2009

LIATAKIS
KONSTANTINOS

TECHNICAL UNIVERSITY

OF CRETE

ELECTRONIC AND COMPUTER ENGINEERING

DEPARTMENT

[IMPLEMENTATION OF BASIC BIOINFORMATICS STRUCTURES IN FPGA AND VLSI]

Contents

ACKNOWLEDGEMENTS

I would like to acknowledge the advice and guidance of Dr. Apostolos Dollas committee chairman who had the idea for this thesis. Furthermore, I would not have finished this thesis without the assistance of Mr. Grigorios Chrysos, Mr.Euripides Sotiriades and Mr. Kyprianos Papadimitriou to whom I am grateful. I also thank Dr. Christos Sotiriou who provided me the necessary technical knowledge. Special thanks go to Mr. Makris who spent many hours helping me with the installation of the CAD tools. My long lasting friends provided material and spiritual help at critical times; Many thanks to them too. Last but not least, I would like to thank my parents and my brother who have supported me all along.

This thesis is dedicated to my family.

INTRODUCTION

Ever since 1953 the year when Watson and Crick unraveled the structure of DNA molecular biology has witnessed tremendous advances. With the increase in our ability to manipulate biomolecular sequences, a huge amount of data has been and is being generated (as we can see in the following figures).

Figure A 1

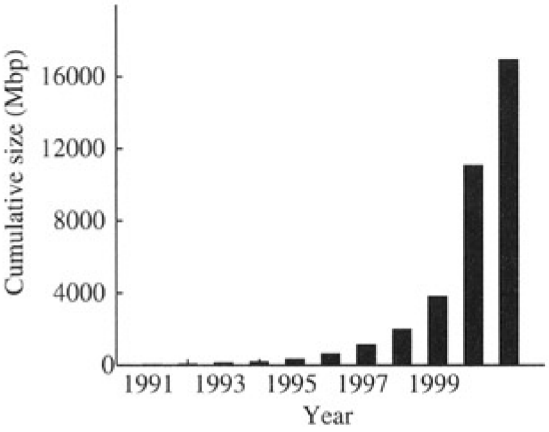


Figure A 2

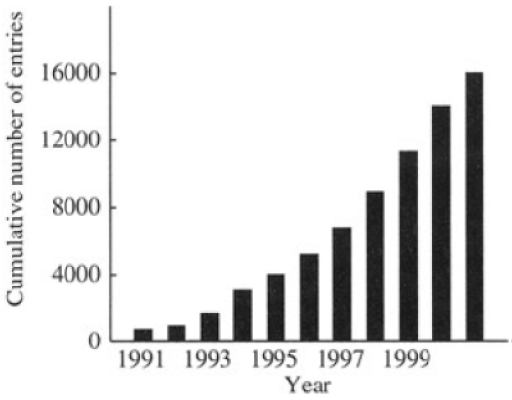


Figure A: (1) Growth of GenBank, the US National Center for Biotechnology Information genetic sequence archival databank. (2) Growth of Protein Data Bank, archive of three-dimensional biological macromolecular structures.

Despite the fact that scientists from the biological sciences are the creators and users of this data, the sheer complexity and size requires the help of other sciences such as mathematics and computer science. Computing contributed not only the raw capacity for processing and storage of data, but also the mathematically-sophisticated methods required to achieve the results. As a result an entirely new field was created, which goes by the general name of computational molecular biology or bioinformatics. One of the most common problems in biology is the following: huge databases are needed in order to store all the data that is being generated. These databases should be able to record changes, as current models may not be suitable. This requires new efficient algorithms for pattern recognition. An algorithm is a type of effective method in which a list of well-defined instructions for completing a task will, when given an initial state, proceed through a well-defined series of successive states, eventually terminating in an end-state. The transition from one state to the next is not necessarily deterministic.
In our try to speed up various bioinformatics algorithms many implementations of these algorithms in hardware (mostly FPGA's) have been proposed. A common characteristic of all these implementation is the use of some basic elements such as:

comparators, shift registers and some mathematic elements such as floating point adders, multipliers and dividers.

The primary objective of this project is the implementation of all those elements in FPGA and VLSI. We will acquire important results about the frequency the power consumption and the cost of each implementation. With these results we will try to come to a final assumption about which hardware implementation of these algorithms is the best.

The organization of this project is as follows:

- First of all we will give an overview of the most common bioinformatics algorithms. For each one of them we will present its basic steps and any existing hardware implementation.
- Afterwards we will refer to the basic elements stated above that each algorithm uses.
- Then we will give the exact flow of the implementation of these elements in FPGA as well as in VLSI.
- We will extract important results from this analysis, we will process them and compare them.
- Finally according to these results we will come to a final assumption and we will talk about further research available concerning this subject.

CHAPTER 1: BIOINFORMATICS ALGORITHMS AND RELATIVE RESEARCH

BIOINFORMATICS ALGORITHMS

In this chapter we will present you the function of the 7 most famous bioinformatics algorithms which are the following:

- BLAST
- FASTA
- Smith-Waterman
- GLIMMER
- Combinatorial Extension(CE)
- RAxML
- Predator

Furthermore for each one of the algorithms stated above we will refer to any existing hardware implementation.

BLAST

The term BLAST is an acronym for Basic Local Alignment Tool. The BLAST program was designed by Stephen Altschul, Eugene Myers, Warren Gish, David J. Lipman and Webb Miller at the NIH and was published in 1990. The BLAST programs are amongst the most frequently used for sequence database searching worldwide (the term database here simply refers to a large set of sequences). BLAST, is an algorithm for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences. A BLAST search enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold. Before we explain in detail how BLAST works we must give the meaning of some terms which will help us later on:

- With the term segment we refer to the substring of a sequence.

- Given two sequences a segment pair between them is a pair of segments of the same lengths one from each sequence. The substrings in a segment pair

have the exact same length. As a result we can form an alignment between them with no gaps. Then we can score this alignment using a matrix of substitution scores. As there are no gaps, no gap penalty function is needed. In the following figure we present an example of a segment pair scored using PAM120 substitution matrix.

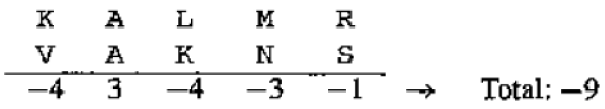


Figure 1 - 1: Scoring example of the BLAST algorithm

- With the term **M**aximum **S**egment **P**air (MSP) we refer to the segment pair that has achieved the highest score. This score is a measure of similarity and can be computed precisely using dynamic programming. However the blast algorithm can make a very good estimation of this number much faster than any dynamic programming method.

Taking into consideration all the definitions stated above BLAST's function can be summarized in the following:

Given a query sentence and a threshold S BLAST returns all the segment pairs between the query and a database sequence with scores above S. The threshold S is a parameter and its value may differ in respect to the user's requirements. As we have already mentioned the alignments reported have no gaps. This is in fact one of the main reasons why BLAST is so fast, because looking for good alignments with gaps is more time consuming.

The way BLAST computes high scoring segments is:

First of all BLAST finds certain seeds which are very short segment pairs between the query and a database sequence. Then BLAST extends these seeds in both directions without including gaps until the maximum score possible for these extensions of this specific seed is reached. BLAST has a certain criterion to stop the extension if the score falls below a limit.

It is reasonable to say that BLAST proceeds through the three following stages:

1. Compile the list with all the high scoring words
2. We search for hits. Each hit gives a new seed
3. We extend the seeds in both directions.

However the steps stated above differ depending on the type of the sequences compared: DNA or Protein.

For protein sequences the list of high scoring words, consists of all words with a length of w (called w-mers), that scored at least T with some w-mer of the query using some PAM matrix to compute the score. W and T are parameters of the program. It is not necessary that this list contains all w-mers of the query. However there is an option to force the inclusion of all w-mers. The most common value for w the seed size is 4 for protein searches.We can scan the database in search for hit based on the list we compiled earlier using two different methods. The first is to arrange the list of words in a hash table. Then for each database word of size w, it is very easy to get their index in the hash table and compare it to the words there. The second approach is to use a deterministic finite automaton to search for hits. It begins from a primary state and then for each character of the database it passes to a next stage. Depending on the state and transition a word from the list is being recognized. The final step of the algorithm for protein sequences is simple: the seeds found earlier are being extended in both directions. To save time the algorithm stops the extension when the score falls below a certain threshold. The segment pair which achieved the highest score and originated from this seed is kept. There is a small chance to miss some important extensions with this approach but is negligible.

For DNA sequences the primary list is compiled only with the w-mers of the query. The way the database is scanned is a lot different from the one described earlier for proteins. Due to the fact that the alphabet has a size of 4, the database is compressed so that each nucleotide can be represented using 2 bits. As a result 4 nucleotides fit in a byte. So except the fact that we save space, the search can be done really fast because every time we compare a byte. Then there is one extra step where we remove from the initial list very similar words which would result in many useless hits. The extension is done in the same way as described earlier for protein sequences.

FASTA

FAST is another family of programs used for sequence database search. The first of these programs was FASTP and was described in 1985 by David J. Lipman και William R. Pearson. The original FASTP program was designed for protein sequence similarity searching. FASTA was described in 1988 and extended the initial FASTP adding the ability to do DNA-DNA and translated protein-DNA searches. FASTA stands for "FAST-All", because it works with any alphabet, an extension of "FAST-P" (protein) and "FAST-N" (nucleotide) alignment.

The FASTA program follows a heuristic approach which contributes to the high speed of its execution. It initially observes the pattern of word hits, word-to-word matches of a given length, and marks potential matches before performing a more time-consuming optimized search using a Smith-Waterman type of algorithm. The size taken for a word, given by the parameter ktup (k-tuple), controls the sensitivity and speed of the program. Increasing the ktup value decreases number of background hits that are found. From the word hits that are returned the program looks for segments that contain a cluster of nearby hits. It then investigates these segments for a possible match.

Three scores must be calculated that describe the sequence similarity results. This is done in the following four steps:

- Identify regions of highest density in each sequence comparison.
  In this step all or a group of the identities between two sequences are found using a look up table. The ktup value determines how many consecutive identities are required for a match to be declared. Thus the lesser the ktup value: the more sensitive the search (k-tup=2 and k-tup=4 are frequently used values for protein and nucleotide sequences respectively). The program then finds all similar local regions, represented as diagonals of a certain length in a dot plot, between the two sequences by counting ktup matches and penalizing for intervening mismatches. This way, local regions of highest density matches in a diagonal are isolated from background hits. To ensure that groups of identities with high similarity scores contribute more to the local diagonal score, BLOSUM50 values are used for scoring ktup matches for protein sequences while an identity matrix is used for nucleotide sequences. The best 10 local regions selected from all the diagonals put together are then saved.
- Rescan the regions taken using the scoring matrices, trimming the ends of the region to include only those contributing to the highest score.
  Rescan the 10 regions taken. This time use the relevant scoring matrix while rescoring to allow runs of identities shorter than the ktup value. Also while rescoring conservative replacements that contribute to the similarity score are takenFor each of the diagonal regions rescanned this way, a subregion with the maximum score is identified. The initial scores found in step1 are used to rank the library sequences. The highest score is referred to as init1 score.
- In an alignment if several initial regions with scores greater than a CUTOFF value are found, check whether the trimmed initial regions can be joined to form an approximate alignment with gaps. Calculate a similarity score that is the sum of the joined regions penalising for each gap 20 points. This initial similarity score (*initn*) is used to rank the library sequences. The score of the single best initial region found in step 2 is reported (*init1*).

- This step uses a banded Smith-Waterman algorithm to create an optimised score (*opt*) for each alignment of query sequence to a database(library) sequence.

A figure is presented below with a graphical overview of the four steps.

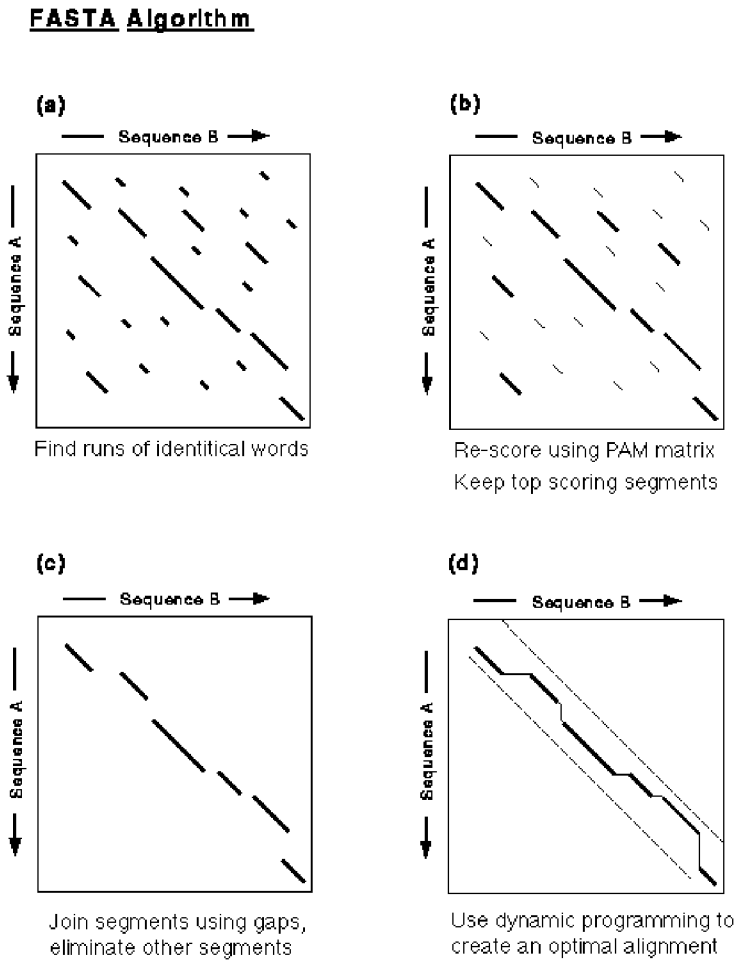


Figure 1 - 2: Description of the FASTA algorithm (Source: http://www.med.nyu.edu).

GLIMMER

GLIMMER stands for Gene Locator and Interpolated Markov ModelER. GLIMMER was the first bioinformatics system for finding genes that used the interpolated Markov model formalism. It is very effective at finding genes in bacteria, archaea and viruses, typically finding 98-99% of all protein-coding genes.

The most reliable way to identify a gene in a new genome is to find a close homolog from another organism. This can be done efficiently with the use of programs such

as BLAST and FASTA which have already been described above. However, many of the genes in new genomes still have no significant homology to known genes. For these genes we have to rely on computational methods of scoring the coding regions to identify the genes. GLIMMER uses a technique called interpolated Markov models (IMMs). IMMs are more powerful than Markov chains and experiments have shown that they produce more accurate results when used to find genes in bacterial DNA.

A fixed order Markov model predicts each base of a DNA sequence using a fixed number of preceding bases in the sequence. For example a 5-order Markov model uses the 5 preceding bases to predict the next one. However, learning such models accurately can be difficult when there is insufficient training data to accurately estimate the probability of each base occurring after every possible combination of five preceding bases. In general, a *k*th-order Markov model for DNA sequences requires 4^*k* + 1 probabilities to be estimated from the training data.
An IMM overcomes this problem by combining the probabilities from contexts (oligomers) of varying lengths to make predictions and using only those oligomers for which we have sufficient training data. It is preferable to use longer oligomers. An IMM uses a linear combination of the probabilities produced by contexts of varying length, to make predictions and gives high weights to oligomers that occur frequently and low weights to those that do not. Using IMMs GLIMMER was developed to identify the coding regions in microbial DNA. It uses a novel approach, based on frequency of occurrence and predictive value to determine the relative weights of oligomers that vary from 1-8. After IMMs have been created for every one of the possible reading frames GLIMMER uses them to score orfs (segments of the genome that may encode a protein). When 2 orfs overlap the overlap area is scored separately to determine which orf I likely to be a gene. As we have already mentioned GLIMMER's ability to identify genes is close to 100%.

