UNIVERSITY OF CALIFORNIA

Los Angeles

Next Generation Dynamically Reconfigurable DSP in 16nm Technology

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical and Computer Engineering

by

Uneeb Yaqub Rathore

2018

ABSTRACT OF THE THESIS

Next Generation Dynamically Reconfigurable

DSP in 16nm Technology

by

Uneeb Yaqub Rathore

Master of Science in Electrical and Computer Engineering

University of California, Los Angeles, 2018

Professor Dejan Markovic, Chair

An increasing number of dedicated accelerators in modern System on Chips (SoCs) have led to large regions of dark silicon. Although highly efficient, these accelerators (ASICs) are inflexible. Where CPUs are flexible in time and FPGAs in space - both architectures suffer from limited efficiencies (in terms of $GOPS/mm^2$ and $GOPS/mW$). The challenge then - is to come up with architectures that implement the "right" amount of flexibility (in both time and space) while simultaneously giving near ASIC performance. Adding to this requirement, these architectures should have fast design-time, yet be easy to program with simple compilers and should support efficient in-field deployment of new algorithms.

As a solution, in this thesis, a UDSP architecture is presented that addresses each of these criteria using hardware/software co-design. This thesis demonstrates a way to make a UDSP array consisting of individual heterogeneous cores well connected by a high-speed routing network and accompanied by a software mapper which eases the overall programming process. As an example implementation, this thesis presents an 81 core chip implemented in TSMC 16nm FF process, with support for FIR/IIR Filters, Matrix Vector Multiply, Small Hardware Neural Nets, FFTs and 1GHz Single Cycle MACs.

The thesis of Uneeb Yaqub Rathore is approved.

Sudhakar Pamarti

Subramanian Srikantes Iyer

Dejan Markovic, Committee Chair

University of California, Los Angeles

2018

To my parents.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to my advisor Professor Dejan Markovic, whose support and advice was vital for completing this work and the Chip. I am thankful for some of the brainstorming sessions incurred with him which helped iron out many of the finer details of this implementation.

I also wish to thank Sumeet Singh Nagi, Sina Basir-Kazeruni Shahroze Kabir and Jiahao Zang, for the uncountable sparing and brainstorming sessions on a wide array of topics – and without whom this work wouldn't have been (in the slightest) possible. Sumeet was responsible for the 16nm implementation of the design and Jiahao Zang was responsible in part for the software backend.

I would also like to acknowledge the support of the Boeing team –namely Monte, Steve and Cindy for their contributions to the design of the Core and their thoroughness on verification. I am indebted to Hariprasad Chandrakumar, Vahagn Hokhikyan, Trevor Black and Alireza Yousefi for the lively discussions they had with me and for their help on some of the topics that I was otherwise stuck on during the course of this project.

And I am forever grateful to my parents, brothers, grandparents and my aunt for their loving support during my studies at UCLA.

CHAPTER 1

Introduction

1.1 Dark Silicon and Leakage Power

Modern SoCs contain many hardware accelerators (Application Specific Integrated Circuits - ASICs) each specialized for a particular task. However, most of the time none or only one of these accelerators is being used. This leads to large areas of the chip being underutilized – regions termed as dark silicon. With shrinking technology nodes, increasing problems with DIBL (Drain Induced Barrier Lowering), gate-oxide leakage current, junction leakage and most importantly sub-threshold conduction, the leakage power to active power ratio is increasing [1, 2, 3]. This means that moving into the future dark silicon is going to become a major concern for leakage. Equations 1.1 and 1.2 show the effect of shrinking channel length on the leakage power. To keep the leakage at a minimum higher threshold transistors need to be used which leads to higher operating voltage resulting in higher active power. This results in a trade-off between the active power and leakage power. Many techniques like power gating schemes [4] and even using different material transistors [5] have been proposed to lower leakage power, but they come at the cost of performance. Another approach to the problem is to reuse the hardware on chip across different algorithms. An example of this is the CPU (Central Processing Unit) that has a fixed compute unit and can do a basic set of memory operations and is re-programmable in time.

$$I_{leak} = \mu_{eff} C_{ox} \frac{W}{L} (m-1) (\frac{kT}{q})^2 e^{-qV_t/mkT} \tag{1.1}$$

$$P_{Total} = P_{static} + P_{dynamic} = (1-\alpha) N I_{leak} V_{dd} K + \alpha N C_{ox} f V_{dd}^2 \tag{1.2}$$

$$N \triangleq transistor\ count$$

$$\alpha \triangleq activity\ factor$$

$$K \triangleq circuit\ constant$$

A CPU coupled with enough memory can process any algorithm (in the span of the primitives that the CPU architecture supports). However CPUs are inefficient due to their excess flexibility and often cannot process large amounts of data at runtime speeds. Extending this architecture is the GPU (Graphics Processing Unit) where the memory and core architecture are specially designed to process matrices. They are an order of magnitude better than CPUs in terms of area and energy efficiency metrics but still 2 orders of magnitude worse than dedicated ASICs. Another approach is to use FPGAs (Field Programmable Gate Arrays) – these spatially configurable structures are programmable down to the bit level - albeit not runtime configurable. FPGAs generally have no temporal configurations so in order to reprogram a small part of the algorithm; the entire program needs to be rewritten. Also due to a large amount of available flexibility that most algorithms don't benefit from, the FPGA's performance metrics are not much better than a GPU. In summary FPGAs are spatially flexible and CPUs are temporally flexible, and their high flexibility trades off with their final performance (in terms of Giga Operations per Joule and Giga Operations per Area – $GOPS/mW$, $GOPS/mm^2$ respectively). This trade-off space is summarized in Figure 1.1 which averages out many architectures in the communications space submitted to ISSCC and VLSI over a 10 year long period [6].

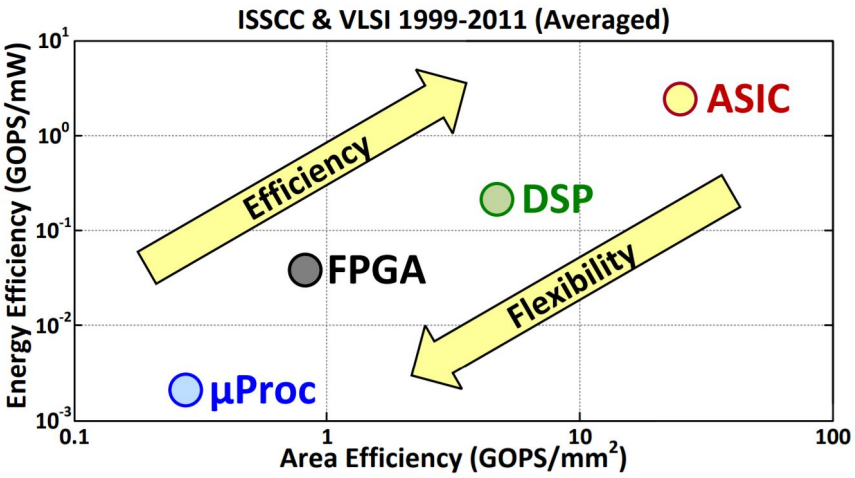


Figure 1.1: Efficiency space.

This begs the question, can there be a domain specific architecture that implements the right amount of flexibility and does so in both time and space? And can such an architecture then trade off some of the spatio-temporal flexibility for a boost in performance in that domain? How close can an architecture get to an ASIC and does this have other desirable benefits like design time savings, IP reuse, scalability and upgradability? During the era of the original x86 architecture, a sizable portion of researchers were exploring the space of very long instruction word (VLIW) processors, some of which were targeted towards answering these questions but over the years they have diminished in their ratio.

This thesis in targeted to explore these questions in more detail through designing an architecture. The architecture presented in this manuscript is called the Universal Digital Signal Processor (UDSP). In addition to presenting a concrete architecture - this thesis reasons and elaborates a design approach to come up with working domain specific architectures. As an example, this thesis also presents a communications domain specific UDSP implemented in TSMC 16nm FF. Many aspects of configurability borrow concepts from CPU compiler theory and FPGAs – and have been tailored to this problem.

1.2 Related Work

The space of reconfigurable architectures using hardware-software co-design is rich with examples of many types of processors, ranging from those specialized for physics processing like ParallAX [7] to those specialized for security [8]. A large class these architectures are different variations of VLIW processors presented first by Dr Fisher in 1983 at the 10th Annual International Conference on Computer Architecture [9]. As such many entities in the space are working on different aspects. Some literature in the space, for example [10], focuses on software aspects of hardware-software co-design in the security domain. Whereas others like [11] turn their attention to power optimization of heterogeneous core designs. [12] focuses on the synthesis aspect of such DSP processors where as [13] Targets reconfigurable DSPs and implements them on FPGAs skipping an entire chapter on the software co-design of reconfigurable systems. Some contributions to this space like [14] (which targets AES encryption) do implement co-design features but are too narrow in their target domain and have a non-generalizable approach. [15] implements an architecture that is too broad and covers a wide variety of applications. This DSP processor uses fewer compute resources and a bus based routing architecture. Their design can handle many operations since the compute units are floating point processors, however their approach is a variation of transport triggered architecture, and therefore it is highly compiler dependent resulting in a processor that relies heavily on memory operations and cannot handle applications that require real-time streaming operations at very high speeds such as baseband modulation or filtering A good attempt is made in [16] to make a configurable DSP. They use a crossbar architecture for their routing network, which is a form of mesh architecture, and they use a 2 ALU based process element as their fundamental compute unit. However they make no attempt at explaining how their approach is generalizable to other domains. [17] Presents a good hardware approach for making generalizable hardware architectures, but lacks in its presentation of any mapping or programming software impact and implications

on their design methodology. Countless such examples exist in the domain of configurable computing and VLIW processors but seldom do they provide a complete hardware/software co-design approach in designing such kernels that are domain agnostic and scalable. In this thesis the UDSP attempts to present such an agnostic approach applied to the domain of a communications DSP.

1.3 Outline of Contributions

A brief outline containing the contributions of this thesis is presented below. Chapter 2 discusses the hardware implementation of the UDSP and goes into detail on how each of the blocks are designed. In this chapter a generalizable approach to the design problem (of reconfigurable hardware) is presented and simultaneously applied to a domain specific accelerator for digital communications processing. Chapter 3 deals with the software backend needed to program the UDSP. It discusses the representation of algorithms, clustering of data flow graphs and approaches for placement and routing onto the UDSP hardware. Chapter 4 discusses the implementation of the 16nm UDSP chip and its verification strategies. Here, the drawbacks of the current chip and improvements for future iterations are mentioned as well. Chapter 5 concludes the thesis.

CHAPTER 2

Hardware Architecture Design

2.1 Structural Overview of the UDSP

The Universal Digital Signal Processor (UDSP) Array consists of 81 identical repetitive blocks, called "Cores", arranged in a grid fashion. In between the individual Cores exists a "Routing Network" responsible for transferring data from one Core to another. On top of the Routing Network exists an "I/O Network" responsible for routing data to and from the UDSP Array and its cores. Adjacent to these data processing layers sits a "Control Module" that is responsible for programming the DSP and reporting on the health of the DSP. Together these four entities, namely the Core, the Routing Network, the I/O Network and the Control Module make up the complete UDSP Array. All data handling entities are individually programmable. For the Core this means that internally it can be reconfigured to take on a particular function from a large set of similar functions. For the Routing Network this means that the connections between the individual Cores are arbitrary and can be reprogrammed to a certain set of connections serving a particular algorithm. For the I/O Network this means that the any external data connection can be purposefully routed to any particular Core and vice versa. In the following sections the analysis and design of each of these sub-entities is covered in more detail and reasoning (both intuitive and mathematical) are presented to justify the design choices made.

2.2 The Core Architecture (Compute Unit)

A Core is the smallest configurable compute subunit in the design. The core is replicated to make a large array which collectively can handle different algorithms belonging to a specific domain (e.g. encryption). Therefore the structure of the Core is highly dependent on the nature of the algorithms the user wishes to implement on the chip. For a particular set of pre-determined algorithms there exists an optimal set of Core designs (in the area/power/speed efficiency sense). Hence the core design needs to be domain specific. The set of applications that this model of the UDSP is targeted towards, includes (but is not limited to) Single cycle MAC, Complex Arithmetic, FIR/IIR/Adaptive Filters, SDF FFT/Butterfly & Simple Convolutional Neural Nets through matrix multiplications. The method to come up with the design for the Core is to identify and reuse the common elements across all algorithms. Given that only a few algorithms will be active at a given time; this saves a dark silicon area and therefore leakage power. However, the associated cost is a slightly larger energy consumption per computation. The goal of a good core design is to keep this overhead at a minimum. Figure 2.1 taken from [6] shows an example of a core design that used this methodology and is manually designed.

Although not apparent, this form of a design (accompanied with multi layered routing

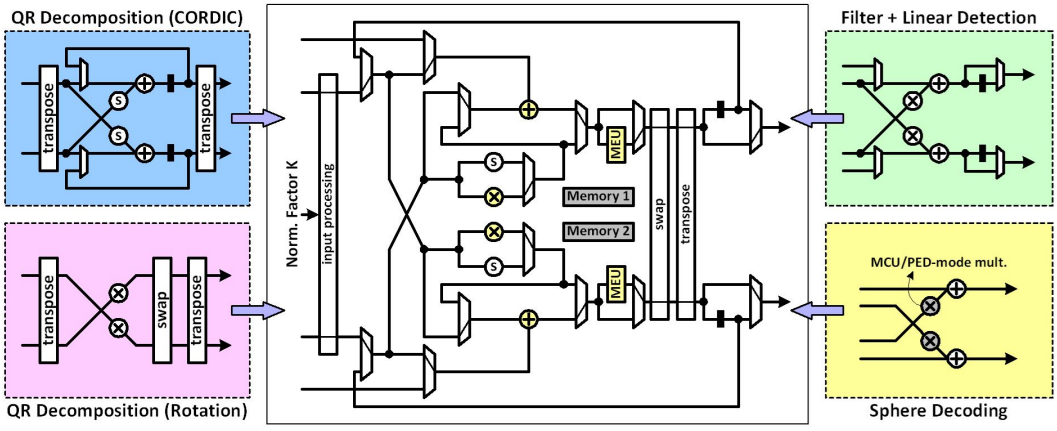


Figure 2.1: Example of a DSP Core

connections) is energy and area efficient if the mapping structures are known a-priori but are difficult to program for arbitrary algorithms. For some of the routing connections between the compute elements in the core, the multiplexer depth is 2 or more. This makes mapping an arbitrary dataflow graph to this core a graph matching problem – which is NP-complete. For the version of the UDSP presented in this thesis, a complete list of algorithms is not known a-priori (as a requirement from the vendor Boeing due to 3rd party NDAs) therefore a multi layered mux approach cannot be used at the core's granularity level as it takes away many types of possible connections in the core. Another requirement is the core needs to be quickly upgradeable without any penalty to the backend software mapper. These constraints cause all routing inside the core to be single hop which also results in the representation of the core as a simpler connectivity matrix and delay matrix pair. These matrices will later serve as an abstraction or a bridge for the software mapper to map algorithms to the core. As a consequence, any change to the core's structure and hence the connectivity matrix and delay matrix is easy to incorporate in the software flow which makes for fluid upgradability.

Support for some intended structures (as provided by the vendor) as well as the general case of filters, FFTs CNNs etc is a requirement for this design. Some of these intended structures are presented in Figure 2.2 (a) – (e). (a) and (b) are direct form structures for FIR and IIR filters, (c) is a complex multiply accumulate unit and is single cycle, (d) is the lattice structure of an FIR filter and (f) is an 8 point FFT.

In order to support these requirements, the core architecture went through several iterations finally resulting in $CoreVersion3.5$ which was later taped out. Some of the nuances of the design were that a single cycle MAC was highly desirable by the vendor so provisions were made to include its support. Since this involved an adder and a multiplier in the critical path, ultra-low threshold voltage cells were used to meet timing for 1GHz. This caused the core to have high leakage power (about 1/3rd of total power) at the cost of performance. Figure 2.3 shows the evolution of the core as requirements matured.

