Performance Computing (HPC) architectures is a suitable choice to address the demands mentioned above. HPC is used to solve complex science problems that need high bandwidth, low latency, and high computing capabilities.

ASIC and DSP processors are built based on dedicated hardware that have been established in this market. The demand for applications that could handle large real-time data streams creates new demands, such as having general purpose microprocessors and more powerful FPGAs. However, programming the FPGAs is not an easy task, since the programmer must have strong knowledge on low-level design using low-level languages, such as VHDL and Verilog. The high-level synthesis is an automated design process to generate a register-transfer level design from an algorithmic description of a desired behaviour. Therefore, high-level synthesis facilitates the programming of the digital systems, such as FPGAs. The high-level synthesis includes three important tasks that are scheduling, allocation, and binding to form the complete control data path and implement it onto the hardware. Scheduling determines the cycle that an operation can be executed. The most famous scheduling algorithm are: list scheduling (ASAP, ALAP), force-directed, time and resource constrained scheduling, and integer linear programming. The Allocation process determines the appropriate number of the processing unit, storage, and interconnection units. Finally, the binding connects placed and scheduled operations according to their data dependencies. Walker et.al[4] introduced a tutorial for the scheduling problem which is used by the high-level synthesis concept.

In this section, we will first describe the mainstream processors for high-performance computing architectures, such as CPU, GPUs, FPGAs, and CGRA. The most popular CGRAs, such as Rapid, rDPA, PipeRench, MorphoSys, REMARK, ADRES, EGRA, and a mesh of parallel computing and communicating nodes are then studied. We will also present the available programming methods to compile REMARK, MorphoSys, PipeRench and RaPiD architectures and explain the state-of-the-art methods in mapping applications on the CGRAs.

Modulo scheduling problem will be addressed in the following section as the most popular method in mapping application onto CGRAs. Finally, the summarized mapping applications onto CGRAs will be presented in two different tables at the end of this section.

2.2 High performance computing architectures

Hardware technologies are very important since they accelerate HPC applications. Based on the quest of HPC, the following components are used as HPC hardware: (i) central processing units (CPUs) that are taking multiple processor cores into account for parallel computing; (ii) graphics processing units (GPUs) that process huge data blocks in parallel ; (iii) Hybrid CPUs/GPUs computing that is a very common solution for supercomputers, as well as its capability for desktop computers and (iv) Field-Programmable Gate Arrays (FPGAs) that are also very useful for a certain class of demanding applications.

Rapid growth and development of complex computation require high-performance computing (HPC) hardware. HPC is used in parallel processing techniques to solve complex engineering problems needing high bandwidth, high computing capability and low latency. Because of this growth, it is impossible to reach high-performance computing by traditional computing systems that contain only one CPU. To reach to higher performance, the HPC utilizes a combination of different hardware platforms such as CPUs, GPUs and FPGA[5], [6] that will be discussed in the following.

2.2.1 Central Processor Unit (CPU)

CPU is a vital part of a computer and contains two essential components, ALUs and Control Units (CUs). The ALU manages the arithmetic and logical operations, whereas CUs can access to the memory to read and execute the instructions[7], [8]. The design of CPU is based on the prefetching and pipelining architecture. A computer architecture that uses this method facilitates fetching the instruction before the current instruction ends and consequently, the throughput of instructions increases. In order to decrease the required time to execute a program and to improve the high-performance computers, the Reduced Instruction Set Computer (RISC) is used in CPU design. Utilizing RISC may increase the number of internal registers inside the CPU in a way that the data flow pipelining will be improved [7]. A brief history of CPU progress is studied in the following.

The first microprocessor was emerged by Intel, 4-bit 4000, in 1970. This chip contained 2,300 transistors with the capability of executing 92,000 instructions per second[8], [9]. Shortly, Intel came with new innovations in CPU evolution in 8008 and 8080. Over time, the new evolution of CPU is continually developed, and others companies, such as AMD and Motorola introduced their products as a competitor to Intel. In 1993, one of the most popular CPUs called “Pentium” was introduced with 60 MHz clock frequency and 100 millions of instructions per second. The evolution on the Pentium continued until 2008, and both Intel and AMD introduced new generation models of CPU. Intel has developed its product and introduced the first CPU, which had 2 billion transistors [7]–[9].

According to Moore’s law, the number of transistors that could be placed inside a chip is restricted and is approximately doubled every two years. However, available single core CPU may not respond to new applications, since they require to be operated at high speed with higher performance without lowering the price. Therefore, the competition of producing CPUs operating at higher frequencies and high performance inside one core has reached a plateau.

The computer architects reached a new approach in order to have better performance; moving the technology towards the multicore instead of using only one core inside a chip. A multicore processor often runs in slower frequencies than one single core, but with an increased calculation throughput. The term “multicore” refers to an integrated processor including two or more processors attached in order to increase the performance via parallel processing. In parallel processing, many calculations are performed at the same time and thus, the large problems can be solved by breaking them into several smaller parts and executing each concurrently[7]–[9].

The multicore processor can execute multiple instructions at the same time in order to increase the speed by high parallel computing algorithms implemented in software. With the use of parallel computing, large problems may be solved faster.

In the following paragraph, the available multicore processors by Intel, AMD and Tilera will be discussed. AMD Opteron 6000 series processors are based on multicore processors (containing 4, 8, 12, and 16). These series of processors support quad-channel memories in order to achieve high bandwidth amount to 51.2 GB/s [6], [10]. The Tilera family processors contain 16 to 100 cores based on Tilera's iMesh on-chip network that are optimized for networking, video and cloud applications. Each core consists of 64-bits very long instruction level. The mesh interconnection technology used in Tilera is based on two modules. First module is used for streaming applications and the second module in memory communication to reach high performance shared memory[11].

The first high-performance architecture of Intel was introduced by Xeon E5 family that supports up to 8 cores with 20 MB shared memory[1]. They have also developed high-performance computing and introduced a new Xeon family. This new generation of Intel is based on multicore processors that extract a good performance from high parallel computing called Intel Xeon Phi, which is based on Intel Many Integrated Core architecture. Intel Xeon Phi coprocessors are PCI Express cards that enable higher performance gains for parallel tasks. Intel Xeon Phi coprocessors provide up to 61 cores, 244 threads and 1.2 teraflops (Floating Point Operations per Second).These coprocessors are categorized within three main product families; Intel Xeon Phi coprocessor 3100, Intel Xeon Phi coprocessor 5100, and Intel Xeon Phi coprocessor 7100.

2.2.2 Graphical Processor Unit (GPU)

GPU has recently become an influential coprocessor as a general purpose processor. GPUs are more efficient to perform parallel processing than CPU. This superiority is due to the basic nature of the GPU based on parallel data architecture and programmable technology[6][12]. GPUs are designed to accelerate demonstration and processing of visual images on a graphical output device. GPUs can process and display millions of pixels, simultaneously, and their design objective was to assist the video processing on devices, such as personal computers, cell phones, , etc.[7]. Design architects employ the natural properties of GPU to solve the complex scientific problems via general purpose processors. The general purpose GPUs are currently used in various HPC application domains such as medical imaging, bioinformatics, and embedded systems and are an ideal option for accelerator devices for massive data-parallel processing [12].

The programming model of GPU is based on a scalable processing array that consists of single instruction multiple threads having several stream processors. There are several memory spaces in GPUs, such as global and local memory. The global memory is accessible by all cores, and local or shared memory is related to each microprocessor[13]. The 1990's years were the beginning of GPUs by the introduction of 86C911 card by S3, which was one of the first standards for the GPU industry. Evolution of GPUs continued to two-dimensional graphics processing in 1990's up to 3D processing graphics processing, which are used in lower-end laptops today[8]. AMD and NVIDIA introduced several models of GPUs, and each one has specifically improved characteristics compared to the previous versions. The newly developed model of GPU called Tesla K80 that was introduced by NVIDIA and comprised 24 GB memory and up to 2.91 TFLOPS double precision performance with 480GB/s bandwidth. In fact, it consists of two GPUs placed inside one packet, where each GPU has 2496 cores [14]. Tesla K80 is ideal for high-performance computing accelerator that requires massive data throughput in single and double precision mode. AMD designed AMD FirePro S10000, and it has 3584 stream cores with the accuracy of 1.48 TFLOPS of double precision or 5.91 TFLOPS of single precision[14].

2.2.3 Field Programmable Gate Array (FPGA)

FPGAs are reconfigurable integrated systems and are semiconductor devices consisted of many logic blocks linking together through programmable routing networking, embedded memory block, and digital signal processing blocks. The logic block is the main component of FPGAs that is implemented in a Lookup Table. LUTs contain a small attached memory that is programmed for the output logic based on the inputs. FPGAs' resources can be configured and linked together in order to create custom instruction pipeline to determine which data is processed. On the other hand, in CPU and GPU topologies the data path are fixed [8], [15].

FPGA is highly based on high-level parallelism and is a perfect choice for implementing a portion of the application that requires extensive parallelism. Xilinx and Altera are two well-known companies to develop FPGAs. Stratix 10 is the newest FPGA introduced by Altera. Stratix 10 device architecture was manufactured on the Intel 14 nm Tri-Gate technology that provides the highest performance and more power efficiency. Stratix 10 SX SoCs hard processor system with

64 bit quad-core ARM Cortex-A53 processor. The debug tools and heterogeneous advanced languages such as OpenCL developed by Altera SDK as design environment facilitate the application implementation on FPGA [16].

Xilinx Ultra Scale architecture is another high accelerator unique device that provides high-performance, high-bandwidth and low latency. It should be noted that the ultra-scale devices are suitable for processing massive data flows, since they have high bandwidth and low latency [17].

2.2.4 Coarse-Grained Reconfigurable Architecture (CGRA)

Based on granularity, reconfigurable computing architecture can be divided into two categories: fine-grained and Coarse-Grained Reconfigurable Architectures [18], [19].

CGRAs are indicated as application-specific reconfigurable devices or embedded FPGAs. CGRAs are introduced to tackle the disadvantage of Fine-Grained Reconfigurable Architectures (FGRAs) for computing application. Some disadvantages of FGRAs are the configuration time, routability and logic granularity. Logic granularity means that the architecture for FGRA is based on logic elements and is not suitable to handle complex signal processing and multimedia computations. The reconfiguration of FGRA is performed at bit-level; therefore the logic blocks are required to operate wide data path, and its routing path may have a huge wide range and poor routability. CGRA operates at the multiple-bit level. Therefore, it has less configuration time than FGRA [20], [21].

CGRAs are consisted of an array of FUs interconnected by a mesh topology network and register files are scattered among the CGRA. Some key characteristics of CGRAs include size, node functionality, topology, and register file sharing. The size refers to the number of FUs that can vary (e.g. 64 FUs); they are arranged as an array of 8×8. The functionality of each FU can be determined to execute an arithmetic or logic operation, such as addition, subtraction or multiplication. There are several configuration networks topology to provide interconnection between FUs. For example, each node can be connected to its four orthogonal or eight diagonal neighbors. CGRAs can include a local memory. The FUs have access to load or store data. Fine-grained architecture is based on bit-level, and CGRA operates at multiple-bit data paths. The size of configuration bit stream of

CGRA is smaller than FPGA and thus, it has a shorter configuration time. The CGRA is becoming an appealing option, since it consists of a large number of computation units with lower cost, power efficiency, and high flexibility. In addition, CGRA is capable of being programmed, i.e. the intensive computational kernels can be mapped to it. It should be mentioned that the CGRA has been used in high performance embedded system [18], [19], [20], [21].

2.3 Coarse-Grained Reconfigurable Architectures

This section introduces the most popular architecture for CGRAs.

2.3.1 Reconfigurable Pipelined Datapath (RaPiD)

RaPiD is a coarse-grained field-programmable that can perform the computational data path as a pipeline. RaPiD consists of ALUs, multiplier, register files and local memories, which can be configured linearly over a bus. These units are interconnected and controlled via a combination of static and dynamic signals. RaPiD has a linear data path that is an alternative approach with 2-D mesh interconnection of PEs. The structure of data path in RaPiD is based on FUs, which are connected to the nearest neighbor fashion[22].

2.3.2 Reconfigurable data path array (rDPA)

The Xputer architecture was one of the first research efforts in the coarse-grained field programmable hardware. Reconfigurable Data Path Array is a reconfigurable device based on field-programmable, which has 32-bits arithmetic logic unit. The rDPA is coarse-grain and consists of a small array called Unit Data Path. Each reconfigurable ALU is also configured by several numbers of rDPAs and can execute some operators of C language as an integer or fix-point data types up to 32-bits length. The mesh network connection is used as interconnection network between rALU, global bus and the bus memory[23].

2.3.3 PipeRench

The PipeRench architecture class consists of a set of physical pipeline stages so-called stripes. Each stripe is made up of the Interconnected Processing Elements (PEs), which contain ALUs and register files. PEs can access to a global bus and receive data from other register files from the previous stripe or the current state through an interconnection network. Meanwhile, each ALU comprised of LUTs plus some circuits like carry chains and zero detection. In PipeRench the aim is to analyze the application's virtual pipeline to map the physical pipeline stage to achieve the maximum execution throughput. Figure 2-1 shows the architecture of PipeRench[24].

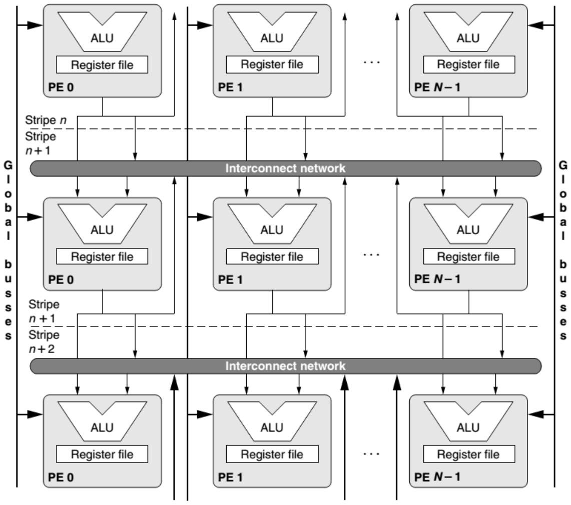


Figure 2-1-PipeRench Architecture [24]©2000 IEEE

MorphoSys is a reconfigurable computing system, which contains a reconfigurable processing unit (as an array of Reconfigurable Cells), a general purpose processor (RISC), and a high bandwidth memory interface. The RCs are interconnected as a 2-D mesh topology and are also coarse-grained. The general processor can control the operation of the RCs. The high-bandwidth interface consists of streaming buffers to transfer data between external memory and RC array. The main component of MorphoSys is an 8×8 RC array, shown in Figure 2-2[25].

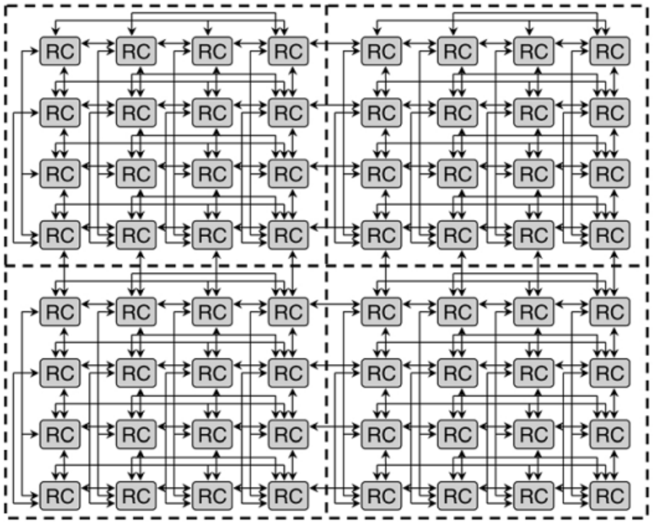


Figure 2-2-MorphoSys reconfigurable computing[25] © 2000 IEEE

2.3.4 Reconfigurable Multimedia Array Coprocessor (REMARK)

REMARK is a reconfigurable coprocessor that is tightly coupled to the main RISC processor. REMARK is designed to accelerate specific application domains, such as multimedia and video/Image processing. It consists of a global control unit of ALU and an 8×8 array programmable logic element called Nano processors. Each Nano processor has a 16-bit data path. The configuration for each Nano element is stored in 32-instruction RAM. Each Nano processor can be connected to the four adjacent Nano processors via dedicated connections. The executions of Nano processors are determined by input signals from the control unit. The input signals can directly configure the instruction for each Nano processor using the main processor. Figure 2-3 shows the architecture for REMARK[26][27].

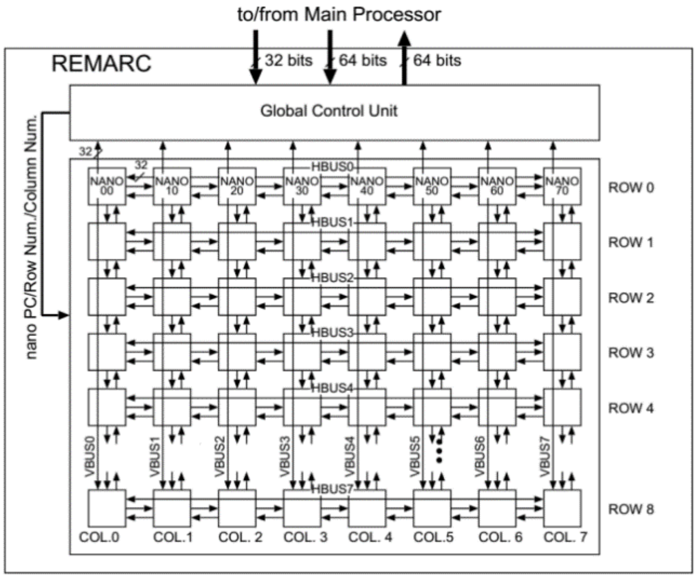


Figure 2-3-REMARK Architecture [26] © 1998 IEEE.

2.3.5 Architecture for Dynamically Reconfigurable Embedded System (ADRES)

The ADRES consisted of a 2-D reconfigurable architecture and comprised of two parts, which couples a Very Long Instruction Word processor and a coarse-grained reconfigurable matrix. The ADRES contains many FUs and register files which are connected via interconnected mesh topology. The FUs can execute the operation at world-level bits and RFs store the intermediate data. For VLIW processor, there are several FUs connected through multi-port register files. The reconfigurable matrix comprised of many reconfigurable cells that contain FUs along with RFs. FUs can be heterogeneous and also support the predicate operation[28].

2.3.6 Expression-Grained Reconfigurable Arrays (EGRA)

EGRA is a platform for the exploration of different designs of CGRA. The EGRA structure is organized as a mesh that consists of three different types of cells i.e. reconfigurable ALU cluster, memories, and multipliers. RACs include heterogeneous arithmetic and logic capabilities to support the complex computation of entire subexpression. Each Cell is connected to its four neighbors and also horizontal-vertical buses. One control unit is instantiated in external of mesh to manage each cell. The architecture for 5×5 tiles of EGRA architecture is shown in Figure 2-4[29].

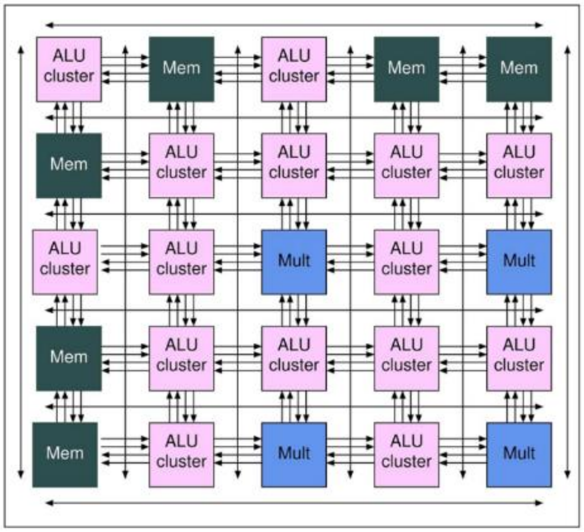


Figure 2-4-EGRA Architecture[29] ©2010 IEEE

2.3.7 A mesh of parallel computing and communicating nodes

The present work is built upon the architecture proposed by Allard et al.[3].The architecture is similar to CGRAs, but it is data-driven, i.e., the register banks are replaced by FIFO Banks (FBs), and the controllers are Token State Machines (TSMs). Thus, the operations start as soon as the operands are available in the FIFOs that contain the operands.

The proposed computing fabric architecture by [3] is comprised of three individual configurable modules i.e. Shared-ALUs, token state machine, and FIFO Banks. The proposed fabric architecture 5×5 is shown in Figure 2-5. As can be seen in this figure, the architecture can be extended in both dimensions. All the computing and routing capabilities of the fabric are concentrated in the SALU, which is consisted of eight independent ALUs, their associated decoder and one central router network, as illustrated in Figure 2-6.

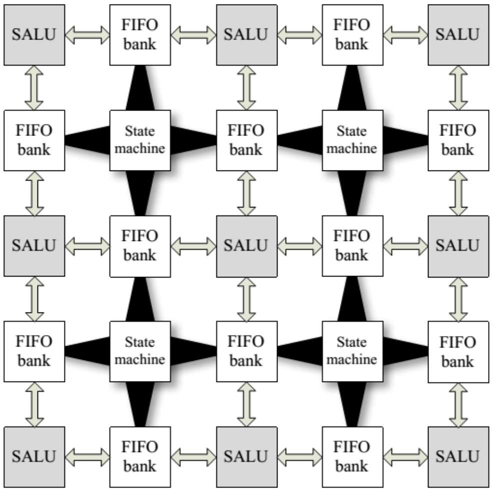


Figure 2-5- CGRA Proposed by Allared [3] © 2010 IEEE

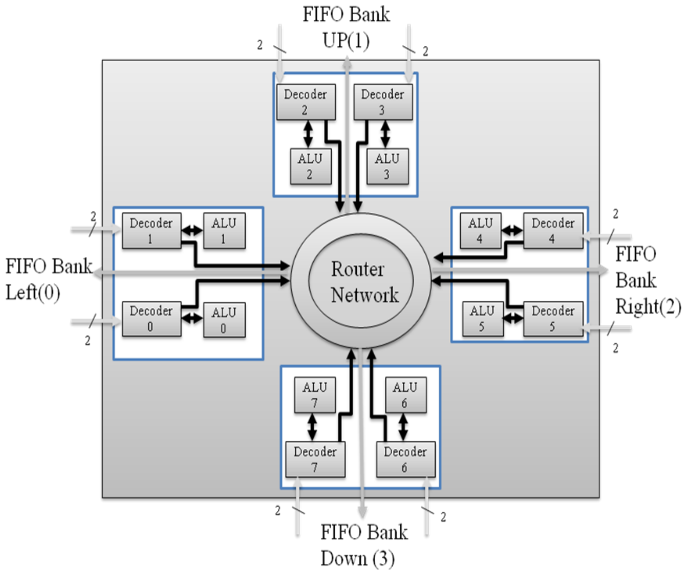


Figure 2-6-The architecture of the SALU [3] © 2010 IEEE

The decoder is used to make the connection between FBs and ALUs. Decoder sends the data token to ALUs to execute the operation. Also, each decoder has two internal buffers to store the operands, temporarily. In order to perform an operation, one operand could come from the adjacent bank, and another one could come from an internal buffer. After executing the operands, the ALU sends the result to the decoder. Thus, the decoder will subsequently forward the result to its final destination through the network. All routing decisions are performed through the router network. In fact, the network router is the vital element of SALU that allows carrying out all data tokens to their destination according to their respective order.

