**CHAPTER-1**

**INTRODUCTION**

**INTRODUCTION**

A field-programmable gate array (FPGA) is an [integrated circuit](https://en.wikipedia.org/wiki/Integrated_circuit) designed to be configured by a customer or a designer after manufacturing – hence the term "[field programmable](https://en.wikipedia.org/wiki/Field-programmability)". The FPGA configuration is generally specified using a [hardware description language](https://en.wikipedia.org/wiki/Hardware_description_language)(HDL), similar to that used for an [Application-Specific Integrated Circuit](https://en.wikipedia.org/wiki/Application-Specific_Integrated_Circuit) (ASIC). [Circuit diagrams](https://en.wikipedia.org/wiki/Circuit_diagram) were previously used to specify the configuration, but this is increasingly rare due to the advent of [electronic design automation](https://en.wikipedia.org/wiki/Electronic_design_automation) tools.

FPGAs contain an array of [programmable](https://en.wikipedia.org/wiki/Programmable_logic_device) [logic blocks](https://en.wikipedia.org/wiki/Logic_block), and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. [Logic blocks](https://en.wikipedia.org/wiki/Logic_block) can be configured to perform complex [combinational functions](https://en.wikipedia.org/wiki/Combinational_logic), or merely simple [logic gates](https://en.wikipedia.org/wiki/Logic_gate) like [AND](https://en.wikipedia.org/wiki/AND_gate) and [XOR](https://en.wikipedia.org/wiki/XOR_gate). In most FPGAs, logic blocks also include [memory elements](https://en.wikipedia.org/wiki/Memory_cell_(computing)), which may be simple [flip-flops](https://en.wikipedia.org/wiki/Flip-flop_(electronics)) or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different [logic functions](https://en.wikipedia.org/wiki/Boolean_function), allowing flexible [reconfigurable computing](https://en.wikipedia.org/wiki/Reconfigurable_computing) as performed in [computer software](https://en.wikipedia.org/wiki/Software).

Field Programmable Gate Arrays (FPGAs) are part of current reconfigurable computing technology, which in some ways represent an ideal alternative for image and video processing. FPGAs generally consist of a system of logic blocks, such as look up tables, gates, or flip-flops, just to mention a few, and some amount of memory, all wired together using a vast array of interconnects. All of the logic in an FPGA can be rewired, or reconfigured, with a different design, according to the designer needs.

FPGAs generally consist of a system of logic blocks (usually look up tables and flip-flops) and some amount of Random-Access Memory (RAM), all wired together using a vast array of interconnects. This type of architecture allows a large variety of logic designs for a number of real time applications.

1

This project system based on the Quartus FPGA STRATIX- II has been used to implement High speed constant multiplication which in turn find its application in DSP filters and real time applications.

The terms used in implementing high speed multiplication involves RPAG, VHDL, FIR and reconfigurable multipliers. In this thesis optimization methods to implement run-time reconfigurable constant multipliers (RCMs) on field programmable gate arrays (FPGAs) are proposed. The performance, hardware effort, reconfiguration time and power consumption of resulting circuits are evaluated. The resulting solutions add some important trade-off points to the design space of RCM on PGAs and make new applications possible.

Multiplication with constants is one of the most frequent operations in digital signal processing (DSP). At the same time, FPGAs have a growing market in DSP applications which were formerly dominated by application specific integrated circuit (ASIC) implementations.

The re-programmability of FPGAs and increasing ASIC manufacturing costs. The costs of the re-programmability of FPGAs are that FPGA designs are typically larger, slower and consume more power than an equivalent ASIC realization. Therefore, optimized implementations of DSP algorithms for FPGAs are getting more and more important. This is one of the reasons why embedded multipliers are present in the fabric of FPGAs. Nevertheless, the drawback of those fixed coarse-grained blocks is their inflexibility in word size and their limited quantity. Limited quantity is particularly critical in industrial applications when low-cost FPGAs with only few embedded multipliers have to be chosen and other parts of a design are competing for DSP resources.

2

The alternative logic-based methods for constant multiplication are requiredwhich are independent of this embedded special purpose hardware but are, on the other hand, efficient enough to narrow the gap to an ASIC realization. Therefore, optimizing the implementation of constant multiplication as shift-add-based circuit is well studied.

The switching between a given limited set of constant multiplications during run-time instead of using larger generic multipliers is important, too. Reconfigurable constant multipliers are used to realize hardware efficient run-time adaptable filters, e.g., for adaptive control and video coding applications. Specifically, in an application with tight reconfiguration time and resource constraints is presented, which motivates the necessity of highly optimized RCMs on FPGAs. An FPGA is used as co-processor in the control loop accelerator.

The Multiplication with constant coefficients is an essential operation in digital signal processing. Initially one of the reasons to put embedded multipliers or DSP blocks into the fabric of field-programmable gate arrays (FPGAs) was to reduce the performance gap between application specific integrated circuits (ASICs) and FPGAs. Nevertheless, the price to pay for those fixed coarse-grained blocks is their inflexibility in word size and limited quantity. Limited quantity is particularly critical in industrial applications, when cheaper and rather small FPGAs with only few DSP blocks have to be chosen due to price pressure. Thus, logic-based constant multiplication methods are needed. Optimizing the implementation of this operation is well studied. Switching between a given set of constants of such multipliers durin grun-time instead of using larger generic multipliers is important to realize hardware efficient run-time adaptable filters; discrete cosine transformation and fast Fourier transform implementations as well as multistage filters for decimation or interpolation like polyphase finite-impulse response (FIR) filters.