The final core architecture is too dense to draw reasonably and is best represented by

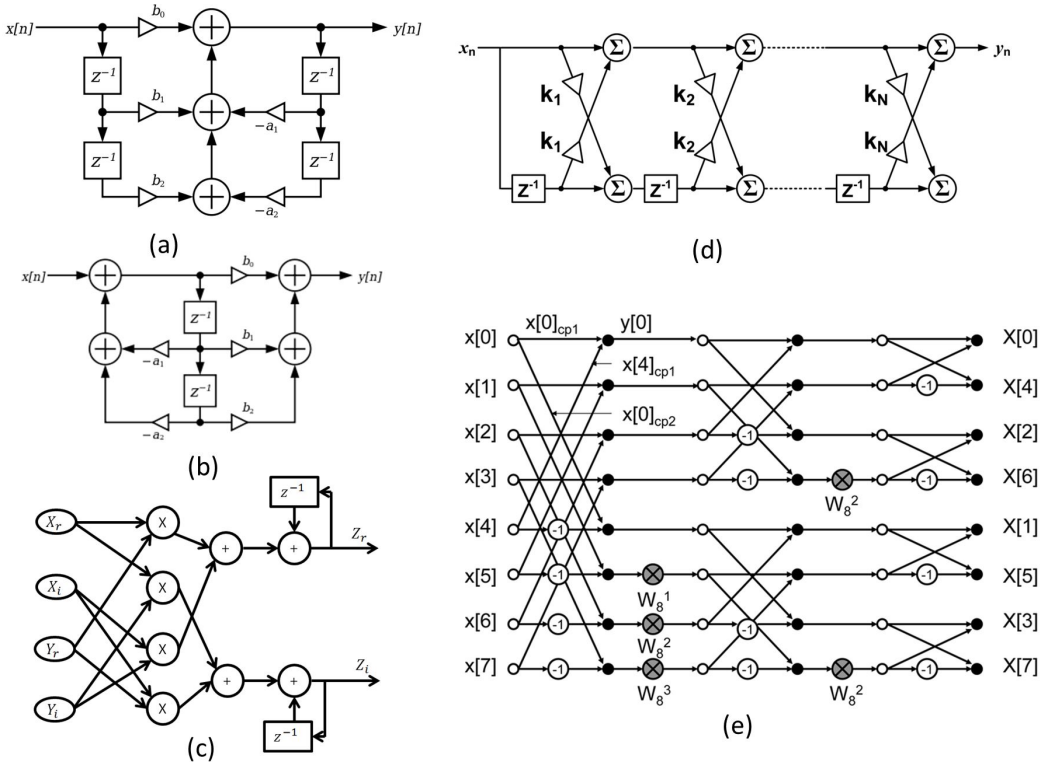


Figure 2.2: Algorithms Supported by Core

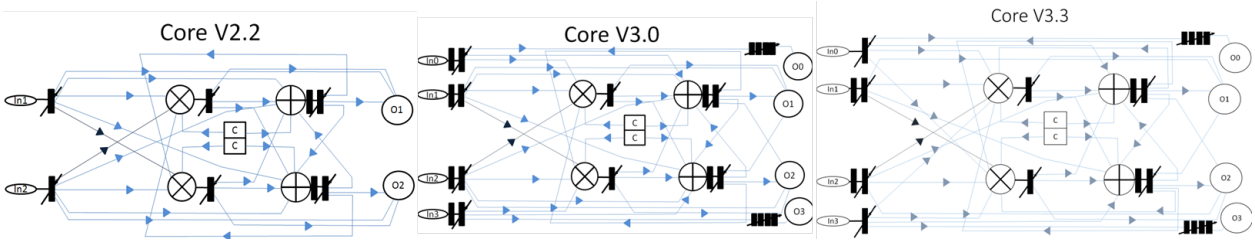


Figure 2.3: Core Architecture Evolution

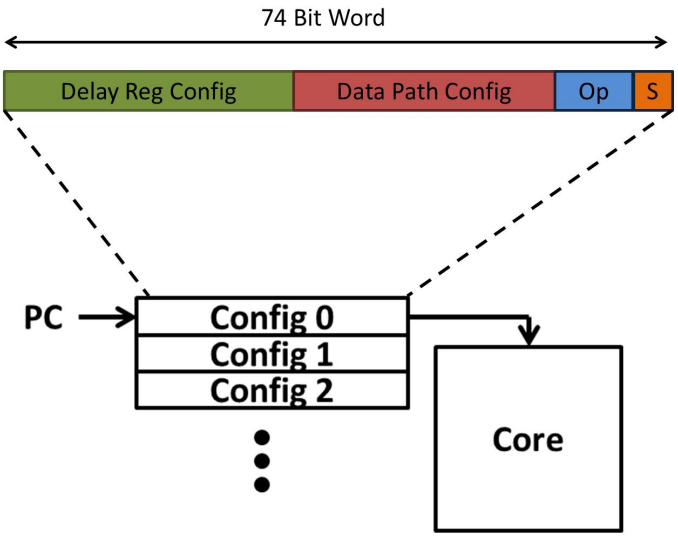


Figure 2.4: Instruction Memory

a connectivity matrix and a delay matrix. These are shown in table 1 and 2 respectively and are elaborated in the Chapter 3. The UDSP also needs to support algorithms like SDF FFT, which change their structure and flow graphs as the algorithms proceed in time. To include this support, 8 instructions for delay and connectivity are kept in a local memory controlled by a program counter as shown in Figure 2.4.

These instructions can be used to change the entire routing and functionality of a single Core. Once these provisions are in place – namely an instruction memory and quick (single cycle) context switching, it is only logical to extend this functionality to time multiplex the different algorithms in the core as this comes with little control overhead. However in order to preserve the internal register information for several algorithms, 4 additional internal state registers are introduced. These registers are used to preserve contextual information of an algorithm in case it needs to be time multiplexed. This addition of the temporal direction is covered in more detail in the software section.

The final version of core has 2 16-bit (16, 15) multipliers, shifters and configurable adders/subtractors. There are 2 configurable delay lines in the core that support long delays (up to 16) and several single delay configurable lines. The core also has 2 constant banks

each containing 8 16-bit constants, and can be programmed to act as a temporary data cache as well. Each core has 4 data inputs and 4 data output. The instruction width is 74 bits, consisting of 38 bits for the delay lines, 20 bits for configuration, 14 bits for selection of computational operations (e.g. addition vs. subtraction) and 2 bits for selecting the temporal states (time multiplexed algorithms). Each core has 8 temporal instructions and 4 internal preservation states per data register.

2.3 The Routing Network Architecture (Network Layer)

This section is divided into three parts. First is an analysis of architectures in literature to better understand the metrics at play in the design of a general routing network. Second is the design of network parameters, incorporating abstract concepts from graph theory. An the third is the UDSP routing layer architecture which takes on a deeper hardware implementation perspective.

2.3.1 Architectures in Literature

This section will review other routing fabrics in literature and contrast the desired benefits and flaws of each of them. A typical FPGA routing network comes in two main flavors, the first being a straight mesh based network and the second being a hierarchical mesh network shown in Figure 2.5 (a) and (b) respectively [18].

Some properties can be deduced about the network from its interconnections. First - a routing network consists of I/O edges from where data is either taken or delivered, and intermediate routing nodes, that can only be used once per route. A routing node becomes unusable for other routes once a route is programed through it. Intuitively, the more the routing nodes and the better they are interconnected with each other, the better the routing layer performs while routes are being mapped, as there is less probability of

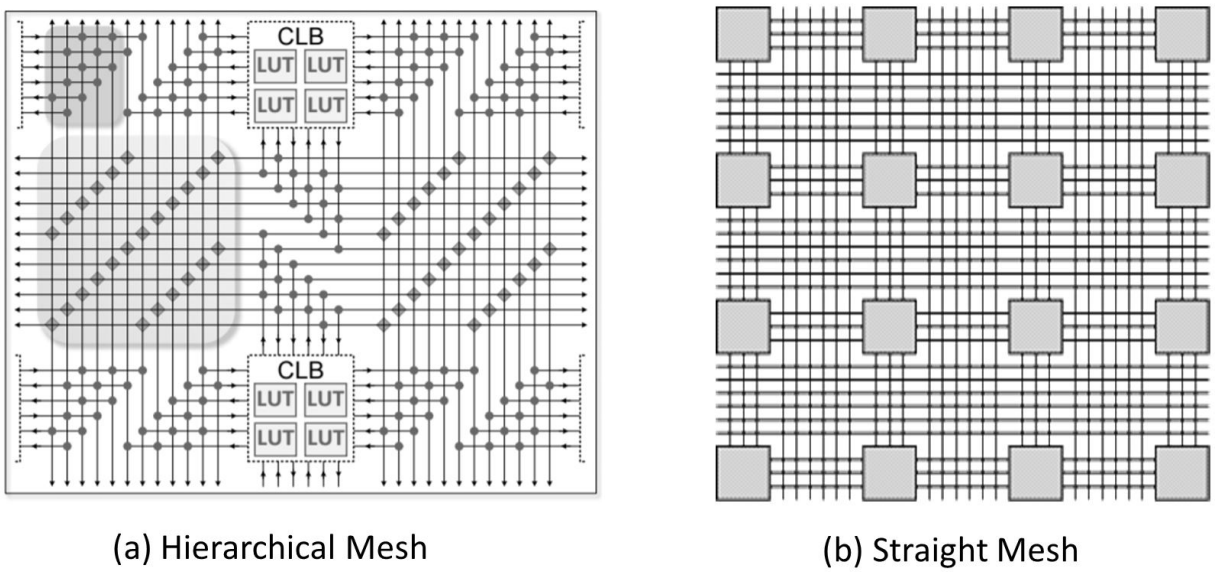


Figure 2.5: Common FPGA Routing Architectures

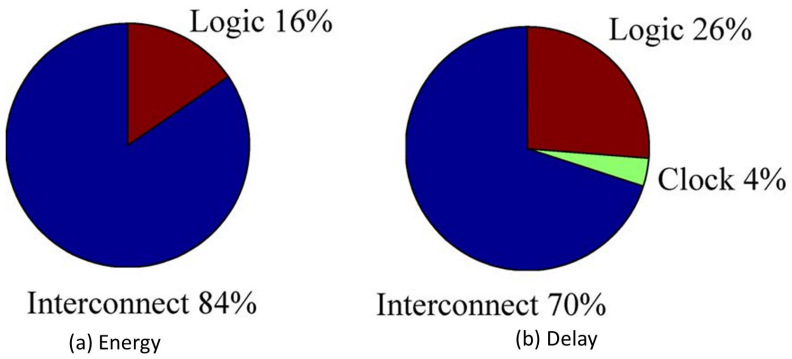


Figure 2.6: FPGA Energy Delay Breakdown (Typical)

conflict. Ideally, in the limit, all I/Os in a routing fabric should be connected to each other in a fully connected fashion, skipping the intermediate routing nodes altogether, however the number of edge connections grows exponentially with order $O(n^2)$. This severely limits the design in terms of its area and energy efficiency [19], [20], [21]. In fact most of the modern FPGAs' area (and hence leakage power) is attributed to its overly complex and dense routing networks. This can be seen in Figure 2.6.

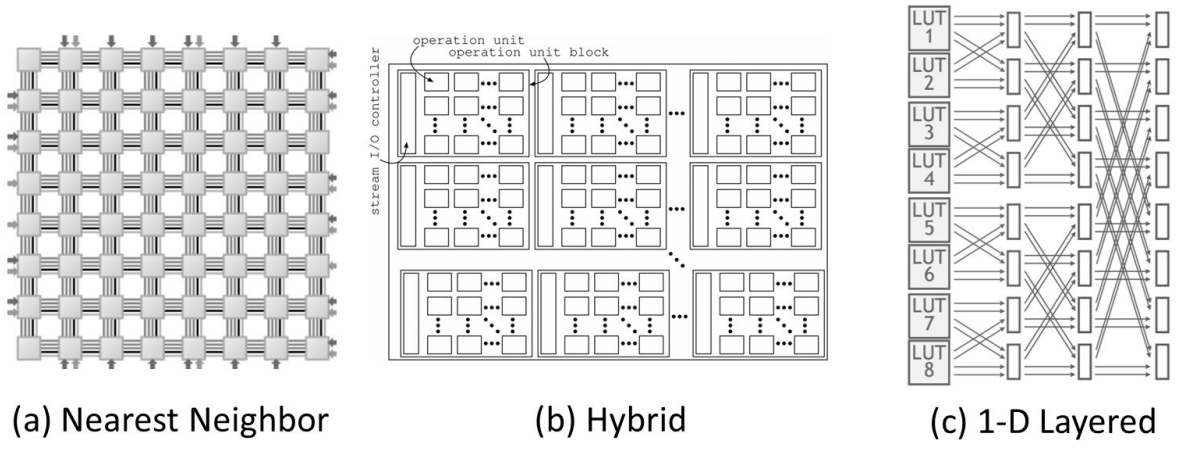


Figure 2.7: Common Routing Topologies

The order of the network is defined as: *the order of the sum of the number of routing nodes and edges present in a network as a function of the number of I/Os supported by the network – namely n.*

A couple of other commonly used routing topologies are shown in Figure 2.7 (a) and (b). In the nearest neighbor topology, I/Os of the compute units are only connected to its neighbors which limits mappability (the ability to map a graph) and utilization (the ratio of the used compute units to unused ones) in a trade-off space. This means that for good mapping the utilization suffers and for good utilization there exist a limited number of algorithms that are mappable. Hybrid layers use a combination of locally dense routing and globally sparse routing or vice versa depending on the application. However due to two different routing boundaries, this topology is harder to automate and requires several iterations of placement and routing combined to converge to a feasible solution. Figure 2.7 (c) shows the routing topology proposed by Fang-Li in 2014 as an efficient alternative [6]. It has a smaller area over head, and is easy to automate (in terms of software programmability) and can operate at very high speeds for data driven real-time applications such as FFTs etc.

Table 2.1 summarizes and compares the pros and cons of each of these topologies, as well as describes the order of each network. The order shows the scalability of each topology and by extension how many I/O and computation nodes it can handle without blowing up in area.

Table 2.1: Routing Topologies

Topology	Order	Pros	Cons
Hierarchical Mesh	$O(n \cdot log_4(n))$	Well Connected	High area penalty
Straight Mesh	$O(n)$	Connectivity dependent on connection sparsity (generally well connected)	Low Speed
Nearest Neighbor	$O(n)$	High Speed	Limited Connectivity / Algorithms
Hybrid	$O(\frac{n}{k}(1 + log_m(\frac{n}{k})))$	High Speed (locally)	Difficult to Automate Fast Mapping
1-D Layered	$O(n \cdot log_2(n))$	Easy to Automate and High Speed	Limited Connectivity / Algorithms

Listed below are some of the key insights which will prove useful in the design and analysis of the UDSP Routing topology. An ideal routing layer should:

1. Support easy connectivity with closer proximity cores for high speed
2. Support long connections in case they are needed
3. Be or order $O(n)$ so that it is infinitely scalable
4. Be easier to design and redesign as well as program and automate
5. Have small overhead in terms of area (and power as well)
6. Be agnostic to the internals of the core and hence support heterogeneity.

Although the constraints are straight forward, complexities arise when they try to counter each other. For example support for 'long connections' coupled with 'high speed' is a deterrent for an 'infinitely scalable' network. In order to use these constraints effectively, the UDSP specific application use case is examined. The 1-D Layered design borrows heavily from the fact that this routing layers' primary function is to accelerate a communications

DSP supporting butterfly structures and lattice FIR filters. It is therefore very effective at performing these tasks but is poor in mapping arbitrary algorithms. As a requirement for the program that this UDSP is being made for, no algorithmic knowledge can be shared a-priori (due to 3rd party NDAs). Hence a routing layer solution that is agnostic to the algorithms needs to be designed. This means that no presumed structure of routing can be exploited (like in the 1-D case) and the interconnection between routing nodes has to be rethought of from first principals. The UDSP also has a speed limitation as its intended use is for wide-baseband processing. The chip has a fixed clock of 1GHz and so does the routing layer. This effectively limits the longest route supported by the network to some finite number of hops 'K'. Another requirement is a design automation speed up- one way of achieving which is to compose the routing layer or repetitive blocks.