The network can accept eight tokens per cycle and return the same number to different destinations. The network router consists of six distinct routers with two different types called type1 and type2. Figure 2-7 shows the block diagram of router type1 and type2. The left side of router type1 connects to the ALUs and router type2. The output of router type1 connects to the ALUs, FIFO Banks, and router type2. While the router type 2 only has communication with router type1.

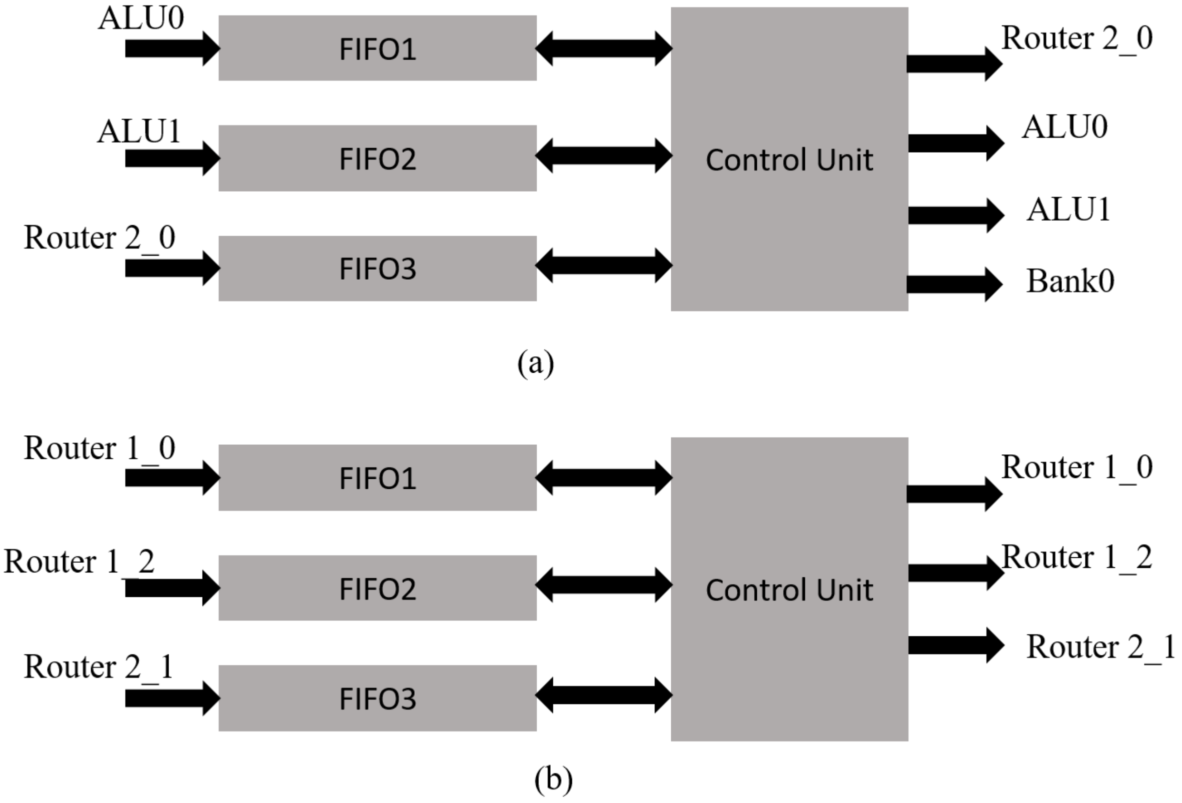


Figure 2-7- Schematic of: a) router type1:router1_0 , b) router type 2: router 2_0 [3] © 2010 IEEE

Each type1router is associated with one side of SALU. In addition, the type 1 router has no contact with other type1, yet they can access to router type 2 to send data token to different paths. A simple round-robin algorithm is used for the network router to send all data token to their destination. The block diagram of router network is shown in Figure 2-8. In this architecture, routing is performed through the central router network. The network can accept eight tokens per cycle and return the same number of tokens to their destinations.

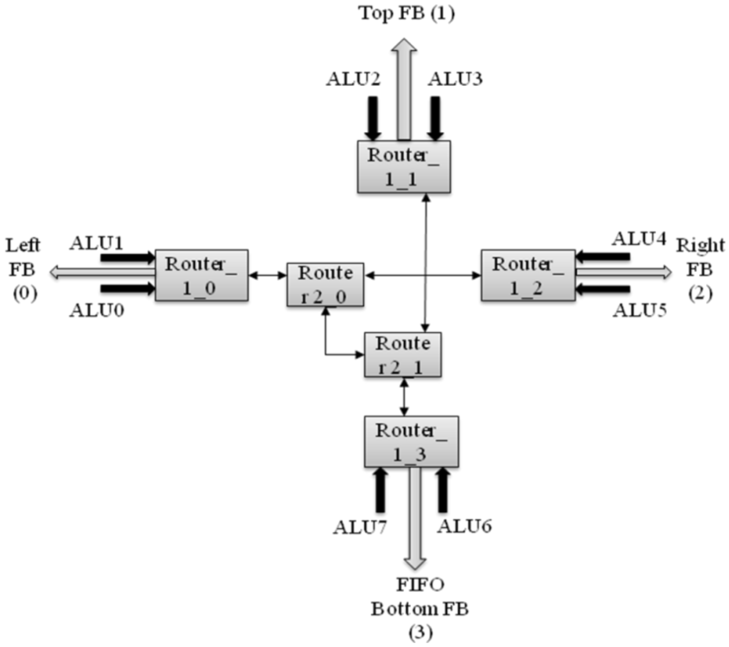


Figure 2-8-Central Router Network [3] © 2010 IEEE

A token state machine is a program unit that stores the required instruction tokens according to a specific application. Instruction can be used in order to control the transmission data tokens between the SALU and RFs. TSM represents the direct interface between the user and the fabric. The instructions must contain all the necessary information to choose the operands, operations, destination, and SALU. Each TSM has eight independent parallel small state machines that each pair corresponds to one FIFO bank.

Figure 2-9 shows the block diagram of a TSM. A FIFO Bank represents the system's memory, which is connected to the two different modules of SALU and TSM. The synchronization by data is the foundation of the architecture, which happens by using data token. FB manages all the traffic required to route the data token and instructions to SALU. Furthermore, it is possible to make concurrent write and read of data tokens. Each FB consists of 16 independent register files R_0 to R_{15}.

The TSM determines the data path and register file. The data token and instruction should be transferred to specified SALU through FBs.

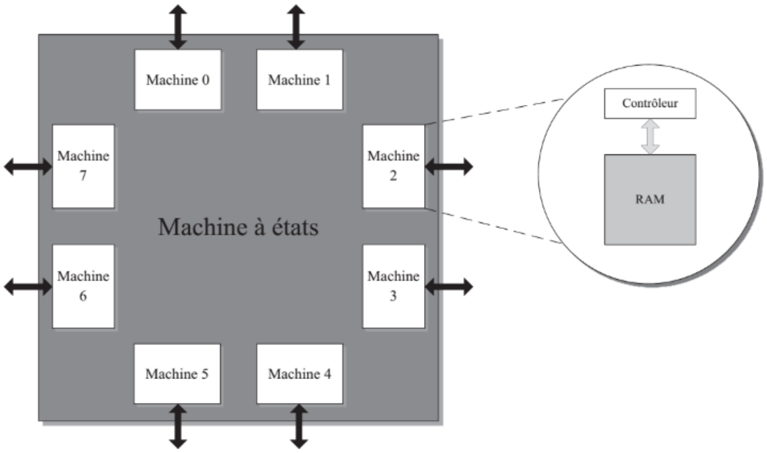


Figure 2-9-Block Diagram of a TSM [3] © 2010 IEEE

2.4 Programming of most popular CGRAs

This section gives the information about the programming of some CGRAs.

2.4.1 Programming REMARK coprocessor

REMARK coprocessor is tightly coupled to a host processor. The programming environment allows programming of both host and coprocessor concurrently. REMARK programming environment is comprised of the REMARK global instruction assembler and the Nano instruction assembler. The GCC compiler programs the host processor. The global instruction assembler begins with global assembly code and generates configuration data and label information. The Nano instruction begins with Nano assembly code and generates the configuration code. Due to utilizing the configurable REMARK architecture, the programmer attaches REMARK assembler instruction into the C program. Using REMARK assembler instruction, the assembled code for the host processor and binary code for REMARK instruction are generated. Finally, the GCC compiler is used again to generate executable code which includes the host processor and the global and Nano configuration data[18][21].

2.4.2 Programming MorphoSys coprocessor

MorphoSys uses GUI-based design tools to compile the application for reconfigurable array and host processor. The programmer has manually to separate the input code between the host processor and reconfigurable array. GUI-based includes mView, mLoad, and MCC. The mView has been developed to help the designer in mapping application to the reconfigurable cells. The mView can operate in two modes, programming mode or simulation mode. Based on both modes, one context file is generated by mView that represents the user-specific application for each cell. For system simulation, each application should be coded into the context words. This context word generates by mLoad using the input file from mView. The MCC is a prototype C language compiler that has been developed to compile code for MorphoSys. After partitioning the code between the host processor (TinyRISC) and the RC arrays, the MCC generates the instructions for TinyRISC processor. These instructions can control the RC array execution for parallel computation[18][27][25].

2.4.3 Programming PipeRench Architecture

The PipeRench compiler maps computations described in a dedicated intermediate single-assignment language so-called DIL into the PipeRench. As mentioned earlier, the PipeRench made up of columns of pipeline stages and the model of the configuration of computation stage which can use the execution of the next stage in the current stage. DIL can be observed as a language to exhibit an intermediate representation of high-level language description such as C. It can also be used to describe pipelined combinatorial circuits. The compiler employs the same internal representation to perform synthesis, optimization and place and route. The compiler constructs a hierarchical acyclic data flow graph as an intermediate representation of the application. The DFG has nodes and edges where nodes represent the operations and edges represent the operands. After the generation of global application's DFG, the compiler does some optimizations over DFG; such optimization includes traditional compiler optimization, for example, common subexpression elimination, algebraic simplification, and dead code elimination. The placement and routing phase is performed via DFG by a deterministic linear-time algorithm which is based on list scheduling[24].

2.4.4 Programming RaPiD Architecture

The programming of RaPiD performs using RaPiD-C, a C-like a language to help the programmer to a map of high-level computation description to the RaPiD architecture. RaPiD-C allows the programmer to specify the parallelism, data movement, and partitioning. The mechanism Wait and Signal are used by RaPiD-C for synchronization and assign right data into the RAM [30], [31].

2.4.5 Different techniques of the mapping applications on the CGRAs

This section provides a brief study of the state-of-the-art methods in mapping applications on the CGRAs.

Ricardo et al. presented a Just-In-Time module scheduling for the mapping application onto CGRA. Their proposed algorithm combines three distinct methods such as a mapping algorithm, a crossbar network, and virtual coarse-grained reconfigurable architecture. A module scheduling algorithm is used in the mapping algorithm to map loops into virtual CGRA. The algorithm is based on a greedy heuristic, and virtual CGRA is a layer on top of FPGA. They have also proposed a CGRA based on crossbar network instead of mesh topology network [32].

The resource constrained mapping of DFG onto CGRA has been presented by Naifeng. The resource constrained mapping problem is formulated using ILP; the produce optimal result is created by ILP for the mapping of the DFG onto the CGRA. In order to accelerate the problem-solving, they have also proposed a heuristic algorithm by using the maximum flow minimum cut algorithm for practical use and large problem[33].

In data-driven mapping using local patterns presented by Gayatri, to accelerate the mapping application on to CGRA, a database of an example of high-quality mapping has been used based on a search tree. The depth of search tree is reduced using placing pattern of nodes instead of single ones. The anytime A* algorithm proposed in this research to find a good solution and improve that solution to place a node on the CGRA. Anytime A* is a greedy algorithm that provides a solution within certain bound to solve the problem of mapping of DFG on to CGRA. To solve the problem mentioned above, they have also used the Anytime Multiline Tree Rollup method in which they try to keep all solution paths diverse to ensure that results from previous steps are stored to avoid repetition and traversing path. They have claimed that their proposed method outperforms the simulated annealing algorithm to placement and routing nodes onto CGRA[34].

Akira et.al have proposed modulo scheduling algorithm to compile loops in a program onto CGRA. Their algorithm consists of resource reservation phase and scheduling algorithm. The resource reservation phase guarantees the resources needed at the steady state such as FUs consumed by operation and routing resources. Resource-aware placement algorithms were proposed to shorten the solution time. In order to map an application onto target architecture, a compact graph has been used in [35].

Yuanqing has proposed an algorithm to map applications written in a high-level language program such C onto CGRA. His proposed algorithm contains 4 phases such as translating source code to a control data flow graph, task clustering and ALU data path mapping, scheduling and resource allocation. In the first phase, the input C program translated into CDFG and some optimization and simplification perform on the CDFG. In the second phase, the CDFG is partitioned into several tasks to assign them on to ALU. In the third phase, the clustered graph is scheduled and mapped to an unbounded number of fully connected ALUs. Finally, the last phase, the scheduled graph in prior phase is assigned to ALU and in the phase, the other resources such as buses, register, memories, etc. are assigned[36].

A routing-aware mapping algorithm has been presented for CGRA by Ganghee. An integer linear programming has been considered for Steiner point routing, i.e., for optimal map application onto CGRA instead of spanning tree based routing. In addition, a fast heuristic mapping algorithm for CGRA that is based on routing aware and incorporated of Steiner point has been presented. The heuristic algorithm contains two phases: list scheduling and quantum-inspired evolutionary algorithm. Using list scheduling the constructed CDFG from the application is scheduled with the given resource constraint to get the initial solution and determine the priority of the node. Dijkstra algorithm is used to find the shortest path between two PEs. The QEA is like a genetic algorithm, and it evaluates each case to reach the best answer of mapping CDFG on to CGRA[37], [38].

Mapping application onto reconfigurable KressArrays proposed by Hartenstein. KressArrays consists of a mesh of a PEs which is also known as reconfigurable data path units. The application written in a high-level programming language are placed and routed on the rDPUs using simulated annealing. A given data path would place and routed on the hardware using simulated annealing based mapper [23][39].

Hyunchul proposed a software pipelining technique for CGRA that leverages module graph embedding referred to graph embedding from graph theory. To place the operations of loop body of the application on CGRA, they have presented three dimensions of CGRA that two of them are related to the FUs and third dimensions assigned to time slots. Module scheduling performed with each set of the operations which are located in the same level of DFG. Three-dimensional scheduling grid is filled for each group of scheduled operation by the skew manner in considering with restricted FUs and time slot available. Also, some cost functions are defined between pair DFG of nodes to reduce the routing path and optimize place and route. These functions are routing cost, affinity cost, and position cost. Routing cost guarantees that producers and consumers are placed close to each other. Affinity cost ensures that the producers with common consumers in DFG are placed together. Finally, position cost ensures that the operations are left-justified on the set of appropriate resources[40].

EMS for CGRA is a research issue in continues of previous work of Hyunchul. Modulo scheduling is a technique in software pipeline of loops to exploit the parallelism of the CGRA. EMS tries to perform the routing of the nodes instead of place nodes first and routing paths followed by placement. During the routing process, if there is a path from the source to the destination of DFG of nodes then placement is done after the routing[41].

Chen has proposed minor graph approach for mapping application onto CGRA. The CGRA mapping problem has been formalized as a graph minor of the module routing resource graph representing the CGRA resources and their interconnects [42].

A retargetable compiler, known as Dynamically Reconfigurable Embedded System Compiler proposed by Mei. He proposed a module scheduling algorithm based on simulated annealing to placement and routing operands on the CGRA. This compiler can parse, analyze transform and schedule plain C program to CGRA[43][44].

Mei introduces a modulo scheduling algorithm to exploit loop level parallelism on CGRA in 2003. Modulo scheduling algorithm used in integer linear program processor such as VLIW to improve the parallelism by executing different loop iteration in parallel. Also modulo routing resource graph proposed as an abstraction of hardware description and enforce to modulo constraint. The proposed algorithm combines the FPGA place and route algorithm with modulo scheduling to achieve a mapping of application onto CGRA [45].

2.4.6 Modulo Scheduling

Modulo Scheduling is a software pipelining technique employed to utilize instruction-level-parallelism in the loops body using overlapping consecutive iterations. The loop body is represented as a data flow graph where the nodes represent the operations, and the edges represent the data dependency among the operations. MS tries to find a pattern to develop it by several iterations of operations. MS utilizes a different approach in which the operation's placement is performed in a cyclic interpretation without any resource conflicts and data dependency violations. The scheduling process includes three stages such as Prolog, Kernel, and the Epilog. The kernel corresponds to the steady-state execution in different consecutive iterations. The instructions of a repetitive pattern of operations are called kernels[46][47].

The goal of MS is to find a valid schedule in which the Initiation Interval (II) is minimized. II is the delay between two successive iterations of the loop body. Ideally, all the loop bodies are processed in parallel (II=0) if there are no dependencies and enough hardware resources. In the worst case scenario, the next iteration of the loop body cannot start before the current one is finished. Initially, the scheduler begins with Minimum II (MII) value between the maximum values of the recurrence-constraints lower bound (RecMII) and the resource constraints lower bound (ResMII). However, if a valid MS cannot be found, the scheduler increases the II by one, and the scheduling is attempted again to find a possible valid MS [42].

MS attempts to explore one model of nodes in DFG that can be executed at the same level. This model, as discussed earlier, is called kernel. The kernel consists of a pattern of DFG nodes. The nodes can be executed as pipeline thanks to the specified pattern. To compute-intensive kernels with high efficiency and flexibility, CGRA architectures are the best candidates. Accordingly, the modified MS is a popular method to map an application in the form of DFG to the CGRA [42], [46][47].

2.4.7 Modulo Scheduling and CGRA

Modulo scheduling is widely used software pipelining technique that is capable of compiling DFG onto a family of heterogeneous CGRA. The goal of mapping is to generate a schedule that explicitly combined with place and route the operation that the application throughput is maximized. This criteria throughput is indicated using initiation interval by modulo scheduling. The II is essentially reflecting the performance of the scheduled and P&R applications onto CGRA, and it plays a central role in exploiting parallelism. Various algorithms have been developed for VLIW processors. However, they have not been successfully applied for CGRA architecture. In another word, the CGRA complexity architecture is much higher than VLIW, due to the complex architecture of CGRA. Thus, the key metric used to map an application onto CGRA is II [32][41]–[43].

The mapping application using MS onto CGRA may give rise to add some different approaches to the scheduler. This difference for scheduling application is mostly due to the hardware characteristics of the CGRA. Modulo scheduling for CGRA considers the scheduling, placement and routing the operations onto function units. Placement determines on which FU of a 2-Dimensional array will place an operation. Scheduling determines in which cycle, an operation can be executed. Finally, routings will connect the placed and scheduled operations according to their data dependencies [32][41]–[43].

In order to map the kernel (defined in Module Scheduling) onto CGRA, each particular cycle of the kernel is mapped on each II configurations of CGRA, where each configuration is referred to one configure the mapping of nodes onto CGRA. Configurations can be stored as a Configuration Context (CC) for CGRA, and they can be updated in every cycle. The CC specifies the functionalities and connectivity among FUs. The CC also includes the direction for each FU to determine where to get its input from prior cycle and where to write its output for the next cycle. In fact, CC is a valid mapped configuration of DFG nodes onto CGRA.

2.4.8 Summary of CGRA mapping algorithms

The characteristics of particular CGRA mainly effect on the compilation techniques. Most CGRAs architectures are non-FPGA based that is coupled to a general purpose processor as a co-processor. Given this, compiling such systems are not as a generic problem similar FPGA-based, because the FPGAs-based has a standard architecture [48] [49].

An overview of placement and routing by well-known CGRA architectures is given in Table 2-1. It can be seen from this table the structure of the CGRA architecture has an important impact on the placement phase. Heuristic placement based on SA and genetic algorithms has been borrowed in synthesis systems for FPGAs [20][18]. PADDI is used a scheduling algorithm in order to resource allocation [20][18][50]. The routing based on greedy algorithms is used only in cases where the routing is restricted to one dimension. Also, the P&R result based on greedy algorithms would not be satisfied as well. The domain specifies which kind of applications can be executed on CGRA, as mentioned in Table 2-1.Table 2-2 provides the summary of the recent mapping algorithms on the CGRAs.

Table 2-1- properties of programming environment of CGRA.

CGRA	Programming	Placement	Routing	Coupling	Domain
REMARC[26]	Assembly	Manual	Manual	Coprocessor	MM
RaPiD[22]	RaPiD-C	SA	Pathfinder	Loose	DSP
PipeRench[24]	DIL	Greedy Linear	Greedy	Coprocessor	Data-Stream
Pleiades [50][20]	C/C++	Direct	-	Coprocessor	DSP
MorphoSys[25]	C	Manual	Manual	Tight	DSP&MM
KressArray[39]	ALE-X	SA	Neighbor	Loose	General-purpose
GARP[51]	C	Tree-matching	Greedy	Coprocessor	General-purpose
PADDI[20][50]	Silage	By Scheduling	Direct	Loose	DSP
MATRIX[52]	Assembly	Manual	Manual	Loose	General-purpose
ADRES[43]	C	SA-MS	Tight	MM	General-purpose

Table 2-2-Technology mappings in recent years.

Authors	Technology Mapping DFG onto CGRA
Ricardo [32]	Modulo scheduling based on a greedy heuristic.
Naifeng[33]	Mapping problem formulated based on ILP.
Gayatri[34]	Using search for local patterns to place node. A* algorithm proposed in this research to find a good solution and improve that solution to place a node on the CGRA.
Akira [35]	Modulo scheduling algorithm based on resource reservation phase and scheduling algorithm.
Yuanqing [36]	The high-level language program is used to map an application onto CGRA.
Ganghee [37]	ILP consider to Steiner point routing in order to reach an optimal map application onto CGRA. Also, a heuristic algorithm based on scheduling and QEA (similar to the genetic algorithm) are combined to routing aware.
Hartenstein[38]	Simulated annealing performs place and route.
Hyunchul [40]	Graph embedding based on modulo scheduling.
Hyunchul [41]	Edge-centric modulo scheduling, the placement, and routing algorithm are combined.
Chen[42]	The mapping problem is formulated based on the minor graph. Algorithm searches for one model of DFG in MRRG. The placement and routing are combined with modulo scheduling and search.
Mei[43], [44],[28]	A module scheduling algorithm based on simulated annealing is used to placement and routing operands on the CGRA.

2.5 Conclusion

In this chapter, a review of the literature on topics specific to CGRAs along with mapping applications algorithms has been presented.