The full form of RPAG is reduced pipeline adder graph. Direct optimisation of pipelined adder graphs is done for high speed multiple constant multiplication. A novel algorithm to solve the pipelined multiple constant multiplication problem is heuristically called as RPAG. It fuses already optimised pipelined constant multipliers.

3

In digital signal processing (DSP) systems finite impulse response (FIR) filters have very much importance since their characteristics in linear-phase and feed-forward implementations make them very useful for building stable high-performance filter architectures. Filters are usually frequency selective networks, which is capable of modifying an input signal in order to facilitate further processing. Thus, digital filters are more preferred to analog due to its high signal integrity.

A FIR filters are mainly applied for different DSP areas because providing virtues of linear phase and system stability. Moreover, the filters with constant coefficients are possible in fir filters.The multiplication with constant coefficients is an essential operation in DSP. Initially one of the reasons to put DSP blocks into FPGA’S was to reduce performance gap between ASICs and FPGAs. Reconfigurability means reusability.

By reconfigurable constant multipliers resource savings up to 75 percent can be done compared to normal LUTS. A lookup table is an array that replaces runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an "expensive" computation or input/output operation.

**1.1 Objective of the project:**

The major theme of this project is to implement a reconfigurable multiplier based on the Look up tables which saves a lot of resources. By using the proposed method, it reduces the run-time of the fusion process, which raises the usability and application domain of the proposed method of run-time reconfiguration. It introduces a new approach to generate pipelined run-time reconfigurable constant multipliers for field programmable gate arrays (FPGAs). It produces results close to the optimum. A lookup table is an array that replaces runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an "expensive" computation or input/output operation.

4

**1.2 Organisation of the thesis:**

This section gives an overview on the organization of the thesis. Moreover, the main contributions of each chapter are provided. The background for this thesis gives an introduction to FPGAs as target technology and introduces the different run-time reconfiguration concepts in detail. In addition to that, the background on multiplier-less constant

Multiplication is provided with a focus on FPGA-specific aspects at the end of the chapter. Logic reconfiguration using reconfigurable LUTs. It is shown how configurable constant multiplication based on LUTs can provide hardware optimized RCM solutions, which outperform generic IP core multipliers and non reconfigurable constant multipliers reconfigured using Partial Reconfiguration. The contributions of this chapter are a new method to generate LUT-based run-time reconfigurable constant multipliers based on Ken Chapman multipliers (KCMs) and two new architectures for LUT-based run-time reconfigurable FIR filters. In addition to a KCM-based FIR filter implementation, an FIR filter architecture and its automatic generation is presented. An algorithm to generate reconfigurable constant multipliers using multiplexers is presented. Previous methods on reconfigurable multiplier-less constant multiplication did not consider costs for registers in the combinatorial path of the circuit, which can be used to split up the combinatorial parts during optimization. However, this is important to realize fast FPGA implementations.

**5**

**CHAPTER-2**

**LITERATURE REVIEW**

**LITERATURE REVIEW**

Multiplication is a computationally intensive problem, especially the design and efficient Implementation on an FPGA where resources are very limited has been more demanding. FPGA based designs are usually evaluated using three performance metrics: speed (latency), area, and power (energy). Fixed point implementations in FPGA are fast and have minimal power consumption. Additionally, a fixed point matrix multiplier unit often requires less silicon real estate in an FPGA or ASIC than its floating-point counterpart. The limitation of fixed point number is that very large and very small numbers cannot be represented and the range is limited to bit-width of the number. There has been extensive previous work in the area of designing an FPGA based system for the computation of fixed point multiplier, a design methodology for synthesizing a family of very compact systolic arrays on FPGA based essentially upon manual mapping at CLB level coupled with VHDL structural-level is discussed. The authorsused matrix multiplication as the benchmark to compare the performance of FPGAs, DSPs and embedded processors. The results show that the FPGAs can multiply two matrices with both lower latency and lower energy consumption than the other two types of devices. This makes FPGA ideal choice for multiplication in signal processing applications.

The multiplication with constant coefficients is an essential operation in digital signal processing. Initially one of the reasons to put embedded multipliers or DSP blocks into the fabric of field-programmable gate arrays (FPGAs) was to reduce the performance between application specific integrated circuits (ASICs) and FPGAs. Nevertheless, the price to pay

for those fixed coarse-grained blocks is their inflexibility in word size and limited quantity.Limited quantity is particularlycritical in industrial applications, when cheaper and rather small FPGAs with only few DSP blocks have to be chosen due to price pressure. Thus, logic-based constant multiplication methods are needed.

6

**CHAPTER 3**

**RECONFIGURABLE CONSTANT MULTIPLIER**

**RECONFIGURABLE CONSTANT MULTIPLIER**

It is based on an optimal algorithm which fuses already optimized pipelined constant multipliers generated by an existing method called reduced pipelined adder graph (RPAG).The results of RPAG are adder graphs representing multiplier-less pipelined constant multipliers using additions, subtractions, and bit-shifts only.The main idea of multiplier-less multiplication as applied in RPAG is to compose a constant multiplication of an addition of shifted inputs.

**3.1 Implementation of PAG Fusion Algorithm**

**3.1.1 Pipelined Adder Graphs:**

The input to the algorithm is PAGs generated with the RPAG heuristic. In general, the presented fusion is not limited to RPAG generated circuits as pipelined MCM input. RPAG is backward-exploring reachable intermediate constants, called predecessors by evaluating the *A*-operation. This leads to a step-wise constant composition, starting with the required output constants. The goal of the heuristic is to select predecessors which result in the lowest number of intermediate constants in the preceding stage and which reduce the adder depth.