2.3.2 Network parameters - *a graph theoretic approach*

This section takes an abstract approach, using graph theoretic concepts to explore and reason the choice of parameters for the UDSP network layer. A *randomly connected graph* is defined as a graph having N nodes, from which 2 are picked at random and connected by an edge; where two nodes cannot share more than one edge. As the number of edges increases the mean number of connections per node $\triangleq \lambda$ increases as well.

The degree of a node is defined as *the total number of edges connected to a node.* The degree distribution of a randomly connected graph follows a Poisson distribution. As an example the degree distribution of a 500 node graph with $\lambda = 2$ is shown in Figure 2.8 (a). Figure 2.8 (b) shows the distribution of edge distances in space if the nodes are randomly placed on a 2-d grid, and (c) shows the distance distribution after the simulated annealing placement algorithm is applied to optimize to cost function of minimizing distances. The simulated annealing algorithm will be covered in more detail in Chapter 3 - the cost function being minimized is given in equation 2.1.

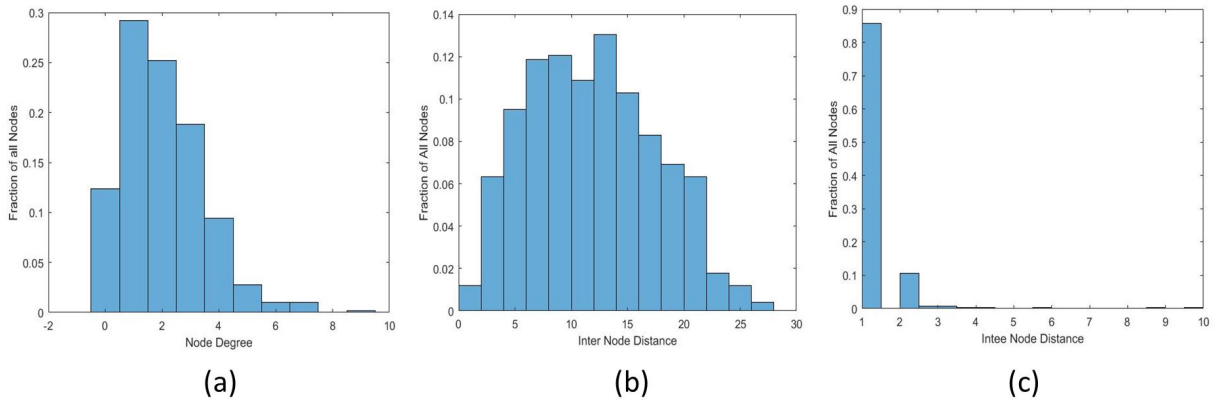


Figure 2.8: Degree and Distance Distributions

$$WiringCost = \sum_{n=1}^{N_{wires}} |n_{xlen}^2 + n_{ylen}^2|^p = 1 \tag{2.1}$$

For this random graph of N nodes, the percolation threshold is defined as $log_e(N)$. If the graph has an average degree distribution λ, and if $\lambda < log_e(N)$ or $\lambda > log_e(N)$ (less than or slightly greater than) the percolation threshold, long wiring connections seldom happen, and after mapping the wiring distances comes out to be negative exponential (empirical evidence). However, if $\lambda >> log_e(N)$ then the length distribution no longer comes out to be negative exponential. This means that wires will be a lot longer, and therefore supporting a 1GHz network would become infeasible. Fortunately, the logarithm is a slowly ascending function, where $log_e(256) \approx log(64) \approx 5$. This means that as long as the random graph that is being mapped has an average degree less than 5, the placement algorithm can effectively minimize connection lengths and solve for the cost function. Since the Core only supports 8 I/Os, it is reasonable to assume that the average degree of a core would be less than 4 (half that of the maximum I/O). Empirical evidence from simulations with $\lambda = 4$ suggests that after placement, more than 90% of connections come within a distance of 3 from any node and can be seen by Figure 2.8 (c). Hence we have a good estimate of the number of hops, 'K', which the network should support. The probability distribution of connections/edges between two nodes on a grid (after cost function minimization) can also be seen pictorially in 2-D in Figure 2.9.

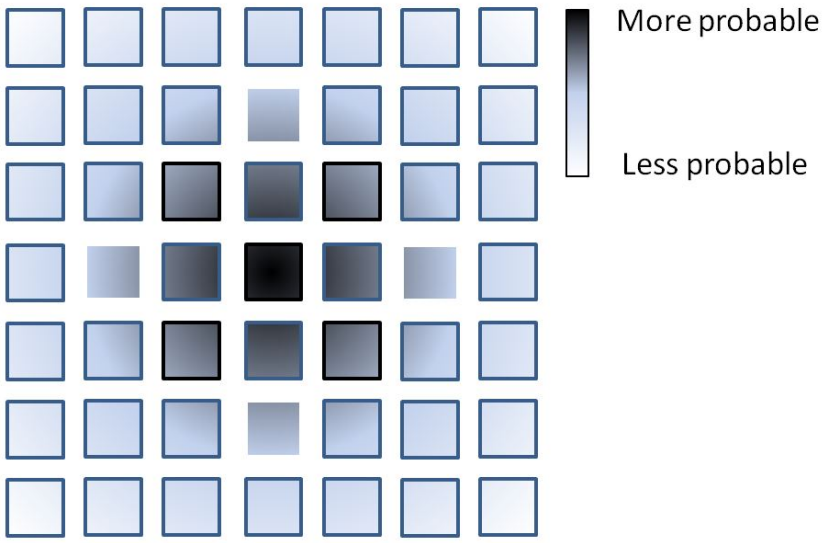


Figure 2.9: 2-D Probability Distribution

2.3.3 3 Layered Fabric and the Vertical Stack

The network in Figure 2.7 (c) covers most of what a desirable interconnect should have, except it has little support for general algorithms. Therefore this architecture is taken as a starting point. It can be observed that although the network is connecting LUTS/Cores/Clusters across a 2-D spacial plane (a physical 2-D chip), the particular network is inherently 1-D and is not able to exploit properly the spacial dimension of nearby LUTS/Cores. From a connection standpoint the compute elements are arranged in a 1-D line whereas on chip they will be arranged on a 2-D grid. To fix this the architecture has to be generalized to a 2-D switch box architecture. In order to hop a certain number of LUTS 'K', the signal needs to find its way to a certain layer 'K'. It is also known that $K = 3$ for this design (to satisfy the speed constraint). Generalizing these concepts, the architecture in Figure 2.10 is presented. It consists of 3 routing layers of 2-D switch boxes that can route signals up and down as well as in all 4 directions in their plane. The first layer is used for nearest neighbor hopping, the

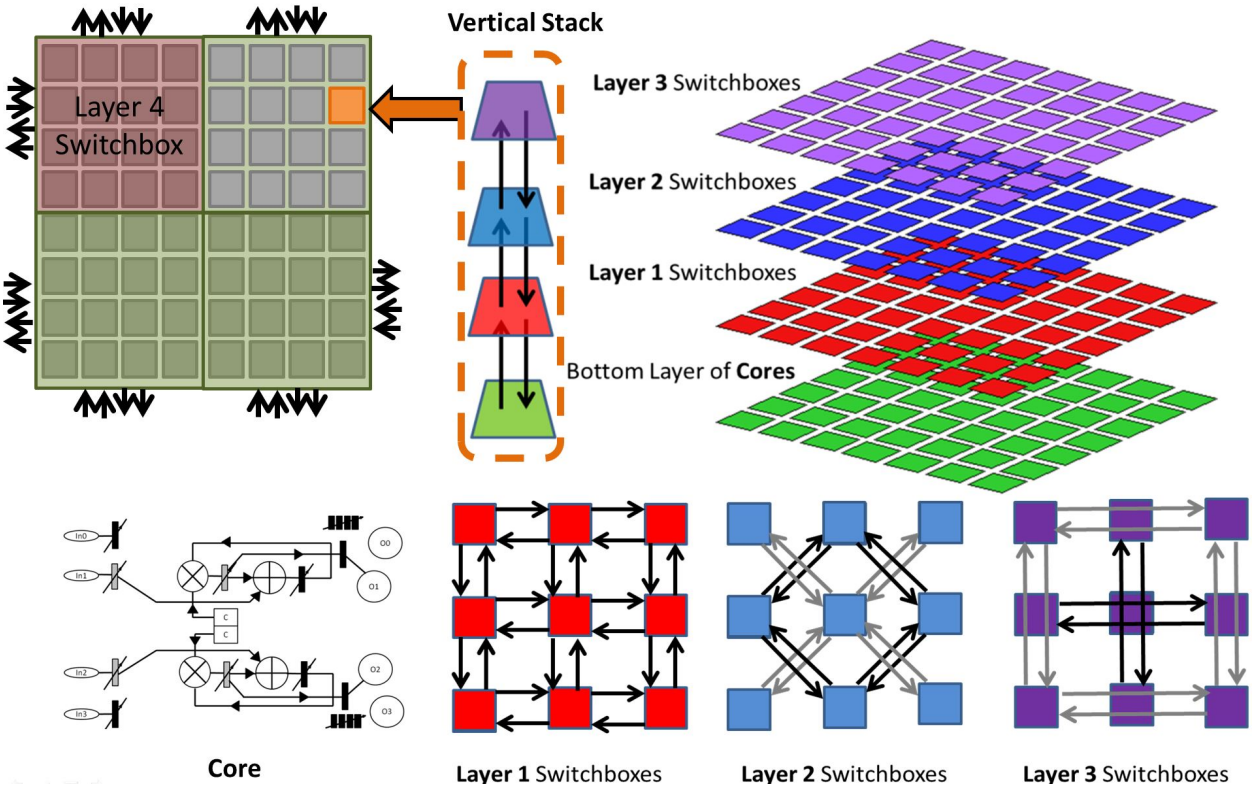


Figure 2.10: Routing Layer Fabric

next layer is used for Cores that are a distance of 2 hops away, and the last layer is used for routes that are a distance of 3 hops away.

The figure only shows a logical 3-D implementation, the actual implementation is still on a 2D silicon wafer. To analyze this layer, it is compared against the metrics described earlier.

1. Supports easy connectivity with closer proximity cores for high speed @ 1GHz
2. Supports long connections in case they are needed (and clock is relaxed)
3. Is of order $O(n)$ and infinitely scalable
4. Highly modular and repetitive design. (Programming and automation is covered in Chapter 3)
5. Agnostic to the internals of the core and hence supports heterogeneity.

The last two constraints left are that after placement cost minimization, there are more nearest neighbor connections and fewer long distance connections. This is easily solved by keeping a larger (8 times larger) amount of routing tokens in the first layer, and reduced tokens in the second and third layers. This also makes the design more hardware efficient, as longer connections are less likely after placement. The routing layer is delay-less which helps in tight retiming of algorithms. It preserves the speed of all DFGs (Data Flow Graphs) that have small iteration bounds (covered in more detail in Chapter 3). Since there is no clock tree, idle power being consumed is also solely due to leakage making the layer is energy efficient. Provisions have been made in the final version to force zero data signals to propagate in unused routing lines, to avoid unnecessary switching losses as well. For hardware synthesis, the three switch boxes above a core are bundled together with the core in what is called the Vertical Stack. In it the routing overhead is about 25% of the total area which is far better than FPGAs. This is because the UDSP is a coarse grain reconfigurable processor whereas FPGAs are finer grain and are required to do bit level manipulations. The Vertical Stack is templatizable and infinitely tile-able (scalable). All switch boxes in the routing layer have 8 instructions and the program counter for them are shared with that of the core, so that the entire UDSP can shift its routing and core functions within a clock cycle. Layer 1 Switch boxes each have 8 tokens with a 90 bit instruction word, layer 2 Switch boxes have 4 tokens with a 24 bit instruction word and layer 3 Switch boxes have 4 tokens with a 24 bit instruction word.

In addition to the three delay-less layers there is another 4rth registered layer - the I/O layer. The *I/O layer* allows connections at the borders of the UDSP to be routed to any of the internal cores and vice versa. It consists of identical replicated switch boxes in an additional layer (similar to the *Routing Network* layers), but unlike the *Routing Network*, does not support multiple instructions (only one instruction). The network consists of delays which should not affect the performance of any of the algorithms since the algorithms' internal loop bounds are never subject to this layer (it only deals with the border input/output

connections). This layer has a very small overhead, with each switch box covering 16 Vertical Stacks at a time. The layer connects to the Upper I/O of the 3rd routing layer of each Vertical Stack. In total the layer spans a 9 X 9 grid of 81 spatial UDSP Cores. For this design the total chip area was the limiting factor, and a total of 81 cores could be manufactured in that reasonable area. On the borders (in the current design) Layer 4 is connected to 4 inputs on the left, 2 inputs/outputs on the top, and one output on the right. Figure 2.10 shows the concept of this layer in conjunction with the *Vertical Stack*.

The internals of each switch box have the potential to contribute to the 'amount of flexibility in routing' and the 'overhead incurred due to that flexibility.' Figure 2.11 (a) shows an ideal switch box where each input has the capability of being connected to every output irrespective of other outputs or connections. This approach is not scalable, furthermore in a real scenario most connections would go unused (underutilized) and would contribute to dark silicon. Adjacent in the same figure is a '2-layer-deep' switch box design with sparse connections (like the one used in the UDSP), which makes the switch box much smaller in area and power at the expense of routing runtime (incurred by the software trying to route through this switchbox) as well as 'Routability' (the ability of a switch box to support different combinations of connections). Intuitively – 'routability' as a metric, can be defined as the amount of $information$ that a switch box can propagate. This can further divided into information propagation for every layer in a switch box including the input and output layer. Table 2.2 below shows an example switchbox matrix that has 7 inputs and 5 outputs that has good routability. A 'TRUE' entry in the table confirms a connection between an input and output. A 'FALSE' entry mean there is no hardware connection between that input and output pair. Routability is formally defined in equation 2.4

Table 2.2: Example: Switchbox Routing Matrix

In/Out	Out0	Out1	Out2	Out3	Out4
In0	0	0	0	1	0
In1	0	1	1	0	1
In2	1	1	1	1	1
In3	1	1	0	0	0
In4	1	1	0	1	1
In5	0	0	1	0	1
In6	1	0	1	1	0

$$\begin{aligned} InputCrossCorrelation_{ij} &= \boldsymbol{Row_i} \cdot \boldsymbol{Row_j} \\ OutputCrossCorrelation_{ij} &= \boldsymbol{Col_i} \cdot \boldsymbol{Col_j} \end{aligned} \tag{2.2}$$

$$\begin{aligned} InputCrossCorrelation &= \sum_{i=1}^{N_{Rows}} \sum_{j=i+1}^{M_{Rows}} \boldsymbol{Row_i} \cdot \boldsymbol{Row_j} \\ OutputCrossCorrelation &= \sum_{i=1}^{N_{Col}} \sum_{j=i+1}^{M_{Col}} \boldsymbol{Col_i} \cdot \boldsymbol{Col_j} \end{aligned} \tag{2.3}$$

$$Routability = (InputCrossCorrelation \cdot OutputCrossCorrelation)^{-1} \tag{2.4}$$

The connections between each of the layers are chosen judiciously by minimizing cross correlations of all input/output combinations, allowing for maximum connectivity possible with a particular level of sparsity. Intuitively the cross correlation of 2 inputs shows how similarly are they connected to the next layer. If cross correlations are minimized while keeping the total number of connections fixed, this results in the maximum amount of information flow from the first layer to the next. Similarly minimizing the cross correlations of the outputs of a switch box maximizes the amount of information they can receive. The second layer is usually fully connected for simplicity, and the correlation analysis is only performed on the first layer. The *variance* of the correlations at the input - is strongly correlated with output cross correlation. This can be seen in Figure 2.11 (b), and it reduces the problem to how only the inputs are connected to the next layer. Therefore to design a switchbox, a

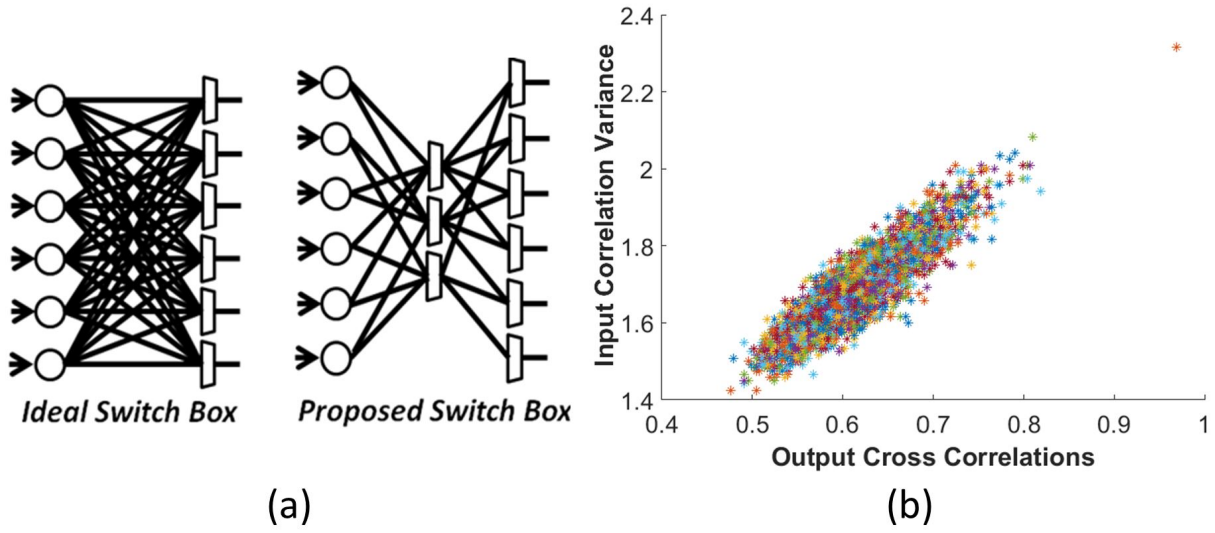


Figure 2.11: Switch Box Design

connection matrix needs to be found such that the cross correlation mean and variance (at the input) is minimized.