Several researchers have proposed some algorithms to compile a program to automatically map an application onto CGRA. There are numbers of automatic design and compiling tools developed to exploit the massive parallelism found in applications and extensive computation resources of CGRA. Some researchers utilize structure or GUI-based design tools to manually generate a design that would be difficult to handle big designs. Some other have only focused on Instruction-Level Parallelism that failed to make utilization of the CGRA efficiently and in principle cannot result in higher parallelism than VLIW. However, ILP is limited in scope and fail to make resources utilization efficiently in CGRA. Some recent researchers have concentrated on exploiting Loop-Level Parallelism on CGRA by applying pipelining techniques such as modulo scheduling. Some Module scheduling algorithms have been proposed based on simulated annealing. It begins with a random placement of operation on the FUs of a CGRA, which may not be a valid modulo schedule. Operations are moved between FUs until a valid schedule is achieved. Simulated annealing techniques result in long convergence time for loops that contain a large number of operations. Some researchers have exploited many greedy algorithms. For example, deterministic place and route, heuristic depth-first placement and priority order placement with backtracking. Greedy or heuristic mapping is the option of choice for many mapping problems due to its speed and determinism. However, in the case of complex problems, it may perform poorly. Integer Linear Programming has received attraction due to its clear representation and the possibility to obtain an optimal solution. ILP has not been shown to be feasible for large scale mapping problems.

In the next chapter, modulo scheduling integrates with the placement and routing algorithm to map a DFG nodes onto CGRA. Modulo scheduling attempts to find a pattern of the DFG nodes that can be executed on the same level by the CGRA. The integrated placement function is done through a recursive function that takes the ordered list of nodes from DFG. The order list of nodes is found based on their mobility. The mobility of a node is the difference between the ALAP and ASAP scheduling methods.

CHAPTER 3 PROPOSED SOLUTION BASED ON MODULO SCHEDULING INTEGRATED WITH PLACEMENT AND ROUTING

3.1 Introduction

Implementing applications on reconfigurable computing architectures (RCAs) is an important research topic due to its potential to accelerate a wide range of applications. However, configuring and programming RCAs is a long-standing challenge. In this section, we propose a design methodology to map an algorithm on an FPGA preconfigured with a Coarse-Grained Reconfigurable Architecture (CGRA). At the lowest configuration level, the architecture of the CGRA is elaborated, synthesized, placed and routed by some hardware design specialist using suitable tools. At the highest level, someone who has no particular knowledge in hardware design is, however, able to configure the CGRA to map an algorithm on a mesh of parallel computing and communicating nodes. For medium and large applications, where the number of nodes varies from tens to thousands, getting a good mapping of applications becomes manually intractable. Founded on well-known mapping and routing algorithms that we have tailored to match our context, we propose a design methodology to automate the mapping of applications on a two-level configurable adaptive hardware fabric.

The second part of this section includes the runtime executing of the applications. From the first part of this section, it is known that the applications in existing architecture have reconfigured one time per each compile[3]. In other words, to execute an application with new data, the entire system needs to be synthesized again. Thus, this process is time-consuming to execute an application with different data. The time-consuming is mostly because this architecture is not tailored for runtime executing applications. It should be noted that some of the applications have different data and can be launched at design time. Therefore, it is not efficient to implement these applications by fixed design. On the other hand, the on-line adaptation of application on hardware may permit significant acceleration which results in the flexibility and adaptability of the platform to run time application.

To address these abovementioned problems the predefined architecture CGRA based on FPGA enhances to a new architecture to runtime executing applications. The new architecture is composed of predefined CGRA coupled with processor-based IPs or MicroBlaze. CGRA FIFO banks can be dynamically fed with new data via CDMA. The CDMA helps to reach a high throughput application running on the CGRA. The MicroBlaze enables the user to control, assemble and manage the data flow on FIFO banks. In fact, the programmer can execute the applications at runtime with different data using high abstraction level of the architecture without to be involved in the low-level design.

3.2 Mapping Applications on Two-Level Configurable Hardware

This subsection addresses the question of mapping complex applications in an architecture proposed previously, which is inspired by CGRAs. Such architecture introduced the concept of two different configuration levels. At the lowest level, a hardware design specialist assembles a kind of dedicated CGRA composed of building blocks such as token-based ALUs, FIFOs, and sequencers (the control path of a token machine). The circuit is then synthesized, placed and routed on an FPGA. At the highest level, people with a background in computer sciences program the token machines to implement an application. Such architecture offers many advantages such as reconfigurability, evolution, high (and low) level programming, low-level parallelism exploitation, etc.

3.2.1 Mapping procedure

The mapping procedure takes a data flow graph (DFG) as input, typically the body of a loop. A maximum number of operations are processed in parallel, and a new iteration of the body loop is started as soon as possible, ideally before the completion of the current one, by using software pipelining techniques inspired of the Iterative Modulo Scheduling. Modulo scheduling is modified in a way that placement and routing phases are integrated into it.

Each node of the data flow graph is an operation that must be placed in an ALU for execution. The inputs, intermediate results, and final results are placed in FBs. The routing is done from one FB to other FB among the fabric through the central router of the SALU when an operation is triggered. Eventually, a “void” operation can be launched only to route the data through the SALU. Finally, all the configurations of the TSM are generated according to a placement and routing process of operations and data transfers.

To ease the understanding of the proposed methodology, it will be illustrated with the small data flow graph (DFG) loop body example in Figure 3-1. This DFG is to be mapped on a small subset of the computing fabrics that has 4 ALUs and 2 FBs, as illustrated in Figure 3-2.

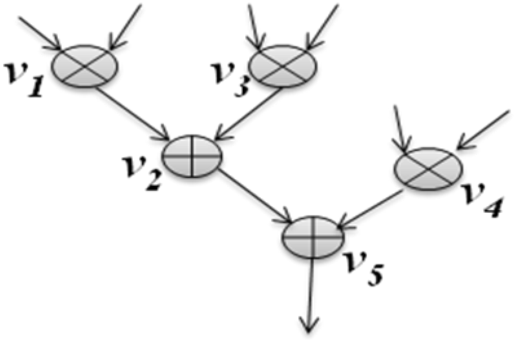


Figure 3-1- Illustrative example input data flow graph loop body

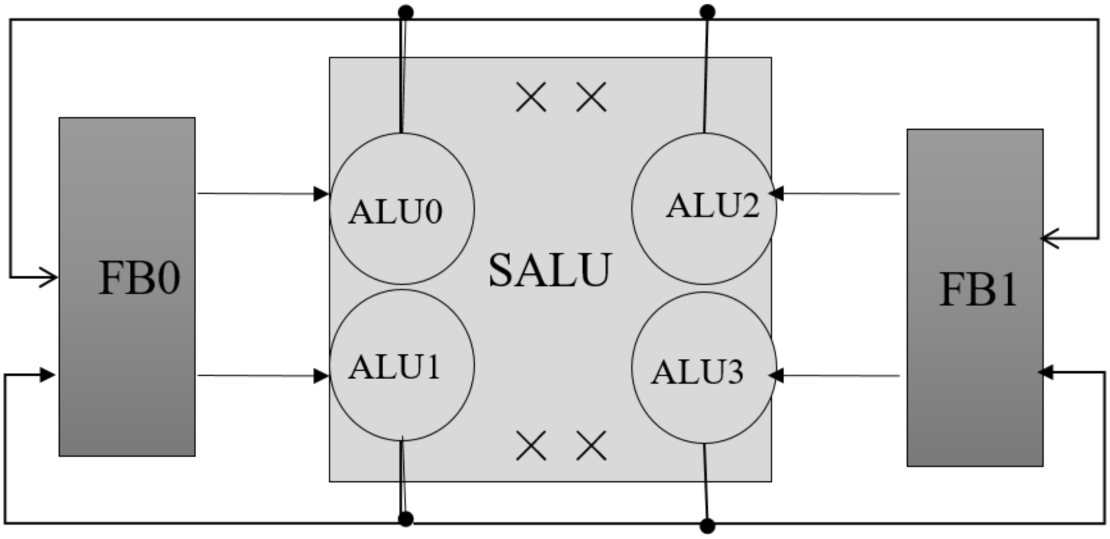


Figure 3-2- Illustrative example target Hardware Architecture Description

In the context of loop implementation, the proposed methodology attempts to minimize the Initiation Interval (II), which is the delay between two successive iterations of the loop body. Ideally, all the loop bodies are processed in parallel (II=0) if there are no dependencies and enough hardware resources. In the worst case, the next iteration of a loop body cannot start before the current one is finished. Function *MII* shows in Figure 3-3 describes how it computes the Minimum

Initiation Interval (MII) from the Hardware Architecture Description (HAD) and the Data Flow Graph Loop Body (DFGLB).

```
Function MII (HAD, DFGLB);
Begin
    1    Nodes= number of vertices in DFGLB;
    2    Size= number of ALU in HDA;
    3    ResMII=Nodes/Size ;
    4    RedMII= Maximum data dependency
    5    II= max (ResMII, RedMII);
    6    Return II;
End
```

Figure 3-3- Calculate the Minimum Initiation Interval (MII)

Data Flow Graph Loop Body and Hardware Architecture Description are specified as the inputs for this *MII* function, where DFGLB is referred to DFG loop and HAD to the property of computing fabric such as FUs specifications, FBs specifications, and the interconnect architecture specification. For this simple example, the HAD is described as pseudo-code in Figure 3-4.

The MII clearly depends on the number of nodes to be computed and available ALUs. In the best case, all the ALUs are active at each clock cycle, and the MII is computed as in line 3 of function *MII* illustrated in Figure 3-3. However, one iteration of the loop body may require some intermediate results computed in the previous occurrence. Such dependencies may increase the MII, which is taken into account in line 4 of function *MII*. The MII is the maximum value issued from those two constraints.

To map DFG nodes on the target architecture, mapping algorithm first orders the nodes by decreasing mobility. The mobility of a node is the difference between the ALAP and ASAP scheduling times as shown in Figure 3-5.

```
Hardware Architecture Description (HAD)
Begin
<Number of FUs= 4>, <Number of FBs=2>
<Function Unit name= “ALU0”>
    <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to>FIFO Bank0
    <out1-connect-to>FIFO Bank0, FIFO Bank1
<Function Unit name= “ALU1”>
    <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to>FIFO Bank0
    <out1-connect-to>FIFO Bank0, FIFO Bank1
<Function Unit name= “ALU2”>
    <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to>FIFO Bank1
    <out1-connect-to>FIFO Bank0, FIFO Bank1
<Function Unit name= “ALU3”>
    <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to>FIFO Bank1
    <out1-connect-to>FIFO Bank0, FIFO Bank1
<FIFO Bank= “FB0”>
     <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to> ALU0, ALU1, ALU2, ALU3
    <out1-connect-to> ALU0, ALU1
<FIFO Bank= “FB1”>
     <port-name=”in1”>,<port-name=”out1”>
    <in1-connect-to> ALU0, ALU1, ALU2, ALU3
    <out1-connect-to> ALU2, ALU3
End
```

Figure 3-4- Pseudo code for Hardware Architecture description example

A null mobility means that the operation should be immediately computed after its parents since it is on the critical path. The higher the mobility, the longer the routing can be without impacting the computation time.

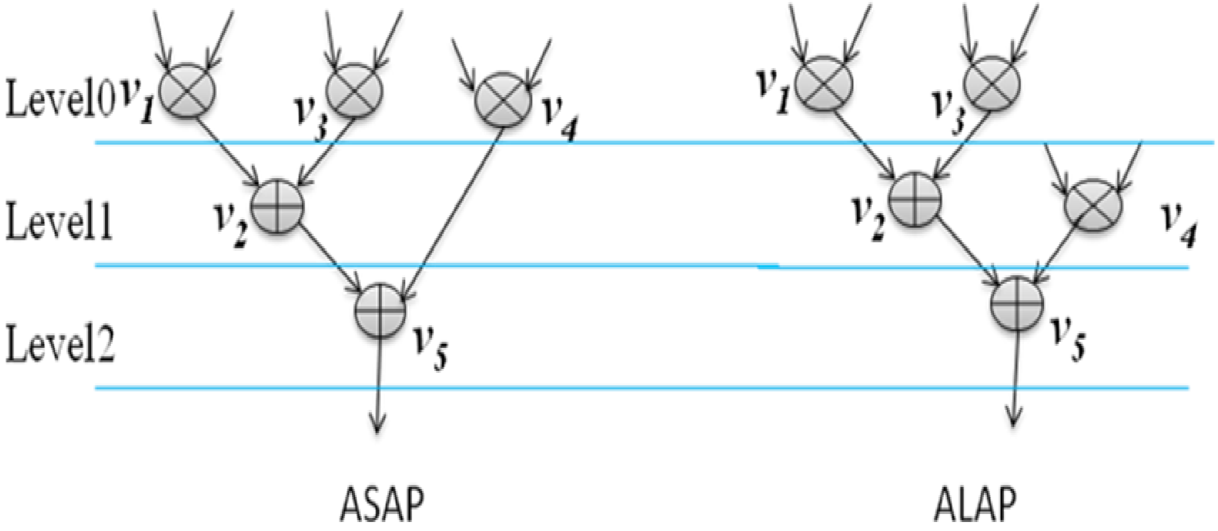


Figure 3-5- ASAP and ALAP scheduling

The ordered of DFG nodes is generated by function *Generate_Ordered_Vertex_List (DFG) which is shown in* Figure 3-6.

```
Function Generate_Ordered_Vertex_List (DFG);
Begin
    For each node $v_i \in DFG$ loop
        //Compute the mobility
        $M_i(v_i) = Level(v_i, ALAP(DFG)) - Level(v_i, ALAP(DFG));$
    End loop;
    List OVL = $\{v_i \in DFG\}$;
    Sort OVL by increasing $M_i$ then by decreasing ASAP level.
Return OVL
End Function;
```

Figure 3-6- create an ordered list of nodes

The ordered list of nodes is sorted by increasing mobility and decreasing ASAP level. In fact, the ordered list of nodes is created based on the critical path in DFG. Thus, the nodes along the critical path that have higher priority should appear earlier.

In our example, the function *Generate_Ordered_Vertex_List (DFG)* illustrated in Figure 3-6 returns the following: M1=1-1=0; M2=2-2=0; M3=1-1=0; M4=2-1=1; M5=2-2=0; OVL = {v1, v3, v2, v5, v4}.

The proposed methodology consists of attempting to map the DFGLB on the HAD with the computed MII. If the attempt fails, the II is incremented until a valid mapping is found. The methodology is detailed in Algorithm I shown in Figure 3-7.

```
Algorithm I: Mapping Loop Body
Inputs: DFG, HAD;
Begin
II= MII (HAD, DFG);
OVL= Generate_Ordered_Vertex_List (DFG):

while (true) {
        MRRG= Gen_Arch_Graph (II, HAD);
         CC=MSPR (OVL, MRRG);
        if (no mapping is found) II++;
        else return CC;
}
End Algorithm
```

Figure 3-7- Main Function of the mapping DFG onto fabric

The inputs for Algorithm I are Initiation Interval (II) along with Ordered Vertex List (OVL). The mapping algorithm will start with MII. For each attempt, the Modulo Routing Resource Graph (MRRG) is constructed for the current II. The MRRG is a graph representing the connectivity resources between the ALUs and the FBs. A configuration is produced for each time slot in the II. An MRRG is illustrated in Figure 3-8 for II=2. All the ALUs and the FBs present in HAD have simply copied in each configuration as well as the routing between them, taking into account that an operand in configuration N produces a result in configuration N+1.

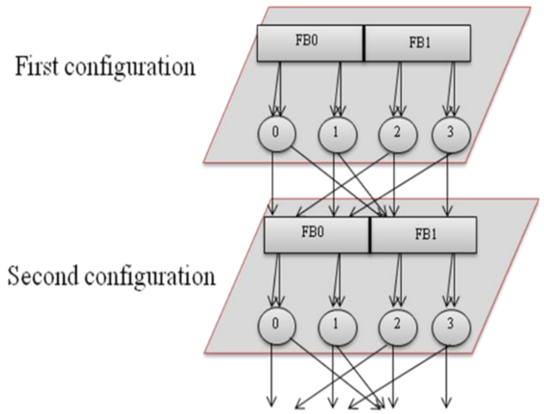


Figure 3-8- Modulo Routing Resource Graph (MRRG) for II=2

MRRG function is shown in Figure 3-9 which returns a Modulo Routing Resource Graph (MRRG) of hardware architecture. This function contains the nodes representing the FUs and the FBs and the edges representing the connectivity among them.

```
Function MRRG (II, HAD);
Begin
    For each vertex $FU_i$, $FB_i \in$ HAD loop
        For j=0 to II-1 loop
            Add vertex $(FU_i)_j$, $(FB_i)_j$ to MRRG
    // add nodes to each II different configurations for CGRA
    For each edge $\in$ HDA loop
    For j=0 to II-1 loop
            Add edges between $(FU_i)_j$ , $(FB_i)_j$ to MRRG
    // add internal edges to each II different configurations for CGRA
    For each $FU_i$, $FB_i \in$ MRRG loop
        For j=0 to II-1 loop
            Add edges between $(FU_i)_{j+II}$ , $(FB_i)_{j+II+1}$ to MRRG

Return MRRG
End Function;
```

Figure 3-9- Generate MRRG

Finally, the mapping function *MSPR* (Modulo Schedule Place and Route) is launched. If a valid mapping is found, the configuration context (CC) is returned as the final result. Otherwise, the II is incremented and a new attempt is launched. The MSPR function is shown in Figure 3-10. This function captures the scheduling plus placement and routing information. The algorithm attempts to find a valid MSPR of DFG onto the MRRG.

```
Function MSPR (OVL, MRRG, CC)
Begin
Pop first node v from OVL that is the successor or the predecessor of an already routed node;
{$p_v$}= placement information of predecessors of v;
{$s_v$}= placement information of successors of v;

{$pr_i$} = Place_ Route (v, $p_v$, $s_v$, MRRG);

if {$pr_i$} is empty, return NULL;

For each $pr_i$ loop {
    Temporarily place and route node v at $pr_i$;
    Recursive call to MSPR (OVL, MRRG, CC);
    if (MSPR is successful) return CC;
    //Backtrack by attempting the other $pr_i$
}
End Function
```

Figure 3-10- Modulo Scheduling Place&Route (MSPR)

The *MSPR* function is the core of the proposed methodology. It attempts to Place and Route (P&R) the nodes *one-at-a-time* based on the ordered list of nodes OVL. Also, each node is mapped onto target if and only if one of its successors or predecessors have already mapped except the first node. MRRG and OVL are used as inputs for *MSPR* function.

The mapping of each node *v* directly depends on its predecessors (p_v) and successors (s_v). Then, for each entrance node, first, the address mapped of pv and sv onto target are determined.

The placement is done through a recursive function that takes the ordered list of nodes OVL that has not been placed and routed yet, the current state of the MRRG and the corresponding state of the CC. The first node is removed from OVL and the function Place_Route returns a set of places and corresponding routes opportunities for that node, taking into account its predecessors that are already placed and routed, and eventually its successors that would already have been placed and routed previously. If there is no place and route opportunity in the given context, the function returns a null value forcing the previous calls to backtrack and try other place and route opportunities at their level. If there are one or several opportunities, they will be tested one by one until the remaining nodes in OVL can be fully placed and routed.

The function Place_Route relies on well-known routing algorithms such as A* or Dijkstra to propose a set of P&R opportunities. Each opportunity has a cost that depends on of the length of the path, the ALU that is already reserved in another configuration, and the affinity of the current node with the rest of the nodes that are already placed.

When each DFG node is mapped onto MSPR, some criteria constraints are performed to satisfy the mapping. For instance, the criteria are the minimum routing function, affinity cost, available resource constraint, and warning-mapping. The minimum routing function indicates the shortest path between the source and destination. The affinity cost determines that the two producers with the common consumers should be mapped as closely as possible to each other. The available resource constraint simply checks the number of available resources of each type (FU, RF) of MRRG to be larger than the number of unmapped DFG nodes. Warning-mapping guarantees that the current mapping can result in a successful mapping in the future or not. If the mapped node cannot pass the constraint test, the algorithm has to choose other opportunities. Finally, the list of opportunities is sorted by increasing cost such that the MSPR function starts by attempting to find a solution with low-cost P&R configurations. The Place_Route function is described in Figure 3-11:

This function (*node_Place_Route*) primarily focuses on routing, and the placement phase occurs during the routing process. As an exception, the first node of OVL is placed in a free place of MRRG. This method clearly improves the compilation time since it eliminates the redundant steps to search an empty position.

When applied to our illustrative example, the MSPR function produces the place and route mapping reported in Figure 3-12. For simplicity of exposition, only the routing edges are presented in Figure 3-12.

```
Function node_Place_Route (v, $p_v$, $s_v$, MRRG)
Use: standard routing algorithms (A* or Dijkstra)
Use: a cost function that depends of the affinity
Begin
If ($p_v$ =null) and ($s_v$= null) Then
        map v into free place of MRRG;
else if ($s_v$= null) Then
        find a set of possible mapping for v near $p_v$
        find a set of routes from $p_v$ to v
else
        find a set of possible mapping for v near $p_v$ and $s_v$
        find a set of routes from $p_v$ to $s_v$ through v
end if
compute the cost of each configuration
sort the list of configurations by increasing cost
Return the list of configurations
End function;
```

Figure 3-11- Place &Route Function

A complete body iteration requires three clock cycles, but a new iteration can already start after only two clock cycles (II=2). The center of the figure represents the steady state (two configurations). In configuration 1, ALU #2 computes the operation v_5 of the previous iteration while ALU #0 and #1 compute operations v_1 and v_3 of the current iteration. In configuration 2, ALU #0 and #3 compute operations v_2 and v_4 of the current iteration. Figure 3-13 shows two different configurations mapped in the CC of the CGRA to support the implementation of this simple example. Those configurations are finally coded into the TSM (one TSM per ALU) where each instruction involves two data token sources (the operands), an operator in the ALU and a target FB to store the result. The following assembly code gives the sequence of instructions for each TSM. OPA$_{v1}$, OPB$_{v1}$ refer to operand A and B of *v1*, respectively.

TSM$_0$: OPA$_{v1}$× OPB$_{v1}$→ OPA$_{v2}$; OPA$_{v2}$+ OPB$_{v2}$→ OPA$_{v5}$.

TSM$_1$: OPA$_{v3}$× OPB$_{v3}$→ OPB$_{v2}$;

TSM$_2$: OPA$_{v5}$+ OPB$_{v5}$→ Result;

TSM$_3$: OPA$_{v4}$× OPB$_{v4}$→ OPB$_{v5}$;

As expected, TSM0, related to ALU0, is configured twice (once for v_1 and once for v_2).

3.2.2 Propose an Assembly Code for Computing Fabric

The token state machine (TSM) is acted as a program memory for the system that contains the required instruction for an application implemented by computing fabric. Each instruction is composed of 47 bits. For some applications that require hundred or thousand instructions, it is extremely inconvenient to make all instructions manually. There is a lack of strong tools that can take defined programs written by high-level language and creates the assembly code associated with the computing fabric.