SMITH-WATERMAN

This algorithm was first presented by Temple Smith and Michael Waterman in 1981. It is a dynamic programming algorithm for local sequence alignment. In other words the algorithm determines similar regions between two nucleotide or protein sequences. As a dynamic programming algorithm it has the ability to find the optimal local alignment with respect to the scoring system being used. The main difference to the Needleman-Wunsch algorithm is that negative scoring matrix cells are set to zero, which renders the (thus positively scoring) local alignments visible. Back tracing starts at the highest scoring matrix cell and proceeds until a cell with score zero is encountered, yielding the highest scoring local alignment. The Smith-Waterman algorithm is very demanding in time and memory resources. More specifically in order to find the alignment of two sequences with lengths m and n respectively O

(m*n) time is needed. This is the main reason why this algorithm is not being used in practice and has been replaced by other algorithms such as BLAST which may not have the ability to find the optimal alignment but is very efficient. Recent work developed by Cray Inc. shows that with the use of reconfigurable logic a speed up of 28x compared to common microprocessors can be achieved. An implementation of the algorithm in a Virtex-4 FPGA showed a speed up, up to 100x over a 2.2GHz Opteron processor.

COMBINATORIAL EXTENSION

There is no exact solution to the protein structural alignment problem. So many heuristic solutions have been proposed. The combinatorial extension method of structural alignment generates a pair wise structural alignment by using local geometry to align short fragments of the two proteins being analyzed and then assembles these fragments into a larger alignment. Based on measures such as rigid-body root mean square distance, residue distances, local secondary structure, and surrounding environmental features such as residue neighbor hydrophobicity, local alignments called "aligned fragment pairs" are generated and used to build a similarity matrix representing all possible structural alignments within predefined cutoff criteria. A path from one protein structure state to the other is then traced through the matrix by extending the growing alignment one fragment at a time. The optimal such path defines the combinatorial-extension alignment.

RAxML

The computation of large phylogenetic trees with statistical models such as maximum likelihood or bayesian inference is computationally extremely intensive. It has repeatedly been demonstrated that these models are able to recover the true tree or a tree which is topologically closer to the true tree more frequently than less elaborate methods such as parsimony or neighbor joining. RAxML is a fast implementation of maximum-likelihood (ML) phylogeny estimation that operates on both nucleotide and protein sequence alignments. Whereas RAxML-III performs worse than PHYML and MrBayes on synthetic data it clearly outperforms both programs on all real data alignments used in terms of speed and final likelihood values. This algorithm was implemented in hardware by Mr. Alachiotis Nikolaos in the Technical University of Crete, achieving a frequency of 267MHz and a speed-up ranging from 45x to 88x (depending on the experiment) when compared to the RAxML software running on a Pentium 4 at 2.66GHz.

Predator

The predator algorithm was first presented by Dmitrij Frishman and Patrick Argos in 1996. Its goal is to predict the secondary structure of a protein and its accuracy reaches 75%. The algorithm takes as inputs the following:

- A sequence of aminoacids of the protein whose secondary structure we want to predict.
- 550 protein chains with a known structure (after X-ray analysis).
- 4 20x20 propensity tables that contain floating point values.
- A 13x20x20 table that contains data for the Nearest Neighbor.
- An array with a size of 5 which contains the values of the threshold, which are necessary to perform the steps for the prediction of the proteins secondary structure.

The output of the predator algorithm is the secondary structure of the protein (Helix, Sheet or Coil).

RELATIVE RESEARCH

Since FPGA were first developed their area, power and speed disadvantage relative to less programmable designs has been recognized. All of the few attempts to measure this gap are mentioned here.

One of the earliest statements to quantify this gap was by Brown et al. Their work reported the logic-density gap between FPGAs and mask programmable gate arrays (MPGAs) to be between eight to 12 times, and the circuit-performance gap to be approximately a factor of three.

More recently, a detailed comparison of FPGA and ASIC implementations was performed by Zuchowski et al. They found that the delay of an FPGA lookup table (LUT) was approximately 12 to 14 times the delay of an ASIC gate. Their work found that this ratio has remained relatively constant across CMOS process generations from 0.25 μm to 90 nm.

Compton and Hauck have also measured the area differences between FPGA and standard-cell designs. They implemented multiple circuits from eight different application domains, including areas such as radar and image processing, on the Xilinx Virtex-II FPGA, in standard cells on a 0.18-µm CMOS process from TSMC, and on a custom configurable platform. Since the Xilinx Virtex-II is designed in 0.15-μm CMOS technology, the area results are scaled up to allow direct comparison with 0.18-µm CMOS. Using this approach, they found that the FPGA implementation is only 7.2 times larger on average than a standard-cell implementation.

Wilton et al also examined the area and delay penalty of using programmable logic. The approach taken for the analysis was to replace part of a non-programmable design with programmable logic. They examined the area and delay of the programmable implementation relative to the nonprogrammable circuitry it replaced. This was only performed for a single module in the design consisting of the next state logic for a chip-testing interface. They estimated that when the same logic is implemented on an FPGA fabric and directly in standard cells, the FPGA implementation is 88 times larger. They measured the delay ratio of FPGAs to ASICs to be two times.

While this thesis aims to measure the gap between standard cells and FPGA implementation, it is noteworthy that the speed, power and area cost of FPGA is even larger when compared to full-custom circuits. Full custom designs tend to be three to eight times faster than semi-custom (standard cell) designs and consume three to ten times less power. Finally a full custom design achieves 14.5 times greater density than standard cell ASIC methodology.

CHAPTER 2: BIOINFORMATICS ALGORITHMS & BASIC CORES

In the following two tables we show the seven bioinformatics algorithms and the mathematical and basic cores that each one of them uses. As the query size can vary according to the user's preferences some of these algorithms may require different sizes for comparators and shift registers. That is why we decided to study four shift registers and three comparators of different lengths.

	Floating Point Adder		Floating Point Multiplier		Floating Point Divider
	SP	DP	SP	DP	DP
BLAST					
FASTA					
GLIMMER	✓				
Smith-Waterman					
Combinatorial Extension(CE)	✓		✓		
RAxML		✓		✓	
Predator					

Table 2 - 1: Bioinformatics algorithms and the mathematical cores they use.

	Shift Registers				Comparators		
	32-bit	64-bit	1024-bit	4096-bit	32-bit	64-bit	128-bit
BLAST	✓	✓	✓	✓	✓	✓	✓
FASTA	✓	✓	✓	✓	✓	✓	✓
GLIMMER	✓	✓	✓	✓	✓	✓	✓
Smith-Waterman	✓	✓	✓	✓	✓	✓	✓
Combinatorial Extension(CE)	✓	✓	✓	✓	✓	✓	✓
RAxML	✓	✓	✓	✓	✓	✓	✓
Predator	✓	✓	✓	✓	✓	✓	✓

Table 2 - 2: Bioinformatics algorithms and the basic cores they use.

CHAPTER 3:
IMPLEMENTATION

In this chapter we will demonstrate the exact design flow for the structures mentioned above in FPGA as well as in VLSI.

IMPLEMENTATION IN FPGA

Due to the fact that the technology we will use for implementing these structures in VLSI is Faraday's 90nm, we chose to find a 90nm FPGA from Xilinx so that the comparison between the results is more accurate. Xilinx has two 90nm FPGA families: the Spartan-3 family and the Virtex-4 family. We decided to use the second (as it is a more advanced generation of FPGA's) and more specifically the XC4VSX55 device. We chose this device and not a larger one because this is big enough to implement all the bioinformatics algorithms we mentioned but not too big so that we get lower values in frequency and higher values in power consumption.

Device	Configurable Logic Blocks (CLBs)(1)				XtremeDSP Slices(2)	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VSX55	128 x 48	55,296	24,576	384	512	320	5,760	8	4	N/A	N/A	N/A	13	640

Table 3 - 1: Specifications of Virtex-4 XC4VSX55 (Source: www.xilinx.com).

The XtremeDSP Slices contain a dedicated 18x18 2's complement signed multiplier, adder logic, and a 48 bit accumulator. Each multiplier or accumulator can be used independently. These blocks are designed to implement extremely efficient and high speed DSP applications.

The Virtex-4 FPGA slice includes:

- Two 4-input LUTs (Look-Up Tables) that can implement any 4-input boolean function, used as combinational function generators (one LUT is marked "F", the other one is marked "G").
- Two dedicated user-controlled multiplexers for combinational logic (MUXF5 and MUXFX). MUXF5 can be used to combine outputs of the slice's LUTs and so to implement 5-input combinational circuit. MUXFX is used to combine outputs of the other MUXF5 and MUXFX (from the other slices).

- Dedicated arithmetic logic (two 1-bit adders, carry chain and two dedicated AND gates for fast and efficient multiplication).
- Two 1-bit registers that can be configured either as flip-flops or as latches. The input to these registers is selected by YMUX and XMUX multiplexers. Note that these multiplexers aren't user-controlled: the path is selected during FPGA programming.

A simplified diagram of a Xilinx Virtex-4 FPGA slice is presented below:

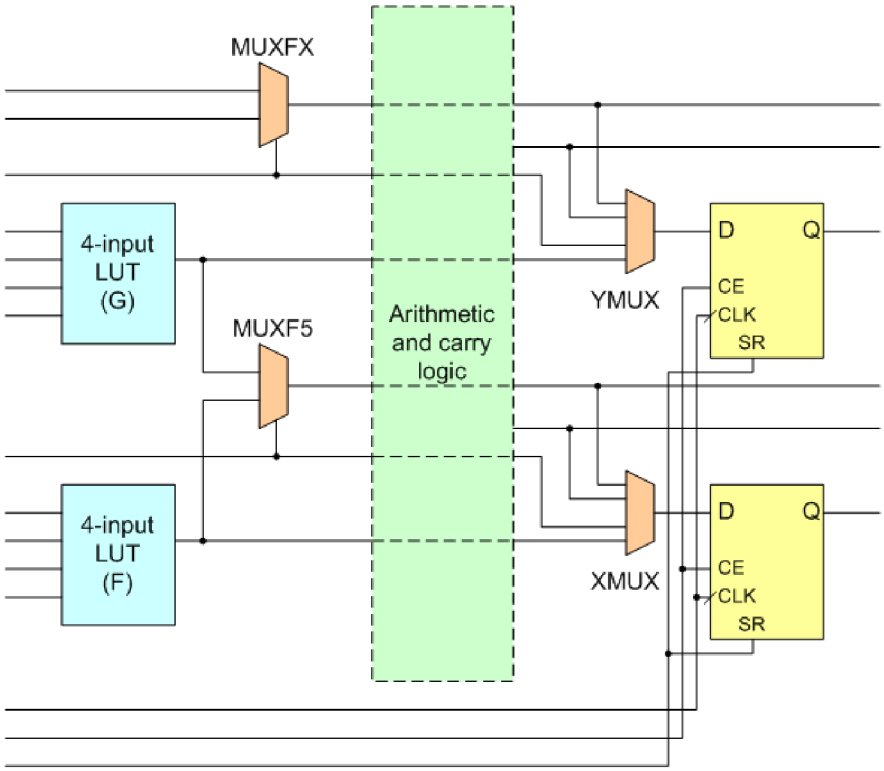


Figure 3 - 1: Block diagram of a slice in Virtex-4 FPGA (Source: http://www.1-core.com/library/digital/fpga-logic-cells).

Using Xilinx Core Generator we generated the following cores:

- Single Precision Floating Point Multiplier
- Double Precision Floating Point Multiplier
- Single Precision Floating Point Adder/Subtractor
- Double Precision Floating Point Adder
- Double Precision Floating Point Divider
- 32-bit Comparator
- 64-bit Comparator
- 128-bit Comparator

The cores that are produced with core generator are optimized for Xilinx FPGA's. For each one of the two multipliers and adders mentioned above we had the option to use some of the XtremeDSP slices that Virtex-4 offers. So in order to have a broader picture we created 8 cores: One single precision floating point adder/subtractor and one single precision floating point multiplier with no DSP usage. We also created one single precision floating point adder/subtractor and one single precision floating point multiplier with maximum DSP usage. The same is done for the double precision adders and multipliers.

The cores produced from Xilinx Core Generator are not suitable for synthesis in other tools (for example Synopsis and RTL Compiler). As a result we downloaded, and simulated 4 codes from www.opencores.org : a single precision floating point adder/subtractor, a single precision floating point multiplier, a double precision floating point adder, a double precision floating point multiplier and a double precision floating point divider. Then we created 4 shift registers of various lengths. A 32-bit, a 64-bit, a 1024-bit and a 4096-bit shift register. We also created 3 comparators of various lengths: A 32-bit, a 64-bit and a 128-bit comparator.

In the following table we present you all the cores we used and how we obtained them for the two design flows:

Core	Core_Gen	Core_Gen_MaxDSP	Opencores	My_VHDL
SP_FP_Add/Sub	✓	✓	✓	
SP_FP_Multi	✓	✓	✓	
DP_FP_Add	✓	✓	✓	
DP_FP_Multi	✓	✓	✓	
DP_FP_Div	✓		✓	
Shift_Reg_32				✓
Shift_Reg_64				✓
Shift_Reg_1024				✓
Shift_Reg_4096				✓
Comparator_32	✓			✓
Comparator_64	✓			✓
Comparator_128	✓			✓

All of the mathematical cores that are mentioned above meet IEEE 754 standard for single or double precision floating point arithmetic. The IEE Standard for Floating Point Arithmetic is the most widely-used standard for floating point computation and is followed by many hardware and software implementations. The current version is IEEE 754-2008, which was published in August 2008; the original IEEE 754-1985 was published in 1985. 32 bits are used to represent a single precision floating point number.

- Sign Bit:1
- Exponent Width:8
- Significant Precision:23 (24 implicit)

Sign	Exponent	Mantissa
32	31……..23	22……0

A double precision floating point number is represented using 64 bits:

- Sign Bit:1
- Exponent Width:11
- Significant Precision:52 (53 implicit)

Sign	Exponent	Mantissa
63	62……..52	51……0

The block diagrams of these cores are presented below:

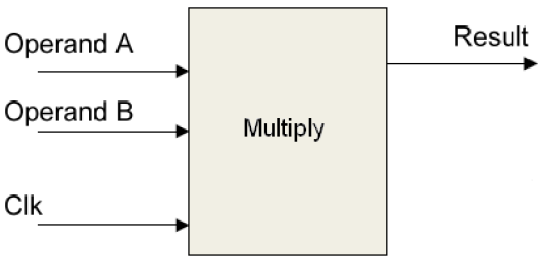


Figure 3 - 2: Multiplier Block Diagram.

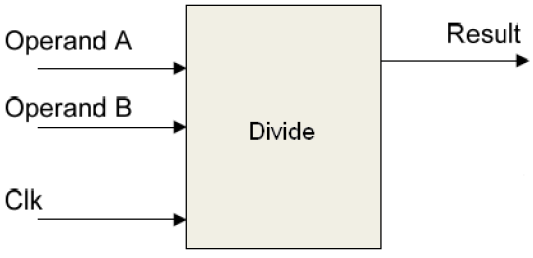


Figure 3 - 3: Divider Block Diagram.

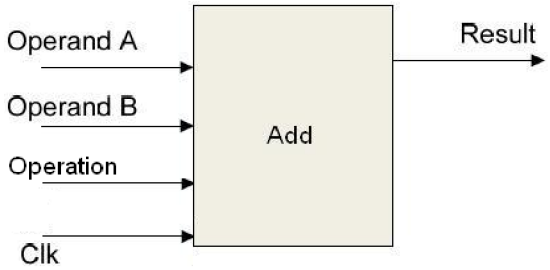


Figure 3 - 4: Adder Block Diagram.