The remaining parameter is sparsity - the number of connections supported by a switch box layer. This depends on the 'degree distribution' of the nodes (Cores) of the underlying graph (Algorithm). In the case of UDSP, the maximum 'degree distribution' of a Core is known to be 8, since each Core has 4 inputs and 4 outputs. This fact is exploited to give the maximum sparsity while maintaining complete flexibility during routing connections by keeping 8 tokens in the first routing layer. Since higher routing layers are expected to be seldom used, their routing tokens are limited to 4 each.

2.4 The Control Architecture

The control module is responsible for programming and debugging the UDSP. As a requirement from the vendor, the external interface protocol to this module needs to be JTAG. A single Vertical Stack has an instruction length of 2Kb (Kilobits), and the control for the UDSP was designed to support a maximum of 256 cores. That gives roughly 512 Kb of

instructions. In order to prevent large instruction memory processing (and hence power and area penalty in hardware), programming is performed in frames. Each frame contains either a Vertical Stack's instruction or it a control instruction. The control module is designed to act as a bridge between JTAG and the UDSP Cores. The software mapper (described in Chapter 3) uses JTAG to communicate the 2Kb frames. Another requirement for the control is to be able to handle instructions from an embedded FPGA elsewhere on chip, as well as directly from external test pins. Figure 2.12 shows the different sections of the control module. All three inputs, JTAG, eFPGA and Direct Serial, can be taken in concurrently. Then a priority checker determines which one has precedence, and that instruction is forwarded for distribution. The distribution module determines the instruction type and the choses the appropriate sub module to deliver the partial or complete instruction to. The Program counter module is responsible for selecting which one of the 8 temporal states of the core is active. The counter is programmable to support multiples forms of temporal jumps and adds the temporal direction to algorithms. Layer 4 programmer programs the I/Os of the UDSP. The Vertical Stack Programmer programs the routing layer as well as the cores, one vertical stack at a time. The Soft Reset block handles the different types of resets in the Cores. Since each core has 4 internal states, there is a reset pin to refresh all of them before a new iteration of the same algorithm is started, without erasing the algorithm from memory. The Observer module is placed to probe data signals from the DSP in order to find out if there are any faults in the system. These outputs can then be serially read from one of the serial interfaces (except eFPGA).

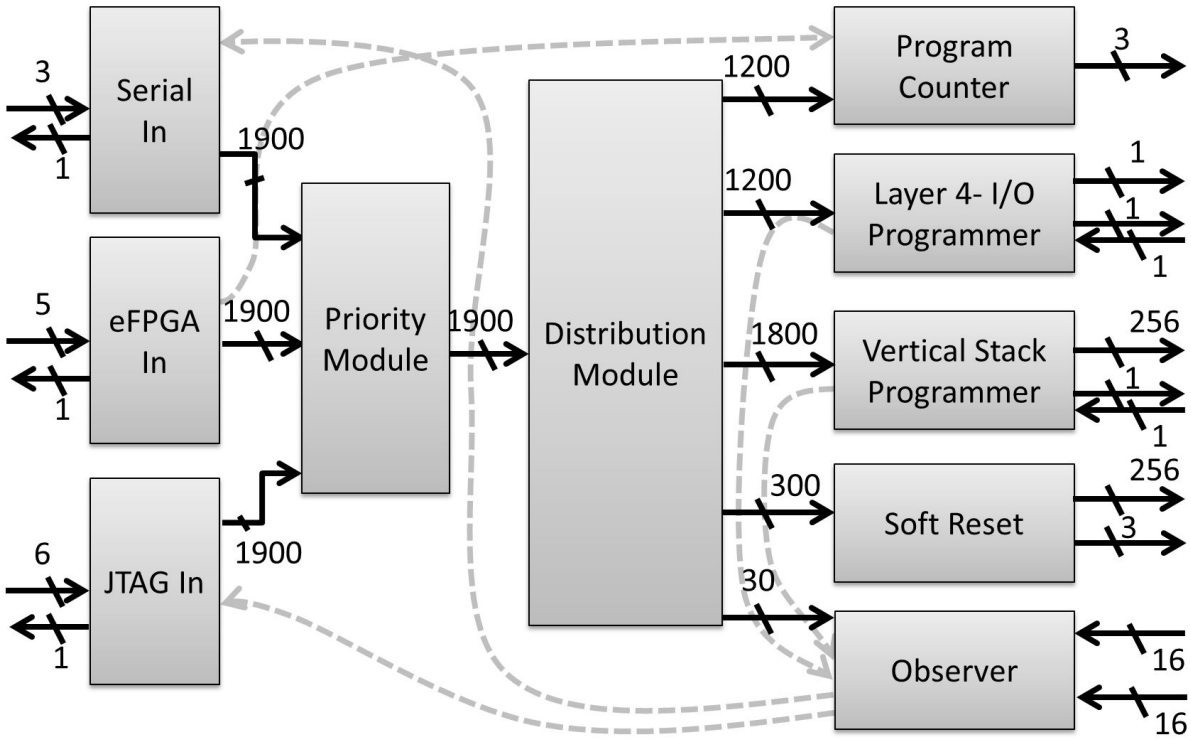


Figure 2.12: Control Module

2.5 Summary of Contributions

In this chapter, the hardware architecture of a universal digital signal processor is presented composed of 3 building blocks. The basic compute unit (Core architecture) is shown which supports fundamental operations e.g. FIR/IIR filters, FFT and single cycle MACs for matrix multiplication. The interconnect fabric is developed which is responsible for high speed data communication between cores and external I/Os. And last the control circuit is made to program the individual vertical stacks as well as probe the design for debugging.

The contributions of this thesis are:

- A 1-hop programmable core architecture for static data flow graphs with instruction memory and internal data states for temporal multiplexing. The core can perform 2 16-bit single cycle MACs at 1GHz

- A novel design of a scalable, delay-less, area and energy efficient routing interconnect that satisfies the 1GHz timing requirement.
- A definition and an in-depth analysis of a novel 'Routability' metric which is used to design the internals of switch boxes for maximum area and energy efficiency metrics.

CHAPTER 3

Software Compiler Design

3.1 Background and Toolflow Overview

The structural design of the UDSP is a co-design between minimizing the hardware that goes into the UDSP and the minimizing the time it takes to program the UDSP. The *ease of programmability (EOP)* is a difficult metric to define, since it varies widely across different architectures of processors. Memory centric processors have different requirements for compilers than parallel data centric processors. One measure of EOP can be the time complexity required for scheduling of hardware resources. Modern compilers and schedulers often rely on Integer linear Programming (ILP) to do the task, and ILP is known to be an NP-Complete problem [22]. This leaves 'time to compile' the only direct way to compare two solutions allowing lesser room to meaningfully compare compilers across architectures. Despite this, intuitive reasoning can give guidelines to design hardware friendlier to compilers and programming. EOP is inversely dependent on the hardware overhead incurred in terms of the overall area and power efficiency of the design. To make programming easier, extra hardware needs to be employed, changing a few data paths to make the flow of data easier. This intern has a Power and Energy efficiency cost on the overall compute capability. An example of this would be a highly programmable x86 processor that has lower compute efficiency to something like a GPU, where the GPU is difficult to program and often relies on the prowess of the programmer as well as the compiler.

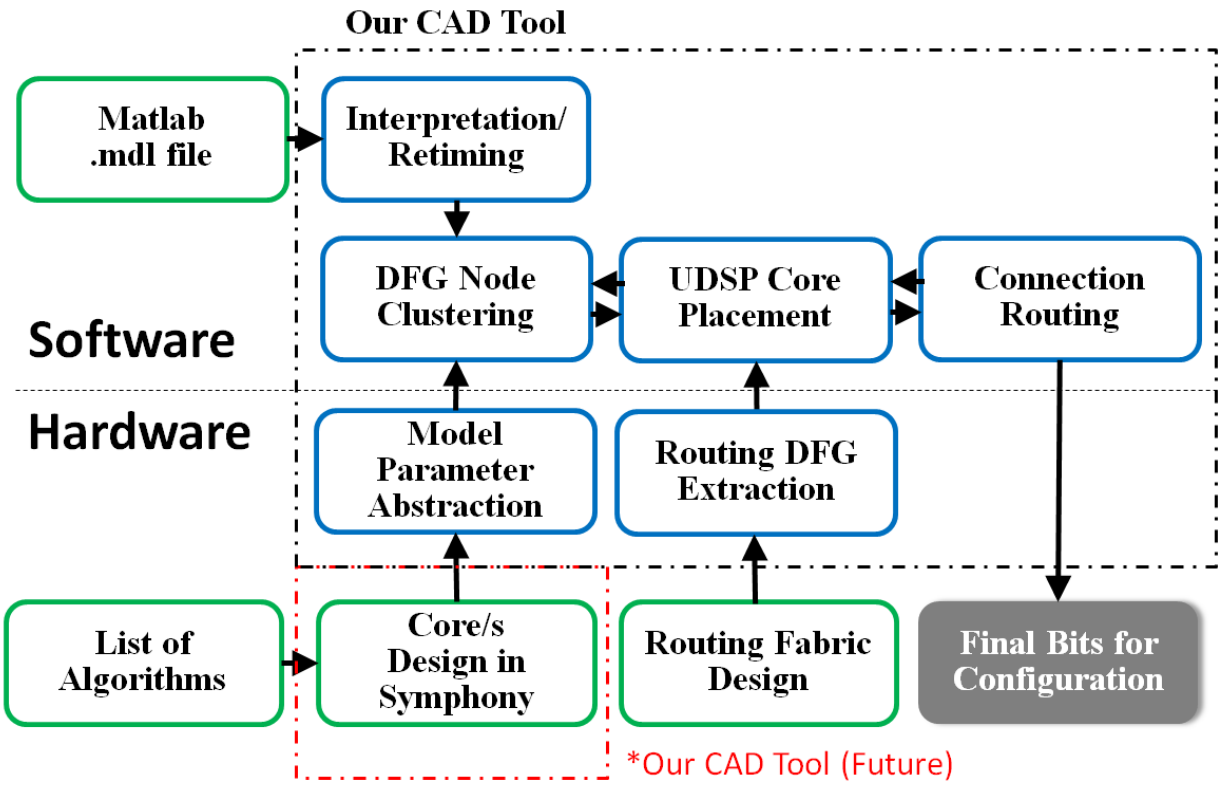


Figure 3.1: Toolflow Overview

Although concrete hardware efficiency metrics exist (such as $GOPS/mW$ and $GOPS/mm^2$), concrete hardware programmability metrics don't exist (EOP is poorly defined) - as a result they are not used in the design process as trade-offs with other hardware metrics. Instead architects often rely on their best judgment to come up with a reasonable compromise. Due to the co-design nature of the hardware and software, and the complex reconfigurability offered by the UDSP, a custom mapper/compiler is required to program the chip. As part of this thesis *Multi Core Mapper* - an in-house software tool, is developed to take user-defined software algorithms and generate the hardware programming bits necessary to map it onto the UDSP Array. The following Figure 3.1 elaborates the scope of the tool.

The tool flow closely resembles an FPGA mapping flow as the underlying problem of configuring flexible hardware is fundamentally the same. This tool is designed to be friendly to Simulink users. It takes an algorithm file as input in .mdl format (Simulink diagram format). The designer of the algorithm can place blocks form a custom UDSP library provided

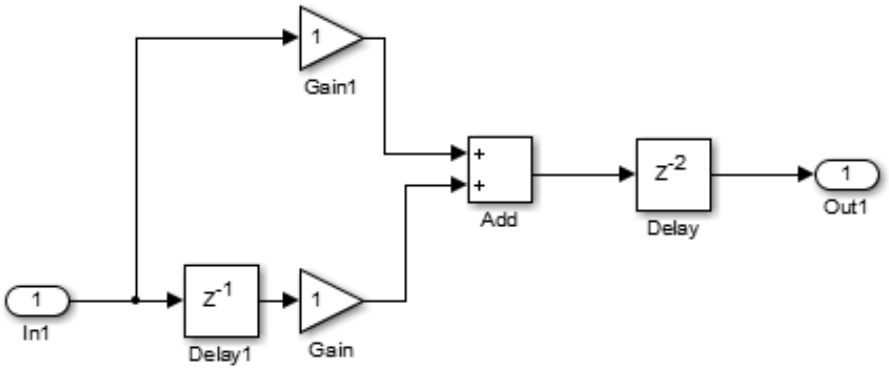


Figure 3.2: A Direct Mapped Architecture Example

which breaks the design down into basic units the UDSP supports. An example algorithm may look like the one shown in Figure 3.2. This data path has 2 gain elements, an adder and a couple of I/Os. In general this representation is referred to as a *direct mapped architecture* of the algorithm.

3.2 Direct Mapped Architecture and Data Flow Graphs

Direct Mapped Architectures, can be divided into two classes. The first class has no loops in the data flow. No computation depends on previous results of a future computation. An FIR filter and the flow graph in Figure 3.2 are examples of this class. The second class contains all algorithms that have loops, referred to as precedence constraints, where a current computation depends on the results of previous future computations. Such architectures have computational dependency and are difficult to pipeline and parallelize. An example of this kind of architecture would be an IIR filter. Figure 3.3 shows examples of both kinds of architectures one being a simple FIR filter and the other being a Complex Multiply Accumulate (MAC) operation. Such complex algorithms can be conveniently represented using high level

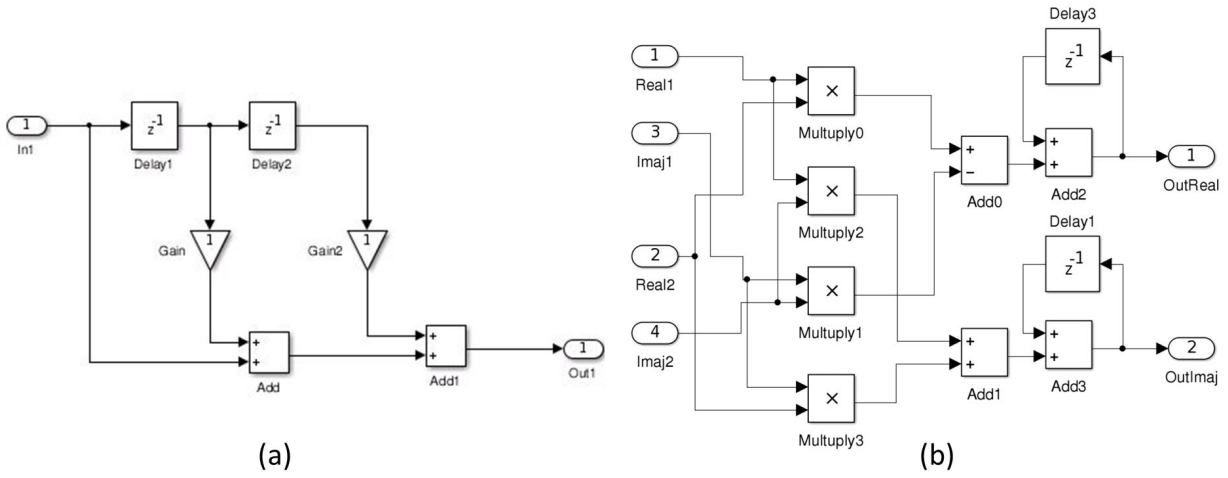


Figure 3.3: (a) FIR Filter, (b) Complex MAC

descriptions. An efficient high level description for analyzing data flow properties is directed graphs – where the nodes of the graph represent computation elements and the edges represent data flows. These are called *Data Flow Graphs (DFG)* and they are more amenable to transformations and easily convertible to Verilog/VHDL and subsequently mapped to hardware and FPGAs. They act as a bridge between algorithms and architectural implementations [23].