**3.1.2 Improved Pipelined Adder Graph Fusion:**

This selection will be the source for the determination of the next preceding stage *s*−1. The procedure is repeated until the input (stage 0) is reached. It is started with the constant mapping *M* of the output stage, the preceding stage *s*, the search width *w* (unlimited for the optimal search) and the costs of the current path current cost, which is zero in the beginning.

7

**3.1.3 Complexity Consideration**

The presented optimal PAG fusion algorithm has to traverse the full search space, its complexity, i.e., the number of possible solutions and branches to find them is an important issue.

**3.1.4 Routing Reconfiguration using Logic Multiplexers on FPGAs**

The programmable routing of an FPGA is fixed during run-time. However, multiplexers. Realized with BLEs can be used to change signal routing during run-time. This is especially interesting in the context of switching between the different circuit alternatives for resource sharing like, e. g., in multiplexer-based run-time reconfigurable constant multipliers (RCMs), A Virtex 6 slice consists of four 6-input LUTs, which can be used as any 6-input logic function . Hence, each LUT can be used as an up to 4:1 1-bit multiplexer.

**3.1.5 Power Consumption**

The power consumption for FPGAs can be estimated using the vendor tools or measured using high precision amplifiers and an oscilloscope. Estimation is done by summing up the power of used resources based on their switching activity.

8

**3.2 IMPLEMENTATION OF HIGH-SPEED CONSTANT MULTIPLICATION**

**3.2.1 Multiplexer Mapping**

As multiplexers are used to switch between the different constants, their mapping to the target FPGA should be as good as possible. This can be achieved by using the explicit resources provided in Quartus Vertex 5-7 slices. In the case of the used Vertex 6 FPGA, our VHDL code generator produces the optimal mapping using Primitives and methods described. This results in a resource optimized multiplexer implementation.

**3.2.2 Switchable Adder Subtractor Mapping**

The fusion of adders with Subtractor leads to the requirement of switchable adder/sub tractors in which the input that is subtracted can be either input *a* or input *b*. The proposed realization of the switchable adder/Subtractor on Xilinx Vertex 5-7 slices can be found. The realization is done using a single LUT to provide the correct carry input and the following LUTs to provide an XOR of the inverted or non inverted inputs, which builds a full adder together with the slice’s carry logic.

**3.2.3 Constant Multiplication on Integrated Circuits**

After this FPGA-specific introduction to run-time reconfiguration, this section provides the background on multiplier-less constant multiplication. Multiplication with constants is an essential arithmetic operation and used in nearly any DSP algorithm. The implementation of this operation on integrated circuits (ICs) is thus a well studied research topic. Instead of using generic multipliers, constant multiplication is implemented multiplier-less using additions, subtractions and bit shifts. This is advantageous as bit shifts can be realized as wires on ICs and special properties in the constant’s number representation can be individually exploited.

9

**3.2.4 Multiple Constant Multiplications**

In the last paragraph, the multiplication with a single constant was considered. In Many applications like, e. g., digital filters the multiplication of an input by several constants, called multiple constant multiplications (MCM), is an important operation. In multiplier-less MCM circuits using addition and bit shifts, intermediate results can be reused among several output constants’ adder graphs. This reduces the required hardware resources and computation effort compared to several separate single constant multiplication (SCM) implementations.

**3.2.5 Implementation of Constant Multiplication on FPGAs**

This section presents two different methods to efficiently implement constant multiplication on FPGAs. First, the already mentioned RPAG algorithm to generate pipelined SCM, MCM and constant matrix multiplication (CMM) circuits is presented. Moreover, LUT-based constant multiplication is introduced as alternative to addition and bit-shift-based multiplier-less constant multiplication.

**3.2.6 Reduced Pipelined Adder Graph Algorithm**

The results of the RPAG algorithm introduced before are adder graphs representing Pipelined constant multipliers using additions and bit shifts only. Previous approaches for Pipelined MCM applied pipelining to existing MCM solutions. However, in the RPAG algorithm pipelining is considered already during the MCM adder graph generation.

**3.2.7 Look-Up Table Based Constant Multiplication**

So far, the focus of the introduction of multiplier-less constant multiplication has been on implementations using additions and bit shifts. Alternatively, constant multiplication can be performed by dividing the multiplication into partial products. These partial products are then realized with LUTs or block RAMs.

10

**3.2.8 Reconfigurable Constant Multiplication using LUTs**

Here, a reconfigurable constant multiplication method using look -up tables (LUTs) is presented. It is based on constant multiplication using LUTs described by Chapman. Reconfiguration is achieved by changing the LUT contents of partial products during run-time. After a short introduction of related work, the architecture of the reconfigurable LUT multiplier is presented and compared to a generic multiplier implementation and constant multipliers reconfigured using Partial Reconfiguration (PR) and the Internal Configuration Access Port (ICAP) filter architectures.

**3.2.9 Reconfiguration Time**

The fastest reconfiguration can be provided by the generic multiplier architecture, whose coefficient can be changed within one clock cycle. For the presented reconfigurable KCM-based LUT multiplier design 32 reconfiguration clock cycles are required until the output of the multiplier is valid again (90*.*65 ns for the slowest design).

11

**CHAPTER 4**

**PROJECT FLOW**

**PROJECT FLOW**

The design flow of Reduced Pipelined Adder Graph is shown below figure. By using the proposed method, it reduces the run-time of the fusion process, which raises the usability and application domain of the proposed method of run-time reconfiguration. It introduces a new approach to generate pipelined run-time reconfigurable constant multipliers for field programmable gate arrays (FPGAs). It produces results close to the optimum. It is based on an optimal algorithm which fuses already optimized pipelined constant multipliers generated by an existing heuristic called reduced pipelined adder graph (RPAG).