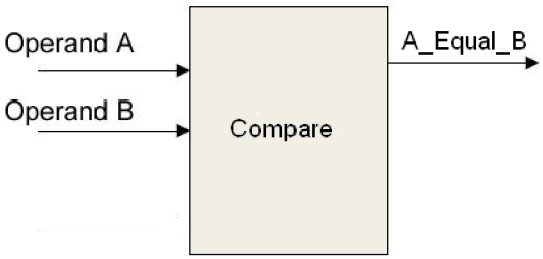


Figure 3 - 5: Comparator Block Diagram.

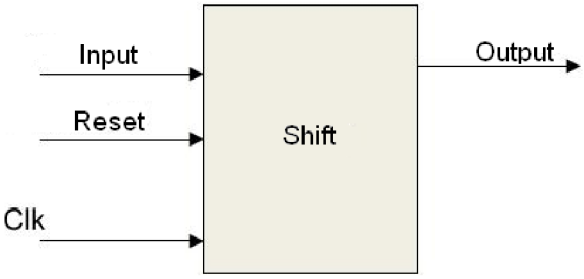


Figure 3 - 6: Shift Register Block Diagram.

We first synthesized each one of the above codes. For all the cores we mentioned above we find out the exact area (this means the number of slices) the design occupies, the timing specifications of the design and its power consumption. In order to find the power consumption of our cores we followed these steps:

- Created a very simple circuit and measured the power consumption of the FPGA with it (a design with a 1-bit input that goes directly to the 1-bit output). The power consumption of this simple circuit was the same even if we changed the size of the input and output (64 bits input and output). This means that every time all the input and output pins (pads) are taken into consideration to measure the power consumption of the FPGA.
- We measured the power consumption of the FPGA with the cores we want to study.
- We subtracted the total power of the simple design from the total power consumption of each one of our cores.

The tool we used for this procedure was Xilinx ISE 9.1 running on Windows XP environment. In order to measure the power consumption of each design we used Xilinx's XPower Analyzer 10.1. The computer we used has a CPU with a frequency of 3GHz and 1.5GB of RAM.

IMPLEMENTATION IN VLSI

For the VLSI implementation we used two tools provided by Cadence Design Systems Inc.: Encounter RTL Compiler and SOC Encounter RTL-to-GDSII System.

Cadence Design Systems, Inc, founded in 1988, is the global leader in software, hardware, methodologies, and services that play essential roles in accelerating innovation in today's highly complex integrated circuits, printed circuit boards, and electronics systems. Companies use Cadence electronic design automation (EDA) technologies and engineering services to design, verify, and prepare advanced semiconductors and systems for manufacturing. These products, in turn, form the foundation of consumer electronics, networking and telecommunications equipment, and computer systems. With locations throughout the world—including China, India, Europe, Russia, Israel, Japan, Korea, Taiwan, and North America—Cadence serves a global customer base.

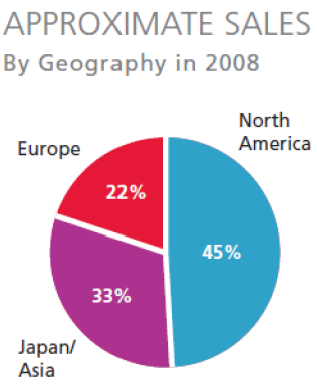


Figure 3 - 7: Cadence Sales Worldwide (Source: www.cadence.com).

Cadence Encounter RTL Compiler offers a unique set of patented global-focus algorithms that perform true top-down global RTL design synthesis. With concurrent multi-objective optimization (timing, area, and power) and support for advances low-power design techniques, Encounter RTL compiler reduces chip power consumption while meeting frequency goals.

The Cadence SoC Encounter RTL-to-GDSII System supports large-scale complex flat and hierarchical designs. It combines advanced RTL and physical synthesis, silicon virtual prototyping, automated floorplan synthesis, clock tree and clock mesh synthesis, advanced low-power implementation, and a complete suite of design for manufacturability, variation, and yield optimization technologies required for advanced node designs. The overview of SoC Encounter's functionality is shown below.

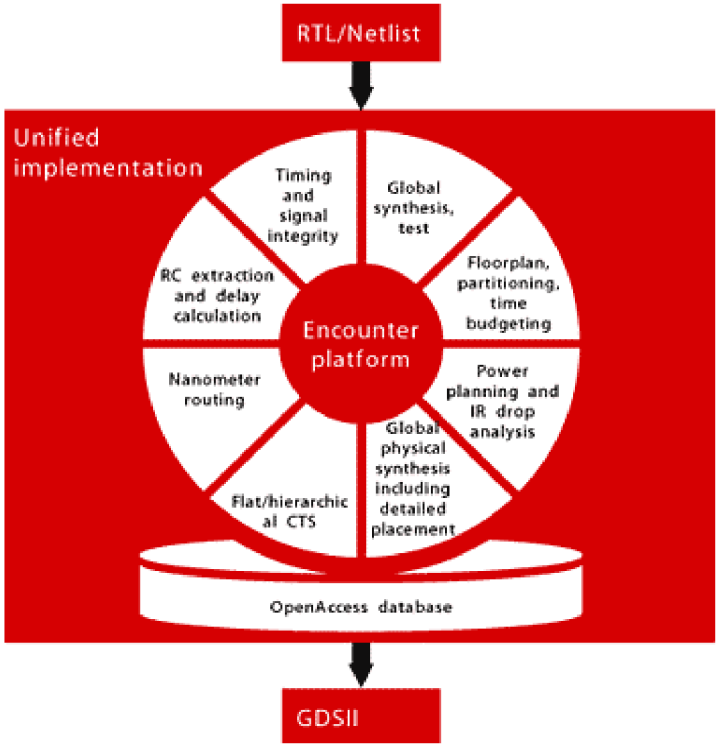


Figure 3 - 8: SoC Encounter Design Flow (Source: http://www.charteredsemi.com).

Both these tools require a Linux, Solaris or IBM platform in order to run properly. We installed them in a computer running CentOS 5.2. CentOS is an operating system based on Red Hat Enterprise Linux, which is the Linux distribution Cadence proposes for the installation of its tools.

The technology we used for the implementation in VLSI is Faraday's 90nm. The FSD0A_A library is a 90 nm standard cell library tailored for UMC's 90 nm logic SP-RVT (Low-K) process. It is optimized for the applications requiring high performance, low operating power consumption, and ultra high density. The characterizations conditions for this technology are shown in table 1 while the general characteristics of Faraday's 90nm are described in table 2:

Operating condition		Min	Typ	Max	Unit
VCC	Core cells	0.9	1.0	1.1	V
	2.5 V I/O cells	2.25	2.5	2.75	V
T_J	Junction operating temperature	-40	25	125	°C

Table 3 - 2: FSD0A_A operating conditions (Source: Faraday FSD0A_A 90 nm Logic SP_RVT (Low-K) Process).

Characteristic	Description
Technology	UMC's 90 nm logic SP-RVT (Low-K) process.
The length of the minimum drawn channel	0.08 µm
Supply voltage	For the core cells: 0.9 V ~ 1.1 V For 32.5 V I/O cells: 2.25 V ~ 2.75 V
Performance	T_d = 18.2 ps/stage (measured from the 101-stage NAND ring in the typical process and operating under 1.0 V, 25 °C)
Gate density	400k gates/mm^2
Power consumption	5.0 nW/MHz/gate (measured from the 2-input NAND, output load = 2 standard loads, in the typical process and operated under 1.0 V, 25 °C)

Table 3 - 3: FSD0A_A General Characteristics (Source: Faraday FSD0A_A 90 nm Logic SP_RVT (Low-K) Process).

In this technology there are 9 different metal layers. One metal layer, metal 9, is made 4 times thicker than the normal, non thick, metal layer. The second and third top metal layers (metal 8 and metal 7) are made two times thicker than the normal, non thick, metal layer. The rest six metal layers (metal 1, metal 2, metal 3, metal 4, metal 5 and metal 6) are made in the normal metal process. The FSD0A_A provides also 8 different fillers cells. These cells have different size so that they can fill every gap in the core. During our design flow the names VCC and GND must be assigned to the power and ground respectively.

The complete design flow from HDL to GDSII is shown below:

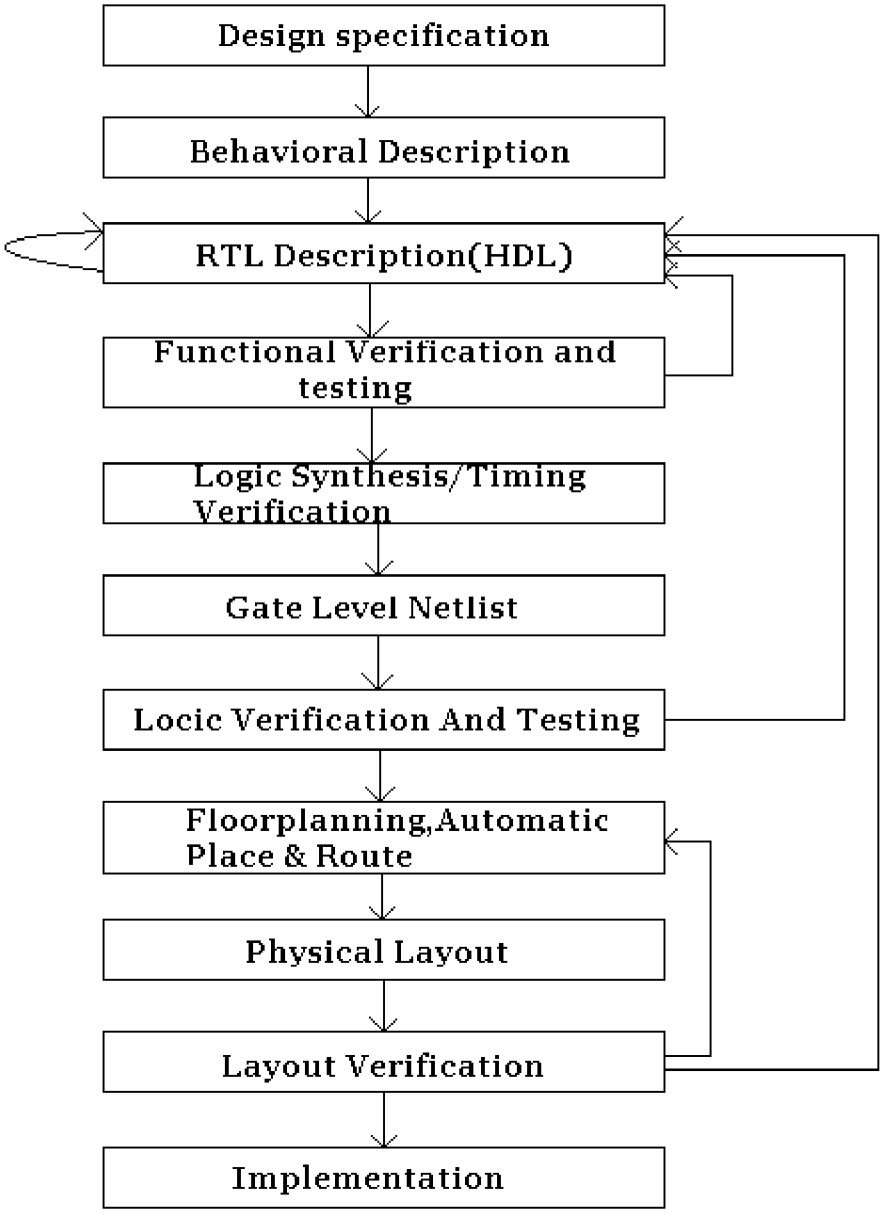


Figure 3 - 9: Complete design flow from HDL to GDSII.

RTL COMPILER STEPS

1. Using RTL Compiler the first step is to read our design. This is done by using the read_hdl command. It reads our vhdl(or verilog) files and creates HDL independent objects in HDL-intermediate format and stores it in design library. The elaborate command elaborates the top-level design to bind all packages and the designs.
2. Then we set the constraints for our design. These typically include defining the clock period, input and output delays etc. After this step, the design is now constrained and ready for mapping and optimization.
3. Afterwards we synthesize our design by typing the following two commands in the rc command prompt :

   ```
   set MAP_EFF high
   synthesize -to_mapped -eff $MAP_EFF
   ```

   These two commands will map and optimize our design with high mapping effort in order to meet the constraints we defined in the previous step. In the end of this step our design is fully-mapped to gate level.

4. In this step we generate the mapped design (.v file) and the design constraints file (.sdc file).These files will be used for placement and routing.

SOC ENCOUNTER STEPS

1. The first step after launching Cadence SoC Encounter RTL-to-GDSII System is to import our design. This procedure requires the following files:
   - A **L**ayout **E**ntity **F**ile (lef) that defines the physical rules, such as the minimum width of the metal layers, the minimum space between various metal layers etc.
   - A .lef file that contains all the physical characteristics of the standard cells.
   - A .lib file that defines the timing specification of the standard cells and is necessary for the timing analysis of our design.
   - The netlist we extracted from RTL Compiler (.v file).
   - The **S**ynopsis **D**esign **C**onstraints file (.sdc file) that contains all the required timing constraints for our design.

   As we only focus on creating cores and not entire chips it is not necessary to add pads to our design.