In a Data Flow Graph, edges may have non-negative integer delays associated with them, and computation nodes may have internal latencies associated with them. A loop is a path in the directed DFG which starts and ends from the same computational node. It is also called a cycle and must contain within it at least one delay element. An example loop is shown in Figure 3.4. An important concept in DFGs is the loop bound which is defined as the ratio of total computation time T_c to the total number or registers or delay elements in the loop T_d – shown in Equation 3.1. For the loop in Figure 3.4, the delays are denoted by the edge weights and the node computation times are indicated inside the nodes (in normalized time units). The loop bound is calculated to be 2 t.u. The loop bound places an upper limit to the speed at which this loop can operate, which can be achieved if the delays are optimally rebalanced. DFGs can have multiple loops, however the critical loop is

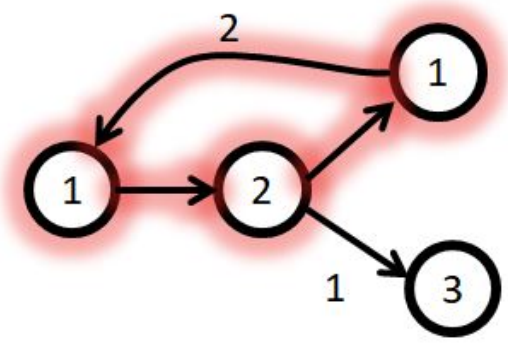


Figure 3.4: An Example DFG Loop

the one with the highest loop bound. This is called the iteration bound of that algorithm. The algorithm or DFG cannot operate faster than that iteration bound which limits the maximum throughput [24].

$$LoopBound = \frac{T_c}{T_d} \tag{3.1}$$

3.3 Model Parameter Extraction (Core)

The UDSP is a direct DFG mappable hardware. Therefore it needs hardware parameters to generate the mapping. The hardware is abstracted into two layers. The core is abstracted separately and the routing layer is abstracted separately. The core is further abstracted as two matrices – the delay matrix and the configuration matrix. The dimension of the matrices is N x M where N is the number of signal generating (input) nodes and M is number of signal sinking (output) nodes. For the configuration matrix - the entry in the i'th row and the j'th column represents the port of the j'th sink node that the i'th source node is connected to. If a particular source node is not connected to a sink node the entry is zero. For example, the output of the Multiplier/Shifter 0 is connected to port 1 of Adder 0, and the output of the same Adder 0 is connected to port 1 of Multiplier/Shifter 0. If all the sources and sinks are

fully connected the matrix would be fully populated. With fewer connections the matrix is sparse. The table 3.1 shows the connectivity matrix of a Core as an example. The matrix is a convenient representation of the possible configurations of a Core. It will subsequently be used by the mapper to map DFGs to this Core.

Table 3.1: Connectivity Matrix (Core V3.4)

Src/Sink	MultShft0	MultShft1	Addr0	Addr1	Const0	Const1	Out0	Out1	Out2	Out3
In0	[2]	[2]	[1]				[1]			
In1	[1]	[2]		[2]	[1]			[1]		
In2	[2]	[1]	[2]			[1]			[1]	
In3	[2]	[2]		[1]						[1]
MultShft0			[1]	[2]			[1]	[1]		
MultShft1			[2]	[1]					[1]	[1]
Adder0	[1]	[1]		[2]				[1]	[1]	
Adder1	[1]	[1]	[2]					[1]	[1]	
Const0	[2]		[2]							
Const1		[2]		[2]						

Table 3.2: Delay Matrix (Core V3.4)

Src/Sink	MultShft0	MultShft1	Addr0	Addr1	Const0	Const1	Out0	Out1	Out2	Out3
In0	[1 2]	[1 2]	[0 1]				[1-8]			
In1	[1 2]	[1 2]		[0 1]	[0]			[1 2]		
In2	[1 2]	[1 2]	[0 1]			[0]			[1 2]	
In3	[1 2]	[1 2]		[0 1]						[1-8]
MultShft0			[0 1]	[0 1]			[0 1]	[0 1]		
MultShft1			[0 1]	[0 1]					[0 1]	[0 1]
Adder0	[1 2]	[1 2]		[1 2]				[1 2]	[1 2]	
Adder1	[1 2]	[1 2]	[1 2]					[1 2]	[1 2]	
Const0	[0]		[0]							
Const1		[0]		[0]						

Table 3.2 shows the delay matrix for the Core. Here the entries on the matrix represent the list of possible delay elements between a source and a sink. If there are multiple entries in a list, it means that connection supports variable delays and can be programmed to one of the values shown. The tool flow later uses this matrix to retime the DFG.

3.4 Interpretation, Retiming and Clustering

The tool flow is implemented in MATLAB. First the .mdl model file is broken down into a DFG that only contains elements supported by the underlying UDSP Cores. These include shifters, multipliers, constant banks and add/subtract arithmetic units. The Graph is then 'levelized' - where each node in the graph is assigned a positive rank such that all incoming connections to that node come from a node ranked less than the rank of this node. As an example the FIR filter shown in Figure 3.3 (a) transforms into Figure 3.5 after the 'levelize' step.

Intuitively, the levels can be interpreted as the flow of the computation, and help in scheduling the tasks by applying algorithms such as ASAP or ALAP. However, if a loop appears, it needs to be broken at an appropriate point to make sure the graph is assigned the rank properly. Once the nodes are each assigned a level, the tool proceeds to re-time the DFG. The constraints of the delay matrix put simply are that every multiplier/shifter has a delay element at its input and every add/subtract unit has a delay element at its output. These simplifications combined with the fact that no delay line can support more than two delays (unless it is specifically a long delay line), the retiming proceeds to balance the delays of the adders and multipliers first by inserting pipelining delays to the flow of the DFG, being careful to not add extra delays to any loops to preserve functionality of the algorithm. Some Delays such as those of Constants are set to 'don't care'. In loops the delays are moved around to satisfy this constraint as well, but no extra delays are added. The effect of retiming on the DFG in Figure 3.5 is shown in Figure 3.6. Once these constraints are met, the tool proceeds to clustering the DFG. The clustering algorithm is a derivation from [25] and is a form of a greedy algorithm which tries to keep as many nodes in a cluster as possible.

In this algorithm, the parent node tries to merge with its child node if possible- if that is not possible due to resource limitations in a cluster then it creates a new cluster. The clusters are one to one mappings to Cores in the DSP. In the functions AnyResourceCombine and

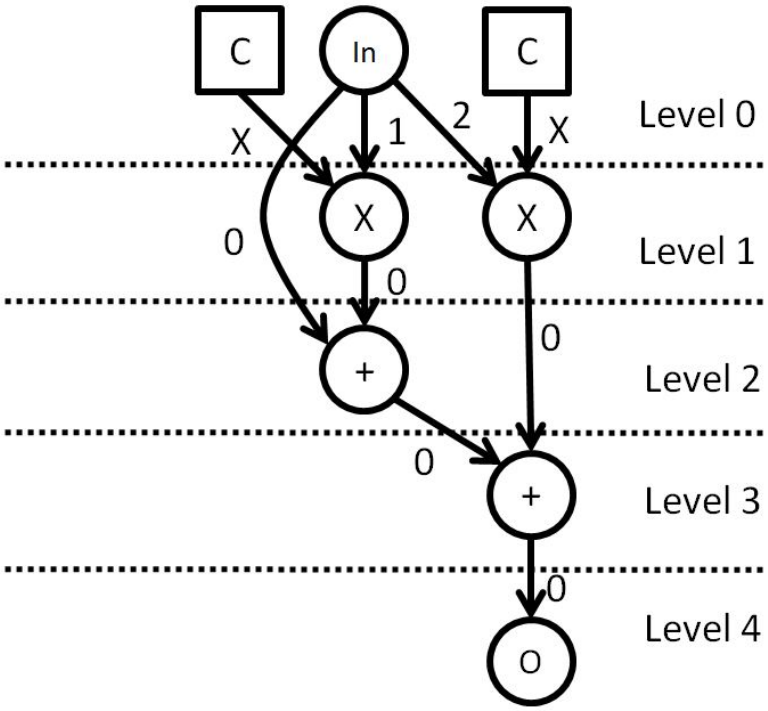


Figure 3.5: DFG After Levelize

Algorithm 1 Clustering

```
1: function AddToClusters(Node,ClusterList)
2:     ChildClusters ← get-children-clusters-of(Node,ClusterList)
3:     if AllResourceCombine(Node, ChildClusters) then
4:         return Combine(Node,ChildClusters)
5:     else if C ← AnyResourceCombine(Node, ChildClusters) then
6:         return AddToCluster(C,Node)
7:     else
8:         return CreateNewCluster(Node)
9: function ClusterLevels()
10:    ClusterList ← empty-list
11:    ClusterListNextLevel ← empty-list
12:    for level ← 0 to toplevel do
13:        for all nodes Node at level level do
14:            NewCluster ← AddToClusters(Node,ClusterList)
15:            add NewCluster to ClusterListNextLevel
16:
17:        ClusterList ← ClusterListNextLevel
18:        ClusterListNextLevel ← empty − list
```

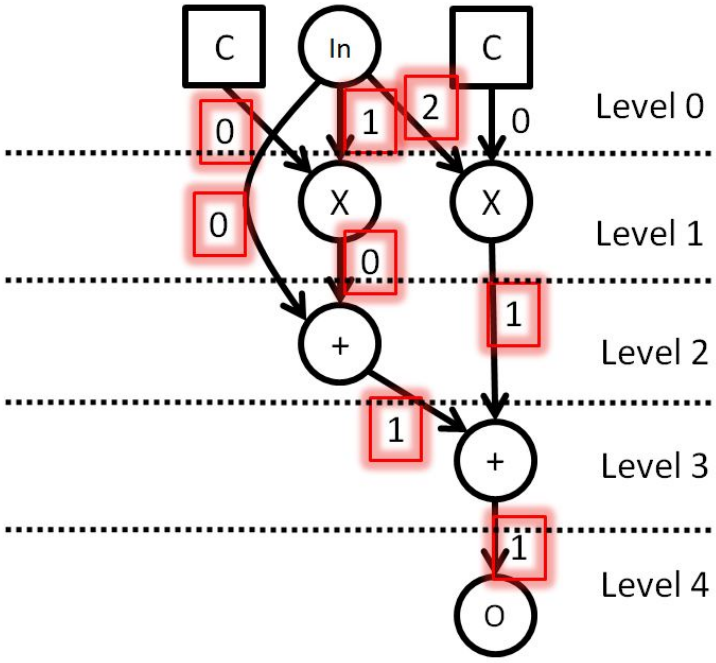


Figure 3.6: DFG After Retiming

AllResourceCombine, the mapper performs a step called 'resource binding'. This is where it checks all the possible states of the configuration matrix and the delay matrix which either allow the addition of a new computation node to this cluster or do not allow it. Since the Core is small and has a limited number of possible states (half of them being symmetric since the core is symmetric), it is possible to brute force this checking function in a negligible amount of time. For more complex core designs, brute force may not be computationally feasible. The clustering algorithm can be seen working on the DFG of the FIR filter in Figure 3.7 (a)-(e). This particular algorithm can be mapped to one Core since everything fits inside a cluster. As a pictorial example, this mapping is shown in Figure 3.7 (f). The actual core connections are much more diverse and complex - as can be seen by the connectivity matrix and the delay matrix and are shown partially for clarity. Figure 3.8 shows an example of how the clustering algorithm will cluster if the input was a multicore mappable input.

Duplication of computation or delay nodes sometimes becomes necessary to reduce strain