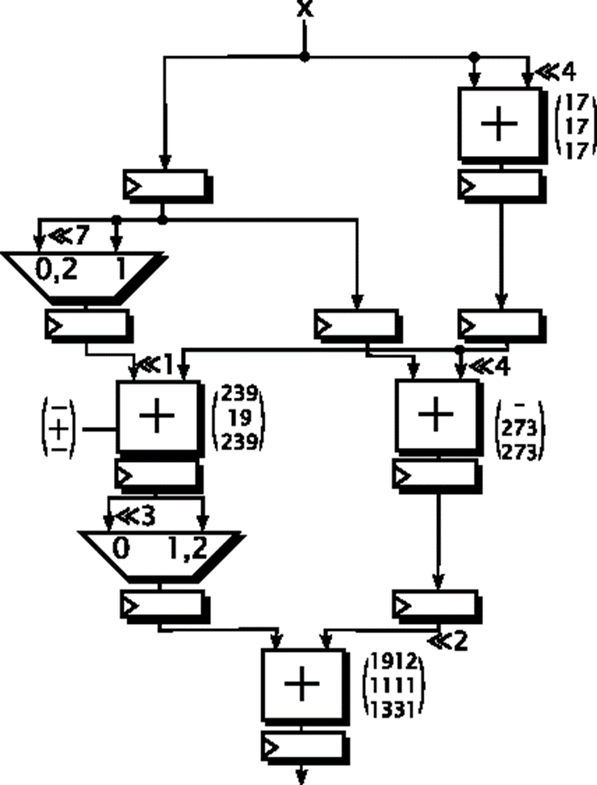
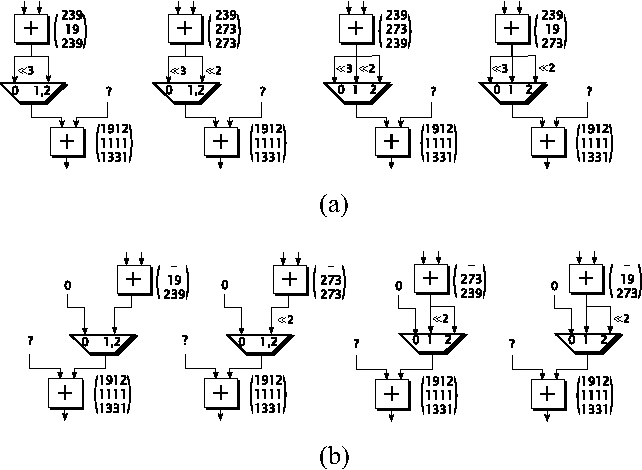


Fig 4.1 RPAG ALGORITHM

12

Fig 4.2: All combinations of the adders in the second last stage for the given output mapping and the given RPAG SCM solutions (a) first preceding adder and (b) second preceding adder.

**Pipelined Adder Graphs:**

The input to the algorithm is PAGs generated with the RPAG heuristic. In general, the presented fusion is not limited to RPAG generated circuits as pipelined MCM input. RPAG is backward-exploring reachable intermediate constants, called predecessors by evaluating the A-operation. This leads to a step-wise constant composition, starting with the required output constants. The goal of the heuristic is to select predecessors which result in the lowest number of intermediate constants in the preceding stage and which reduce the adder depth. Two more examples for such a circuit of a pipelined SCM realization can be found in Fig. 5.2 which are used as running example. The stage *s* denotes the pipeline depth of each. The inputs to the algorithm are PAGs generated with the RPAG heuristic.

13

The inputs to the algorithm are PAGs generated with the RPAG heuristic. In general, the presented fusion is not limited to RPAG generated circuits as pipelined MCM input. However, RPAG was chosen as it proved to outperform state-of-the-art MCM methods when these are optimally pipelined. The results of RPAG are adder graphs representing multiplier-less pipelined constant multipliers using additions, subtractions, and bit-shifts only. The main idea of multiplier-less multiplication as applied in RPAG is to compose a constant multiplication of an addition of shifted inputs. This is beneficial because a constant shift is only a wire in hardware. All constants can be formally represented as a operation is performed. A multiplication by 17 could, for example, be realized as an addition of which is defined as Aq(u, v) = |2l1u + (−1)sg2l2v|2−r (1) with q = (l1, l2, r, sg), where *u* and *v* are the input constants,*l*1, *l*2, and rare shift factors and the sign bit sg ∈ {0*,* 1} denotes Whether an addition or subtraction the input with the input left-shifted by 4 (multiplication by 16). This can be seen in the leftmost example in Fig. 2. In the following subtraction, 17 times the input is subtracted from 256 times the input, which corresponds to a constant multiplication by 239. Finally, this intermediate result is left shifted by three to get the final result of 1912 times the input.

If the constant to multiply with is known in advance, this kind of realization is much cheaper in terms of resources than implementing a generic multiplier. In order to automatically generate such constant multipliers, RPAG is backward-exploring reachable intermediate constants, called predecessors by evaluating the *A*-operation. This leads to a step-wise constant composition, starting with th required output constants. The goal of the heuristic is to select predecessors which result in the lowest number of intermediate constants in the preceding stage and which reduce the adder depth. Two more examples for such a circuit of a pipelined SCM realization can be found in Fig. 2, which are used as running example. The stage *s* denotes the pipeline depth of each.