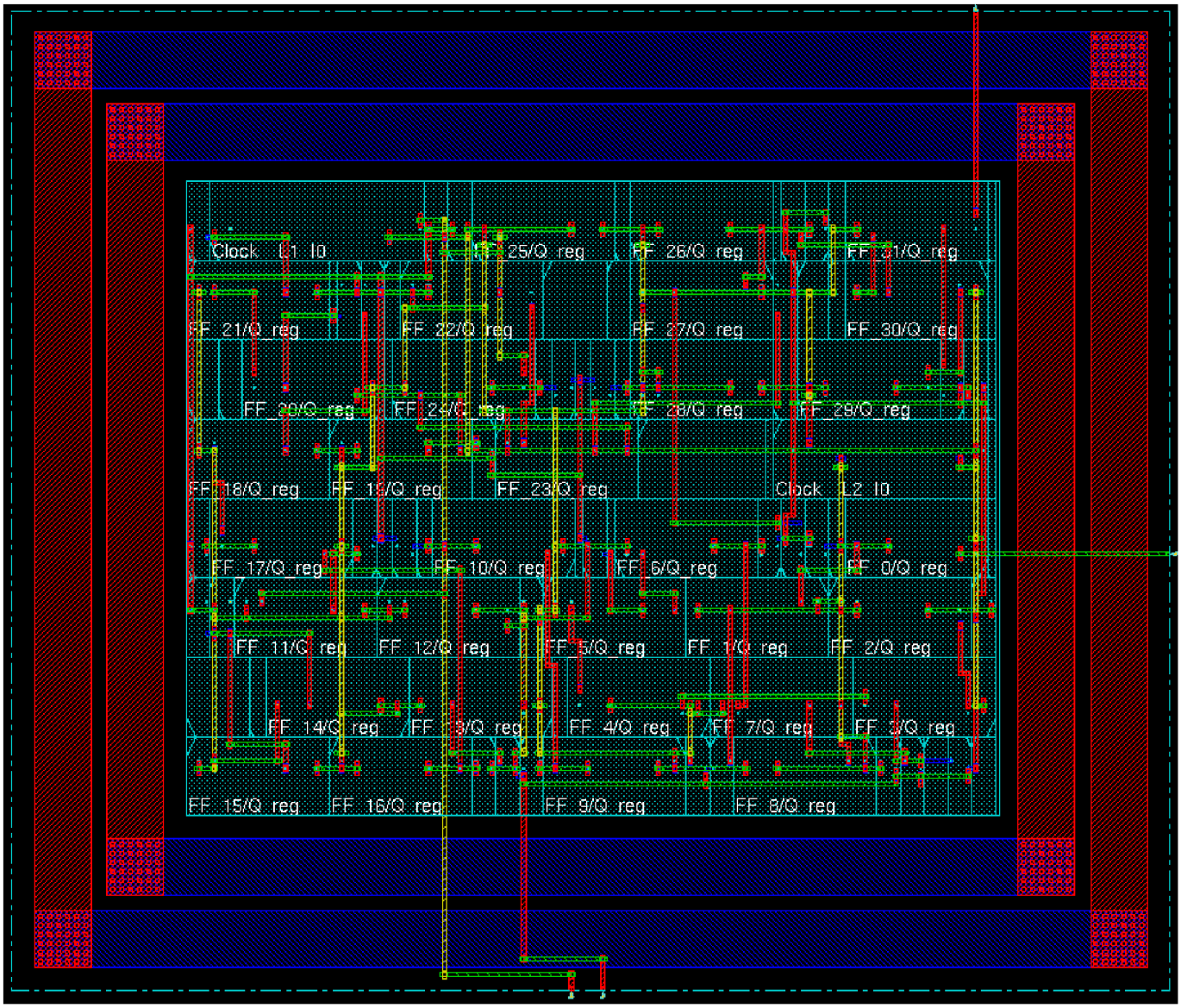
2. We choose the operating conditions for our design. These are the BCCOM (minimum conditions) and WCCOM (maximum conditions) which we described earlier.
3. In this step we perform floor planning of the device. We specify the aspect ratio of the die (usually set to 1.0) and the utilization of the core. Depending on the type of the design we must specify this number appropriately. High utilization ratios (>0.75) may result in very high congestion during routing which will make the design practically unroutable. For the cores we created ourselves(the comparators and the shift registers) the core utilization is over 0.7, while in the 3 cores(Single precision multiplier, single precision adder and double precision multiplier) we started our implementation using high utilization ratios(0.5-0.7) but all our tries failed during DRC check. So we implemented these three cores using utilization ratios varying from 0.35-0.5. Then we specify the core-to-IO boundary distances that are applied to the four sides of the core (Left, Bottom, Right, Top) in order to allow the later introduction of power and ground rings. The size depends entirely on the design. In our small designs we use spacing of 11 nm and in the larger designs spacing of 26 or 30 nm.
4. The next step in the flow is power planning. More specifically we create the power rings and power stripes which will distribute power in the cells of our design. We choose the metal layer to use for the power rings (which will be used as VCC and GND) and the desired width and spacing. Furthermore we specify the width and the set-to-set distance of the power stripes. The number of the power stripes must be chosen carefully as too few power stripes may result in poor power distribution while too many may result in high congestion and more power consumption. We prefer in most of our designs not to use metal1 and metal2 layers for the power planning, in order to be able to place cells under the metal layers.
5. After we finish the power planning there is a typical step, in which we define that all cells will connect to the VCC and GND signals.
6. The fifth step consists of the standard cell placement. SoC encounter tries to place all standard cells in legal position. We choose whether we want the placement to be timing driven (derives necessary timing information from the .sdc file of our design) or not and also whether we want congestion optimization (if turned on the placer tries to minimize congestion).
7. Then comes the phase of special route (sroute) which does the routing for special nets such as the power nets VCC and GND.
8. The next step in the flow is trial routing. This not actual physical routing but gives us a first impression of the congestion and the timing of our design.
9. In this step we perform an in place optimization before designing the clock tree. During in place optimization (IPO) the size and the structure of the gates changes in order to increase their driving capability and reduce delay.

10. After we finish the PreCts IPO we must design the clock tree for our design. For this procedure we must first specify in a file, all the desired constraints. This file contains information about the clock root (or roots if we have more than one clocks), the required clock period, the maximum required delay, maximum available clock skew etc. Furthermore in this file, in the parameter named Buffer we list all the standard cells which will be used as amplifiers during clock tree synthesis. After the clock tree specifications file has been created we import it and run clock tree synthesis which takes some time in order to complete.
11. In this step we perform another Trial Route, now taking the synthesized clock tree into consideration and we let the Encounter run in place optimization again with the option –postCTS.
12. Then we try to fix the remaining violations by executing the commands fixDRCViolations and fixSetupViolations. The first focuses in geometrical violations and the second in timing violations.
13. The next step consists of the routing. We first determine all the parameters as for example: if the procedure will be timing driven, if the router will attempt optimization in time, if it will alter the size of the gates etc. After we specify these attributes we start the global and detail routing using NanoRoute. It takes some time for the routing to complete. The tool performs violation checks, DRC (Design Rule Checking) and it also reports errors and warnings after the process is complete.
14. In order to finish our design we must fill the empty space of the core with fillers. Filler cells are added in order to make metal-1,n-well and p-well continuous. As we already mentioned Faraday's 90nm technology provides 8 fillers cells of various sizes in order to fill every gap of the core.
15. Finally we must make sure that our design has no violations. So we run DRC and connectivity check. The first checks for and DRC errors while the second checks if there is any unconnected/floating net. None of the cores we designed had any violations of any kind.

After we have done all the above steps we analyze timing and synthesize the power plan in order to find out the timing specifications and power consumption of our design. All the results we extracted will be shown in the next chapter.

We will now present you all the masks we produced by following the above steps. Note that in the case of the Single Precision Floating Point Adder/Subtractor and in the case of the Double Precision Floating Point Multiplier the density of the core is low (approximately 40%) but that was the only way to avoid DRC violations).The double precision floating point adder and divider failed DRC check but the results we extract concerning frequency and power consumption are close to be accurate.