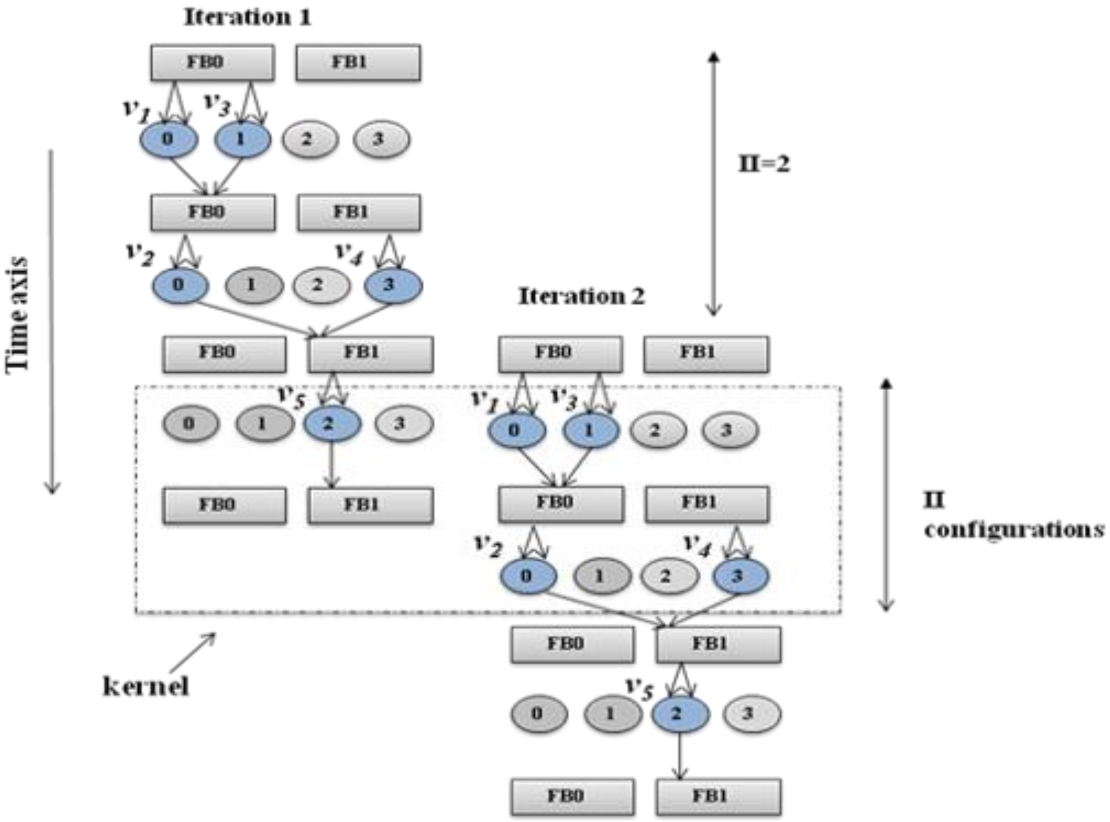


Figure 3-12-MSPR result for our illustrative example Corresponding II=2

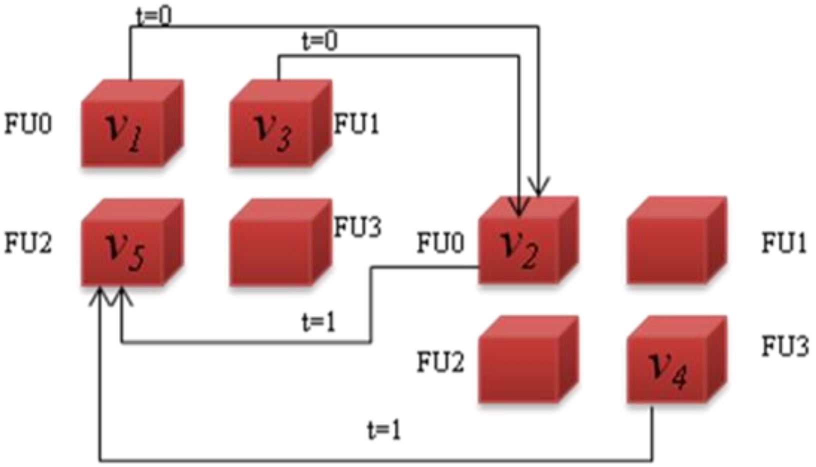


Figure 3-13-Two configurations (left and right) to map our sample DFG

The Python language is used To create the automatic assembly code for our computing architecture; The user can write a program in a specific order based on the instruction format of computing fabric. The Python language tool compiles the code and generates the assembly code worked with the computing fabric. In Appendix A, each word of the proposed assembly language is explained which is used to generate coding of instruction.

3.2.3 Generates the instruction bits for TSMs using WinTim32 Application

WinTim32 is a fully functional Meta assembler and 1.5-pass assembler based on the Texas Instruments Meta-Assembler known as "TIM”[53]. An assembler translates human-readable symbolic assembly language programs into binary machine language that can then be loaded into the computer’s memory. A Meta assembler allows the user to define the instruction formats for any machine. Once the user defines instruction formats, the Meta assembler then serves as an assembler. A Meta assembler is useful for people that are designing a new computer since they can use it to assemble programs for the new computer without writing a new assembler from scratch. WinTIM is a Meta assembler used to convert the symbolic strings of a source program to machine language code.

The full C++ source code is also available for the WinTim32 program. The source code has been modified based on our requirements to generate the constant VHDL files. This VHDL file contains all the instruction bits memory assigning to computing fabric.

To produce this VHDL file, the first step is to define all of the instruction formats and mnemonic names. WinTIM and definition tables process the definition file are produced by the assembly process.

The second step is to assemble the assembly language program for the new instruction formats using the instruction definition tables produced in the definition phase. In this step, the Meta assembler functions as a conventional assembler as it converts symbolic assembly language in a *.src file into binary machine language.

Figure Appendix B 1 shows the instruction format with mnemonic names based on our computing fabric. The source code of WinTim32 is modified based on our requirements, then the given definition and VHDL file are generated. This VHDL files for one tile of computing fabric is shown in Figure Appendix B 2. Therefore the VHDL file contains one TSMs with eight small tsm such as tsm0-7. The tile of the computing fabric and the number of small tsm are defined as parameters, i.e., the user can change them based on the size of the computing fabric, and the number of tsm require in one tile. Binary instructions bits in VHDL files are matching with their assembly codes.

3.3 Runtime Executing Applications on Parallel Computing and Communicating Nodes

The second part of this section intends to enhance the existing CGRA to exploit the benefits of the runtime applications.

The proposed architecture contains a 2-D mesh computing fabric coupled with two Microblazes. Microblaze is a virtual soft core microprocessor based on 32-bit the Reduced Instruction Set Computer (RISC) architecture. The RISC architecture is optimized for implementation in Xilinx FPGA in which the instruction and data buses are separated from each other. The Microblaze processor architecture balances execution performance against implementation size. It is highly customizable and supports a lot of configuration options.

These processors act at the overall runtime management, resemble, data flow and controlling of data tokens through parallel computing mesh (called computing fabric). In addition, these processors provide the performing IO with the external world via UART and other available interfaces.

To execute an application on preconfigured CGRA on FPGA at runtime, the CGRA should be enhanced to reload the contents of the FIFO banks dynamically in computing fabric. CGRA in new architecture is defined as a custom peripheral Intellectual Properties (IP). This IP is attached to the Processor Local Bus (PLB) where the Microblaze is defined as a microprocessor. In order to execute the applications at runtime with possible high throughput, a Central Direct Memory Access (CDMA) is used. Using the CDMA, the CGRA's FIFO banks can dynamically feed with new data token. Then, it leads to support runtime execution applications through CDMA by feeding embedded FIFO banks inside the CGRA.

It is shown that the central router network in SALU has a bottleneck that is a strong limitation to implementing large data flow graphs. In fact, the SALU can produce eight data tokens per cycle but is only able to route four data tokens in the central router. This problem will be addressed and fixed by duplicating the central router and adapting the decoders accordingly. We will then discuss the new architecture to support runtime execution applications.

3.3.1 Modified Fabric:

In the new version of the fabric, some deficiencies have been resolved such as 1) internal buffer inside the SALU, 2) central router network 3) the mesh topology is modified to the Torus-mesh topology in order to utilize the edges boundary of fabric.

The prior internal buffers inside the SALU could store and load single data token whereas, in the updated version of the computing fabric, internal buffers can store and load a streaming data. As presented in Section 2.3.7, the SALU is composed of three elements; ALUs, decoders, and a Central Router Network. It should be mentioned that the SALU defines generating, calculation and the routing of the data tokens. The router could convey four data tokens to four distinct destinations (up, down, right and left) at the same time. This structure may affect the performance of the throughput of the system. To increase the throughput, the structure of the router should be modified to send at least two data tokens to each side simultaneously. According to the number of ALUs on each side, the structure of the SALU is modified. The possible solution is that the router is duplicated two times and each pair of the decoder with its ALU on each side is connected to the one router. Figure 3-14 shows the block diagram of the new network router.

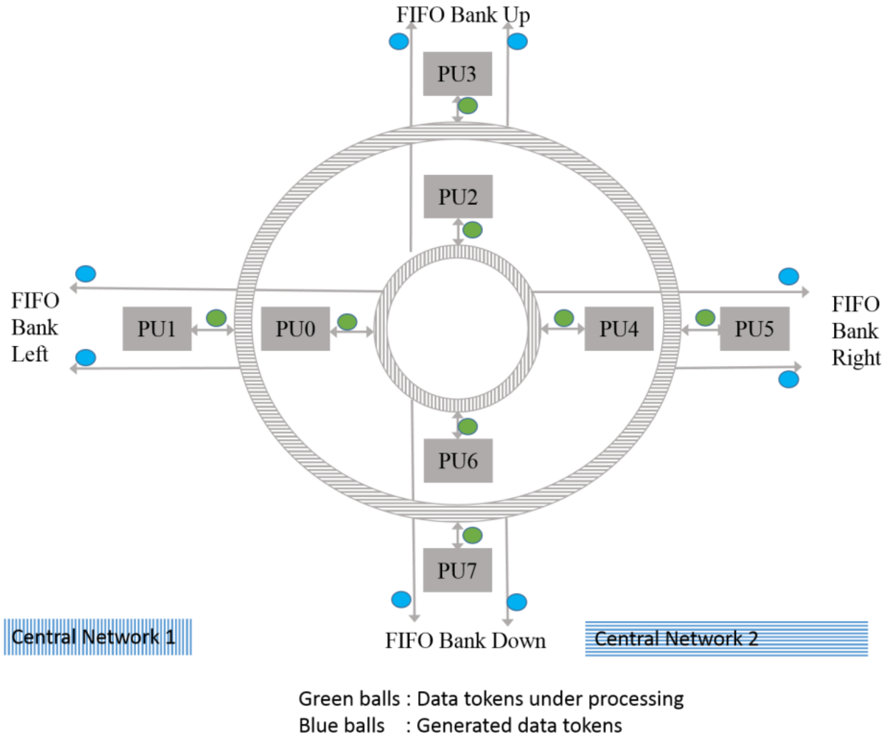


Figure 3-14- New Network Router.

It is seen from Figure 3-14 that the network can accept eight tokens per cycle and return the same number of tokens to their destinations. Figure 3-15 illustrates merging data from two routers to each FIFO bank. Each FIFO bank contains 15 small FIFOs. It should be taken into account that the single FIFO is accessible only by one router at each clock cycle.

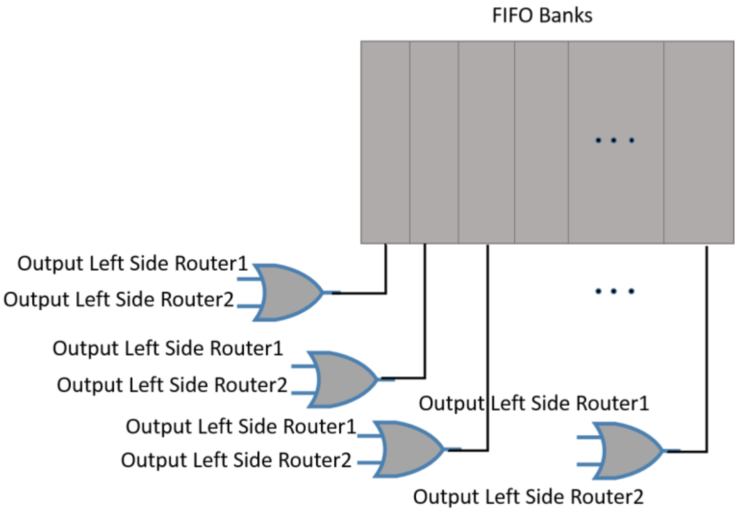


Figure 3-15- Merging Data from two routers to a FIFO Bank

The assigning data tokens from routers to other FIFO banks (UP, Right and downsides) are following the same roles as shown in Figure 3-15.

3.3.2 Runtime Executing Application on the Computing Fabric

3.3.2.1 Why Microblaze is used

In this section, a Microblaze would be coupled with the parallel computing and communication nodes via Processor Local Bus (PLB). The flexibility of Microblaze gives this opportunity to the users to balance the required performance of their application against the area logic cost of the soft processor. Another advantage of using Microblaze is its ability to integrate customized Intellectual Property (IP) cores. The Microblaze with IP can dramatically improve acceleration in software execution time due to the algorithms being executed as parallel in hardware instead of executed sequentially in software. One of the restrictions to Microblaze is the nature of RISC processor architecture. Modern RISC processor includes two inputs and one output execution unit as ALU.

It is known that the Microblaze is a pipeline architecture that is composed of three or five pipeline stages such as Fetch, Decode, Execute, Memory and Write Back. In general, each stage takes one clock cycle to be completed. Thus, to complete an instruction, it takes three or five clock cycles to be completed.

Based on the nature of the RISC processor architecture, the maximum throughput is restricted. In another word, if an application needs a high rate of the calculations, the RISC architecture is not useful. The modern applications that require several instructions are not suitable for this kind of the architecture since they need more execution units. The user IP core is an alternative to solve this problem to execute more than one instruction per cycle. However, attaching user IP to the processor may have its restriction such as throughput of transferring data between IP and Microblaze.

Computing fabric as IP attached to the MicroBlaze gives this opportunity to the users to execute two different applications on the same platform. In addition, the user can separate the critical and sequential application codes from each other that could be implemented on CGRA and MicroBlaze, respectively.

3.3.2.2 Attaching User IP to the Microblaze via PLB

The parallel communicating nodes or fabric is implemented as a user IP, and it is attached to the Microblaze via PLB. A channel is required between computing fabric and Microblaze to access to each FIFO bank in the fabric. Figure 3-16 illustrates the general view of communicating between computing fabric and Microblaze. The processor shown in Figure 3-16 acts at the overall run-time management, resemble data flow and controlling of data tokens in FIFO banks in parallel computing mesh.

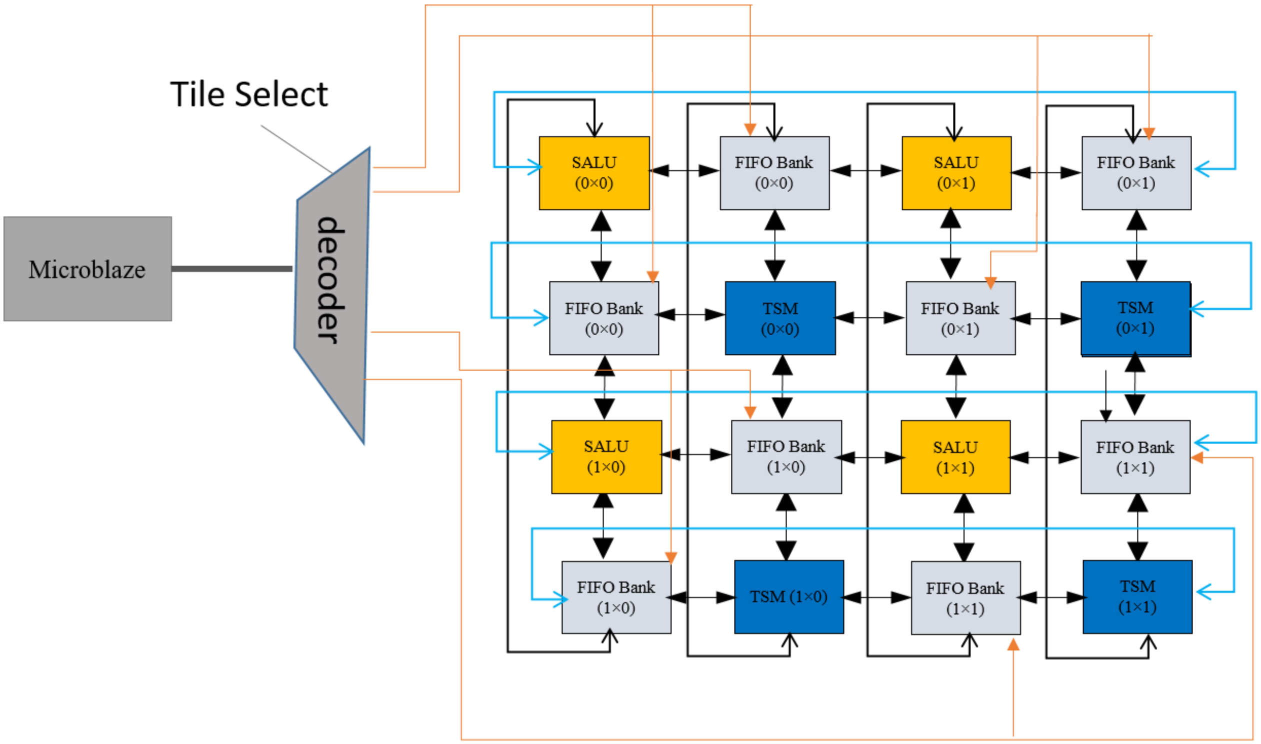


Figure 3-16 - Microblaze coupled with the fabric

The size of the fabric in Figure 3-16 is 2×2 tiles, and each tile has a specific identification (ID) from 1 to 4, where 1 is referred to the tile “00” and 4 to the tile “11”. Each tile of the fabric has 32 FIFOs that are placed in two different FIFO banks. The decoder in Figure 3-16 is responsible for connecting right data to tiles using the ‘*tile select*’.

There are two methods to transfer the data to the fabric. The first method is shown in Figure 3-17. In this method, the *write enable* (*Wr_en*) signal associated with each FIFO and their data are separated. This method consumes more clock cycles since it needs two instructions per each data; one for choosing the FIFO (Wr_en) and other for assigning data (Data_In). In addition, the read process by Microblaze from fabric contains two links as shown in Figure 3-17.

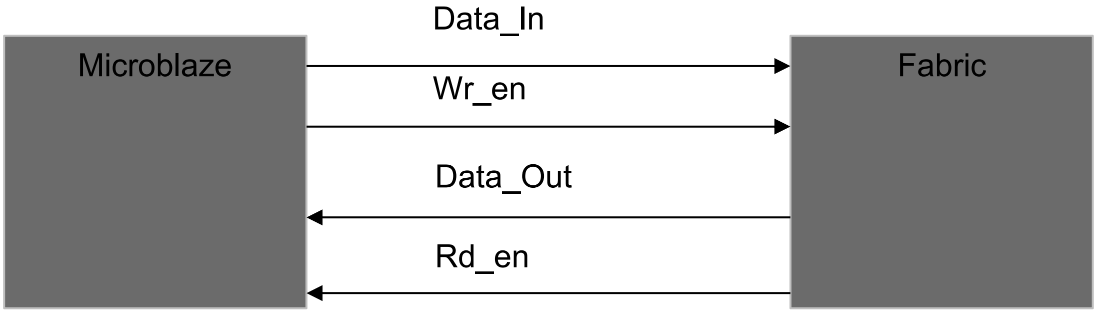


Figure 3-17- Double Link between Microblaze and Fabric for writing and reading process

The second method used as a single link between Microblaze and fabric is shown in Figure 3-18. In this method, one link for writing to the fabric contains the both “Wr_en” and “data” signals. In addition, in the read process, one link is enough to read data from fabric.

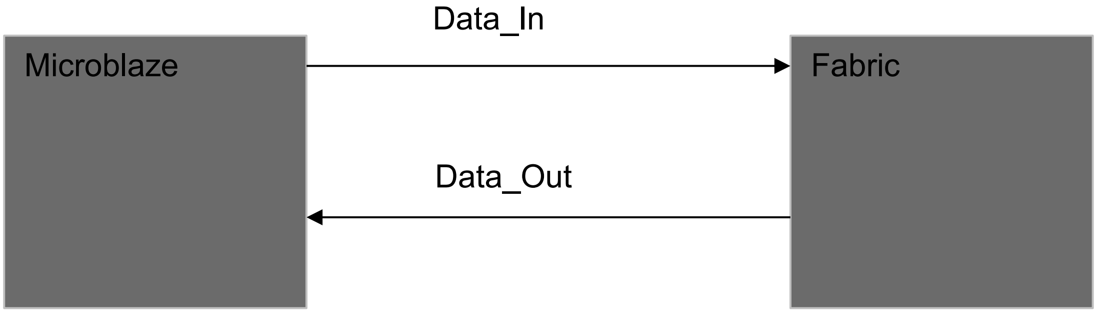


Figure 3-18- Single Link communicating between Microblaze and fabric for writing and reading process

3.3.2.3 Microblaze and streaming data

In this section, a bottleneck while the objective is to transfer streaming data between Microblaze and computing fabric over PLB is explained and a possible solution to this problem is proposed.

As I mentioned in 3.3.2.2, the computing fabric is attached to the MicroBlaze as a co-processor. There is an already established single link between MicroBlaze and computing fabric. This link is useful when the burst transfer is not the objective and is more efficient when the single beat data is the final point. In fact, the C code does not support the burst data between the processor and its attached co-processor. For example Table 3-1 gives the equivalent assembly code according to C code on the left side, where “*Pointer to Fabric”* is pointed to the computing fabric address.

It is seen from this table that the C compiler for this simple code uses three instructions. For each line of code, a co-processor (computing fabric) receives the data within 12 clock cycles, i.e., MicroBlaze needs 12 clock cycles to send a single beat communication data over the PLB.

Table 3-1-Equivalent assembly code for a simple C code

Pointing to address of fabric	Assembly code

*(Pointer to Fabric) = 207; //data [0];
*(Pointer to Fabric) = 198; // data [1];
	107 *(Pointer to Fabric) = 207;
	00000780: lwi r3, r19, 36
	00000784: addik r4, r0, 207
	00000788: swi r4, r3, 0
	108 *(Pointer to Fabric) = 198;
	0000078c: lwi r3, r19, 36
	00000790: addik r4, r0, 198
	00000794: swi r4, r3, 0

Computing fabric architecture is based on data-driven which means that the operations start as soon as the operands are available in the FIFOs that contain the operands. Data token results are produced in one clock cycle if and only if data token inputs are available in their ports. Therefore if each input data token is fed within 12 clock cycles, then its result is generated in the next clock cycles by the computing fabric. It should be noted that the application is configured in low latency with high-throughput. In fact, the computing fabric goes to standby until the next data arrives. Given this, computing fabric is not a good candidate to support runtime execution due to its big latency and low throughput as well as its cost space. In other words, computing fabric capability is blocked by its channel to wait for receiving the new data tokens.

3.3.2.4 Direct Memory Access (DMA):

The main reason to choose the DMA as an option is to overcome the abovementioned problem for PLB burst transfer. Using DMA, it is possible to read data directly from memory by the user Peripheral IP- core or co-processor. In this work, the central DMA (CDMA) core, provided by the

Xilinx, is used. The user defines the length of the burst data transfer through CDMA. The maximum throughput using CDMA to transfer the data between source and destination address is half data per cycle. With this rate of the transmission, computing fabric can generate each data tokens in two clock cycles.