14

**Improved Pipeline Adder Graph Fusion:**

Just like RPAG, the proposed PAG fusion is backward exploring. Starting with the constant mapping of the output stage all PAGs are fused stage by stage. The basic idea is to combine those intermediate values in the respective preceding stage to share the same adder, which leads to a minimal overhead of possibly necessary multiplexers or switchable adder/subtractors. To do so, all combinations of intermediate values are evaluated and their costs are calculated separately and stored in a cost matrix. Multiplexers can appear at the inputs of the successive stage in the following cases.

1) Input has a different shift value.

2) Input has a different source.

As described before, the target is to select the overall best mapping *M* for the specific stage *s*. This selection will be the source for the determination of the next preceding stage *s*−1. The procedure is repeated until the input (stage 0) is reached. A simplified pseudo-code of the generalized fusion process is given in Listing 1. It assumes that the overall best solution and costs are globally known. It is started with the constant mapping *M* of the output stage, the preceding stage *s*, the search width *w* (unlimited for the optimal search) and the costs of the current path current cost, which is zero in the beginning. Compared to the algorithm presented in [20] the algorithm was generalized, such that it can be used both as heuristic and in an optimal way. In contrast to an arbitrary search through the whole search space, which was done in the former version, the search is now improved and based on a sorted cost matrix. More details on the search width are provided in Section II-D. In the running example used here, the three SCM graphs generated with RPAG (see Fig. 2) are fused starting with the desired output mapping *M* = {1912; 1111; 1331}, meaning that the resulting circuit can be switched between these three values. This will be called an SCM circuit with three configurationsin the following.

15

The enumeration of all adder combinations of the second last stage consisting of {239} for the first, {19}, {273} for the second and {239}, {273} for the third SCM solution, respectively, is given in Fig. 3. For the constant 1912 only one adder is required in stage two, but two adders are required in the other SCM circuits. This fact is considered by the insertion of a don’t care “−.” Due to a separate cost calculation for a specific combination, some of the multiplexer inputs are unknown from a local point of view. These are marked with a question mark. They do not have any contribution to the currently considered adders’ multiplexer costs, which is a main advantage of the proposed method as the costs for each combination can be calculated and evaluated separately. The cost evaluation is following the assumption that the multiplexers will be realized as a cascade of 2:1 mux. Thus, *N* −1 2: 1 multiplexer are required to switch between *N* configurations, which leads to a contribution of each used multiplexer in cost MUX = N – 1/ N. As a zero input can be realized by resetting the succeeding register, these inputs are not considered as multiplexer inputs as our implementation targets pipelined implementations. multiplexer cost for each mapping is stored in a multidimensional cost matrix C. The cost matrix for the combinations of the current stage can be found in Table I in a 2-D representation. For example, the first entry in the first row corresponds to the leftmost mapping in Fig. 4(a) in which two multiplexer inputs, each with a cost MUX of 23 are used. To get a valid solution a selection of one mapping for the first preceding adder in Fig. 4.2(a) will directly force the selection of the corresponding mapping for the second adder in Fig. 4.2(b) or reduces the selectable possibilities for other adders in a more general case. This means each valid mapping solution for a specific stage consists of selections with a unique row and column index. Thus, finding the cheapest mapping solution *M* for a specific stage reduces to finding the valid solution with the lowest sum of costs.