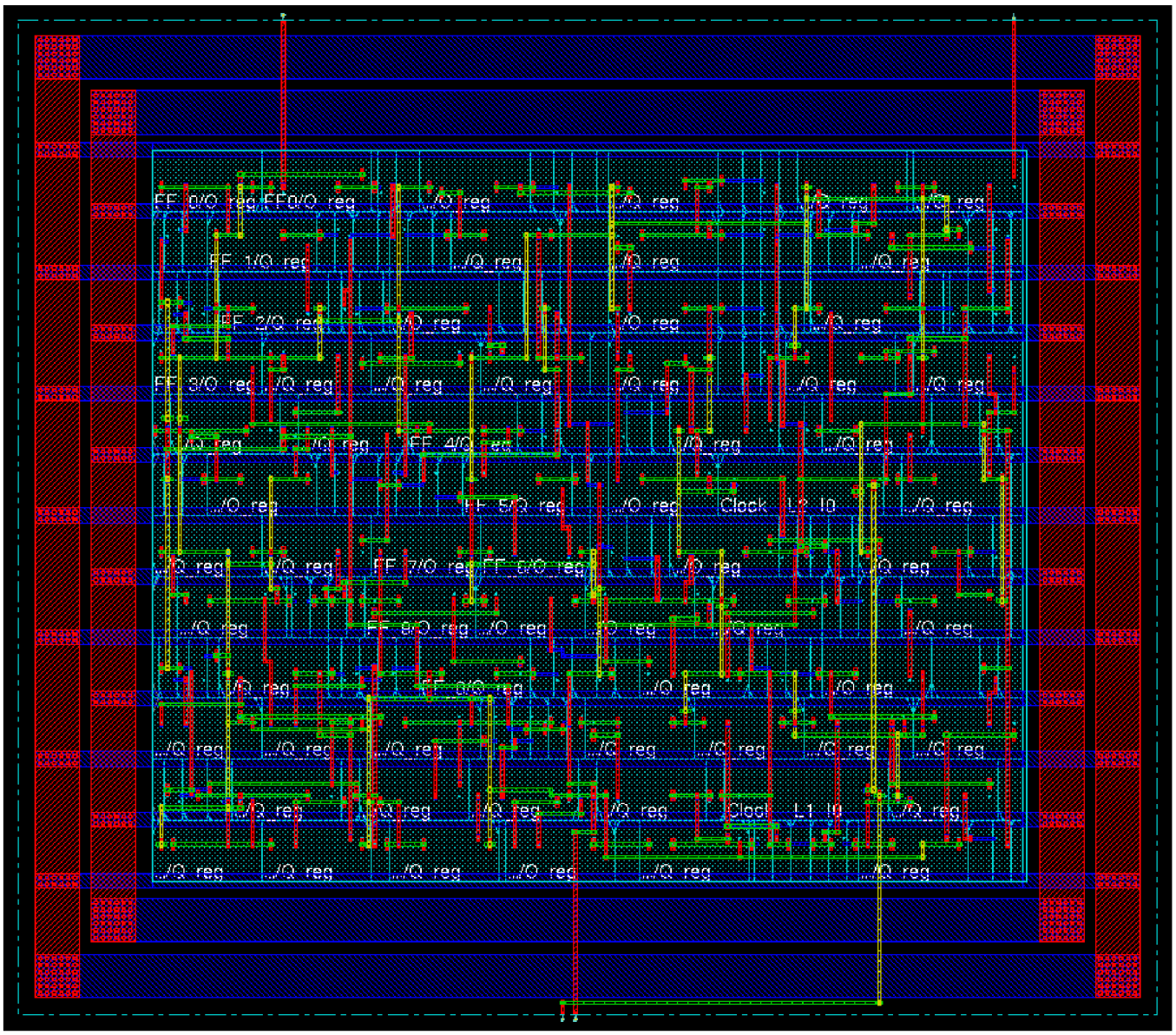
Shift Register 32-Bits



Dimensions	Width (nm)	Height (nm)
	28.685	22.4

Mask 3 - 1: Shift Register 32-Bits

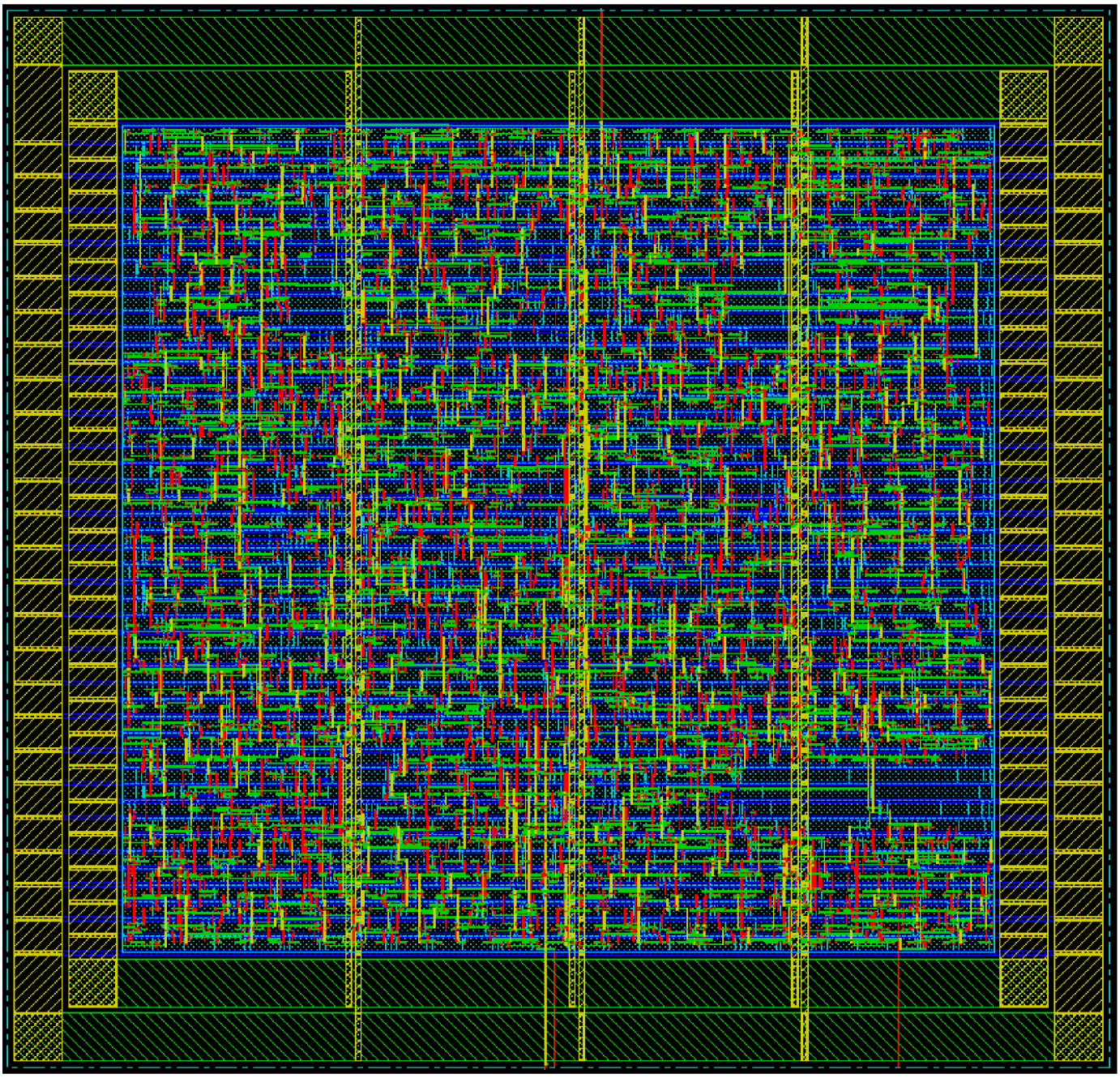
Shift Register 64-Bits



Dimensions	Width (nm)	Height (nm)
	40.205	33.6

Mask 3 - 2: Shift Register 64-Bits

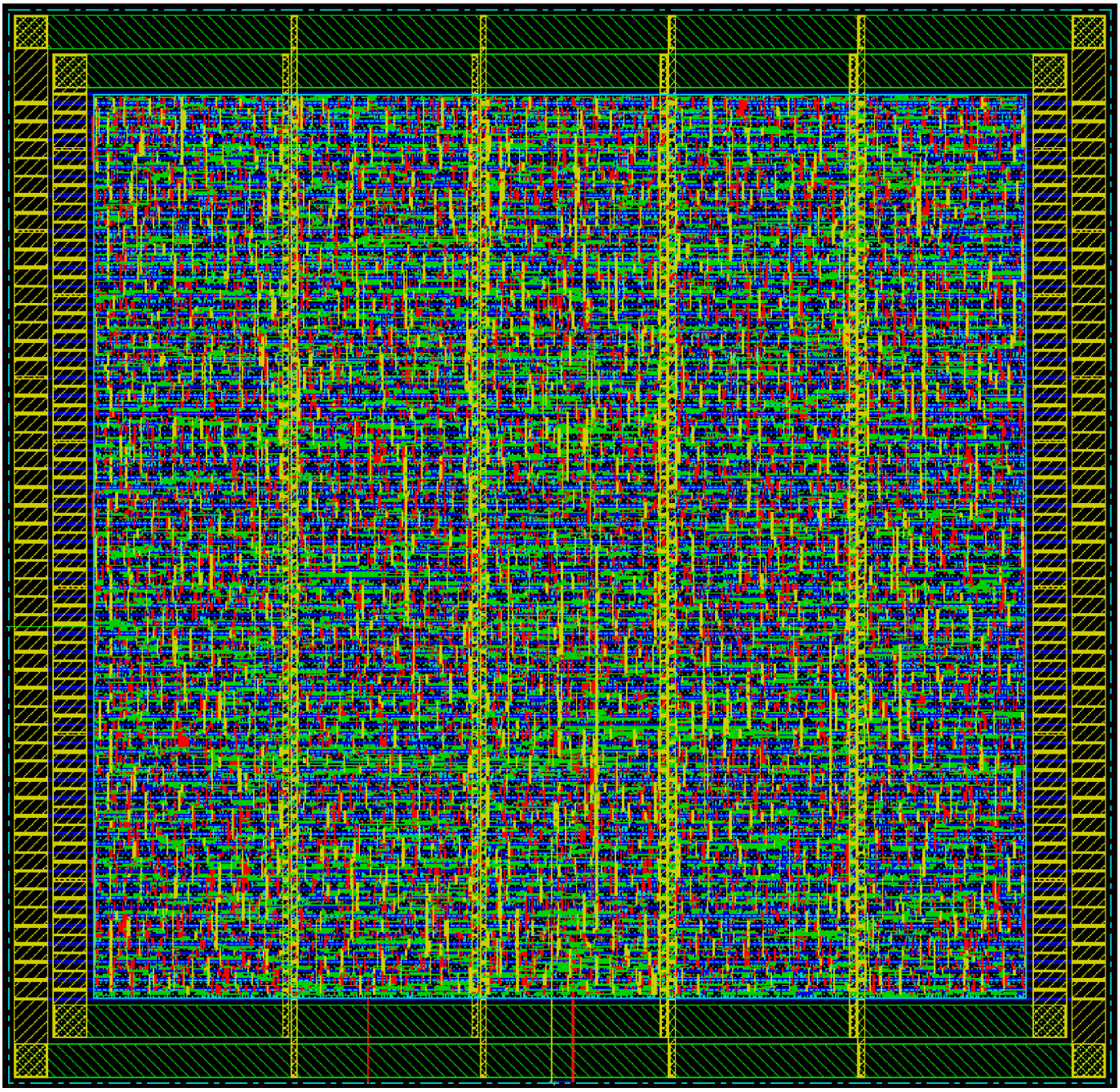
Shift Register 1024-Bits



Dimensions	Width (nm)	Height (nm)
	144.565	137.2

Mask 3 - 3: Shift Register 1024-Bits

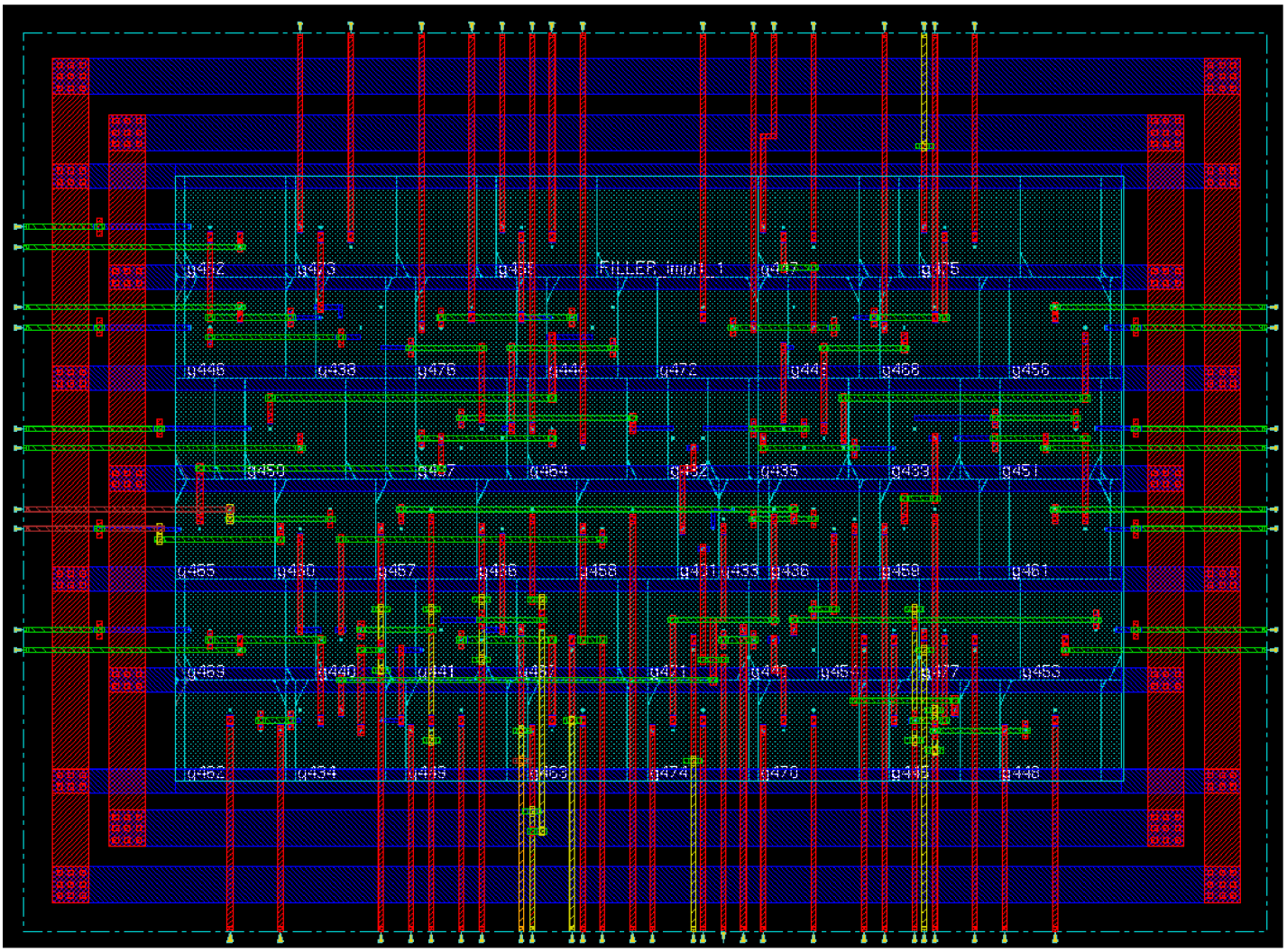
Shift Register 4096-Bits



Dimensions	Width (nm)	Height (nm)
	286.21	277.2

Mask 3 - 4: Shift Register 4096-Bits

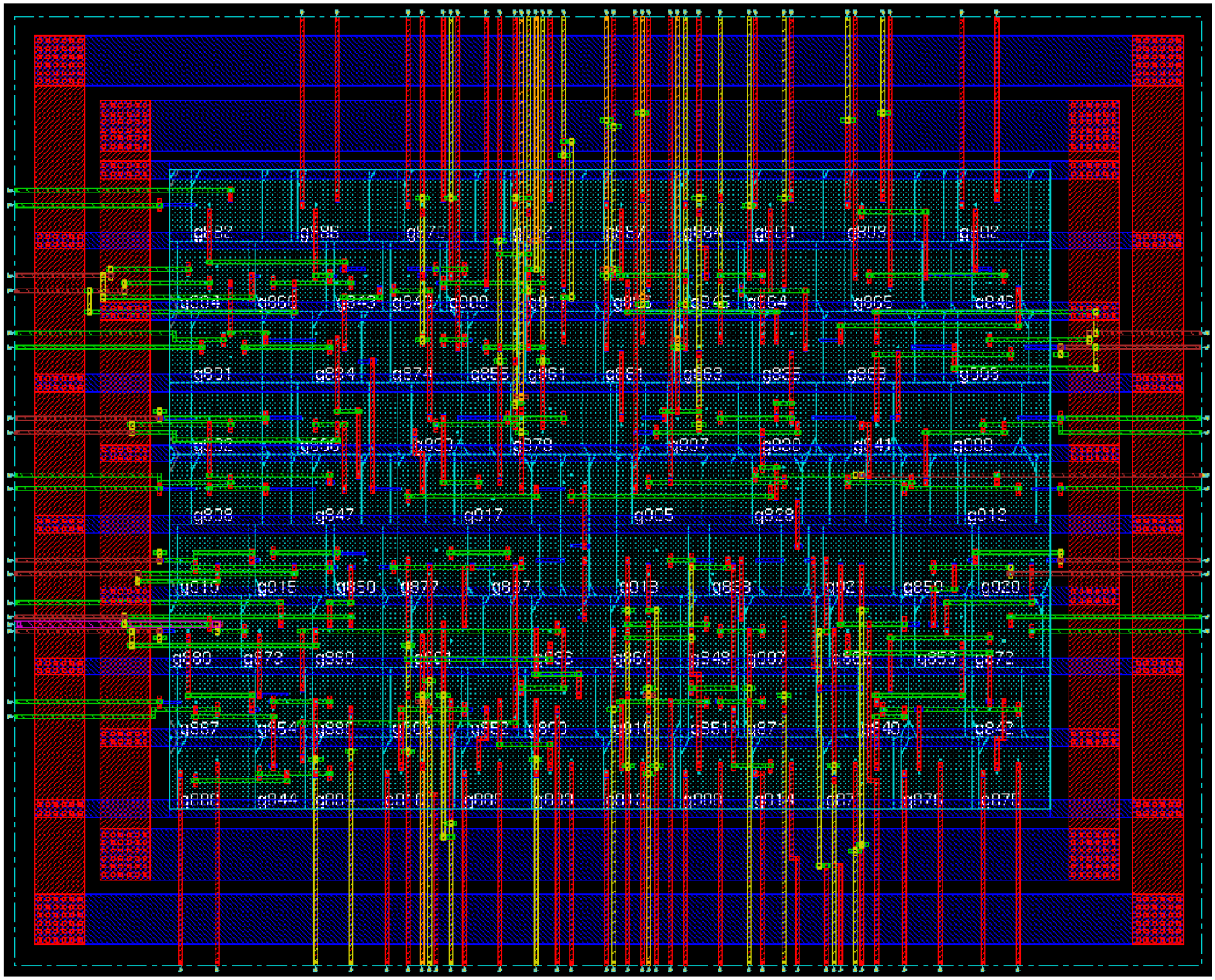
Comparator 32-Bits



Dimensions	Width (nm)	Height (nm)
	26.385	16.8

Mask 3 - 5: Comparator 32-Bits