Another advantage of CDMA transfer over the burst is its decoder module in Figure 3-16. This decoder could be implemented in the software instead of the hardware via VHDL code. This decoder in hardware needs 64 registers corresponding to each FIFO in one tile of the computing fabric. Among these registers, 32 are used to write into FIFOs and the remaining 32 registers for reading from FIFOs. A drawback to these registers is that the decoding address corresponds to each FIFO consumes the large logic area which can increase the timing critical path. However, this decoding module can be transferred to the software. Table 3-2 gives the implemented software decoder corresponding to each FIFO. In order to write onto FIFOs, the tile number should first be defined at the beginning.

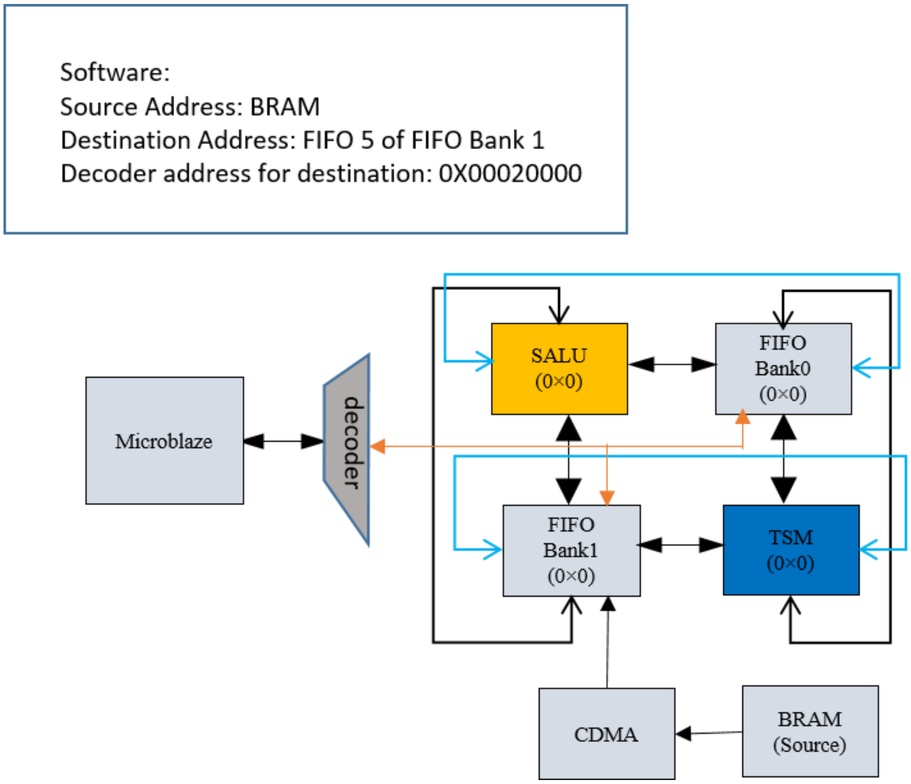
Figure 3-19 shows an example of the burst data transfer by CDMA which is separated from Microblaze. It is seen from this figure that the destination address is the FIFO5 of FIFO bank1. In the first step, MicroBlaze selects the corresponding address to the destination FIFO. Then, when the data arrives, it should be connected to the destination FIFO which is already determined by the Microblaze.

Table 3-2- Implemented software decoder to select each FIFOs in one Tile of computing fabric

FIFO Bank UP		FIFO Bank Down	
FIFO_0: 0X00000001	FIFO_8: 0X00000100	FIFO_0: 0X00001000	FIFO_8: 0X00100000
FIFO_1: 0X00000002	FIFO_9: 0X00000200	FIFO_1: 0X00002000	FIFO_9: 0X00200000
FIFO_2: 0X00000004	FIFO_10: 0X00000400	FIFO_2: 0X00004000	FIFO_10: 0X00400000
FIFO_3: 0X00000008	FIFO_11: 0X00000800	FIFO_3: 0X00008000	FIFO_11: 0X00800000

Table 3-2- Implemented software decoder to select each FIFOs in one Tile of computing fabric (continued)

FIFO Bank UP		FIFO Bank Down	
FIFO_4: 0X00000010	FIFO_12: 0X00001000	FIFO_4: 0X00010000	FIFO_12: 0X01000000
FIFO_5: 0X00000020	FIFO_13: 0X00002000	FIFO_5: 0X00020000	FIFO_13: 0X02000000
FIFO_6: 0X00000040	FIFO_14: 0X00004000	FIFO_6: 0X00040000	FIFO_14: 0X04000000
FIFO_7: 0X00000080	FIFO_15: 0X00008000	FIFO_7: 0X00080000	FIFO_15: 0X08000000



Software:
Source Address: BRAM
Destination Address: FIFO 5 of FIFO Bank 1
Decoder address for destination: 0X00020000

Figure 3-19- A simple example of access to a FIFO by CDMA

Once the data is placed in predefined FIFO by CDMA, it spreads out among computing fabric. The software is responsible for spreading out data to other parts of the computing fabric based on its demand. The advantage of this method consumes a less amount of the logic resources. Also, the process of data developing to the entire part of computing fabric is easier by the software. An example shows the efficiency of this method in Figure 3-20.

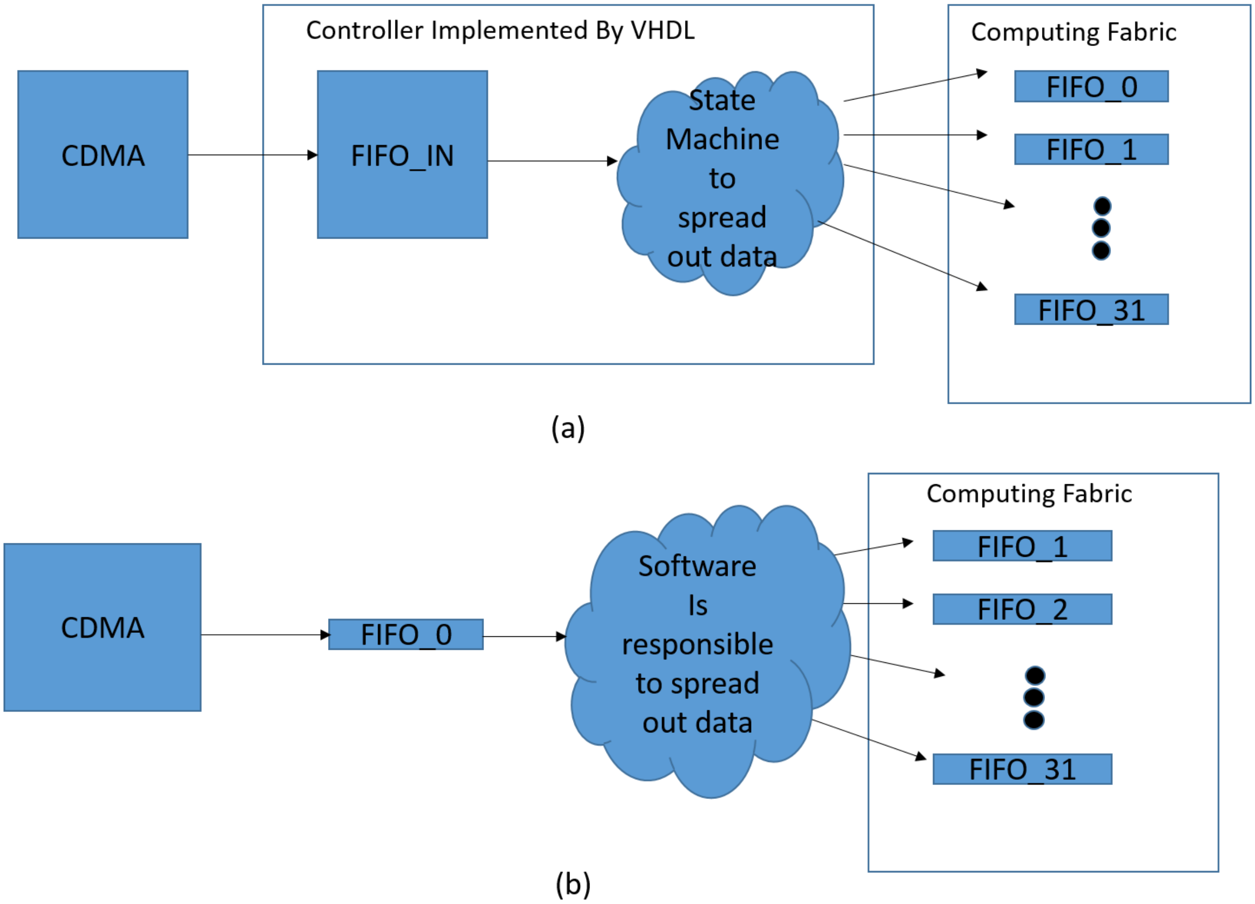

Controller Implemented By VHDL
CDMA
FIFO_IN
State Machine to spread out data
Computing Fabric
FIFO_0
FIFO_1
FIFO_31
(a)
CDMA
FIFO_0
Software Is responsible to spread out data
Computing Fabric
FIFO_1
FIFO_2
FIFO_31
(b)

Figure 3-20- a) spread out data by Hardware b) spread out data by the software

Figure 3-20 (a) shows the data spreading out the procedure by the hardware. To this end, a controller should be attached to receive data and transmit it to the computing fabric. This controller contains a FIFO to buffer the input data arriving from CDMA and a complicated state machine. The number of the states in state machines depends on the number of application's inputs. If the application has 32 inputs, the state machines need more logics. For example, for three inputs the state machines are configured as three states. Each state connects one input to a FIFO. Each state is transferred to other after in each clock cycle. The transferring of states are continued until to send all inputs to the FIFOs.

Figure 3-20 (b) shows a similar process as in part (a), yet the system does not need any controller. The user can decide if CDMA should connect to FIFO and defines its code to spread out data from FIFO source to the other FIFOs.

3.3.3 General View of Runtime Executing Application Hardware

The proposed architecture contains a 2-D mesh computing fabric coupled with two Microblazes. The high-level block diagram of the prototype system is illustrated Figure 3-21. It is known that some applications exhibit multi-thread parallelism which can enhance the overall performance.

These processors provide the performing IO with the external world via UART and other available interfaces. The dual-core microprocessors available in this architecture are independent. It is seen from this figure that the link to each microprocessor is equal to have access to the FIFO banks. In this case, through one microprocessor writes the data tokens into FIFO banks and others read the data tokens from FIFO banks.

The CDMA gets data from the source address. In this case, it is BRAM attached to the PLB. CDMA transmits the data to the destination address (computing fabric). It should be noted that both the source and destination address should be accessible by the CDMA. It means that when CDMA is attached to the PLB, then their source and destination devices must be attached to the PLB not anywhere else.

Figure 3-22 shows the screen shot of the implemented proposed architecture using Embedded Development Kit (EDK) tools to a runtime execution of the applications. It can be seen from this figure two Microblazes along with CDMA are defined as masters. Computing fabric also is attached as a co-processor to the PLB. BRAM_img is the source address which the data are stored there, and CDMA can access to it.

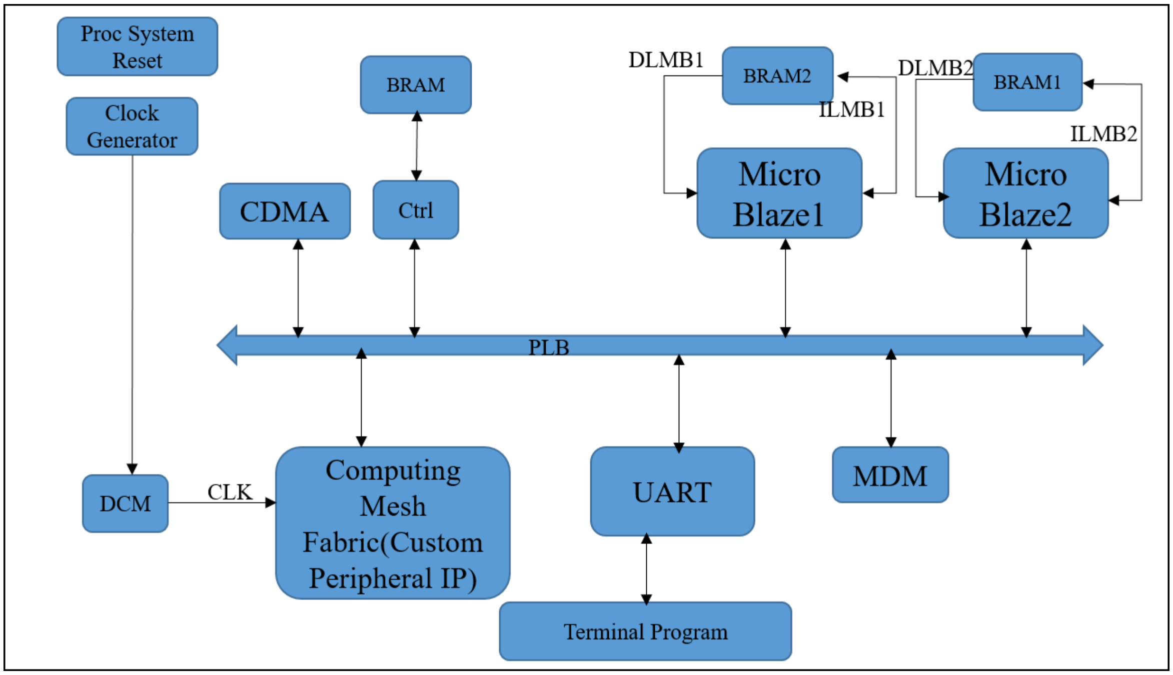
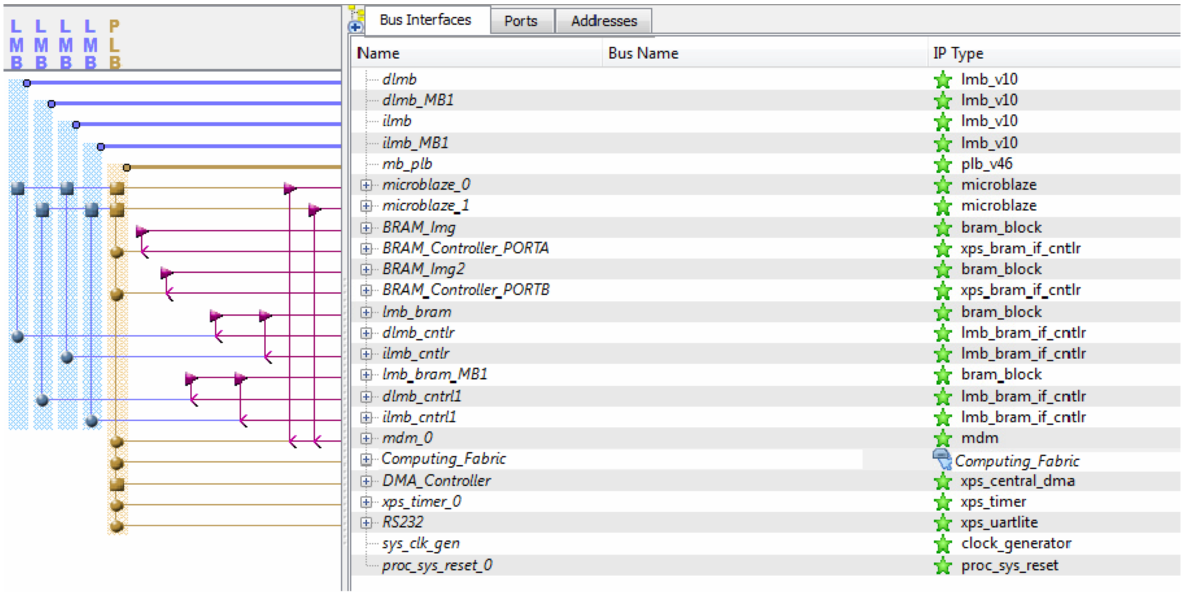


Figure 3-21- Runtime Execution Applications Architecture



Name	Bus Name	IP Type
dlmb		lmb_v10
dlmb_MB1		lmb_v10
ilmb		lmb_v10
ilmb_MB1		lmb_v10
mb_plb		plb_v46
microblaze_0		microblaze
microblaze_1		microblaze
BRAM_Img		bram_block
BRAM_Controller_PORTA		xps_bram_if_cntlr
BRAM_Img2		bram_block
BRAM_Controller_PORTB		xps_bram_if_cntlr
lmb_bram		bram_block
dlmb_cntlr		lmb_bram_if_cntlr
ilmb_cntlr		lmb_bram_if_cntlr
lmb_bram_MB1		bram_block
dlmb_cntrl1		lmb_bram_if_cntlr
ilmb_cntrl1		lmb_bram_if_cntlr
mdm_0		mdm
Computing_Fabric		Computing_Fabric
DMA_Controller		xps_central_dma
xps_timer_0		xps_timer
RS232		xps_uartlite
sys_clk_gen		clock_generator
proc_sys_reset_0		proc_sys_reset

Figure 3-22- Screen shot of the Implemented architecture to support runtime execution of the applications using EDK

3.4 Conclusion

The first part of this section introduced a design methodology to automate the process of mapping applications onto an FPGA-based computing fabric developed in previous work. Made of parallel computing and communicating nodes, the fabric resembles CGRA but it is data-driven, and it has an original communication network that deserves a custom place and route methodology. The proposed methodology, founded on the Modulo Scheduling algorithm, converges rapidly towards a solution with a low initiation interval. The foundation of the methodology is a list scheduling algorithm that takes into account the mobility of a node, the affinity with other nodes and the already placed and routed predecessors and successors.

The second part of this section introduced a new model of high abstraction level runtime execution of the application. The proposed architecture is shown to make the abstracts away from FPGA to high-level design. Easy-to-use and flexibility of the proposed architecture have provided an opportunity for those applications requiring to be dynamically executed. However, the proposed architecture provides a simple and fast method for programmers enabling them to reload new data to their applications on hardware.

.

CHAPTER 4 EXPERIMENTAL RESULT

4.1 Introduction

Implementing the proposed architecture for mapping on runtime reconfigurable is not simply due to imposing many challenges and surprises. In the case of the proposed solution, the important aspect is to prove the use of the resources against the performance. The resources could be measured by suitable tools such as Planahead or ISE Xilinx tools, for an application implemented onto our computing fabric. Also, the performance may be comparable while an application is implemented in two distinct hardware architectures. To analyze the performance of the system, especially in digital circuit design, several parameters may be involved. One which may affect the performance is how long a design takes to reach to a final point of its test and implementation. It should be noted that some applications need a designer with a good knowledge in hardware design. It is known that developing an application on hardware; it is not always an easy task. For example, to implement an application in VHDL, one may need a strong background in the hardware design roles as well as time and budget. Once the designer decides to develop an application or reconfigure it with new data, all previous attempts from the beginning of a design should be taken into account.

According to the existing challenges, this section provides an experimental result of the computing fabric where the designer does not need to be involved in the low-level design, yet can implement the application onto FPGA platform.

The result in this work is obtained by using a manual implementation of the proposed mapping algorithm onto computing fabric. The main objective is to let FPGAs become the mainstream computing hardware. This may be achievable in a future work when the proposed methodology will be automated, optimized and compared with other approaches in the field of the high-level configuration of FPGAs.

Finally, the proposed architecture is elaborated, synthesized, placed and routed on Xilinx Virtex-5 family using suitable tools. In addition, the runtime matrix multiplication result shows that the proposed architecture can lead to high throughput to perform runtime applications.

4.2 Simulation and manual mapping application on computing fabric

The results presented in this section are based on the new architecture that enables eight data tokens to be produced and routed at each clock cycle. The topology of computing fabric is based on Torus -Mesh topology.

Our methodology is manually applied to two tangible examples taken from the DSP world. The first example is a matrix multiplication algorithm (the RGB-YCbCr transform), which is illustrated in Figure 4-1. Since the inputs are high-fanout nodes, these nodes are advantageously duplicated to enhance the MII. In this case, the R, G, and B should be repeated since they are used three times by the restricted ALUs on one side of SALU. Otherwise, the II can be increased, and it will affect the maximum throughput of the circuit.

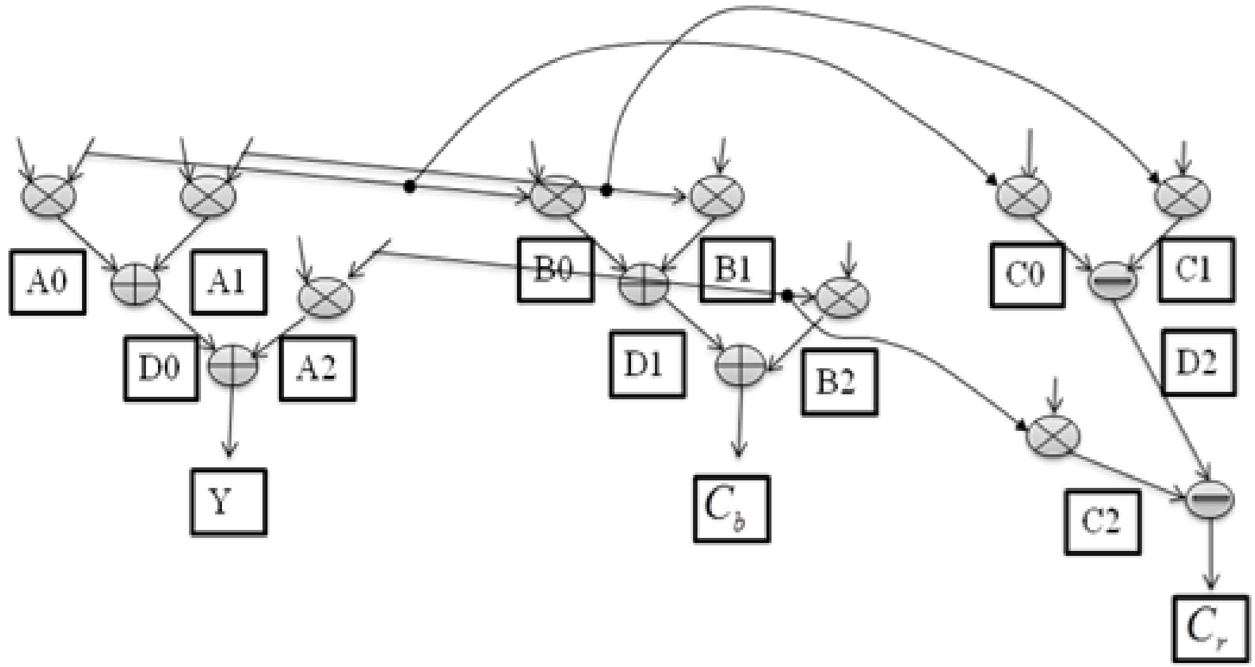


Figure 4-1- RGB-YCbCr DFG application

Two SALUs (SALU00 and SALU01) and their associated FB are considered as the target architecture (2-D torus illustrated in Figure 4-2).