16

**CHAPTER 5**

**RESULTS**

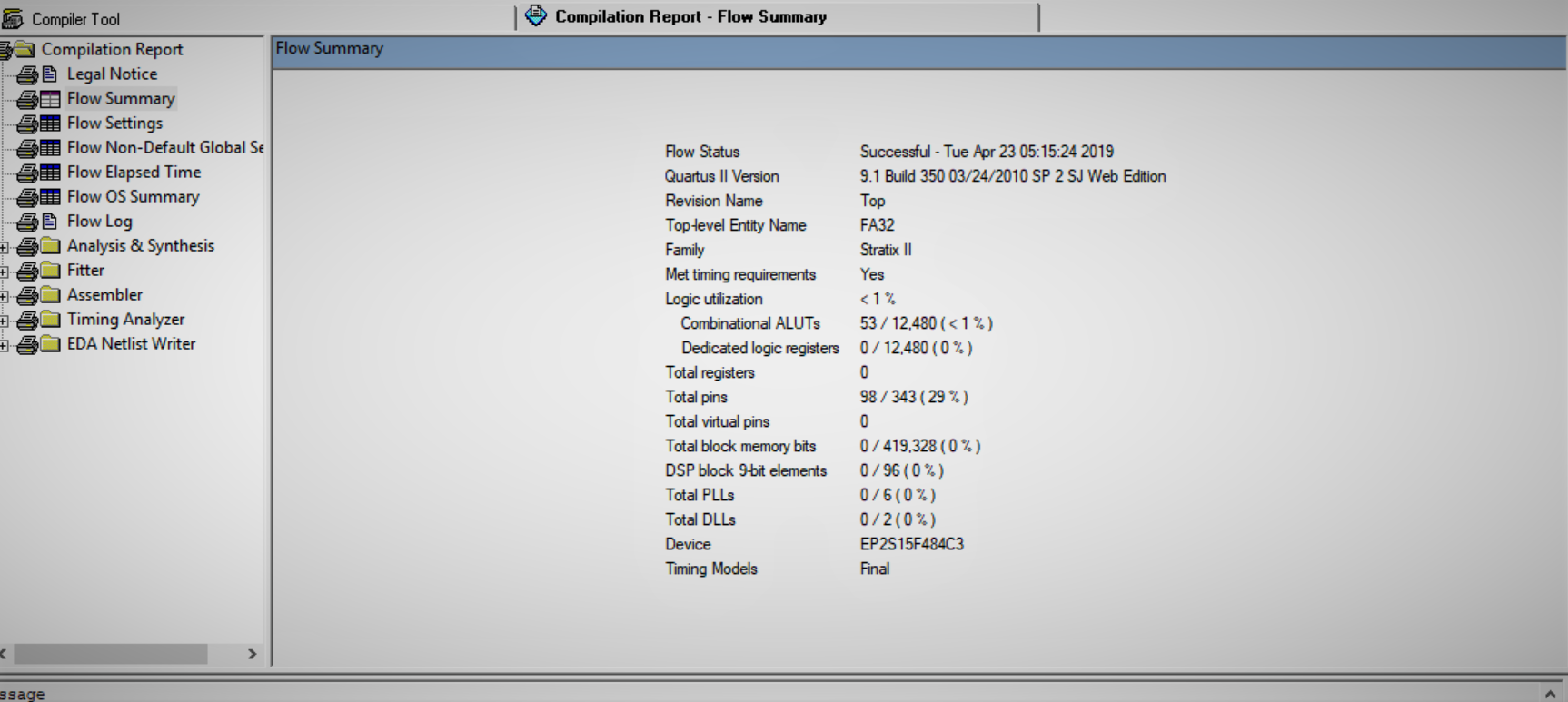
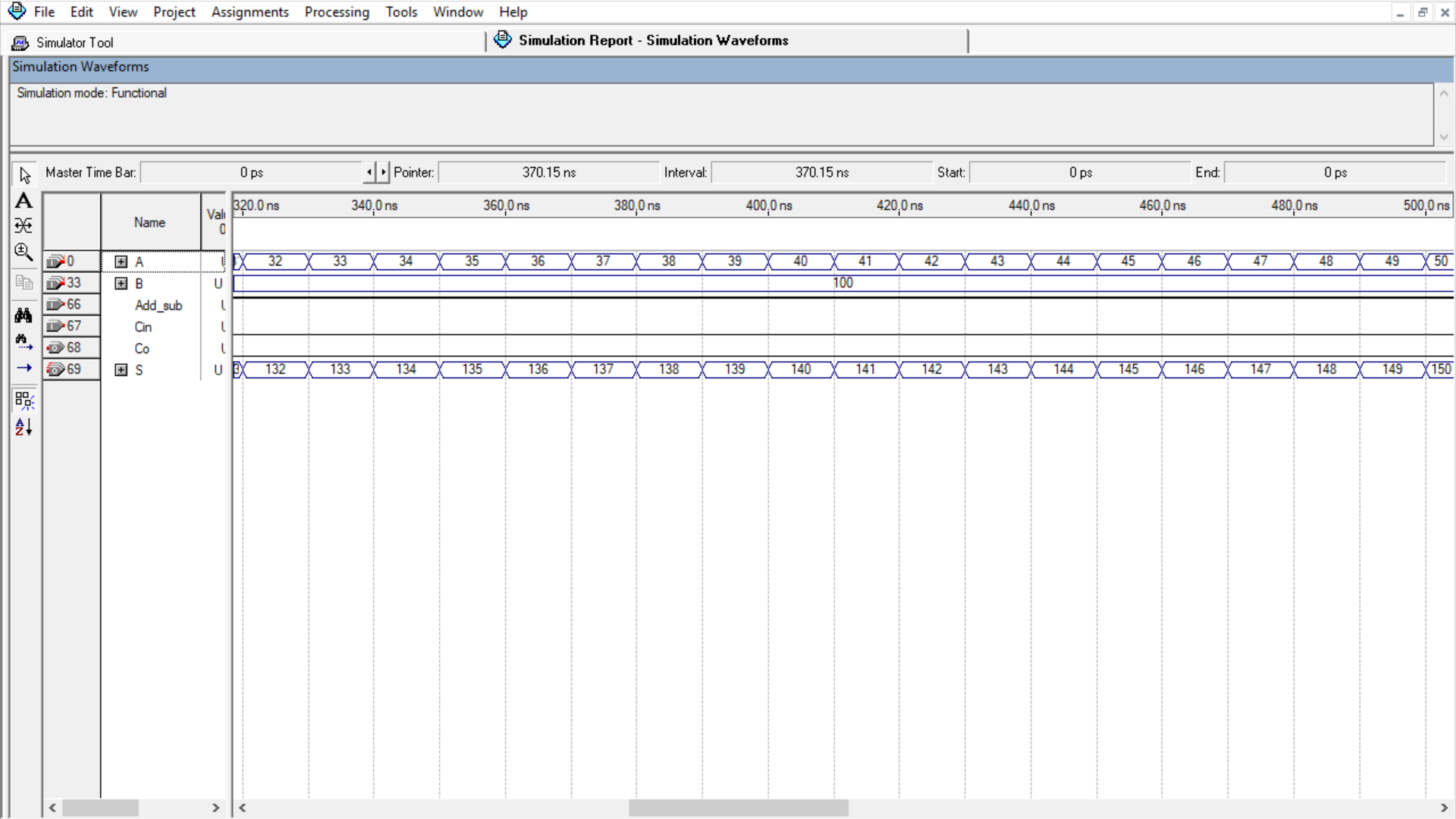
**RESULTS**