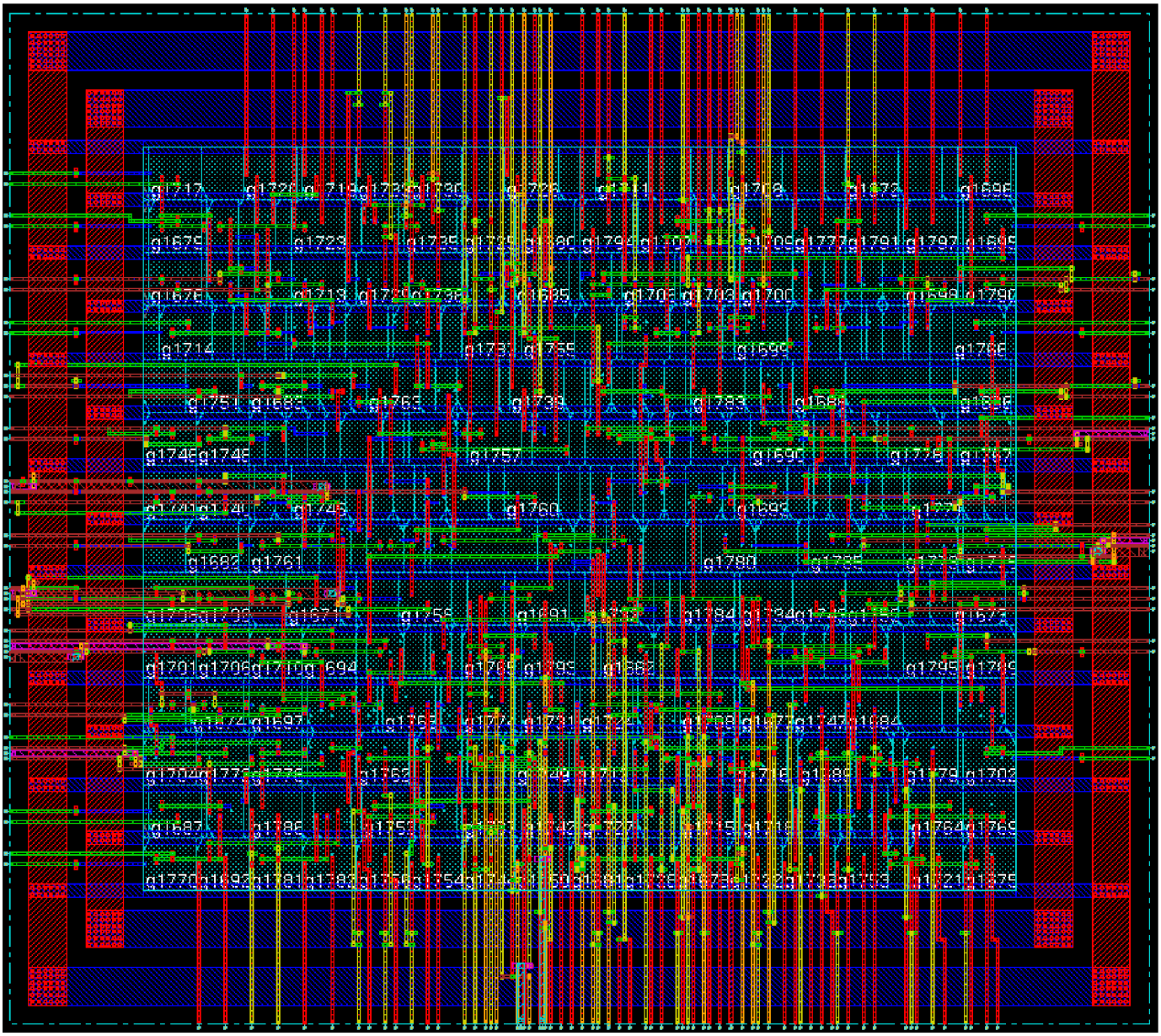
Comparator 64-Bits



Dimensions	Width (nm)	Height (nm)
	32.55	25.2

Mask 3 - 6: Comparator 64-Bits

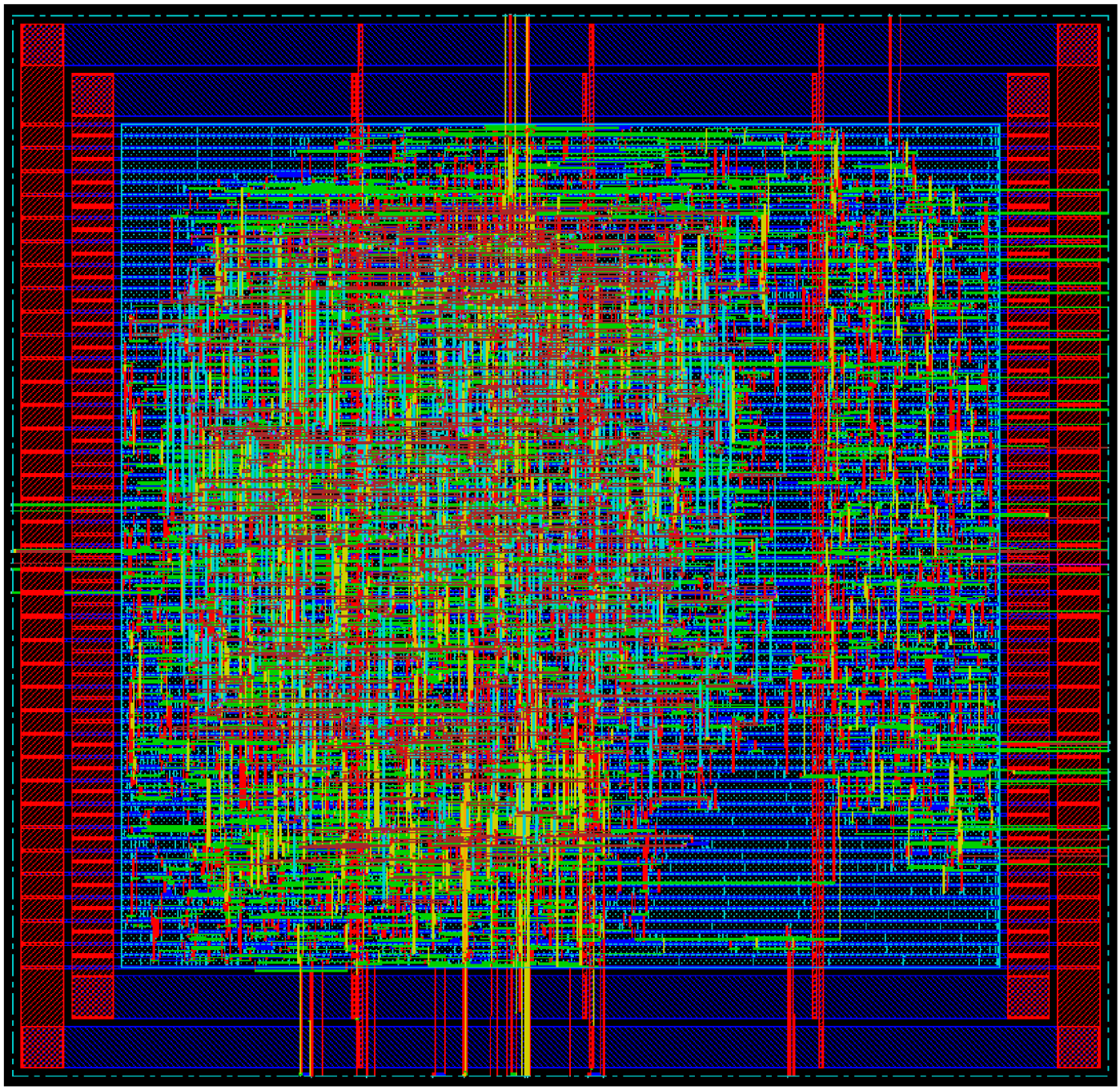
Comparator 128-Bits



Dimensions	Width (nm)	Height (nm)
	45.95	39.2

Mask 3 - 7: Comparator 128-Bits

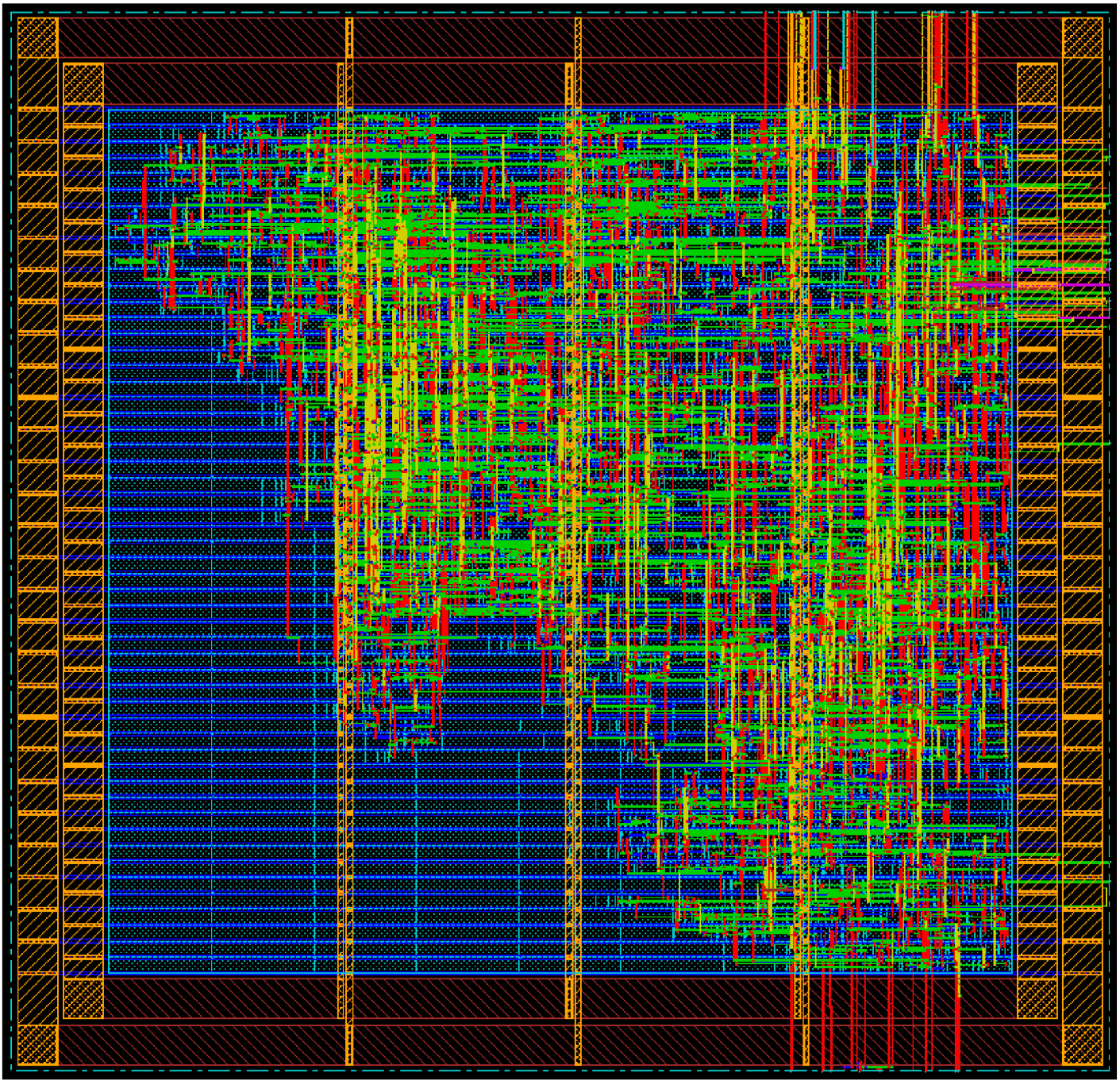
Single Precision Floating Point Multiplier



Dimensions	Width (nm)	Height (nm)
	209.67	201.6

Mask 3 - 8: Single Precision Floating Point Multiplier

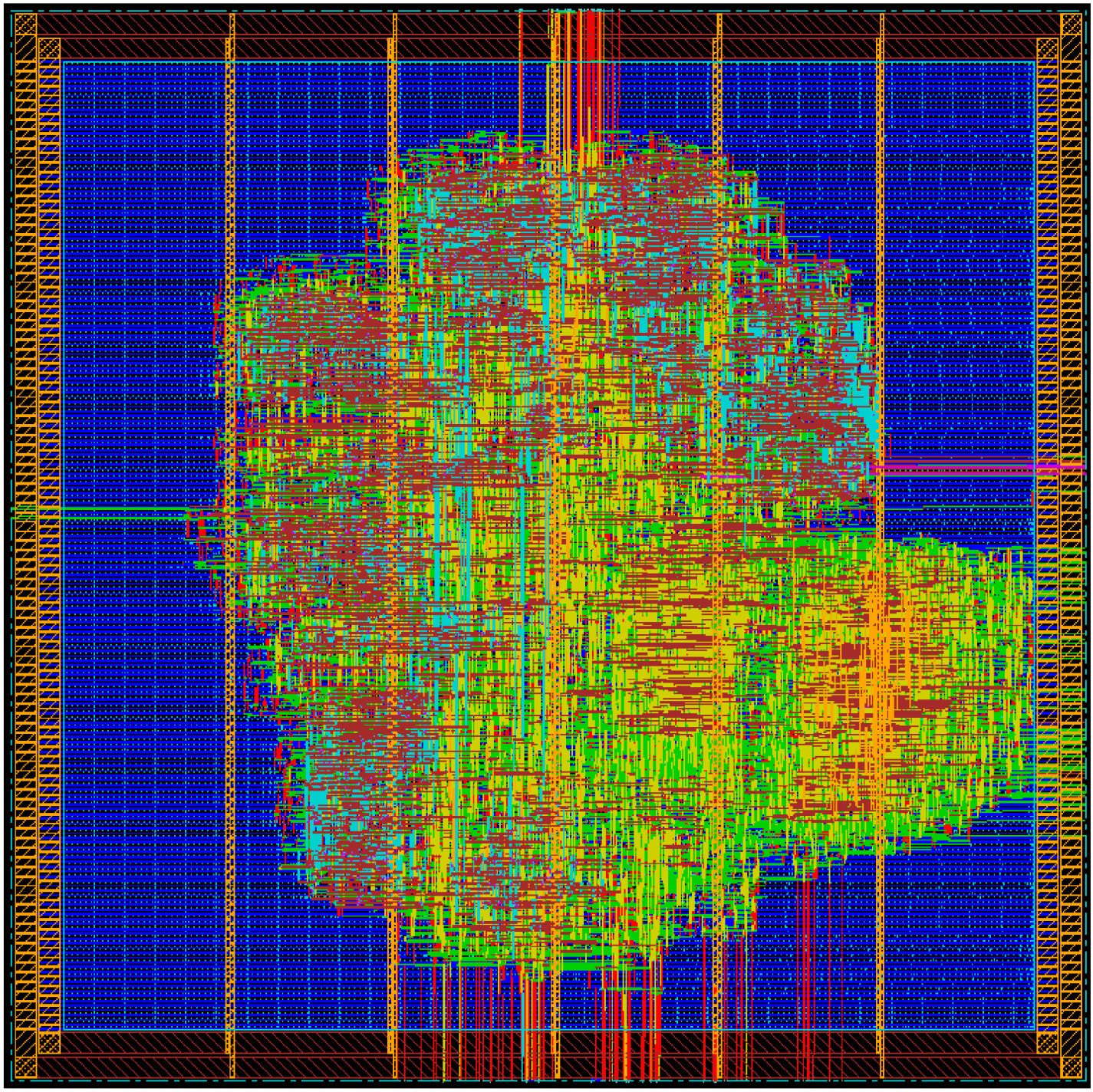
Single Precision Floating Point Adder-Subtractor



Dimensions	Width (nm)	Height (nm)
	157.995	151.2

Mask 3 – 9: Single Precision Floating Point Adder-Subtractor.

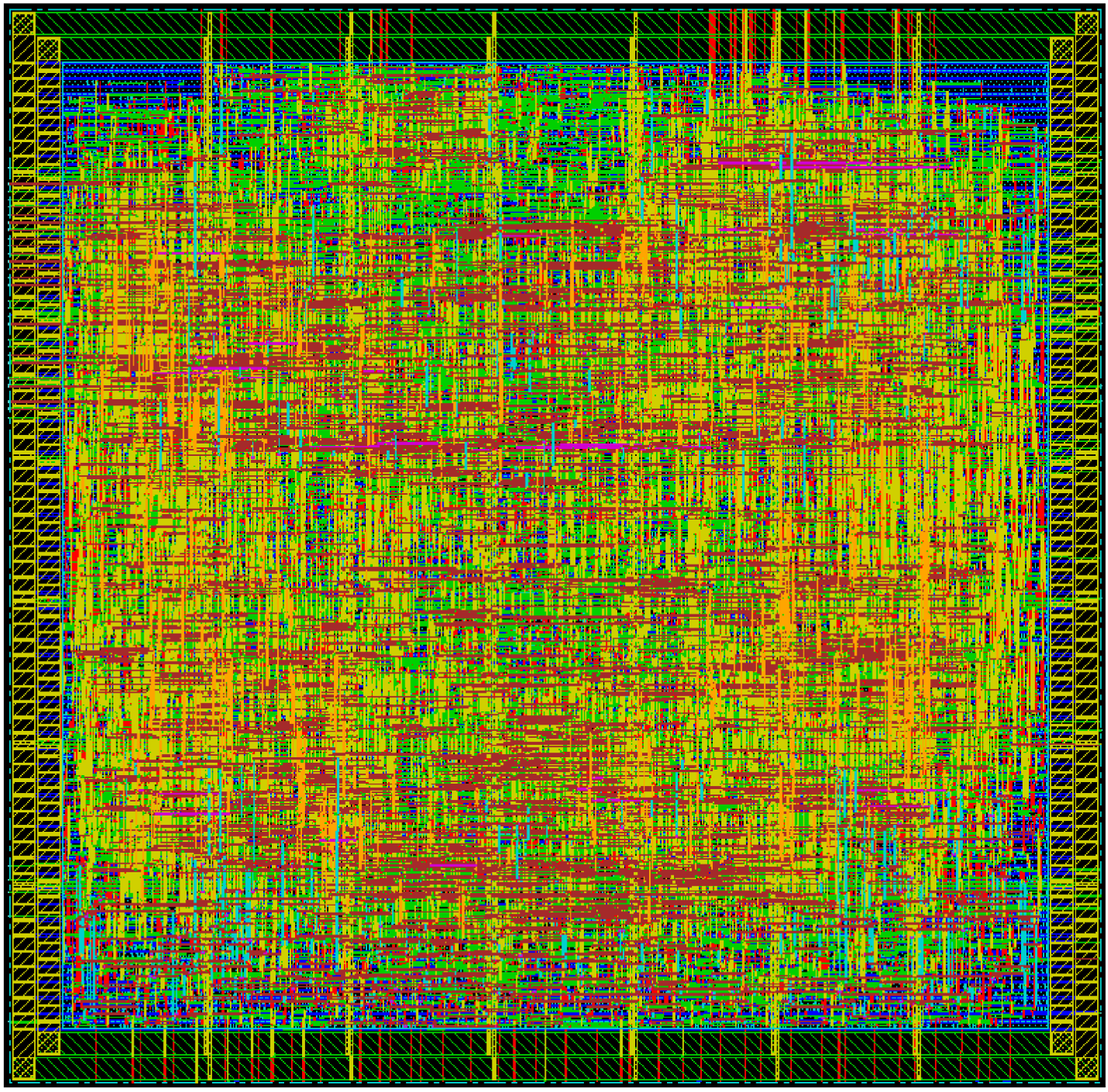
Double Precision Floating Point Multiplier