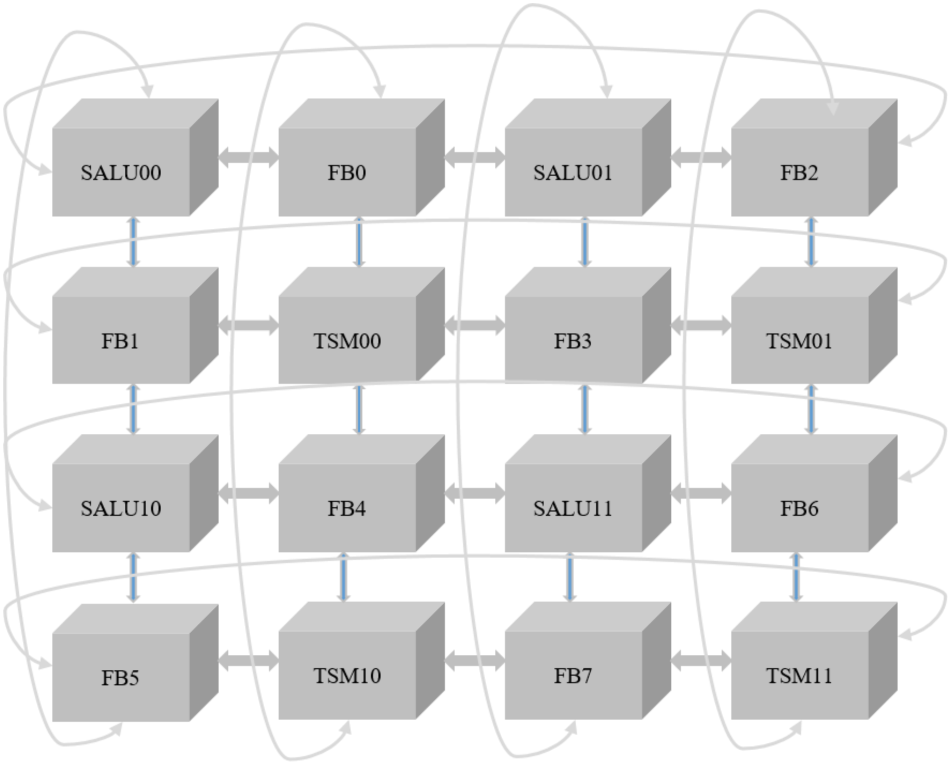


Figure 4-2-Target architecture (SALU00, SALU01, and their FBs only)

The ordered list of nodes is computed, and the MII is found to be one. The MRRG is constructed and illustrated in Figure 4-3, where the Figure 4-2 is flattened to one row for the convenience of drawing. Nevertheless, the topology remains the same.

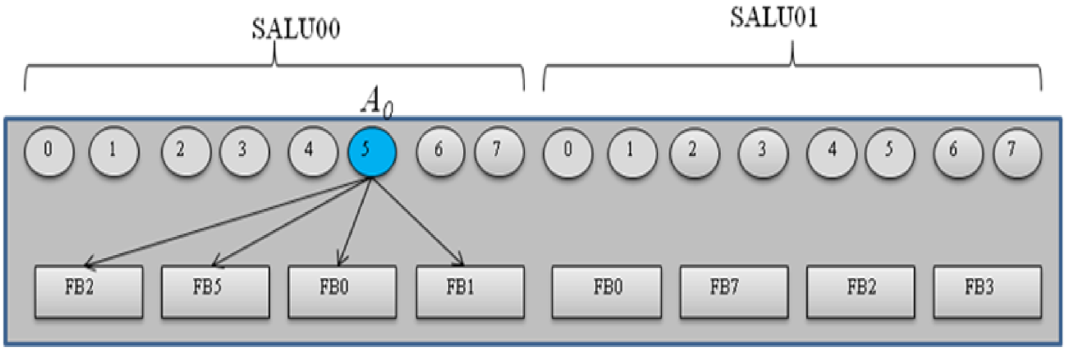


Figure 4-3- MRRG of the target architecture composed of two SALU

In the next phase, the algorithm creates a Modulo Schedule Place and Route. MSPR is a subgraph of MRRG capturing the scheduling, placement, and routing information. The algorithm attempts to find a valid MSPR of DFG onto the MRRG. MSPR is a 3-D demonstration of P&R related to the time axis. The first node is A_0, which is mapped onto SALU00: ALU5. As can be seen from Figure 4-3, the A_0 output can be routed to FB_0, FB_1, FB_2 or $FB_{5,}$ where each route has a cost. In this example, FB0 has a high cost since it is connected to SALU00: ALU5, which is already used to compute A_0. The other nodes have the same cost. The algorithm will randomly choose FB_2 to route the A_0 result. The remaining FB are kept as potential candidates if the algorithm has to backtrack at a given time.

In order to reduce the search space, the algorithm attempts to restrict it to distance of predecessors or successors from specific node. If there exists no available place in this specified address, the algorithm should increment search space by one. In this step, the algorithm checks the possibility for each routed to determine whether the selected destination creates a warning map in the future of processing mapping or not. For example, FB_0 is a high-cost place, because A_1 will be mapped in the future in this place. If the output of A_0 is mapped into FB_0, in this case, a conflict will happen, and II is increased. Thus, the throughput decreases. In another word, for each FB only two ALUs are assigned via a SALU.

The next node that reads from the ordered list is D_0 since it is a direct successor of A_0. It is mapped to ALU_0 because A_0 has already been mapped to FB_2 during the previous step. This process is continued until all the nodes are mapped onto the MSPR. The resulting mapping is shown in Figure 4-4. For simplicity, only the relevant routing edges are drawn. It can be seen from this figure that each ALU is not used more than once, leading to a unary II, even if the full loop body requires three cycles to complete. The final configuration context of the mapping is shown in Figure 4-5.

When the configuration context of given application is prepared, the assembly code is written for each tsm according to FIFOs. The number of tsm should be equal to the number of ALUs. Then for this example, we would have 15 programmed tsms to determine the routing data tokens from the sources (FIFO banks) to destinations (FIFO banks) through two SALUs.

In order to implement this application onto the fabric, the written assembly code by the Python is used as an input file for WinTim. Thus, WinTim can generate the instruction binary file for embedded program memory within the fabric.

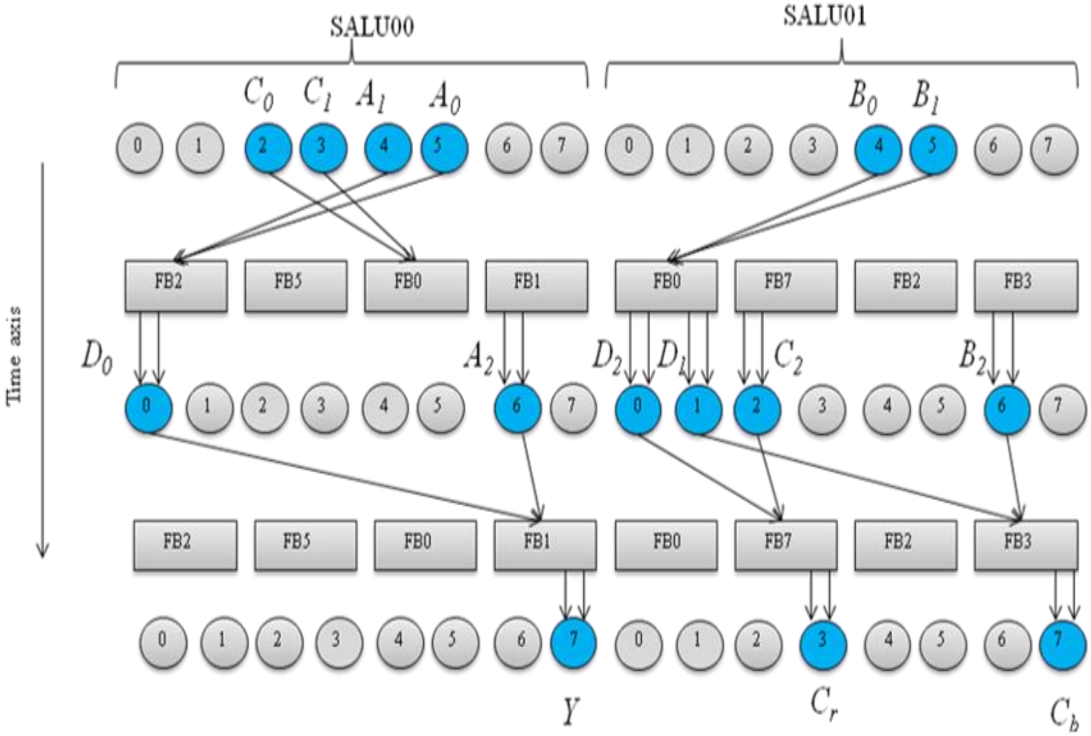


Figure 4-4-MSPR of Y output of RGB-YcbCr DFG application (II=0).

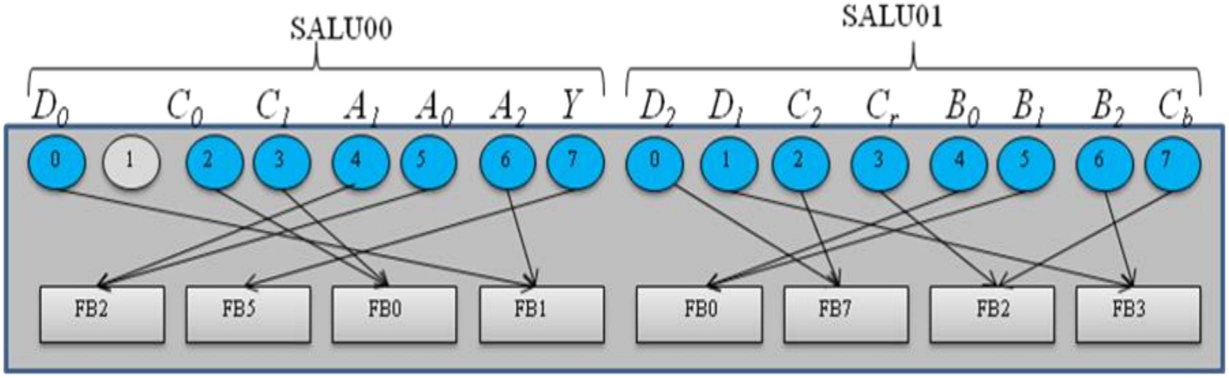

SALU00
SALU01
D_0 C_0 C_1 A_1 A_0 A_2 Y
D_2 D_1 C_2 C_r B_0 B_1 B_2 C_b
0 1 2 3 4 5 6 7
FB2 FB5 FB0 FB1 FB0 FB7 FB2 FB3

Figure 4-5- Final configuration context

In order to validate our configuration, the fabric has been manually configured and simulated in VHDL. The achieved configuration of the application (RGB-YC_bC_r) by the ISIM Xilinx tools is then simulated. A screenshot of the simulation is illustrated in Figure 4-6. The application is tested for several consecutive inputs data Red, Green, and Blue. As can be seen from this figure, results demonstrate that the 15 ALUs receive process and produce one operation at each clock cycle on average, leading to the announced unary II. The results observed at the outputs (Y, Cb, and Cr) have also been validated.

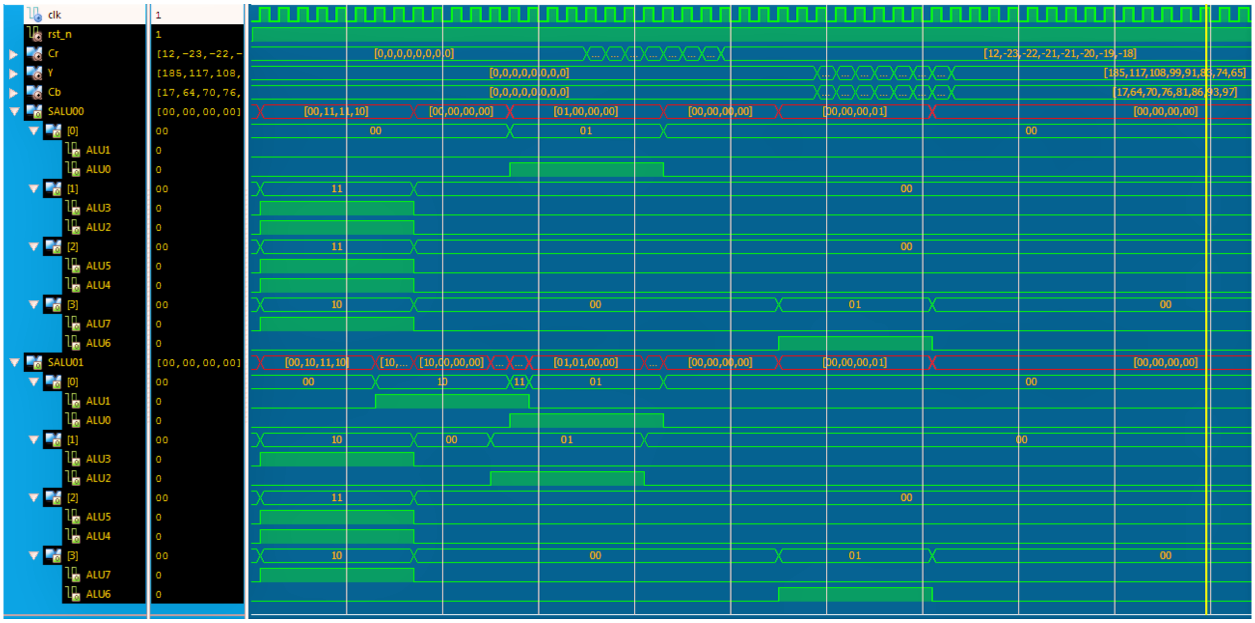


Figure 4-6- Simulation Result Based given configuration context for RGB-YCbCr application

The second example is a 4-point FFT, as shown in Figure 4-7. In this example, we attempt to map the DFG in a single tile (8 ALUs). Figure 4-8 illustrates the MSPR obtained at the end of the proposed methodology. ALU #0 and #5 have been used twice, leading to II=MII=2. Figure 4-9 shows the result simulation FFT4-point onto fabric along with II=2. ½ decreases the obtained throughput for this example.

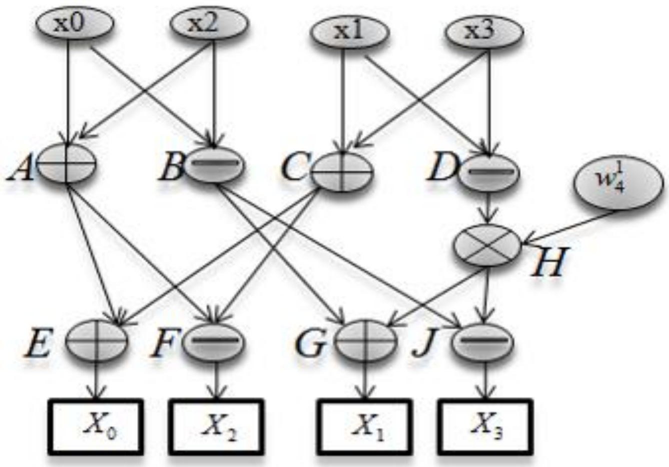


Figure 4-7- DFG of a 4-point FFT

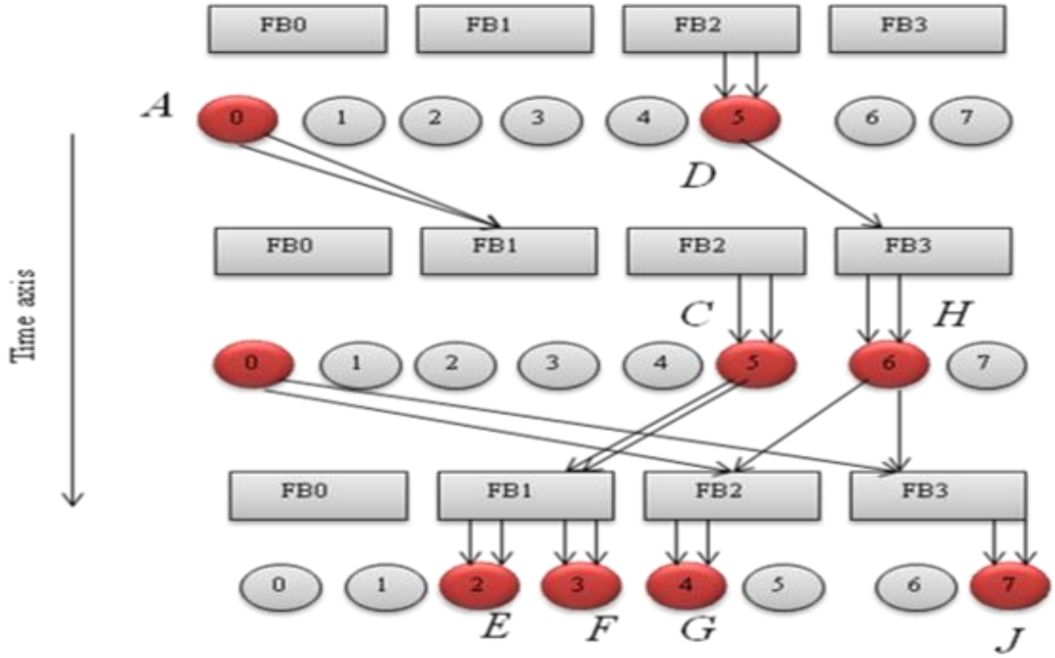


Figure 4-8-MSPR of a 4-Point FFT (II=2)

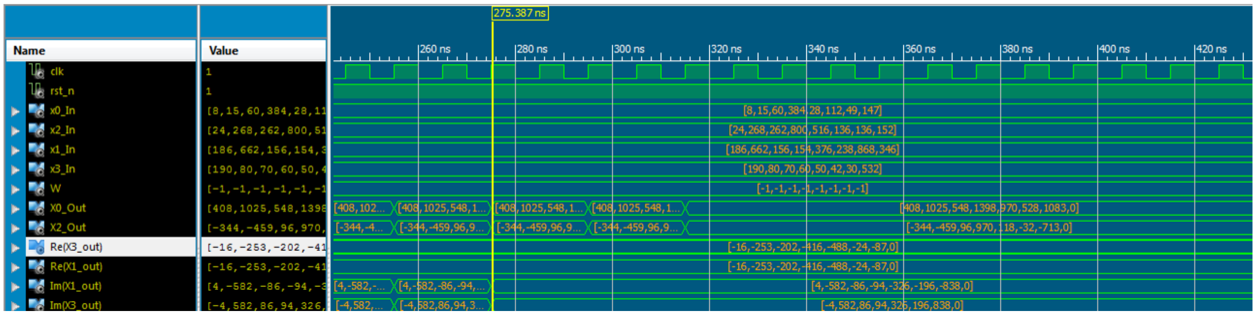


Figure 4-9-FFT 4-point Simulation (II=2)

4.3 Runtime RGB-YCbCr Transform on the CGRA

This section provides the obtained result to runtime executing an application. The idea to runtime executing for the computing fabric is discovered when the system attempts to be configured with new data tokens. The first version of the computing fabric architecture is not tailored for runtime execution applications. It should be noted that some of the applications can be configured at design time. Thus, it is not efficient to implement these applications by a fixed design.

The proposed architecture in order to runtime executing applications is elaborated, synthesized, placed and routed on FPGAs Virtex-5, using XILINX EDK 14.5 tools.

Two different ways implement the runtime matrix multiplication algorithm (RGB-YCbCr transform) on the proposed fabric. First, only the RGB-YCbCr transform is implemented by the MicroBlaze. This implementation is totally based on the software, and the application is configured in software. The software provides the new data which is predefined in a Block RAM. Then, the application is configured with new data periodically after each result. However, the achieved throughput may not be efficient for some applications that need to get a result in every few clock cycles. The nature of the RISC processor as well as transferring C code to assembly code by C compiler result in obtaining low throughput results.

The second method of RGB-YC$_b$C$_r$ transform is implemented by computing fabric which the MicroBlaze feeds its data inputs. Computing fabric is implemented as a custom peripheral IP attached to the PLB. It has been described in former that the computing fabric is a full pipeline parallel computing architecture. Architecture is data-driven, which means that the operations start as soon as the operands are available in the FIFOs that contain the operands. The token data results are produced in one clock cycle if and only if data token inputs are available in their ports.

Based the architecture's properties, the generated data tokens depends more on the arrival data token rates stored in their FIFO banks. Then, application throughput can be directly determined by a channel that can receive and transfer data tokens from computing fabric.

Figure 4-10 shows the block diagram showing that the MicroBlaze is responsible for transferring application's inputs to the computing fabric. The application (RGB-YCbCr) is already well configured in computing fabric. Microblaze is ready to transfer data from memory and feed it into a FIFO in computing fabric. Data spreading out for application is performed by computing fabric.

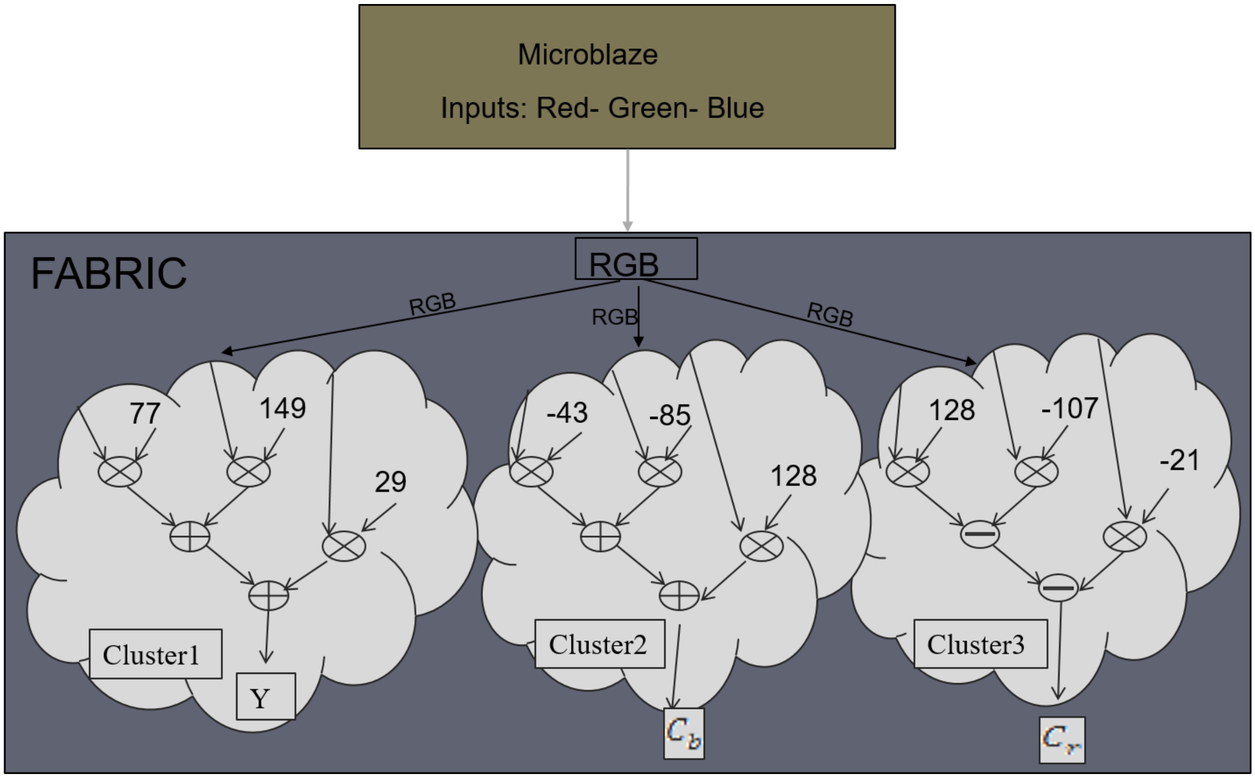


Figure 4-10- Block Diagram of Data transferring from Microblaze and computing fabric

Figure 4-11 shows the clock cycles captured by the chip scope required for MicroBlaze to transfer three R-G-B set to the fabric. However, this transferring is not efficient since each transferring takes 12 clock cycles, i.e., the computing fabric is free more than 90% of its time. The end point of this result, shown in Figure 4-11 determines for eight different R-G-B, MicroBlaze needs 304 clock cycles to feed computing fabric.