Fig 5.1: Compilation report of 32-bit Full adder

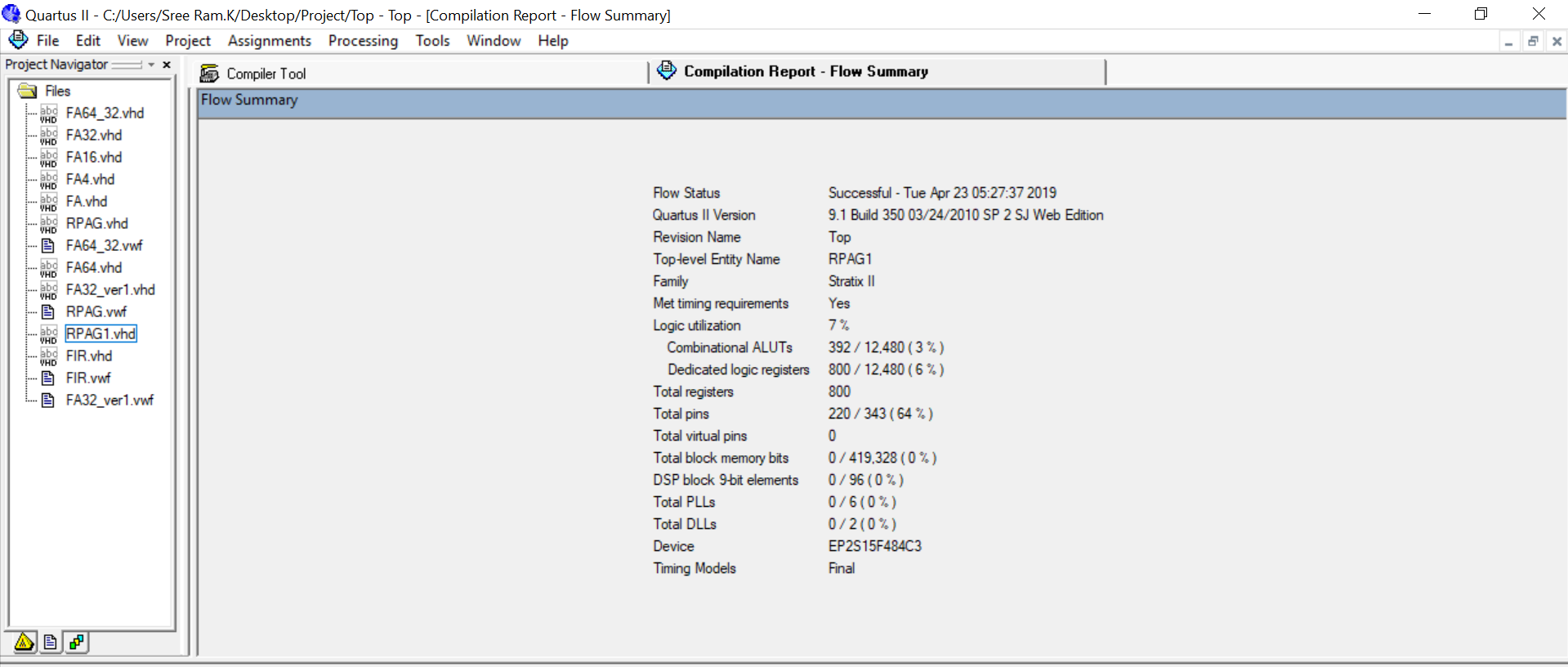
The compilation report of 32 -bit full adder is shown in above figure 5.1. Full adder is a combinational circuit. The delay of 32- bit full adder is 20.191 nano seconds.

17

Fig 5.2 Simulation report of 32-bit Full adder

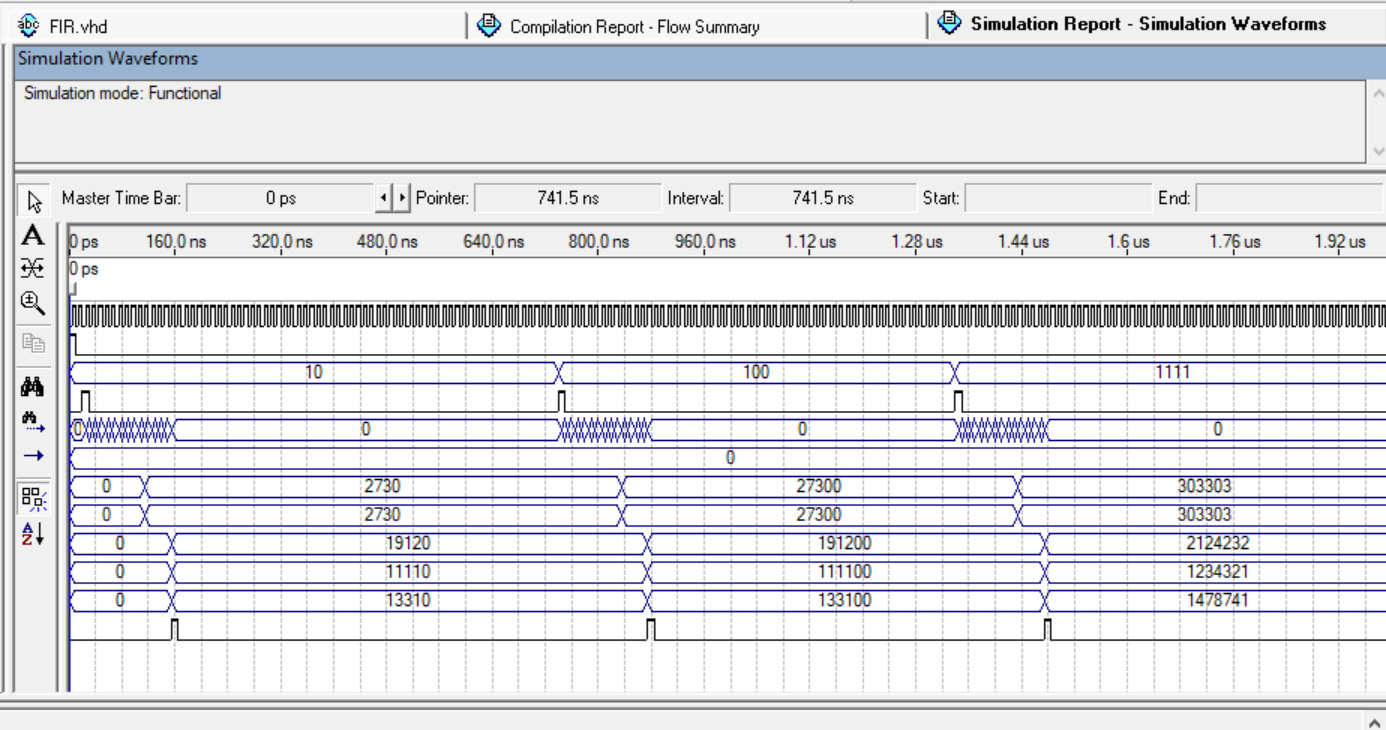
The simulation report of 32-bit full adder is shown in above figure 5.2. The inputs are A & B.A is incrementing in regular steps .Addition or Subtraction is indicated by bits 0 and 1.The Output is carry and sum.