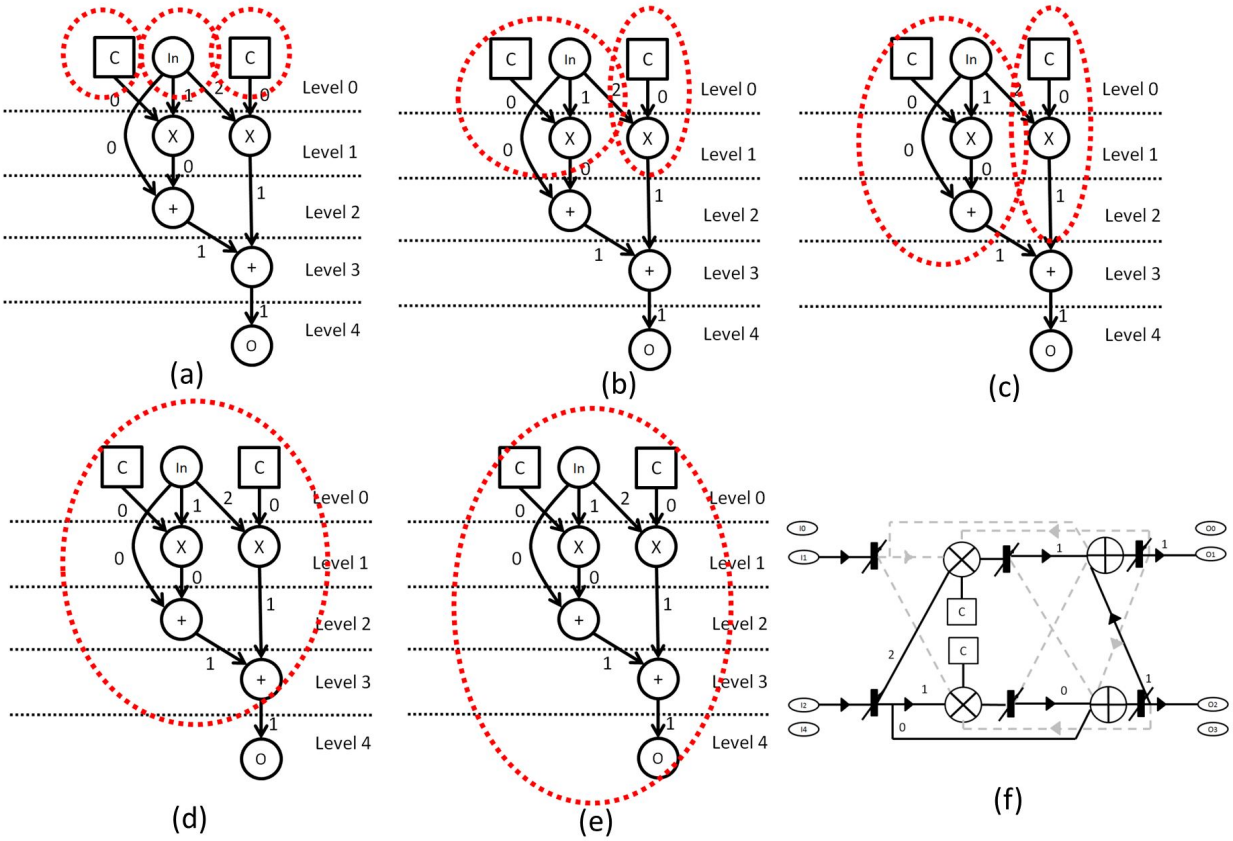


Figure 3.7: Steps During Clustering

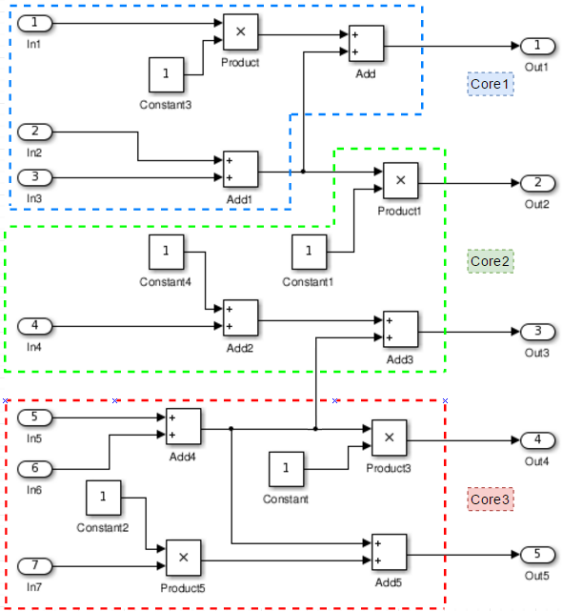


Figure 3.8: Multicore Clustering

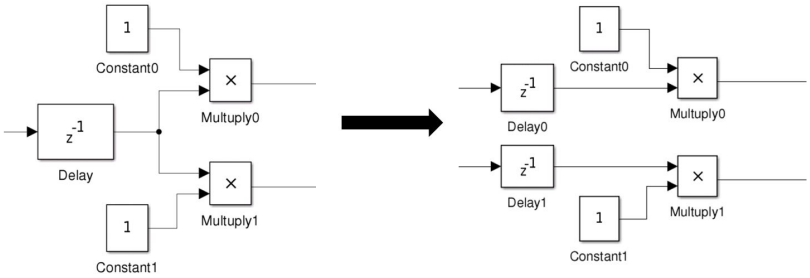


Figure 3.9: Delay Split

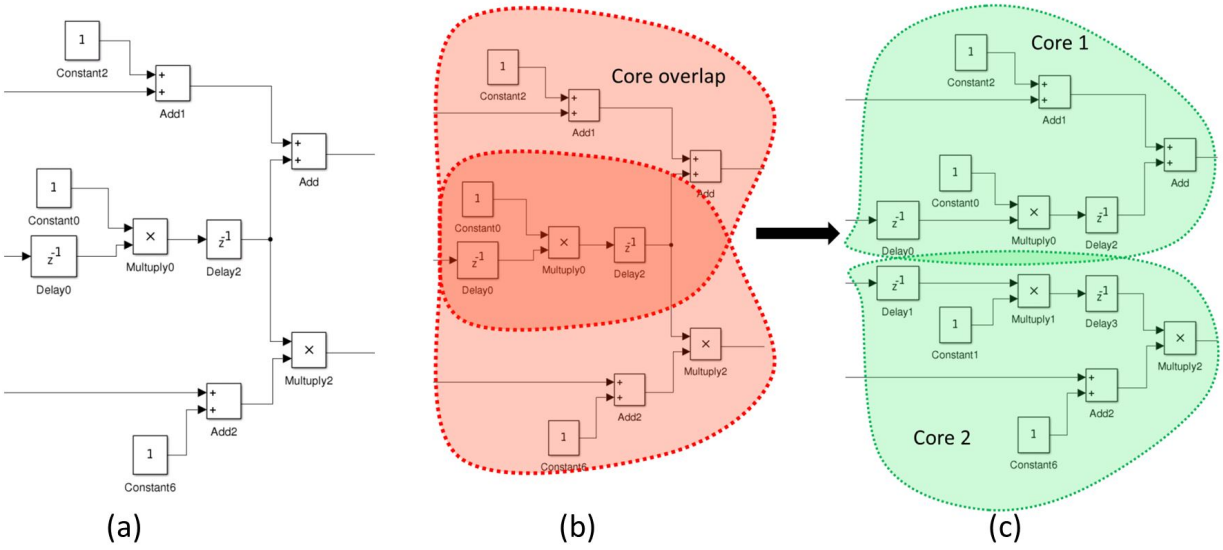


Figure 3.10: Core Split

on the routing layer. This comes at an extra power penalty but makes the design mappable – a necessary cost. Figure 3.9 shows a frequent case of duplicating delays. Whereas Figure 3.10 (a)-(c) shows a less frequent case of duplicating a computational node when the routing layer is limited in the number of connections it can offer.

3.5 Routing DFG Extraction (Routing Layer)

This step is the analogous to the Model Parameter Extraction step, but for the routing layer, which is the interconnect fabric that binds the UDSP cores. Due to several possible connections in a switch box and an exponentially more in the Vertical Stack, storing the interconnect information is not straight forward. A pseudo adjacency matrix (like the connectivity matrix of the core) is not viable here due to storage constraints. The exponential number of vertices in the routing layer makes the matrix size large. Instead we represent this as a DFG where the core I/Os are the start and end points of the data words and each mux in the path is like a node, with many data inflows and one data outflow. The graph model is directly extracted from the hardware Verilog as it eliminates the possibility of human error in assigning the configuration bits to the multiplexers in the path. Direct extraction also saves time if the hyper parameters of the routing fabric are changed (for example in later design iterations). This representation of the routing nodes is inefficient for computation as compared to the adjacency matrix, however, it requires less storage memory. Figure 3.11 shows this pictorially at the scale of a single switchbox (note: the entire routing layer has 243 switchboxes). Here (a) is the supposed actual hardware implementation of a switch box, (b) is the equivalent DFG and (c) is an example of how it would look after the last step of routing connections.

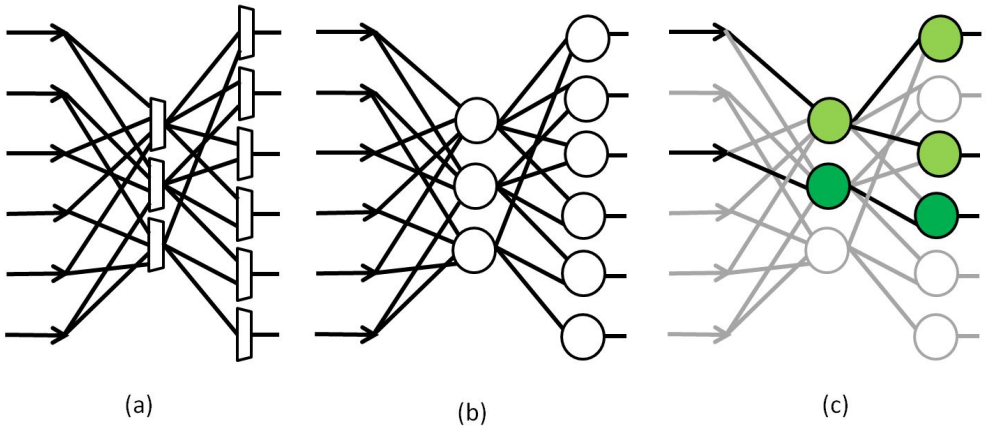


Figure 3.11: Routing Model Extraction

3.6 Placement and Routing

The 2-D discrete placement problem is a well-known problem from the FPGA space. The problem is as follows:

Assign the clusters made in the clustering step to logical cores on the UDSP grid, where the goal is to minimize the routing distance (under the assumption that), this would allow successful routing later on. An additional goal for the UDSP is to meet timing constraints of <1ns end to end since the UDSP is advertised to run at 1GHz.

This additional goal is met by co-designing the hardware and software. During the design of the routing layer (from Chapter 2), the number of layers are kept at 3 - the intuitive argument being, with the number of input/outputs from one core not exceeding 8, it is highly unlikely that an optimally placed graph has longer connection lengths than a distance of 3. Using this, the Vertical Stack is designed to have an I/O to boundary delay of <500ps for any of the three layers' outputs or inputs. Although layer 1 should have a smaller routing delay than layer 2, with layer 3 having the largest delay (since the signal has to travel through layer 1 and 2), by relaxing the timing on the first two layers all the way up to 500ps, the synthesis tool optimizes the sizes of the gates that had far reaching connections (layer 3) vs. those that had smaller distance connections (layer 1). This flexibility allows the tool to lower the power

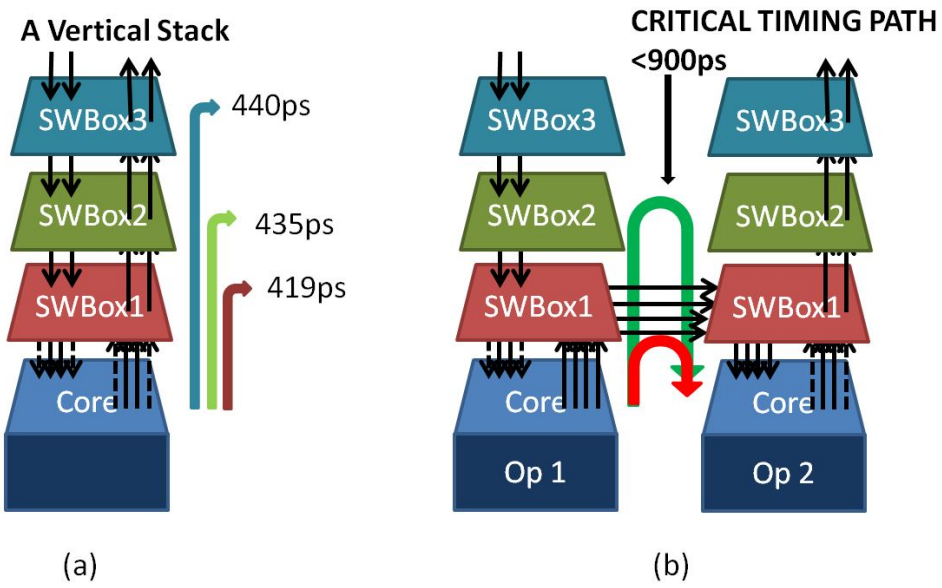


Figure 3.12: Vertical Stack Timing

and area of the design while still meeting the timing on all layers equally. Figure 3.12 (a) shows the different timing delays for a single Vertical Stack satisfying the 500ps constraint. Figure 3.12 (b) shows two vertical stacks communicating with each other through different layers satisfying the timing by a margin of 100ps. For the mapper to guarantee operation of the UDSP at 1GHz it needs to ensure that no connection length exceeds '3'. Anything larger than 3 would require 2 hop routing and would break down the 1ns timing constraint.

This is a significant constraint difference between an FPGA and a UDSP, where the former has the flexibility of adapting its speed according to the critical path of the algorithm, the later has a fixed timing to meet and must adapt the algorithm (through retiming or duplication etc.) and the placement of cores to do so. Restating the placement problem:

Assign the clusters made in the clustering step to logical cores on the UDSP grid, where the goal is to minimize the routing distance (under the assumption that), this would allow successful routing later on, ***with no distance exceeding 3****.*

Placement is an NP-Complete problem, which means that there is no efficient way to compute a solution using a computer program. Several heuristics have been developed, of which 3 are compared here and the best selected. A brief over view of each is as follows:

Partitioning based placement: Decomposes a complex system into smaller subsystems, and recursively applies min-cut partitioning to map the cluster list onto the UDSP core grid. It has the advantage of having a flexible cost function where the user can minimize the net cut or the edge cut etc, and it is a move based algorithm, which means it does one move per iteration and is suitable for timing driven placements. The disadvantage of this approach is there can be lots of 'indifferent' moves which represent a very flat region in the optimization space. This limits the kinds of cost functions that can be used.

Simulated Annealing: Mimics the annealing process used to gradually cool metal to produce higher quality structures [26]. The initial placement is random, and the algorithm then iteratively swaps the position of two clusters in the map if it reduces the cost - the cost being the total wire length. The annealing approach also accepts some swaps that increase the cost function with some probability which in the optimization space translates to getting out of a local minimum. The advantage of this approach is that it has a flexible cost function, where the user can optimize solely over the wire length or have a mix of the time it takes to converge vs. max wire length. Simulated annealing also has the potential to reach the globally optimal solution given enough time, but it is slow. The algorithm is shown in 2:

Algorithm 2 Simulated Annealing

```
    function SimulatedAnealing()
2:      Temperature ← initial-temp
        Placement ← initial-placement
4:      while Temperature > final − temp do
            for Few − Iterations do
6:              NewPlacement ← Perturb(Placement)
                ΔC ← Cost(NewPlacement) - Cost(Placement)
8:              if ΔC < 0 then
                    Placement ← NewPlacement
10:             else if Random(0, 1) > e^{−ΔC/Temperature} then
                    Placement ← NewPlacement
12:         Temperature ← ScheduleAneal(Temperature)
```

Vectoring: Mimics the motion of particles (cores/clusters) attached to each other with springs (wires). It is an idea inspired by physics and proposed as part of this thesis. The system of particles is initialized and is allowed to settle to its lowest energy state with particles probabilistically swapping positions in a particular direction based on the then total force on them. Longer wires exert more force than shorter wires. The advantage of this approach is that it is straightforward to include in the cost function the value of '3' as the maximum routing distance, since any connection of length more than 3 can be forced to exert infinite force. The disadvantage is that the solution may get stuck in a local minimum and depends highly on the initial placement. Many simulations are running in parallel to can diminish this problem.

The basic cost function that for all these algorithms is shown in Equation 3.2. The p^th norm is to vary the degree of penalty for longer connections. Figure 3.13 shows the before and after mapping of running the placement algorithm. This simple example illustrates how the placement algorithm works on the UDSP grid. A real example of mapping is shown in Figure 3.14 which shows a random initial placement on the left and a possible solution found through simulated annealing on the right [27].

$$WiringCost = \sum_{n=1}^{N_{wires}} |n_{xlen}^2 + n_{ylen}^2|^p \tag{3.2}$$

The UDSP has 8 temporal instructions, which means that the 2-D placement problem can be expanded to a 3-D placement problem with the third dimension being *time*. The wire distance is not straightforward to cast into the temporal direction since there is no physical distance is between two temporal instances of the same core, but there is a delay of one cycle to switch the instruction of the core. In this cycle the data needs to be registered and held in the core. Figure 3.15 illustrates the same algorithm as Figure 3.13 including optimization over the temporal direction. This example shows the ease with which the temporal reconfigurability can be incorporated into the UDSP design.