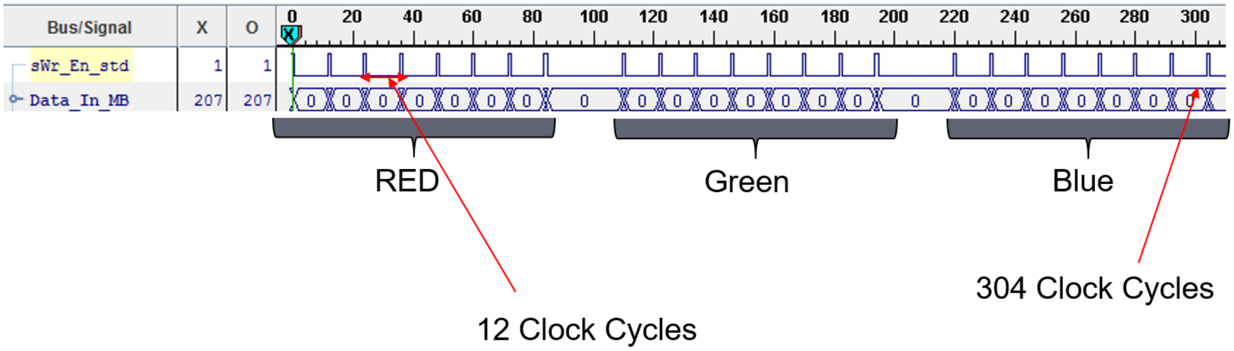


Figure 4-11- Capturing data transfer by the chip scope

The data transferring between Microblaze and its custom peripheral IP over PLB could be defined as a bottleneck. Thus, we have decided to change the channel and use the CDMA to access the data tokens directly by the computing fabric. In fact, by using CDMA, the MicroBlaze is bypassed, and computing fabric can obtain data tokens directly from the block RAM. To this end, CDMA takes the controlling of the PLB to transfer the whole predefined data length from BRAM to the computing fabric in burst mode. Figure 4-12 shows the eight words transfer by CDMA to the fabric. As can be seen from this figure, in each clock cycle one input is launched from memory and transferred to the fabric. With this data rate, computing fabric can generate data token in every cycle. Maximum throughput is then obtained with this created channel by CDMA.

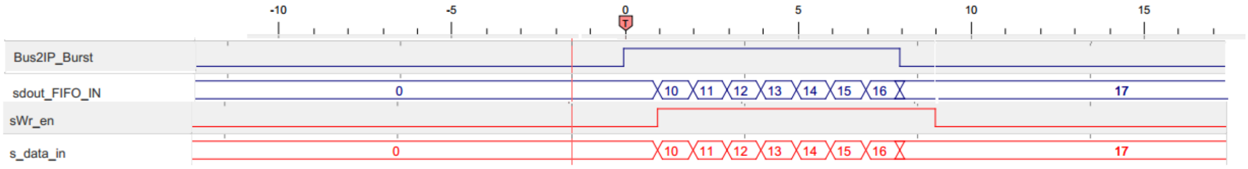


Figure 4-12- Eight words transferring By DMA to fabric

The data length is defined in the software and CDMA can transfer this length of data to the computing fabric. In fact, data includes three words R-G-B as inputs for implementation RGB-YCbCr application on the computing fabric. The application is well configured on computing fabric based on proposed methodology, i.e., in every clock cycle, three output Y, Cb, and Cr result if and only if the data tokens arrive. Table 4-1 gives the number of clock cycles required to perform RGB-YCbCr application in runtime configuration.

Table 4-1- require clock cycle to perform RGB-YCbCr application (length is 340 *3 R-G-B).

Platform	Consuming Clocks Cycle	Number of operations
Microblaze	107436	15300
Computing fabric using DMA	2172	15300

The data length is arbitrary selected as 1020 words. Microblaze is taken 107436 clock cycles to perform 1020*15 operations while computing fabric consumes 2172 clock cycles to do the same operations. As can be seen from this table, The total time to send our 1020 data elements is now 2172 clock cycles, which takes into account the time to configure the DMA. In this configuration, a computer scientist with a pre-synthesized FPGA configured with our proposed architecture would obtain a speedup of nearly 50.

Table 4-2 shows the resources utilization by the proposed architecture on target device platform.

Table 4-2- Resources Utilizing by computing fabric 2×2.

	#Slice Registers	#Slice LUTs	#Occupied Slices	DSP48E Slices
Computing Fabric	23784(24%)	37357(38%)	14153(58%)	42(33%)

The computing fabric is occupied more resources against the Microblaze while it extremely improves the throughput. As another advantage, we can consider that the FIFO’s capacitance in computing fabric is eight words. The data length is 1020 words for R-G-B that is transferred into the computing fabric’s FIFO. Then, FIFO’s data almost 128 times is reconfigured on runtime without the computing fabric needs to be synthesized, placed and routed again.

4.4 Conclusion

As a conclusion, the computing fabric can do runtime execution of applications along with the obtains high throughput result. In order to overcome to bandwidth bottleneck of the Microblaze, we made a channel using DMA. The channel provides a high-speed data transferring between computing fabric which is introduced as custom peripheral IP attached to the PLB. The proposed architecture is shown to make the abstracts away from FPGA to high-level design. Easy-to-use and flexibility of the proposed architecture have provided an opportunity for those applications requiring to be dynamically executed. It has been shown that most of the existing architectures that support runtime execution of applications, could only be implemented on a specific platform with the user having knowledge about the roles of hardware design. However, the proposed architecture has provided a simple and fast method for programmers enabling them to reload new data to their applications on hardware by high abstraction level implementation through software. The obtained result for RGB-YCbCr application is demonstrated a significant improvement factor of 98% compared to the MicroBlaze.

CHAPTER 5 CONCLUSION AND FUTURE WORK

In this thesis, we have introduced a design methodology to automate the process of mapping applications onto an FPGA-based computing fabric. The employed fabric has been developed using parallel computing and communicating nodes. Although this fabric resembles CGRA, it is data-driven and has an original communication network that deserves a custom place and route methodology.

By advancing the FPGA devices, the available resources increase. In other words, the number of logic elements, logic blocks, and block memories may increase. While these advancements provide more flexibility and capacity for designers, handling these resources will make new issues. The resources may adversely affect the systems if careful considerations are not taken into account. Thus, the use of available resources and time are two crucial factors in developing algorithms on FPGA platform using low-level hardware programming languages such as VHDL and Verilog.

In this thesis, we have synthesized, placed and routed a coarse-grained reconfigurable architecture FPGA according to the Allard’s method. The CGRA provides a solution for designers to utilize the current FPGA’s resources more effectively. The CGRA consists of multiple cores running in parallel. The proposed method automates the process of mapping applications on the CGRA based on the Modulo Scheduling algorithm. It is shown that the proposed solution converges rapidly with a low initiation interval.

Preliminary results, obtained by a manual implementation of the proposed methodology, led to the best possible initiation intervals with a high occupation rate of the computing resources in the fabric. It has been encouraging since we have obtained better performances than the current state of the CGRA and associated tools. It has been shown that the computing fabric precludes any complexity of the hardware implementation, yet maintaining flexibility available in the software.

However, it is known that the existing architecture for computing fabric has reconfigured one time per each compile[3]. In other words, to execute an application with new data, the entire system needs to be synthesized again. Thus, this process is time-consuming to execute an application with different data. This is mostly due to the fact that this architecture is not tailored for runtime executing applications.

It has been shown that the proposed architecture provides a simple and fast method for programmers enabling them to reconfigure their applications on hardware by high abstraction level implementation through software. It has also been shown that easy-to-use and flexibility of the proposed architecture provide an opportunity to applications requiring to be dynamically executed.

CGRA in new architecture has defined as a custom peripheral Intellectual Properties (IP). This IP is attached to the Processor Local Bus (PLB) where the Microblaze is defined as a microprocessor.

In order to overcome the possible bottleneck in transferring streaming data between Microblaze and computing fabric over PLB, the DMA has been proposed to execute streaming applications on the CGRA.

It should be noted that the application has configured in low latency with high-throughput onto computing fabric by the automated mapping methodology. In fact, the computing fabric goes to standby until the next data arrives. In view of this, computing fabric was not a good candidate to support runtime execution due to its low throughput. In other words, computing fabric capability was blocked by its channel to wait for receiving the new data tokens.

In order to execute the applications at runtime with possible high throughput, a Central Direct Memory Access (CDMA) has used. Using the CDMA, the CGRA’s FIFO banks are able to feed dynamically with new data token. Then, it leads to supports runtime execution applications through CDMA by feeding embedded FIFO banks inside the CGRA. The obtained result shown a significant improvement to execute runtime application on the computing fabric. For example, a FIFO almost 128 times has recharged with new data through CDMA on runtime without the computing fabric needed to be synthesized, placed and routed again.

Based on this new architecture along with its automated mapping methodology the computing fabric is more accessible to support advanced features such as runtime parallelism processing. In addition, the automated mapping application gives the advantage of the simplicity and flexibility of software development.

CHAPTER 6 FUTURE IMPROVEMENT

In order to improve this work some improvements are listed in the following as future works such as:

1. The proposed method may be automated and optimized in the field of the high-level configuration of FPGAs. The proposed method is becoming an appealing option for designers since they focus on the application-oriented in the software rather than hardware design roles.
2. We have attached the computing fabric in the new design to the PLB due to that the PLB is supported by a different prototype of Xilinx family. Since as we know, to support stream data processing the Advanced eXtensible Interface (AXI) is more suitable. But, it is supported for a new generation of Xilinx products family. Then, the other improvement could transfer the computing fabric from PLB to the AXI to reach higher throughput applications.
3. It is known that the existing architectures have reconfigured one time per each compile. In other words, to reconfigure the architecture, the entire system needs to reconfigure again. Consequently, this process is time-consuming to execute an application with different configurations. This is due to the fact that this architecture is not tailored for run-time reconfiguration. In order to runtime reconfiguration of architecture, the token state machine will also change by the microprocessor in the runtime instead of the design time. On the other hand, the on-line adaptation of application on hardware may permit significant acceleration resulting in the flexibility and adaptability of the platform to run time application. The online adaptation of application on hardware may be achievable in the future works.

BIBLIOGRAPHY

[1] Intel, "Intel® Xeon Phi^{TM} Product Family," 2015. [Online]. Available: http://www.intel.com/ content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[2] Nvidia, "Tesla K40 and K80 GPU Accelerators for Servers | NVIDIA," 2015. [Online]. Available: http://www.nvidia.com/object/tesla-servers.htm.

[3] M. Allard, P. Grogan, Y. Savaria, and J. David, "Two-level Configuration for FPGA : A New Design Methodology Based on a Computing Fabric," *Circuits Syst. (ISCAS), 2012 IEEE Int. Symp.*, pp. 265–268.

[4] R. A. Walker and S. Chaudhuri, "Introduction to the Scheduling Problem," *IEEE Des. Test Comput. 12.2*, p. 60–69., 1995.

[5] H. S. Luka Daoud , Dawid Zydek, "A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing," *Adv. Syst. Sci.*, pp. 483–492, 2014.

[6] B. Liu, D. Zydek, H. Selvaraj, and L. Gewali, "Accelerating high performance computing applications: Using CPUs, GPUs, hybrid CPU/GPU, and FPGAs," *Parallel Distrib. Comput. Appl. Technol. PDCAT Proc.*, pp. 337–342, 2012.

[7] R. H. Landau, "A Beginner's Guide to High–Performance Computing." [Online]. Available: http://www.shodor.org/media/content/petascale/materials/UPModules/beginnersGuideHPC/moduleDocument_pdf.pdf.

[8] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing Performance Benchmarks among CPU , GPU , and FPGA," *E-project-030212-123508*, pp. 1–113, 2012.

[9] B. Schauer, "Multicore Processors–A Necessity," *ProQuest Discov. Guid.*, no. September, pp. 1–14, 2008.

[10] AMD, "AMD Opteron^{TM} 6000 Series Platform," 2016. [Online]. Available: http://www.amd.com/en-us/products/server/opteron/6000.

[11] Tilera, “TILE-Gx Processor Family – Product Brief,” *Development*, pp. 1–2, 2011.

[12] Y. Zhai, E. Mbarushimana, W. Li, J. Zhang, and Y. Guo, “Lit: A high performance massive data computing framework based on CPU/GPU cluster,” *Proc. - IEEE Int. Conf. Clust. Comput. ICCC*, 2013.

[13] J. Fang, A. Varbanescu, and B. Imbernón, “Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi,” *Iwbbio*, pp. 579–588, 2014.

[14] AMD, “AMD FirePro^{TM} S10000 High-Density, High-Performance Server Graphics.” [Online]. Available: https://www.amd.com/documents/FirePro_S10000_Data_Sheet.pdf.

[15] S. Settle, “High-performance Dynamic Programming on FPGAs with OpenCL,” *Ieee-Hpec.Org*, 2013.

[16] Altera, “STRATIX 10 FPGA AND SOC.” [Online]. Available: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html?utm_source=Altera&utm_medium=press_release&utm_campaign=gen_10.

[17] Xilinx, “UltraScale Architecture and Product Overview,” 2016. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[18] “Programming of Coarse Grain Reconfigurable Platforms.” [Online]. Available: http://xputers.informatik.uni-kl.de/papers/publications/Nageldinger4.pdf.

[19] R. Hartenstein, “Coarse grain reconfigurable architectures,” *Proc. ASP-DAC 2001. Asia South Pacific Des. Autom. Conf. 2001 (Cat. No.01EX455)*, pp. 1–6, 2001.

[20] A. Chattopadhyay, “Ingredients of adaptability: A survey of reconfigurable processors,” *User Model. User-adapt. Interact.*, vol. 2013, 2013.

[21] A. L. Bjorn De Sutter, Praveen Raghavan, “Coarse-Grained Reconfigurable Array Architectures,” *Handb. Signal Process. Syst. SE - 6*, pp. 553–592, 2010.

[22] P. F. Carl Ebeling, Darren Cronquist, “RaPiD—Reconfigurable pipelined datapath,” *Field-*

Programmable Log. Smart Appl. New Paradig. Compil., pp. 126–135, 1996.

[23] R. W. Hartenstein, J. Becker, and R. Kress, “High-performance computing using a reconfigurable accelerator,” vol. 8, no. November 1995, pp. 429–443, 1996.

[24] S. Copen and R. Reed, “PipeRench : A Reconfigurable Architecture and Compiler,” *Comput. 33*, vol. 4, pp. 70–77, 2000.

[25] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, “MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications,” *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, 2000.

[26] T. Miyamori and K. Olukotun, “REMARC : Reconfigurable Multimedia Array Coprocessor,” *IEICE Trans. Inf. Syst. 82*, vol. 2, pp. 389–397, 1999.

[27] K. Choi, “Coarse-Grained Reconfigurable Array: Architecture and Application Mapping,” *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 4, pp. 31–46, 2011.

[28] B. Mei, “ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix,” *F. Program. Log. Appl.*, vol. 53, no. 9, pp. 61–70, 2003.

[29] G. Ansaloni, P. Bonzini, and L. Pozzi, “EGRA: A coarse grained reconfigurable architectural template,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 6, pp. 1062–1074, 2011.

[30] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. 2009.

[31] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, “Specifying and compiling applications for RaPiD,” *Proceedings. IEEE Symp. FPGAs Cust. Comput. Mach. (Cat. No.98TB100251)*, 1998.

[32] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, “A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures,” *Proc. - 2013 Int. Conf. Embed. Comput. Syst. Archit. Model. Simulation, IC-SAMOS 2013*, pp. 188–195, 2013.

[33] N. Jing, W. He, and Z. Mao, “Resource Constrained Mapping of Data Flow Graphs onto Coarse-Grained Reconfigurable Array,” *23rd IEEE Int. SOC Conf. IEEE*, pp. 260–265, 2010.

[34] G. Mehta, K. K. Patel, N. Parde, and N. S. Pollard, “Data-driven mapping using local patterns,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 32, no. 11, pp. 1668–1681, 2013.

[35] A. Hatanaka and N. Bagherzadeh, “A modulo scheduling algorithm for a coarse-grain reconfigurable array template,” *Proc. - 21st Int. Parallel Distrib. Process. Symp. IPDPS 2007; Abstr. CD-ROM*, 2007.

[36] Y. Guo and P. M. Heysters, “Mapping applications to a coarse grain reconfigurable system,” *Asia-Pacific Conf. Adv. Comput. Syst. Archit. Springer Berlin Heidelb.*, pp. 221–235, 2003.

[37] G. Lee, K. Choi, and N. D. Dutt, “Mapping multi-domain applications onto coarse-grained reconfigurable architectures,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 30, no. 5, pp. 637–650, 2011.

[38] G. Lee, S. Lee, K. Choi, and N. Dutt, “Routing-aware application mapping considering steiner points for coarse-grained reconfigurable architecture,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5992 LNCS, pp. 231–243, 2010.

[39] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, “Mapping Applications onto reconfigurable KressArrays,” *Int. Work. F. Program. Log. Appl.*, pp. 385–390, 1999.

[40] H. Park, K. Fan, M. Kudlur, and S. Mahlke, “Modulo Graph Embedding : Mapping Applications onto Coarse-Grained Reconfigurable Architectures,” *Proc. 2006 Int. Conf. Compil. Archit. Synth. Embed. Syst. ACM*, pp. 136–146, 2006.

[41] H. Park, K. Fan, S. a. Mahlke, T. Oh, H. Kim, and H. Kim, “Edge-centric modulo scheduling for coarse-grained reconfigurable architectures,” *Proc. 17th Int. Conf. Parallel Archit. Compil. Tech. - PACT ’08*, p. 166, 2008.

[42] L. Chen and T. Mitra, “Graph minor approach for application mapping on CGRAs,” *FPT 2012 - 2012 Int. Conf. Field-Programmable Technol.*, vol. 7, no. 3, pp. 285–292, 2012.

[43] B. Mei, M. Berekovic, and J. Y. Mignolet, “ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors,” *Fine-and Coarse-Grain Reconfigurable Comput.*, pp. 255–297, 2008.

[44] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “DRESC: a retargetable compiler for coarse-grained reconfigurable architectures,” *IEEE Int. Conf. Field-Programmable Technol. 2002. (FPT). Proceedings.*, pp. 166–173, 2002.

[45] H. D. M. and R. L. B. Mei, S. Vernalde, D. Verkest, “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling,” *Comput. Digit. Tech. IEE Proc.*, vol. 150, no. 5, 2003.

[46] B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops,” *Prof. Eng.*, vol. 7, no. 21, pp. 63–74, 1994.

[47] V. H. Allan, R. B. Jones, M. Lee, J. Allan, and V. H. Allan, “Software Pipelining,” vol. 27, no. 3, 1995.

[48] M. Ashraful and A. Tuhin, “Mapping Applications to Coarse-Grain Reconfigurable Architectures Coarse- grained Reconfigurable Ar- Coarse-grained Reconfig- urable Architectures Hartenstein surveyed the works done so far in the,” pp. 1–6, 2006.

[49] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm, and J. Hammes, “Automatic compilation to a coarse-grained reconfigurable system-opn-chip,” *ACM Trans. Embed. Comput. Syst.*, vol. 2, no. 4, pp. 560–589, 2003.

[50] G. Theodoridis, D. Soudris, and S. Vassiliadis, “A Survey of coarse-grain reconfigurable architectures and cad tools basic definitions, critical design issues and existing coarse-grain reocnfigurable systems,” *Fine-and Coarse-Grain Reconfigurable Comput.*, pp. 89–149, 2008.

[51] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “Garp architecture and C compiler,”

Computer (Long. Beach. Calif)., vol. 33, no. 4, pp. 62–69, 2000.

[52] E. Mirsky and a. DeHon, “MATRIX: a reconfigurable computing architecture with configurable\ninstruction distribution and deployable resources,” *FPGAs Cust. Comput. Mach. 1996. Proceedings. IEEE Symp.*, pp. 157–166, 1996.

[53] G. I. of T. Eric VanHeest, Mitch Kispert, “WinTim32,” 1999. [Online]. Available: http://users.ece.gatech.edu/~hamblen/book/wintim/.

APPENDIX A – ASSEMBLY CODES FOR PROPOSED ARCHITECTURE

This appendix gives the required information to write an assembly code for the proposed CGRA. The assembly codes write by the Python language. The assembly codes should be covered all 47 bits of instructions.

Table Appendix A 1 gives all required bits that can be used in one instruction. For each word on the right side of this table, there is a specific word to define its assembly language.

Table Appendix A 1-Instruction word in token state machine.

Bits	Description
46-43	Operation
42-39	Condition
38	SALU select
37	Flag modification
36	Operand A(FIFO Bank/ SALU Buffer)
35	Deep Reading of Operand A
34-30	Address Operand A
29-26	Address deep reading Operand A
25	Operand B(FIFO Bank/ SALU Buffer)
24	Deep Reading of Operand B
23-19	Address Operand B
18-15	Address deep reading Operand B
14-12	FIFO bank or SALU destination

Table Appendix A 1-Instruction word in token state machine (continued).

Bits	Description
11	Write to FIFO
10	Write as double address or choose the buffer
9-5	Result destination 1
4-0	Result destination 2

Bits 46-43 defines the operations. Table Appendix A 2 gives the operation assembly codes.

Table Appendix A 2-Operation Assembly word.

Bits 46-43	operation	Assembly
0000	Addition	ADD
0001	Subtraction	Minus
0010	Multiplication	Multiple
0011	Multiplication with truncation	MuTrunc
0100	Shift Left Logic	SLL
0101	Shift Right Logic	SRL
0110	Shift Right Arithmetic	SRA
0111	AND	AND
1000	OR	OR
1001	XOR	XOR
1010	NOT	NOT

Table Appendix A 2- Operation Assembly word (continued).

Bits 46-43	operation	Assembly
1011	FLIP	FLIP

Bits 42-39 as default are set to 0. Each FIFO bank is associated to two SALUs and they can be defined by one bit (Bit 38). Table Appendix A 3 illustrates the corresponding assembly code for each state of bit 38.

Table Appendix A 3- SALU Selection along with its Assembly code.

Bit 38	Description	Assembly code
0	Select SALU that is located in left or upside of FIFO banks	SaluLeftUp
1	Select SALU that is located in Right or Downside of FIFO banks	SaluRightDown

Flag modification is set to 0 as a default inside instruction word. Bit 36 defines the operand A whether it is in FIFO bank or SALU buffer (Table Appendix A 4).

Table Appendix A 4- Operand A and corresponding Assembly code.