Dimensions	Width (nm)	Height (nm)
	567.215	565.6

Mask 3 – 10: Double Precision Floating Point Multiplier

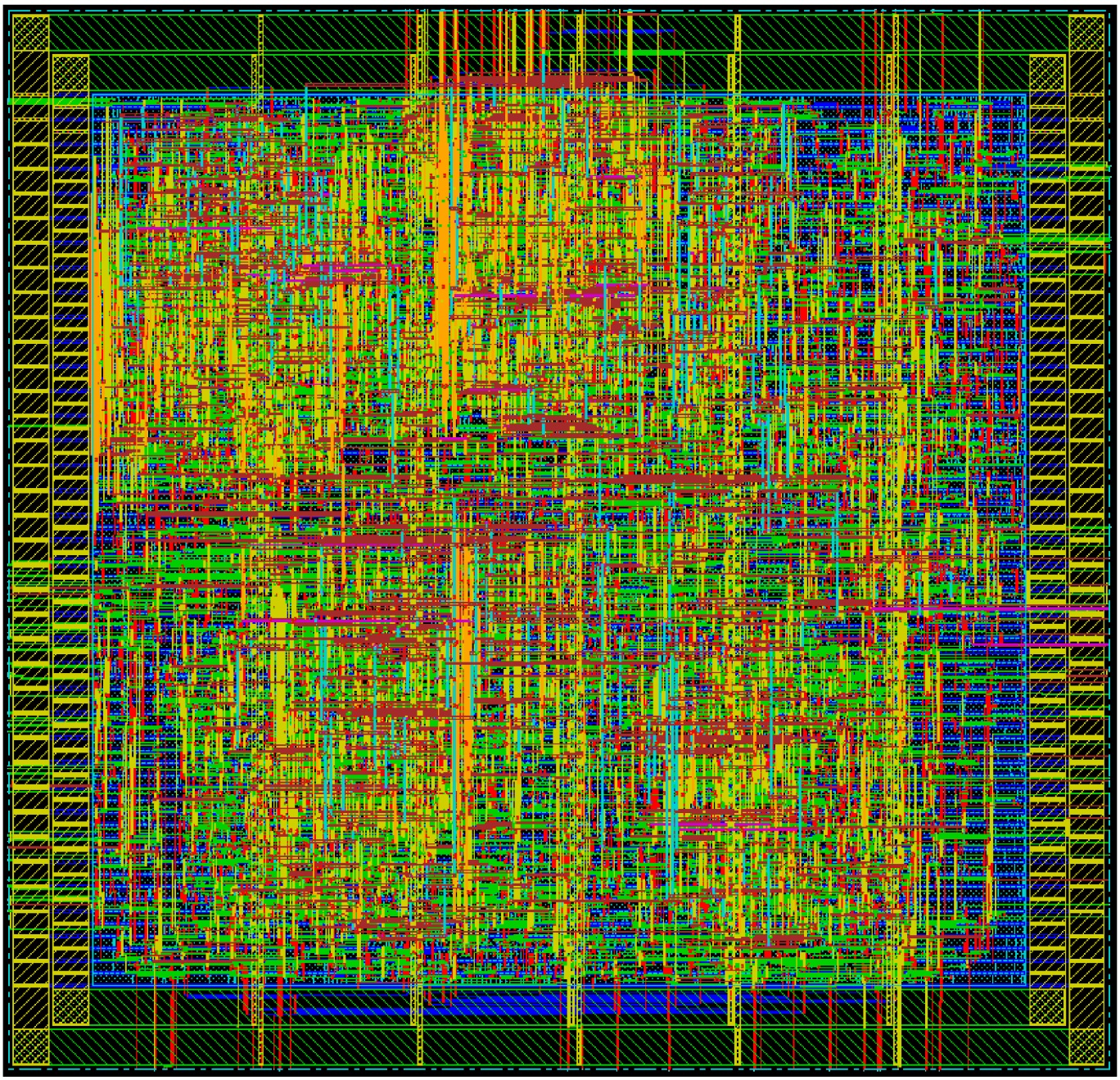
Double Precision Floating Point Adder



Dimensions	Width (nm)	Height (nm)
	354.525	347.2

Mask 3 - 11: Double Precision Floating Point Adder

Double Precision Floating Point Divider



Dimensions	Width (nm)	Height (nm)
	211.355	201.6

Mask 3 - 12: Double Precision Floating Point Divider.

CHAPTER 4: RESULTS

In this chapter we will present you and compare all the results we gathered from the previous design flows.

Math Cores

	FPGA		VLSI	
	Frequency(MHz)	Power(mW)	Frequency(MHz)	Power(mW)
Single Precision FP Multiplier Coregen	332	41	-	-
Single Precision FP Multiplier Coregen_MAX_DSP	445	51	-	-
Single Precision FP Multiplier Opencores	117	48	193	1.72
Single Precision FP Adder/Subtractor Coregen	451	44	-	-
Single Precision FP Adder/Subtractor Coregen_MAX_DSP	418	47	-	-
Single Precision FP Adder/Subtractor Opencores	132	39	238	1.543
Double Precision FP Multiplier Coregen	221	66	-	-
Double Precision FP Multiplier Coregen_MAX_DSP	353	103	-	-
Double Precision FP Multiplier Opencores	37	83	156	10.47
Double Precision FP Adder Coregen	355	57	-	-
Double Precision FP Adder Coregen_MAX_DSP	355	63	-	-
Double Precision FP Adder Opencores	180	144	278*	14.86*
Double Precision FP Divider Coregen	266.089	131	-	-
Double Precision FP Divider Opencores	32	75	166*	3.4*

Table 4 - 1: Frequency and power consumption of all the arithmetic cores.

(The Double Precision FP adder and Divider from opencores.org are marked with an asterisk as they had DRC violations after the VLSI implementation).

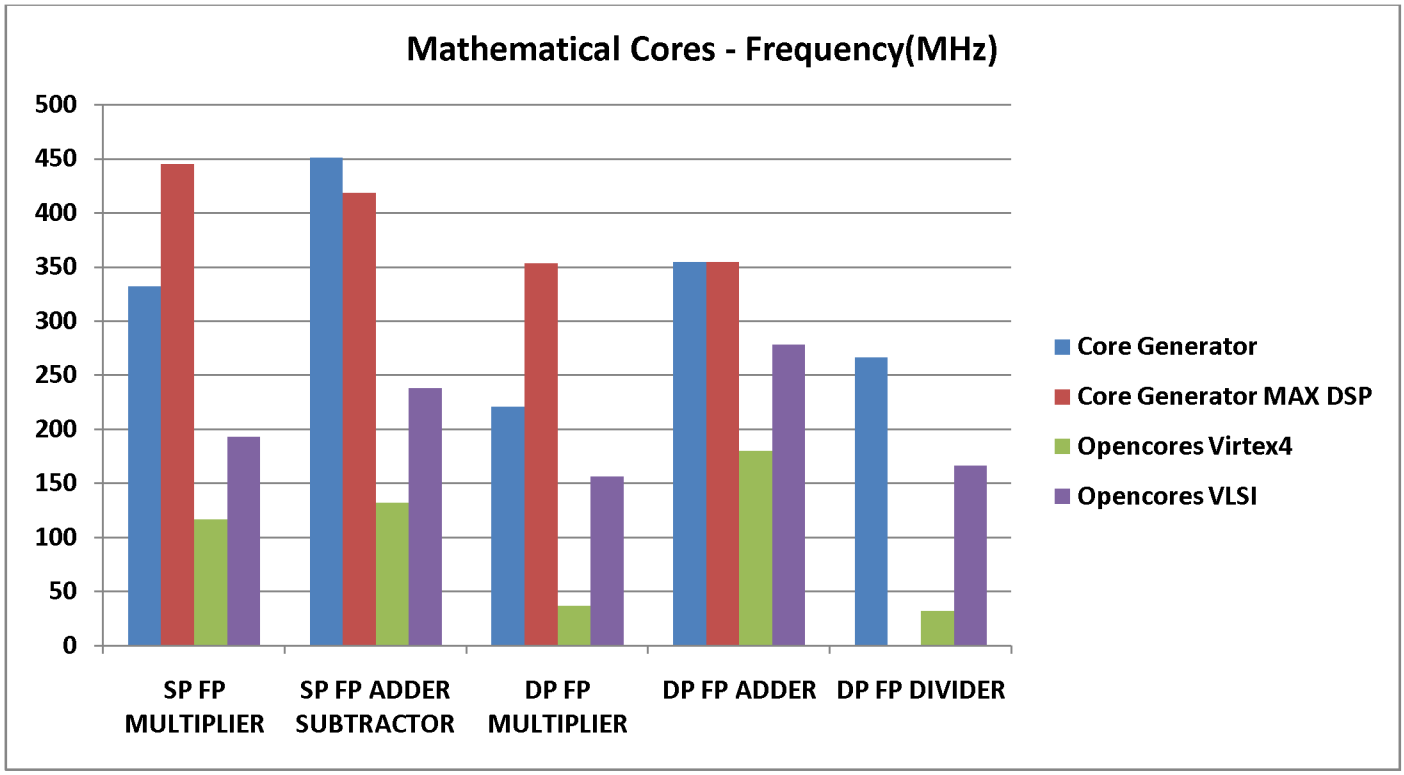


Diagram 4 - 1 : Frequency in MHz of the five mathematical cores depending on their implementation.

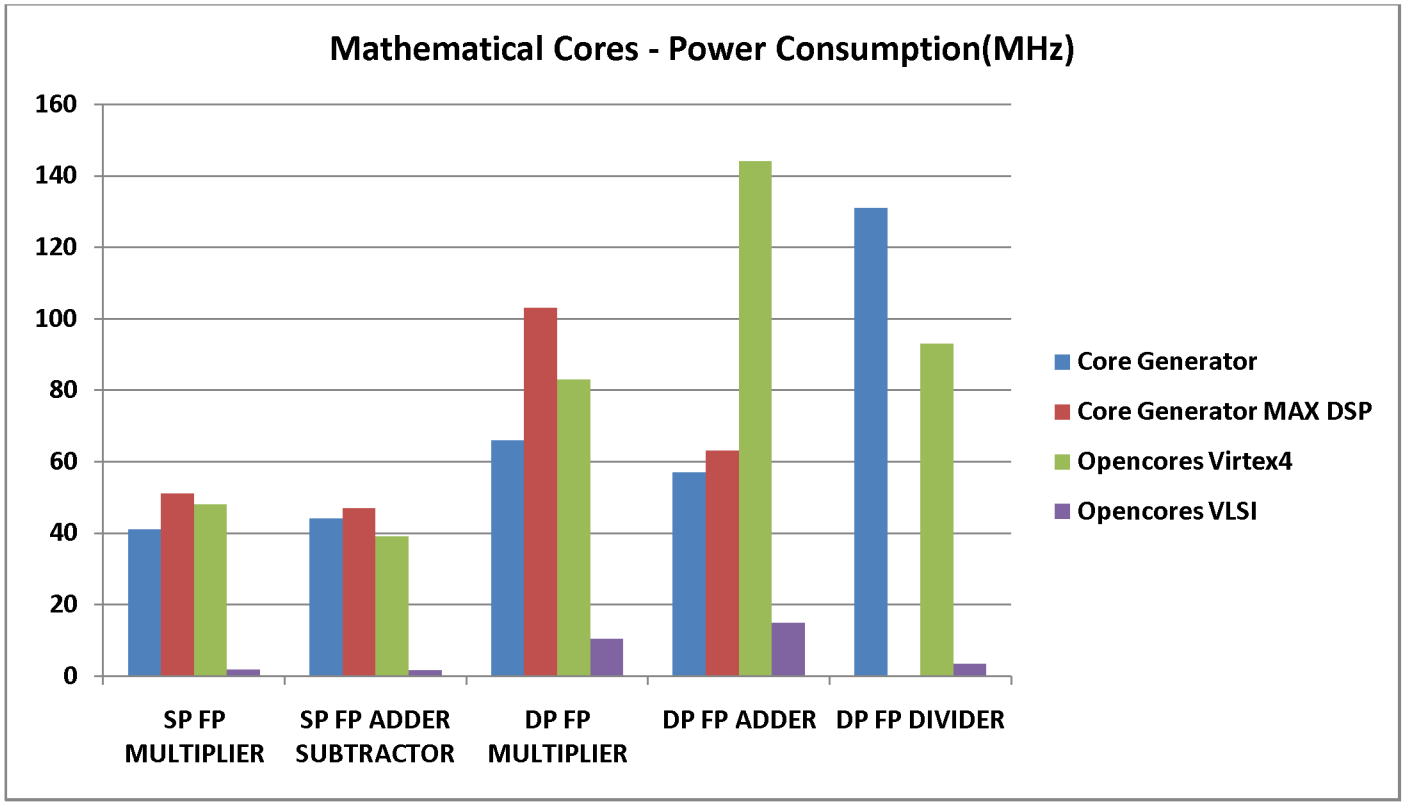


Diagram 4 - 2 : Total Average Power Consumption of the five mathematical cores depending on their implementation.

In all the above diagrams we can see that, by using EDA tools all the mathematical cores we produce are 1,5x to 2x faster compared to the respective FPGA implementation, while in the case of the double precision floating multiplier and divider we get 4x faster designs. However, these values are still a lot lower than these of the Virtex4 especially when we use the maximum available number of DSPs. When it comes to power consumption however, we can see that all the designs we produce using EDA tools, are more power efficient. The gap between the power consumption of the VLSI and the respective FPGA implementation varies between 10-25x depending on the design. This is because FPGAs contain LUTs and routing channels which are connected via bit streams, as they are made for general purpose and because of re-usability. Research has shown that the dynamic power consumption of a LUT is 500 times greater than the power of an ASIC gate. As a result, in every FPGA design we have extra circuitry which is not necessary and results in wastage of power.

Comparators

	FPGA		VLSI	
	Delay (ns)	Power(mW)	Delay (ns)	Power(mW)
Comparator_32 Coregen	6	3	-	-
Comparator_64 Coregen	6.711	4.82	-	-
Comparator_128 Coregen	7.412	8.72	-	-
Comparator_32	6.303	3.04	0.267	0.0131
Comparator_64	6.439	4.97	0.33	0.0246
Comparator_128	6.711	9.39	0.365	0.0518

Table 4 - 2: Frequencies and power consumption of all comparators.

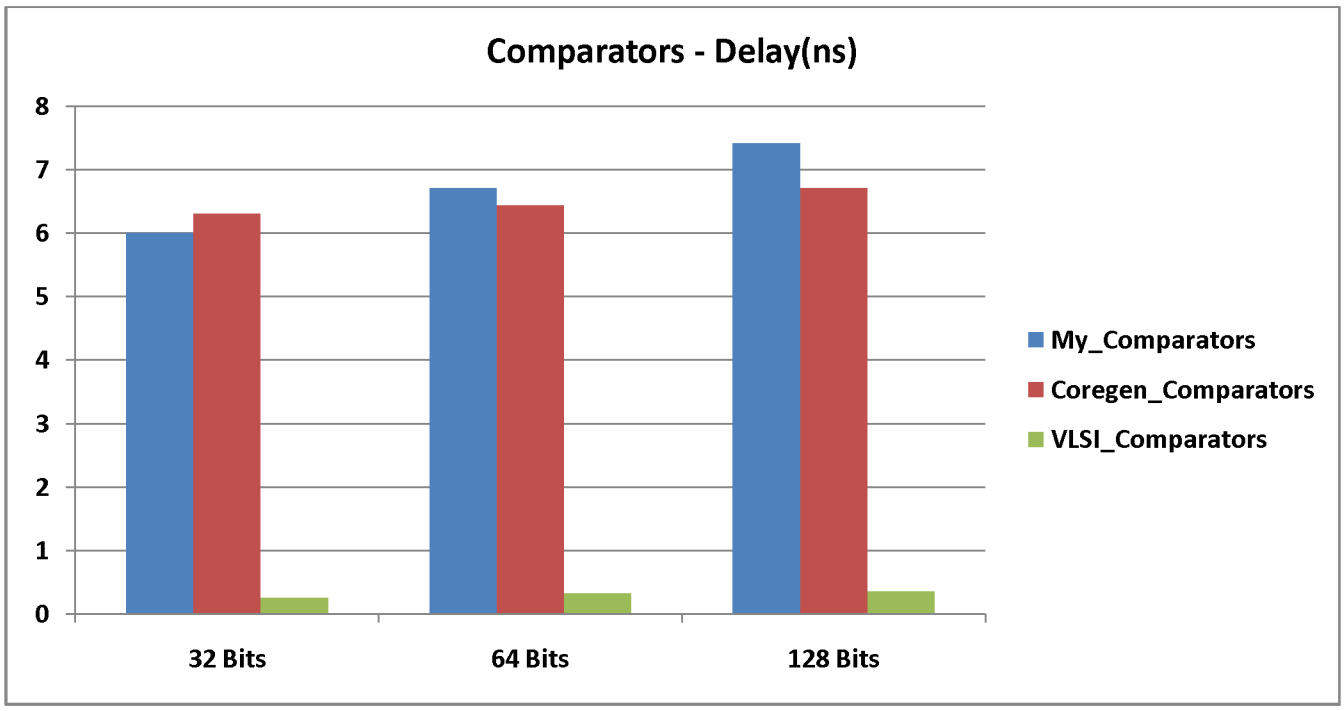


Diagram 4 - 3: Delay (in ns) of the three different comparator implementations according to the number of bits.