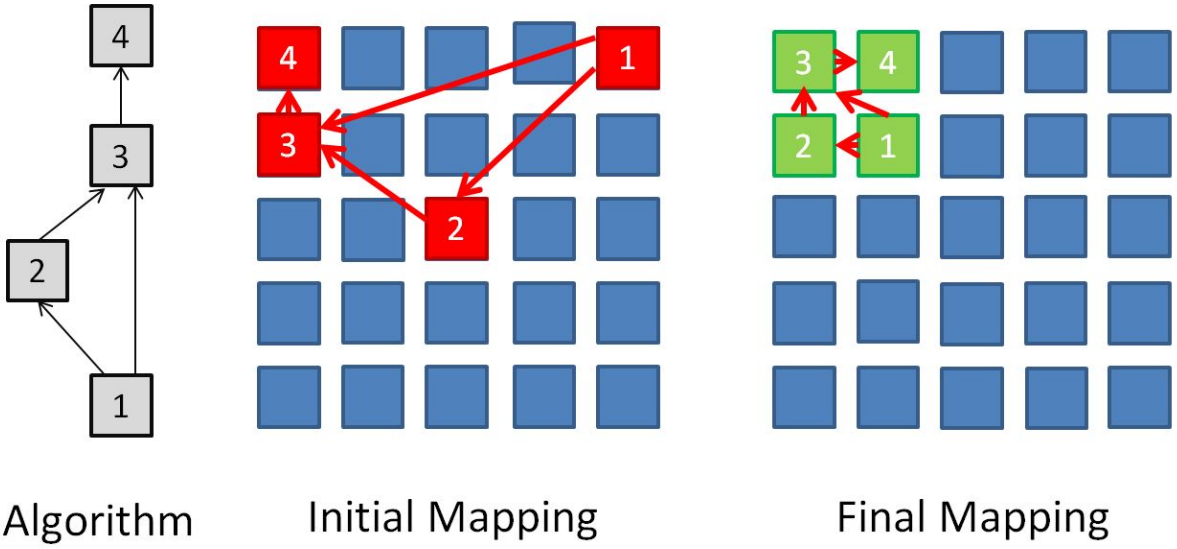


Figure 3.13: Cost Function Based Placement (Simple)

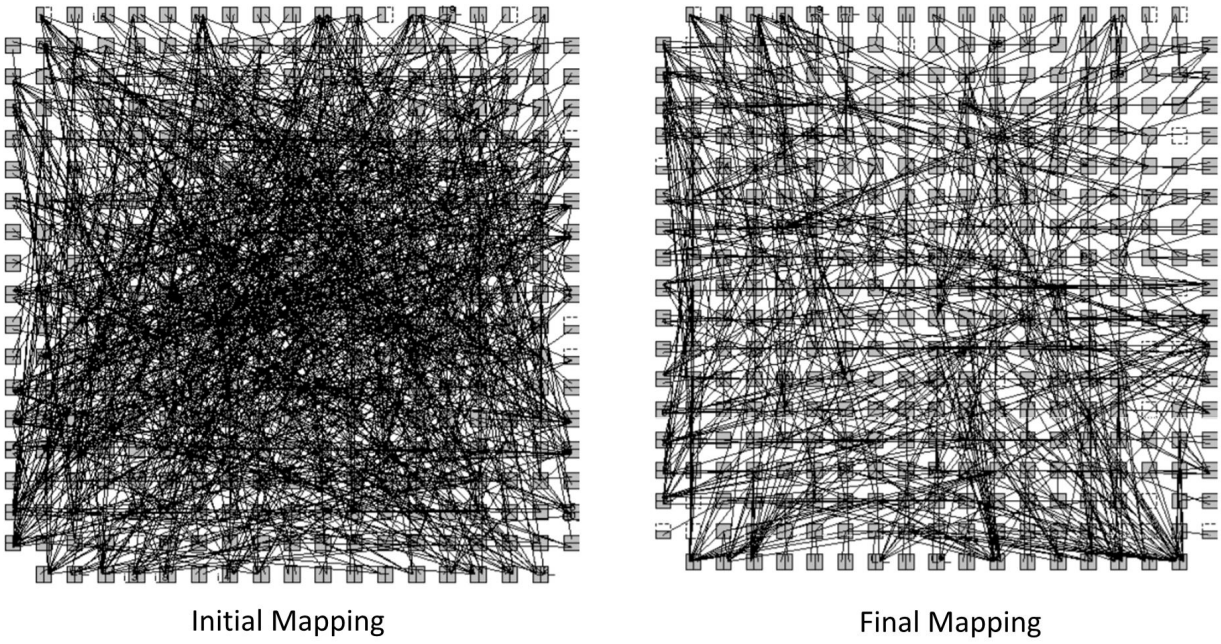


Figure 3.14: Cost Function Based Placement (Real)

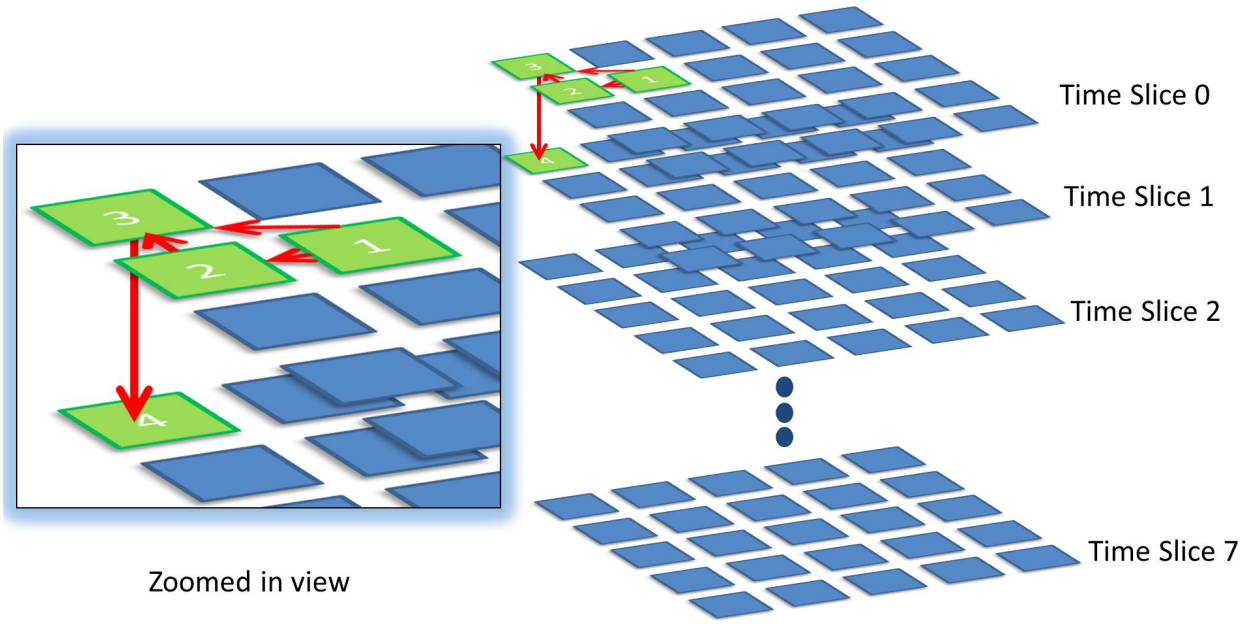


Figure 3.15: Temporal Mapping in 3-D

From the tested algorithms for placement, after their implementation and comparison, the performance ranking (in terms of final cost function value) is: Simulated Annealing (first), Vectoring (second), and Partition Based (third). The runtime ranking (in terms of lower runtime) is: Vectoring (first), Simulated annealing (second) and Partition Based (third). To get these comparisons, random graphs are generated with random connections and the placement algorithms are run on these graphs to optimally place them. The results of all three algorithms are then compared. For the UDSP the Simulated Annealing algorithm is chosen due to its superior performance. However, a general initial temperature setting is not feasible since some designs might have very few cores being utilized and others might have many more. To get the initial temperature based on the DFG being mapped, initially the DFG is placed randomly, then first N 'pair swaps' are made and the standard deviation of the cost function for all of these swaps is computed. The initial temperature is set as a scalar multiple (20X) of this cost function which ensures that in the beginning all swaps are valid.

After placement of the cores, the tool flow proceeds to routing connections between them. The inputs to these set of routing algorithms is the routing layer's extracted equivalent DFG and the placement of Cores on the UDSP grid, along with edge inputs and outputs including those tied to layer 4 of the routing layer. Some considerations for routing are:

1. Any one of the core's output can be connected to any other core's input, even its own.
2. Cores more than a distance of 3 away cannot be connected while satisfying 1GHz clock frequency.
3. The routing layer is delay less except for layer 4, so any long distance connections should go through layer 4.

A trivial solution to routing is to use the shortest path algorithm. This eliminates the traffic balancing problem. Since the underlying DFG is hardware multiplexer based, routing through a node is equivalent to eliminating that node away such that no other connection can reuse that node. The shortest path algorithm is iteratively applied, and each time it is applied the nodes it traverses are taken away from the graph and a connection is marked as satisfied. This process is repeated until all connections are satisfied. This method is very computationally efficient as well as very resource efficient but may result in conflicts in routing, rendering the routing incomplete. There are two possible ways to implement the shortest path algorithm – Dijkstra and Bellman-Ford. Dijkstra has slightly better time complexity. However Bellman-Ford has the ability to deal with one output being mapped to multiple inputs [28]. As a final choice the Bellman-Ford algorithm is chosen for routing.

The problem of routing conflicts remains as it is possible that two shortest paths share the same multiplexer, and because of the order in which the routing algorithm received the two paths, one of them can no longer use that multiplexer. In the worst case, the particular multiplexer is critical to the second route (i.e. it can only be satisfied with the said multiplexer in the path). To resolve this, for every single route, multiple non-optimal paths (in the shortest distance sense) are chosen and kept in memory. The cross correlations of all

paths are calculated, and starting from the most correlated paths, the routing is executed. This reduces the possibility of conflicts however doesn't guarantee a valid routing solution. Another possible solution to this problem is derived from Google's PageRank algorithm [**?**]. In this algorithm, every data input node and data sink node (Cores) in a graph is assigned a constant emission value, and the rest of the nodes are all assigned a zero emission value initially. In each iteration of the simulation, the value of every routing node is discounted by a certain factor α and summed into all its neighbors in the graph. After many iterations the graph converges to the properly assigned ranks of every node. Intuitively, this kind of ranking gives a high rank to nodes that have high connections going through them. This means that multiplexers that are in higher demand will be ranked higher giving a sense of how important a multiplexer is and how many connections are affected due to occupying this multiplexer for a particular path. This information is then used to assign multiplexers in order of least importance. That is connections using the least used multiplexers are mapped first in the hope that the demand for high rank multiplexers would decrease as the routing proceeds. An example illustration of PageRank convergence is shown in Figure 3.16. Here the red node has very high importance since it routes many connections through it.

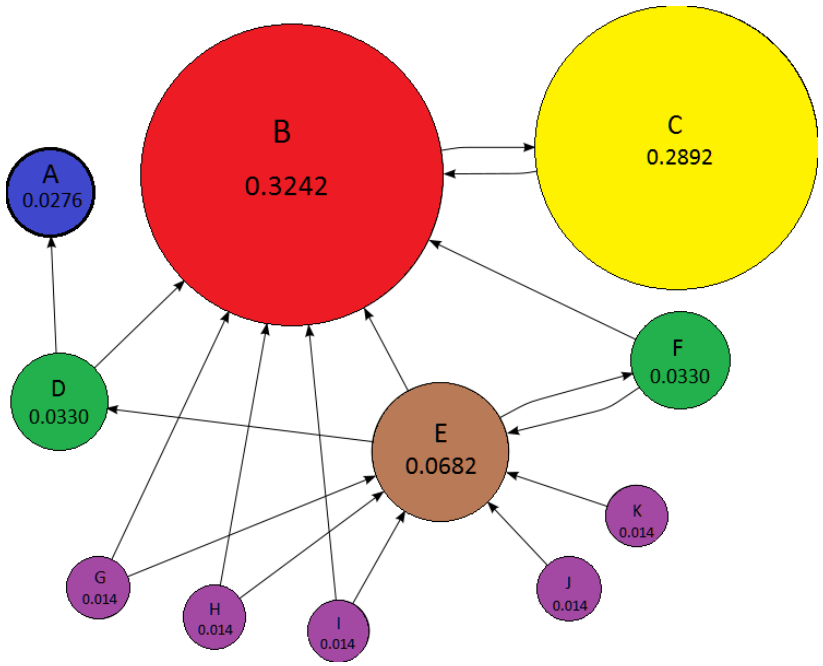


Figure 3.16: Google PageRank Example

3.7 Summary of Contributions

In this chapter, the design flow of a *Multicore Mapper* has been presented with 3 main steps. The first is clustering where the equivalent data flow graph is retimed, and formed into 'Core mappable' blocks called clusters. The next is placement, where these Cores are placed on a 2-D grid whilst solving a cost function to minimize the routing distance between these cores. If the temporal direction is also used, then the placement grid is 3-D. After placement the last step of routing is performed, where the connections to these cores are routed through the interconnect fabric using modified shortest path algorithms. Figure X15 summarizes these steps pictorially.

The contributions of this thesis are:

- A multicore mapping tool flow which can take static graph based algorithms from Simulink and generate byte code which can be programmed into the UDSP through JTAG.
- A novel and efficient way (applicable to this architecture) to use the temporal direction for fast programmability without additional processing by considering time as the 3rd

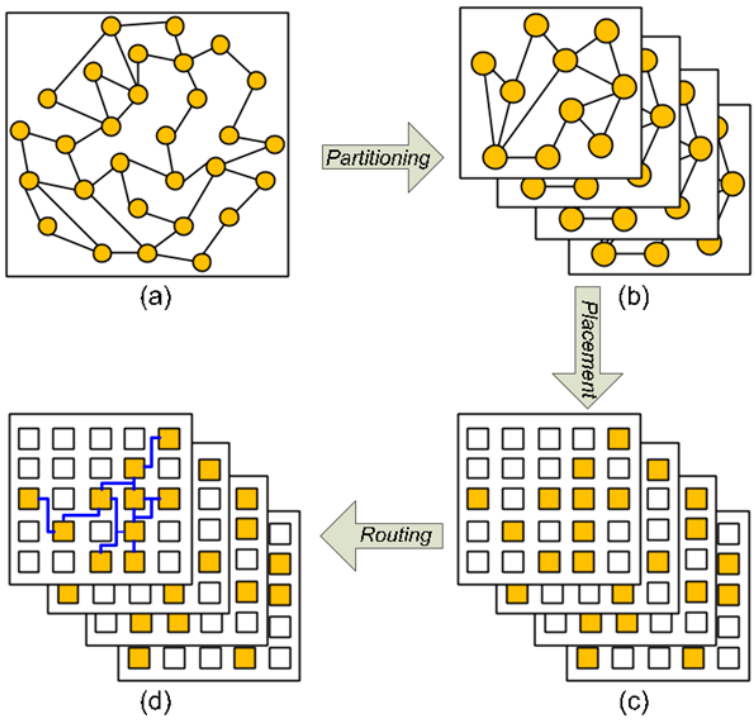


Figure 3.17: Multicore Mapper Flow Summary

spacial dimension.

- Analysis of FPGA tool flows, to leverage and improve upon them for the UDSP application. Table 3.3 highlights some of the key differences between the two tool flows.

- A novel physics inspired *Vectoring* algorithm for placement which presents the best runtime and moderate performance.

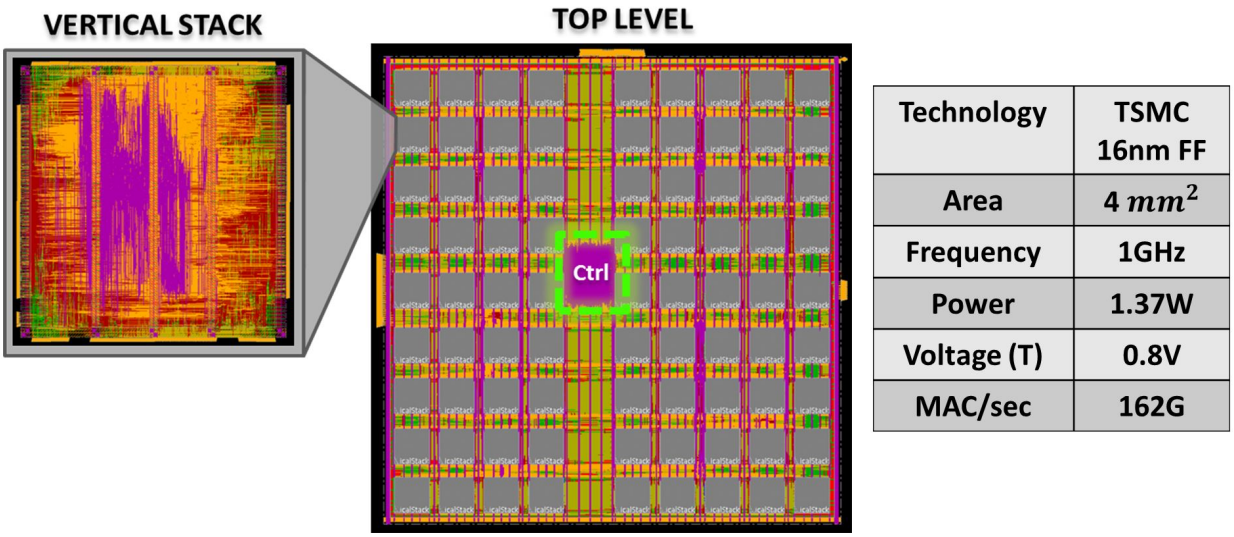
Table 3.3: FPGA vs UDSP Tool Flow

Step	UDSP	FPGA
Clustering (Granularity)	Block Level	Gate Level
Clustering (Functionality)	16-bit Arithmetic	All Possible
Placement (Critical Path)	Single-stage	Multi-stage
Placement (Frequency)	1GHz Target	Not-fixed
Routing	Multi-layer	Mesh based

CHAPTER 4

Results and Future Work

4.1 The 16nm Chip



Technology	TSMC 16nm FF
Area	4 mm^2
Frequency	1GHz
Power	1.37W
Voltage (T)	0.8V
MAC/sec	162G

Figure 4.1: Chip Micrograph and Specs

Figure 4.1 shows the micrograph of the resultant chip and the final performance numbers respectively. The design consists of repetitive blocks which allows for the easy and automated creation of large arrays. It also allows for fast hierarchical verification since only test benches (pertaining to the original design algorithms) need to be written for a single 'type' of Core. This chip has been taped-out in TSMC 16nm FF.