Bit 36	Description	Assembly code
0	Operand A is located in SALU buffer	OperandABuff
1	Operand A is located in FFO bank.	OperandA

Bit 35 defines whether the reading operand A from FIFO bank is normal or deep. In deep reading, the data token is not removed from FIFO banks. It only makes a copy of data token (operand) and sends it to the SALU. The corresponding assembly code for deep reading of operand A is "*DeepA*". As a default, this bit is 0 and the normal reading is performed.

Bits 34-30 are responsible for defining where is the operand A as given in Table Appendix A 5.

Table Appendix A 5- The address of operand A and corresponding assembly code.

Bits 34-30	Description	Assembly Code
5B#00000	FIFO number 0	R0
5B#00001	FIFO number 1	R1
5B#00010	FIFO number 2	R2
5B#00011	FIFO number 3	R3
5B#00100	FIFO number 4	R4
5B#00101	FIFO number 5	R5
5B#00110	FIFO number 6	R6
5B#00111	FIFO number 7	R7
5B#01000	FIFO number 8	R8
5B#01001	FIFO number 9	R9
5B#01010	FIFO number 10	R10
5B#01011	FIFO number 11	R11
5B#01100	FIFO number 12	R12
5B#01101	FIFO number 13	R13
5B#01110	FIFO number 14	R14
5B#01111	FIFO number 15	R15

Bits 29-26 defines the address for deep reading as given in Table Appendix A 6. As it has been mentioned earlier, the FIFO deeps are eight and thus; a user can access to each element of them using the deep address.

Table Appendix A 6- deep reading address and assembly code.

Bits 29-26	Description	Assembly Code
4B#0000	Element 0	Addeep0
4B#0001	Element 1	Addeep1
4B#0010	Element 2	Addeep2
4B#0011	Element 3	Addeep3
4B#0100	Element 4	Addeep4
4B#0101	Element 5	Addeep5
4B#0110	Element 6	Addeep6
4B#0111	Element 7	Addeep7

From bits 25- 15 are allocated to the operand B, the assembly codes are the same as operand A. The only difference is where the '*A'* is changed to '*B'*.

Bits14-12 determines the destination FIFO bank or SALU buffer. When the destination is the SALU buffer and not the FIFO banks, these bits are used to decode the address to correspond each buffer in SALU sides. The destination address for each decoder along with their assembly codes are given in Table Appendix A 7 .

Table Appendix A 7- SALU Buffer as a destination with their assembly codes.

Bits14-12	Description	Assembly codes
3B#000	Decoder Buffer 0 is the destination	Decoder0
3B#001	Decoder Buffer 1 is the destination	Decoder1
3B#010	Decoder Buffer 2 is the destination	Decoder2
3B#011	Decoder Buffer 3 is the destination	Decoder3
3B#100	Decoder Buffer 4 is the destination	Decoder4
3B#101	Decoder Buffer 5 is the destination	Decoder5
3B#110	Decoder Buffer 6 is the destination	Decoder6
3B#111	Decoder Buffer 7 is the destination	Decoder7

Since bits, 14-12 are used to determining FIFO bank as destinations. Table Appendix A 8 gives corresponding assembly code to determine the destination address for generated data tokens resulted by SALU.

Table Appendix A 8- FIFO bank destination with assembly codes.

Bits14-12	Description	Assembly codes
3B#000	FIFO Bank Left from SALU side	BankFIFOLeft
3B#001	FIFO Bank UP from SALU side	BankFIFOUP
3B#010	FIFO Bank Right from SALU side	BankFIFORight
3B#011	FIFO Bank Down from SALU side	BankFIFODown

Bit 11 determines whether SALU wants to write in FIFO bank or not. To this end, there is assembly code '*WriteFIFO'* that forces '1' to bit 11. Since the destination is not FIFO bank, it is not necessary to write in the FIFO bank and then assembly code "NoWriteFIFO" is allocated to this. "NoWriteFIFO" is forced '0' to bit 11.

Bit 10 has two options: first, if the target destination is the FIFO bank and second if the target destination is SALU buffer. For each option, the specific assembly code is assigned according to Table Appendix A 9 and Table Appendix A 10.

Table Appendix A 9- Double address with its assembly code.

Bit 10 as write to FIFO bank	Description	Assembly Codes
1B#0	Result send to only one destination	Default 0
1B#1	Result can send to two different destination	BothAddress

When the address destination is SALU buffer, two buffers are assigned to each decoder that are determined by bit 10. The user can send data tokens to left or right buffers of the predefined decoder (by bits 14-12).

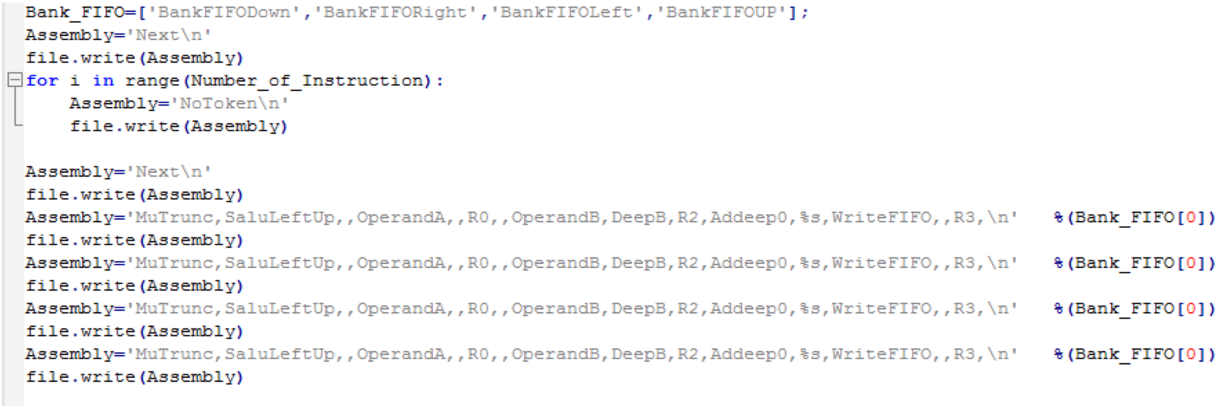
Table Appendix A 10 - select SALU Buffer and its assembly code.

Bit 10 as write to SALU buffer	Description	Assembly Codes
1B#0	Select Buffer Right of SALU	BufferRight
1B#1	Select Buffer left of SALU	BufferLeft

Bits 9-0 determines the destination addresses for which the assembly codes are equivalent to what given in Table Appendix A 5.

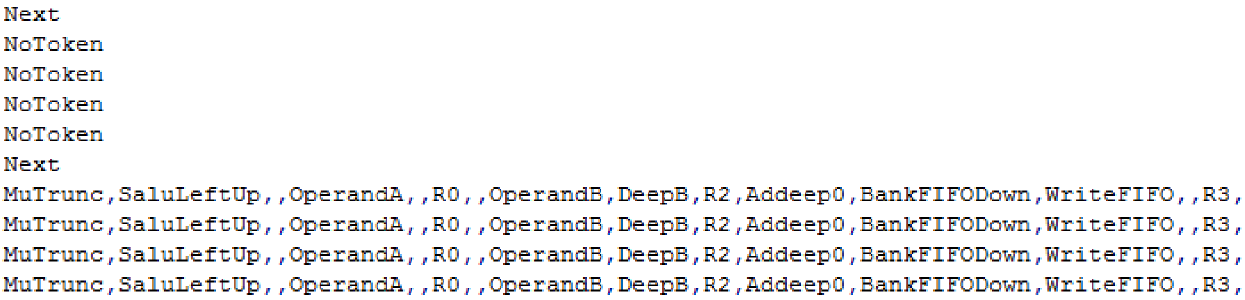
There is three extra assembly command codes '*Next', 'TSMNext'* and 'NoToken'. To separate TSM0-7 from each other, the command '*Next'* is used. '*TSMNext'* is used when the computing fabric has more than one tile. 'NoToken' demonstrates that there are no more instructions in TSM.

Figure Appendix A 1 shows a simple code conversion from Python as source code to the assembly codes acceptance to computing fabric. Evidently, each small tsm is composed of four instructions. When one tsm is not programmed, it should be filled up with ‘*NoToken’* and 'Next' command separates the tsms.



```
Bank_FIFO=['BankFIFODown','BankFIFORight','BankFIFOLeft','BankFIFOUP'];
Assembly='Next\n'
file.write(Assembly)
for i in range(Number_of_Instruction):
    Assembly='NoToken\n'
    file.write(Assembly)

Assembly='Next\n'
file.write(Assembly)
Assembly='MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,%s,WriteFIFO,,R3,\n'   %(Bank_FIFO[0])
file.write(Assembly)
Assembly='MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,%s,WriteFIFO,,R3,\n'   %(Bank_FIFO[0])
file.write(Assembly)
Assembly='MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,%s,WriteFIFO,,R3,\n'   %(Bank_FIFO[0])
file.write(Assembly)
Assembly='MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,%s,WriteFIFO,,R3,\n'   %(Bank_FIFO[0])
file.write(Assembly)
```

(a)

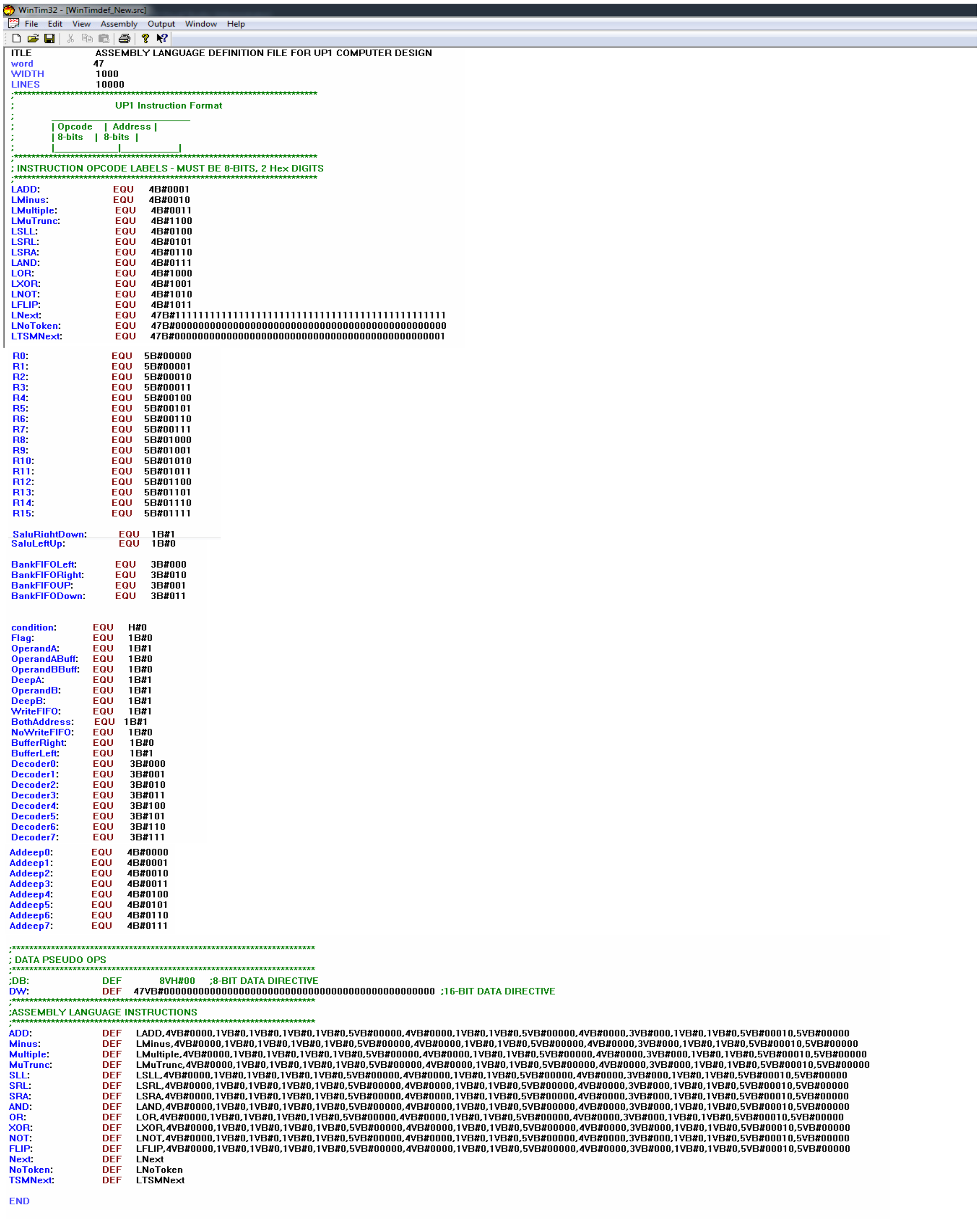


```
Next
NoToken
NoToken
NoToken
NoToken
Next
MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,BankFIFODown,WriteFIFO,,R3,
MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,BankFIFODown,WriteFIFO,,R3,
MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,BankFIFODown,WriteFIFO,,R3,
MuTrunc,SaluLeftUp,,OperandA,,R0,,OperandB,DeepB,R2,Addeep0,BankFIFODown,WriteFIFO,,R3,
```

(b)

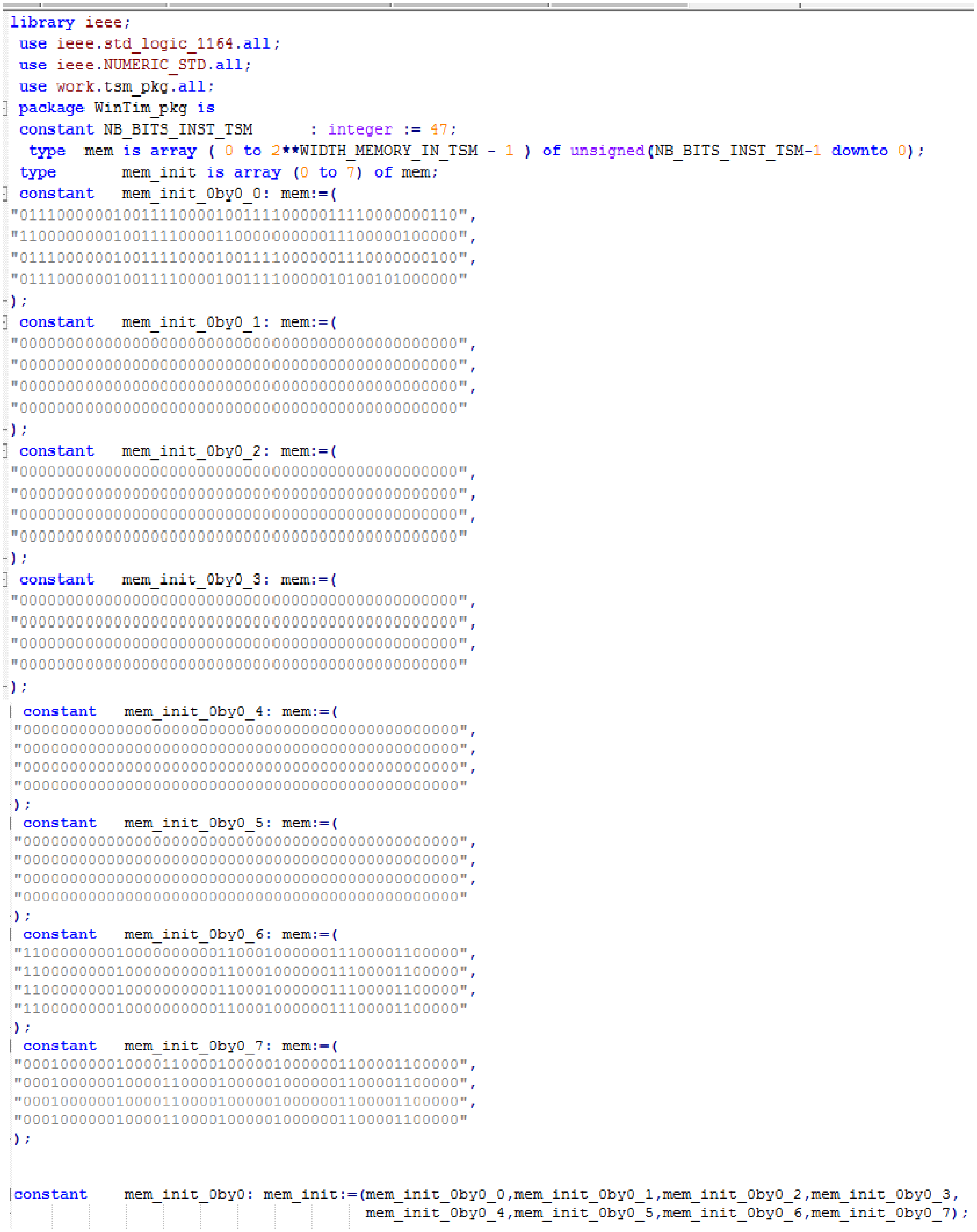
Figure Appendix A 1- Simple example of create assembly code, a) Python language b) assembly codes

APPENDIX B – WINTIM32



```
ITLE                   ASSEMBLY LANGUAGE DEFINITION FILE FOR UP1 COMPUTER DESIGN
word                   47
WIDTH                  1000
LINES                  10000
;*******************************************************************
;                         UP1 Instruction Format
;              ___________________________
;              | Opcode    | Address |
;              | 8-bits    | 8-bits  |
;              |___________|_________|
;*******************************************************************
; INSTRUCTION OPCODE LABELS - MUST BE 8-BITS, 2 Hex DIGITS
;*******************************************************************
LADD:                  EQU    4B#0001
LMinus:                EQU    4B#0010
LMultiple:             EQU    4B#0011
LMuTrunc:              EQU    4B#1100
LSLL:                  EQU    4B#0100
LSRL:                  EQU    4B#0101
LSRA:                  EQU    4B#0110
LAND:                  EQU    4B#0111
LOR:                   EQU    4B#1000
LXOR:                  EQU    4B#1001
LNOT:                  EQU    4B#1010
LFLIP:                 EQU    4B#1011
LNext:                 EQU    47B#11111111111111111111111111111111111111111111111
LNoToken:              EQU    47B#00000000000000000000000000000000000000000000000
LTSMNext:              EQU    47B#00000000000000000000000000000000000000000000001

R0:                    EQU    5B#00000
R1:                    EQU    5B#00001
R2:                    EQU    5B#00010
R3:                    EQU    5B#00011
R4:                    EQU    5B#00100
R5:                    EQU    5B#00101
R6:                    EQU    5B#00110
R7:                    EQU    5B#00111
R8:                    EQU    5B#01000
R9:                    EQU    5B#01001
R10:                   EQU    5B#01010
R11:                   EQU    5B#01011
R12:                   EQU    5B#01100
R13:                   EQU    5B#01101
R14:                   EQU    5B#01110
R15:                   EQU    5B#01111

SaluRightDown:         EQU    1B#1
SaluLeftUp:            EQU    1B#0

BankFIFOLeft:          EQU    3B#000
BankFIFORight:         EQU    3B#010
BankFIFOUP:            EQU    3B#001
BankFIFODown:          EQU    3B#011


condition:         EQU    H#0
Flag:              EQU    1B#0
OperandA:          EQU    1B#1
OperandABuff:      EQU    1B#0
OperandBBuff:      EQU    1B#0
DeepA:             EQU    1B#1
OperandB:          EQU    1B#1
DeepB:             EQU    1B#1
WriteFIFO:         EQU    1B#1
BothAddress:       EQU   1B#1
NoWriteFIFO:       EQU    1B#0
BufferRight:       EQU     1B#0
BufferLeft:        EQU     1B#1
Decoder0:          EQU     3B#000
Decoder1:          EQU     3B#001
Decoder2:          EQU     3B#010
Decoder3:          EQU     3B#011
Decoder4:          EQU     3B#100
Decoder5:          EQU     3B#101
Decoder6:          EQU     3B#110
Decoder7:          EQU     3B#111
Addeep0:           EQU    4B#0000
Addeep1:           EQU    4B#0001
Addeep2:           EQU    4B#0010
Addeep3:           EQU    4B#0011
Addeep4:           EQU    4B#0100
Addeep5:           EQU    4B#0101
Addeep6:           EQU    4B#0110
Addeep7:           EQU    4B#0111

;*******************************************************************
; DATA PSEUDO OPS
;*******************************************************************
;DB:                 DEF          8VH#00     ;8-BIT DATA DIRECTIVE
DW:                  DEF    47VB#00000000000000000000000000000000000000000000000  ;16-BIT DATA DIRECTIVE
;*******************************************************************
;ASSEMBLY LANGUAGE INSTRUCTIONS
;*******************************************************************
ADD:                 DEF    LADD,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
Minus:               DEF    LMinus,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
Multiple:            DEF    LMultiple,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
MuTrunc:             DEF    LMuTrunc,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
SLL:                 DEF    LSLL,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
SRL:                 DEF    LSRL,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
SRA:                 DEF    LSRA,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
AND:                 DEF    LAND,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
OR:                  DEF    LOR,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
XOR:                 DEF    LXOR,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
NOT:                 DEF    LNOT,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
FLIP:                DEF    LFLIP,4VB#0000,1VB#0,1VB#0,1VB#0,1VB#0,5VB#00000,4VB#0000,1VB#0,1VB#0,5VB#00000,4VB#0000,3VB#000,1VB#0,1VB#0,5VB#00010,5VB#00000
Next:                DEF    LNext
NoToken:             DEF    LNoToken
TSMNext:             DEF    LTSMNext

END
```

Figure Appendix B 1-Instruction formats and mnemonic names definition file by WinTim32



```
library ieee;
 use ieee.std_logic_1164.all;
 use ieee.NUMERIC_STD.all;
 use work.tsm_pkg.all;
 package WinTim_pkg is
 constant NB_BITS_INST_TSM       : integer := 47;
  type  mem is array ( 0 to 2**WIDTH_MEMORY_IN_TSM - 1 ) of unsigned(NB_BITS_INST_TSM-1 downto 0);
 type        mem_init is array (0 to 7) of mem;
 constant    mem_init_0by0_0: mem:=(
"01110000001001111000010011110000011110000000110",
"11000000001001111000011000000000011100000100000",
"01110000001001111000010011110000001110000000100",
"01110000001001111000010011110000010100101000000"
);
 constant    mem_init_0by0_1: mem:=(
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000"
);
 constant    mem_init_0by0_2: mem:=(
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000"
);
 constant    mem_init_0by0_3: mem:=(
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000"
);
 constant    mem_init_0by0_4: mem:=(
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000"
);
 constant    mem_init_0by0_5: mem:=(
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000",
"00000000000000000000000000000000000000000000000"
);
 constant    mem_init_0by0_6: mem:=(
"11000000001000000000011000100000011100001100000",
"11000000001000000000011000100000011100001100000",
"11000000001000000000011000100000011100001100000",
"11000000001000000000011000100000011100001100000"
);
 constant    mem_init_0by0_7: mem:=(
"00010000001000011000010000010000011000011000000",
"00010000001000011000010000010000011000011000000",
"00010000001000011000010000010000011000011000000",
"00010000001000011000010000010000011000011000000"
);


constant     mem_init_0by0: mem_init:=(mem_init_0by0_0,mem_init_0by0_1,mem_init_0by0_2,mem_init_0by0_3,
                                       mem_init_0by0_4,mem_init_0by0_5,mem_init_0by0_6,mem_init_0by0_7);
```

Figure Appendix B 2- Generated VHDL file for one TSM by WinTim