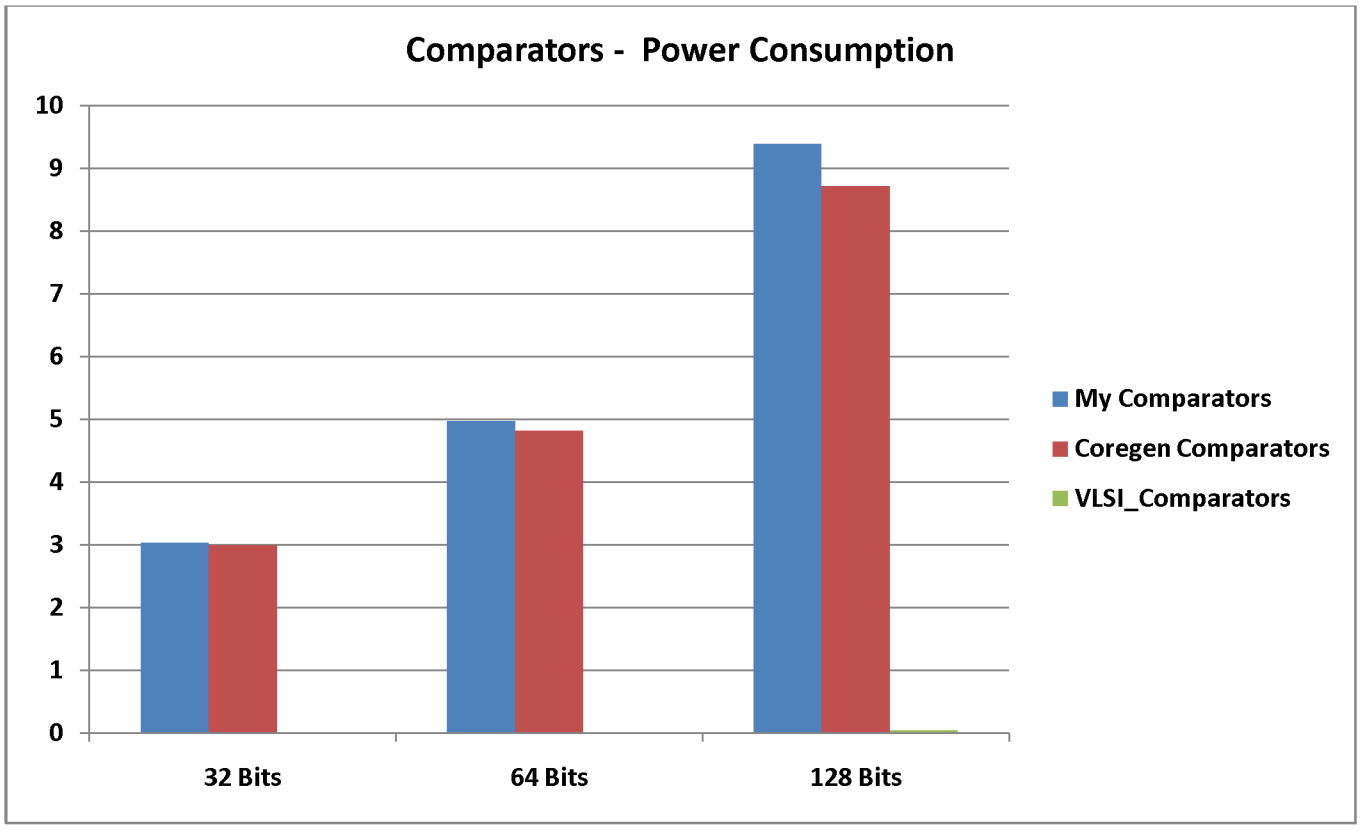


Diagram 4 - 4: Power consumption (in mW) of the three different comparator implementations according to the number of bits.

As we can see in diagram the 3 comparators we designed almost have the same delay compared to those we produced from Xilinx Core Generator. With the use of EDA tools we can achieve very low delay values, almost 20 times lower than these of the Core Generator Comparators. FPGAs use look-up tables, which are relatively slower than ASIC gates. Recent research by Zuchowski showed that the delay of an FPGA LUT was approximately 12 to 14 times the delay of an ASIC gate. This ratio has remained relatively constant across CMOS process generations from 0.25µm to 90nm. In all three implementation we can see that the power consumption grows linearly with the increase of the number of bits. The power consumption is again lower in the designs we made using EDA tools.This gap in power consumption varies between 180-300x.

Shift Registers

	FPGA		VLSI	
	Frequency(MHz)	**Power(mW)**	**Frequency(MHz)**	**Power(mW)**
Shift Register 32 Bits	353.107	10	910	0.5107
Shift Register 64 Bits	353.107	11	833	0.923
Shift Register 1024 Bits	353.107	47	680	8.366
Shift Register 4096 Bits	353.107	159	666	26.726

Table 4 - 3: Frequencies and power consumption of all shift registers.

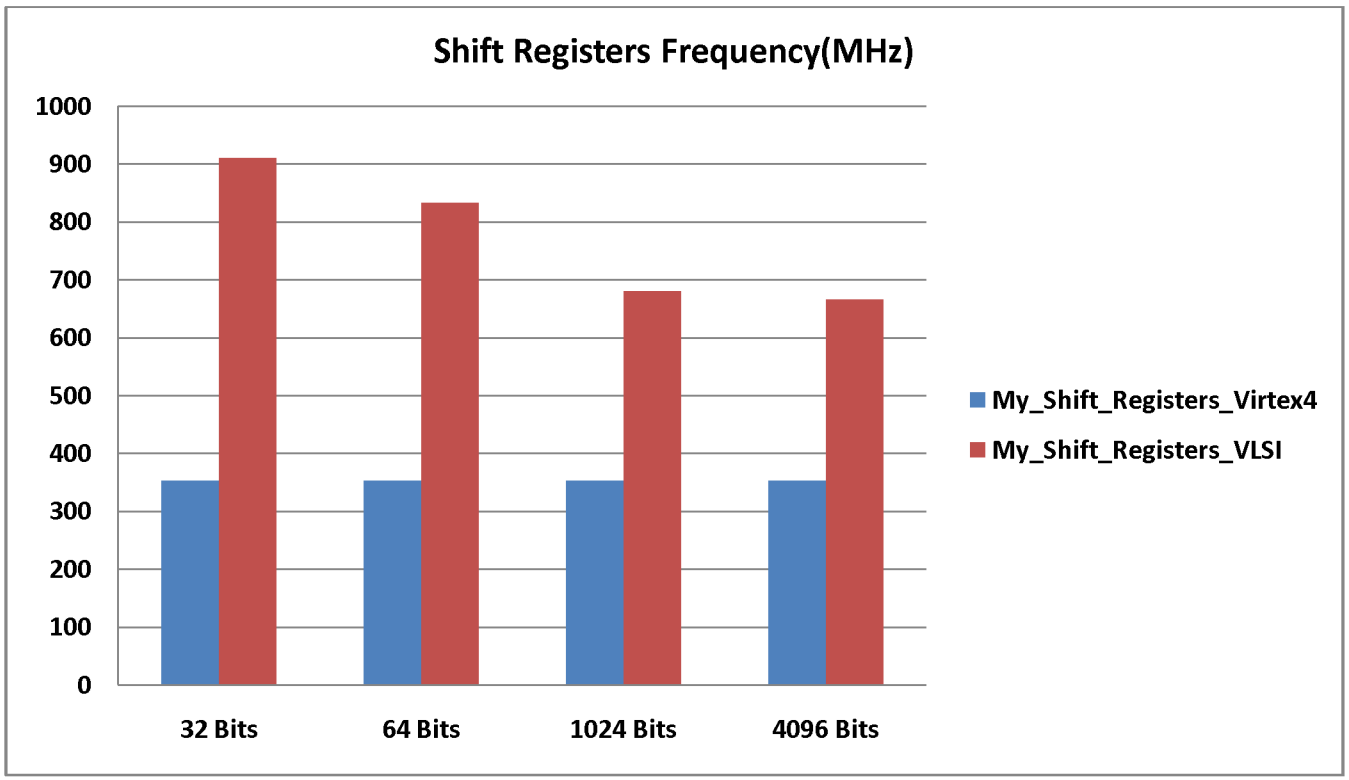


Diagram 4 - 5: Frequency (in MHz) of the two different shift register implementations according to the number of bits.

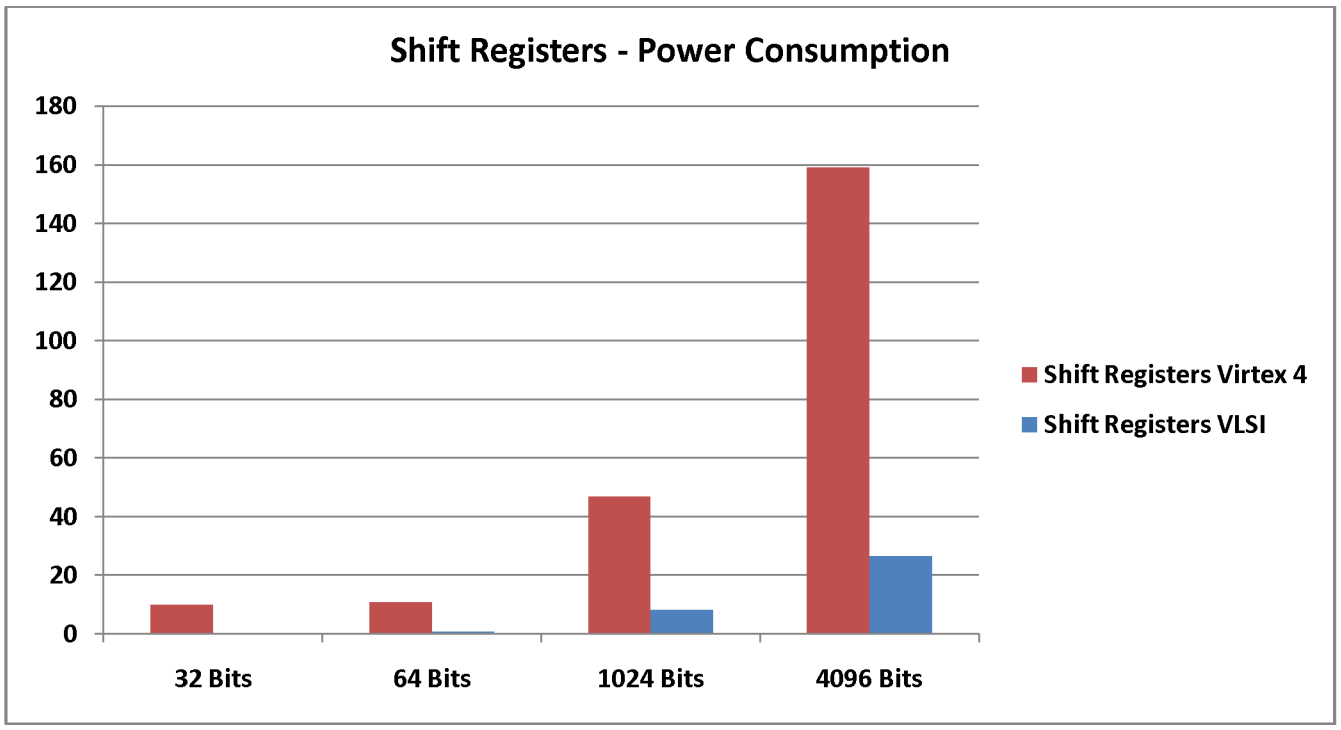


Diagram 4 - 6: Power consumption (in mW) of the two different shift register implementations according to the number of bits.

From the first diagram we can see that the frequency of the 4 shift registers implemented in Virtex4 stays the same. This is because the lookup tables in the FPGA are ordered in rows. Each row of LUT's follows the previous. As a result the critical path is the same even if the number of bits changes. In general the VLSI implementations have 1.8-2.5x higher frequency than the respective implementation in Virtex 4.

We can see from diagram that there is a linear growth in power consumption of both VLSI and FPGA designs when the number of bits becomes larger. Once again the shift registers we designed consume more power in Virtex-4 than in VLSI. The gap between them ranges between 5-20 times.

CHAPTER 5: CONCLUSION AND FUTURE WORK

Conclusion

Finally, the conclusions about the implementation of basic structures, that are common in bioinformatics algorithms, along with some ideas about future research concerning this subject, are presented.

As we saw before, ASIC implementations of sequential circuits are 1-4 times faster than the FPGA implementations and also consume a lot less power. However the designs we can produce using the core generator are a lot faster. The cores that are available in public are not optimized for standard cell implementation. As a result the designs we produce using these VHDL or VERILOG codes have low density, which results in slower and more power consuming circuits. Also, the cost of a standard cell implementation is very high compared to the cost of a Virtex4 FPGA. Only in the case of the 3 combinational circuits, their ASIC implementation was faster and more power efficient than those generated from Xilinx. Without writing optimized HDL structural codes the best alternative is to use the Xilinx's cores.

Future Work

There are many research proposals relative to this subject. Some of them are the following:

- Implementation of structural HDL code for these cores, standard cell design, and later comparison with cores produced from the core generator.
- Using the cores produced from the core generator and reverse engineering techniques we can get their VHDL codes and then using the EDA tools we can produce better standard cell designs.
- Design of these basic cores in full custom and later comparison with cores produced from the core generator.
- Design of the bioinformatics algorithms we mentioned above, using standard cell ASIC methodology.
- Design of the bioinformatics algorithms we mentioned above, in full-custom.

List of Tables And Figures

Bibliography

Cadence Design Systems. (2007). Encounter® Command Reference.

Cadence Design Systems. (2007). Encounter® Menu Reference.

Cadence Design Systems. (2007). RTL Compiler® Command Reference.

Dollas, A., & Sotiriades, E. (2007). A General Reconfigurable Architecture for the BLAST Algorithm. Department of Electronic and Computer Engineering, Technical University of Crete, Chania.

Faraday, Technology Corporation. (2006, September). 90 nm Technology Standard Cell Library.

Faraday, Technology Corporation. (2006). FSD0A_A 90 nm Logic SP-RVT (Low-K) Process.

Gyvez, J. P. (2008). Cadence Encounter Manual. Faculty of Electrical Engineering Eindhoven University of Technology.

Gyvez, J. P. (2008). Cadence RTL Compiler Manual. Faculty of Electrical Engineering Eindhoven University of Technology.

Kuon, I., & Rose, J. (2007). Measuring the Gap Between FPGAs and ASICs.

Lesk, A. M. (2002). *Introduction to Bioinformatics.*

Salzberg, S. L., Delcher, A. L., Kasif, S., & White, O. (1998). *Microbial gene identification using interpolated Markov.*

Setubal, J., & Lesk, A. (1997). *Introduction to Computational Molecular Biology.*

Shindyalov, I. N., & Bourne, P. E. (2001). A Database And Tools For 3-D Protein Structure Comparison And Alignment Using The Combinatorial Extension(CE) Algorithm.

Shindyalov, I. N., & Bourne, P. E. (1998). Protein Structure Alignment By Incremental Combinatorial Extension(CE) Of The Optimal Path.

Smith, T., & Waterman, M. (1981). *Identification of Common Molecular Subsequences.*

Smith-Waterman algorithm. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Smith-Waterman_algorithm

Sotiriades, E., Chrysos, G., Dollas, A., Pnevmatikatos, D., Papaefstathiou, I., Mplemenos, G.-G., et al. (2008). Special purpose processors for performance boosting of Bioinformatics Algorithms. Department of Electronic and Computer Engineering, Technical University of Crete, Chania.

Sotiriou, C. P. (2008). *SoC Encounter How To.* Retrieved from Department Of Computer Science University Of Crete: http://www.csd.uoc.gr/~hy523/socencounterhowto_2008.html

Stamatakis, A., & Ott, M. (2008). Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures.

Xilinx. (2007). Virtex-4 Family Overview.