4.2 Functional Verification

Each Vertical Stack has a 2Kb wide instruction. And the UDSP array has a total of 162Kb of instruction memory. A complete verification of every functional state is not feasible in a reasonable about of time. Instead, functional verification is performed and a certain subset of necessary functions is checked. Since the Core is designed to perform a certain set of operations, test benches are made to cover those particular algorithms. Additionally, the Array consists of identical blocks. It is therefore sufficient to test one Core extensively and the rest functionally. The Routing Network also benefits from a repetitive design as all switch boxes inside a layer are identical. Testing each type of switch box is sufficient. However the interconnections of the complete fabric are of critical importance as well. Testing the fabric involves testing a relatively infinite space of programming instructions to cover all connection scenarios which is not feasible. Since we assure with a 90% confidence interval, particular lengths of connections, we can use this fact to reduce the test space. The verification tests reduce to picking sets of 2 Cores that are 1,2 and 3 length apart and forcing feed through connections between them. This process is automated by the mapping tool. Figure 4.2 shows pictorially an example of fabric verification for two adjacent cores using a layer 1 switchbox. For post-silicon verification, the same test benches are used as the pre-silicon verification by connecting an FPGA and a serial data link to the UDSP and repeating the tests.

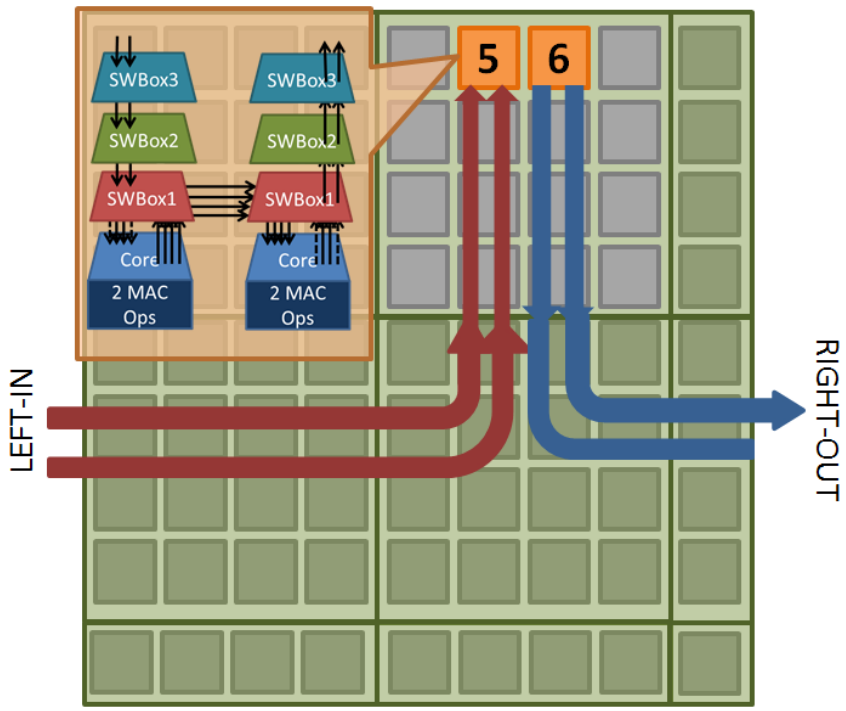


Figure 4.2: Functional Verification Example

4.3 DSP Templates and Heterogeneity

Because of the modular nature of the UDSP, heterogeneous tiles or templates of cores and the routing fabric can be constructed and placed in the chip. A tile of the UDSP consists of a Vertical Stack where the design confers to a certain set standard, in terms of the number of I/Os per core and the internal 1-hop nature of connections within the cores compute units. Custom tiles or templates are modifiable to serve a certain set of desired algorithms. Repositories of these tiles and their associated helper files (like test benches) can be made and kept for repetitive use in designs. Each tile shares the routing network design but has its own core's connectivity matrix and delay matrix (compute resources). This leads to the reuse of the mapping tool flow to map algorithms to a heterogeneous array of cores. These blocks of IP's are either *soft* in nature, meaning the end user can change the internals of the design (like sizing, retiming), or *hard* in nature where the Post P&R version of the Vertical

Stack is given as a black box to designers. The top level UDSP has the same set of options with Soft IP scripts that allow for fast creation of an arbitrary NxM heterogeneous UDSP Array. The result is rapid deployment of IP, with fair amount of designer control (if the designer wishes to customize the design).

4.4 Future Work

In the current version of the UDSP, the Vertical Stack has been over provisioned with 8 instructions and 4 temporal state registers. This has significant costs on area and power overheads. In addition to this, support for very tight loop bound algorithms like the Single Cycle MAC operation at 1 GHz requires the design to use ultra-low voltage threshold (uLVT) cells to guarantee timing closure. The uLVT cells are faster but suffer from high leakage (currently greater than 1/3 of the total power consumption of the chip). In a repetitive pattern design (like UDSP or FPGA) significant time is required to experiment with floor plans at the top level that can then lead to significant area savings. However the current design was relaxed at the top level due to time constraints of the tape-out schedule - there is room for improvement in the area efficiency as well. A requirement for the design is to have the data flight time between two Vertical Stacks be less than 1ns for timing closure. Since the routing network consists of very long delay less wires (which are never going to be instantiated by the programming software), a constrained optimization of paths has to be performed while sizing the gates. Because of these reasons, currently the UDSP is 10x away from an equivalent ASIC in terms of area and energy efficiency metrics. To rectify this, in future iterations, a heterogeneous array of cores is intended to be used, which will be pipelined, relaxing timing constraints and allowing the use of standard voltage threshold (SVT) cells, lowering the leakage power significantly. Adding to this will be clock gating to shut off unutilized/unmapped cores. Cutting down on the number of instructions from 8 to 2 and reducing the temporal states also improves the area and energy efficiencies.

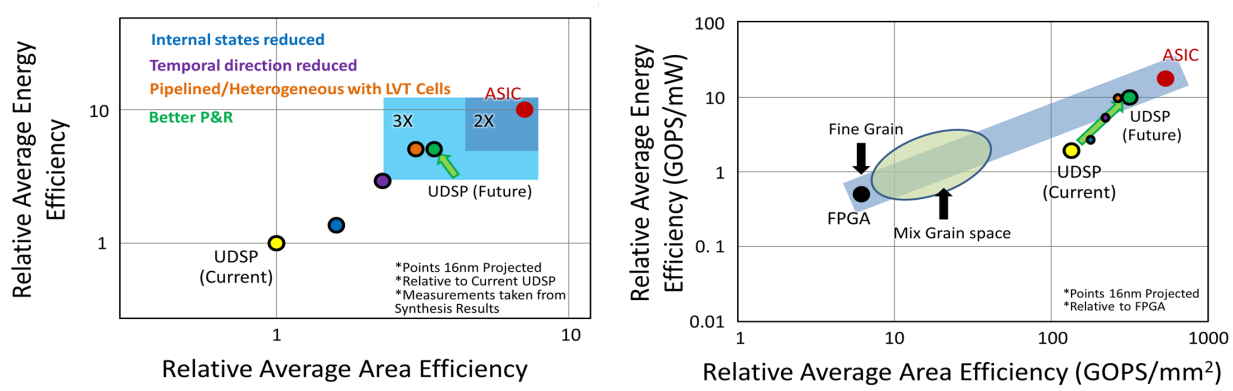


Figure 4.3: Proposed Improvements and Impact

Allotting more design time to P&R will also reduce the underutilized spaces causing an increase in area efficiency. The simulated results of all these proposed improvements are summarized graphically in Figure 4.3 (a) and (b). The goal of the next iteration is to incur a reconfiguration penalty of no more than 3x from and ASIC.

CHAPTER 5

Conclusion

The idea behind the UDSP is to find a way of implementing multiple ASICs in a small area in a concentrated fashion. This saves Dark Silicon (and leakage power). The method used is to take these domain specific algorithms, identify the common hardware elements and reuse them on chip (across all algorithms), where only one or two algorithms are active at a given instance. However this approach does not provide speed up in design time. To solve this, the set of algorithms (Data Flow Graphs) are observed to find their minimum sub-structure which can span all algorithms. This sub-structure/graph is a reusable 'template'. Its reuse is leveraged in design time savings as only the template needs to be designed and the original algorithms become software mappable by tiling these templates in hardware. However template reuse introduces area and energy overheads on all algorithms in the set. Here the design becomes qualitative in analysis rather than quantitative, and to minimize this overhead requires manual designer intervention to architect the core template by using experience and basic reasoning. It becomes qualitative because a formal measurement of 'flexibility' is nonexistent in literature, unlike Power/Area efficiencies. The design requirements for the UDSP go a step further, demanding that the original algorithms being used to design the UDSP, (which are supposed to be static and known) are now arbitrary albeit with some degree of similarity (domain specific). This means that the starting point for the templates is no longer known. To solve this whilst keeping a template approach (not sacrificing any speedups), an efficient and flexible Routing Network design is important. The network,

like its original counterpart (the core), must be modular and templatizable, yet be fast and efficient, have a small small-footprint and be scalable and have support for heterogeneity of sub-structures/graphs/cores. More over since the original algorithm is now random (to a degree), probabilistic mapping arguments, based on graph theory, are used in the design of the Routing Network to guarantee more than 90% mapping of arbitrary algorithms. A custom in-house mapping tool is developed to program the UDSP. The software and tool flow borrows concepts from literature as well from FPGAs, adapting them to serve the constraints of the current problem. A new vectoring algorithm is proposed and its performance compared to state of the art placement algorithms resulting in a final placement algorithm – a hybrid between simulated annealing and vectoring.

The result is a UDSP which, without prior knowledge of specific algorithms, is able to give near ASIC like performance across a domain, the development of which is automatable without the requirement of a designer, saving design time. The current version of the UDSP, however, does not come inside a factor of 3x to an ASIC, due to the over ambitious nature of the extra constraints on the chip which are not necessary or are experimentally placed in there. In the future iterations of the UDSP, these provisions intend to be removed to show that this is a valid, cheap and fast approach for implementing an efficient chip, building up IP template libraries and further reducing design/layout times.

References

[1] Frank, Michael P. The Future of Computing Depends on Making It Reversible. IEEE Spectrum: Technology, Engineering, and Science News, IEEE Spectrum, 25 Aug. 2017, spectrum.ieee.org/computing/hardware/the-future-of-computing-depends-on-making-it-reversible.

[2] Sridhar, Ram. 2015. Dark Silicon Overview, Analysis and Future Work.

[3] Taylor, M.B. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. in Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE. 2012.

[4] N. Meena and A. M. Joshi, "New power gated SRAM cell in 90nm CMOS technology with low leakage current and high data stability for sleep mode," 2014 IEEE International Conference on Computational Intelligence and Computing Research, Coimbatore, 2014, pp. 1-5.

[5] M. Prasad and U. B. Mahadevaswamy, "A novel approach for leakage current reduction of 14nm NMOS device using HfO2 dielectric," 2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT), Mysuru, 2017, pp. 129-134.

[6] Yuan, F. (2014). Energy-Efficient VLSI Architectures for Next-Generation Software-Defined and Cognitive Radios. UCLA.

[7] Y. Thomas, F. Petros, P. Sanjay, R.Glenn (2007). ParallAX: An Architecture for Real-Time Physics. ISCA 2007.

[8] Chao Li, Jun Zhou, Yuan Jiang, Canfeng Chen, Yongjun Xu and Zuying Luo, "A reconfigurable and scalable architecture for security coprocessor," 2010 5th IEEE Conference on Industrial Electronics and Applications, Taichung, 2010, pp. 1826-1831.

[9] FISHER J.A. Very long instruction word architectures and the ELI-512, Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 140-150, 1983.

[10] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," in Computer, vol. 36, no. 4, pp. 68-74, April 2003.

[11] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., 2003, pp. 81-92.

[12] M. S. Bright and T. Arslan, "Synthesis of low-power DSP systems using a genetic algorithm," in IEEE Transactions on Evolutionary Computation, vol. 5, no. 1, pp. 27-40, Feb 2001.

[13] A. Saha and A. Sinha, "Re-configurable DSP Processor - A New Computing Platform for Software Radio Applications," 2013 Fourth World Congress on Software Engineering, Hong Kong, 2013, pp. 225-228.

[14] Xu Deqiang, Cary Ussery and C. Hongyi, "AES implementation based on a configurable VLIW DSP," 2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology, Shanghai, 2010, pp. 536-538.

[15] A. Saha and A. Sinha, "Re-configurable DSP Processor - A New Computing Platform for Software Radio Applications," 2013 Fourth World Congress on Software Engineering, Hong Kong, 2013, pp. 225-228.

[16] P. Sinha, A. Sinha and D. Basu, "A novel architecture of a re-configurable parallel DSP processor," The 3rd International IEEE-NEWCAS Conference, 2005., 2005, pp. 71-74.

[17] A. M. Obeid et al., "Flexible reconfigurable architecture for DSP applications," 2014 27th IEEE International System-on-Chip Conference (SOCC), Las Vegas, NV, 2014, pp. 204-209.

[18] Larry Page , Sergey Brin , R. Motwani , T. Winograd. (1998)The PageRank Citation Ranking: Bringing Order to the Web.

[19] S. Hauck, G. Borriello and C. Ebeling, "Mesh routing topologies for multi-FPGA systems," Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, MA, 1994, pp. 170-177.

[20] S. Narayanan, G. V. Varatkar, D. L. Jones and N. R. Shanbhag, "Computation as Estimation: A General Framework for Robustness and Energy Efficiency in SoCs," in IEEE Transactions on Signal Processing, vol. 58, no. 8, pp. 4416-4421, Aug. 2010.

[21] Ian Kuon; Russell Tessier; Jonathan Rose, "FPGA Architecture:Survey and Challenges," in FPGA Architecture:Survey and Challenges , 1, Now Foundations and Trends, 2008, pp.124-

[22] Chang, Chia-Ming & Chen, Chien-Ming & King, Chung-Ta. (1997). Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. Computers & Mathematics with Applications. 34. 1-14.

[23] Nanda, Rashmi . DSP architecture optimization in Matlab/Simulink environment. University of California Los Angeles, UCLA, 2008, pp. 14-16.

[24] D. Y. Chao and D.T. Wang. Iteration bounds of single-rate data flow graphs for concurrent processing. IEEE Transactions on Circuits and Systems, 40(9):629-634, July 1993.

[25] Jong-eun Lee, Kiyoung Choi, and Nikil D. Dutt. 2003. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. SIGPLAN Not. 38, 7 (June 2003), 183-188.

[26] Xiao Guo, Teng Wang, Zhihui Chen, L. Wang and Wenqing Zhao, "Fast FPGA placement algorithm using Quantum Genetic Algorithm with Simulated Annealing," 2009 IEEE 8th International Conference on ASIC, Changsha, Hunan, 2009, pp. 730-733.

[27] Yonghong Xu An introduction to FPGA Placement. May 2003

[28] Chen, S. (2014). Shortest Path Problems: Dijkstra, Bellman-Ford, and Floyd-Warshall. Lecture.

[29] Wang, C. (2013). Building Efficient, Reconfigurable Hardware using Hierarchical Interconnects. UCLA.

[30] Bic, Lubomir, et al. Advanced Topics in Dataflow Computing and Multithreading. IEEE Computer Soc. Press, 1996.