18

Fig 5.3: Compilation Report of RPAG .

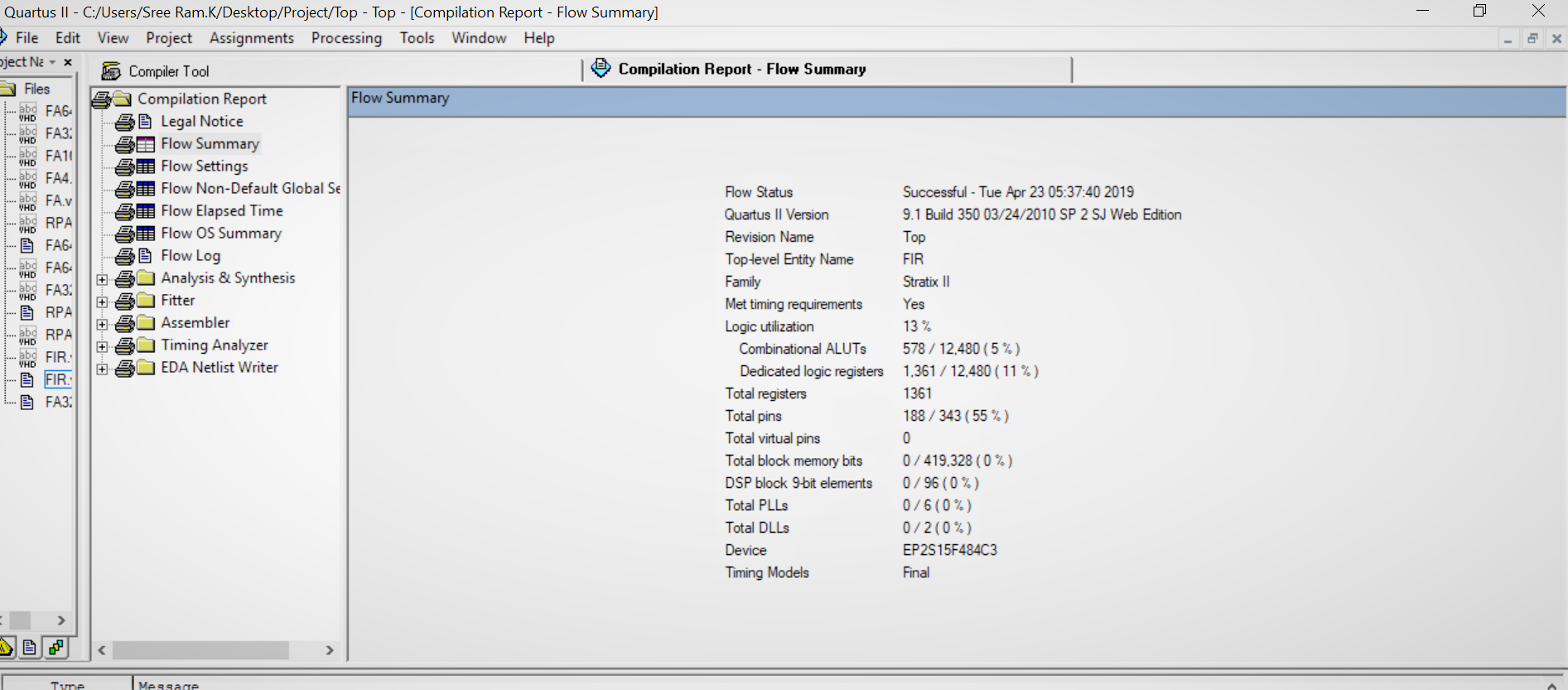
The compilation report of RPAG is shown above figure 5.3. Acronym for RPAG is reduced pipelined adder graph. It is optimized pipelined constant multipliers generated by an existing method. The frequency obtained in RPAG is 143.10MHz.

19

Fig 5.4: Simulation report for RPAG

The simulation report for RPAG is shown in above figure 5.4. The main idea of multiplier-less multiplication as applied in RPAG is to compose a constant multiplication of an addition of shifted inputs. The outputs are switched between constants 1912, 1111, 1331.

20

Fig 5.5: compilation report for FIR

The compilation report for FIR is shown above figure 5.5. Acronym for FIR is Finite impulse response. FIR filter has very much importance since their characteristics have stable high-performance. The frequency obtained for FIR is 131.61MHz.

21

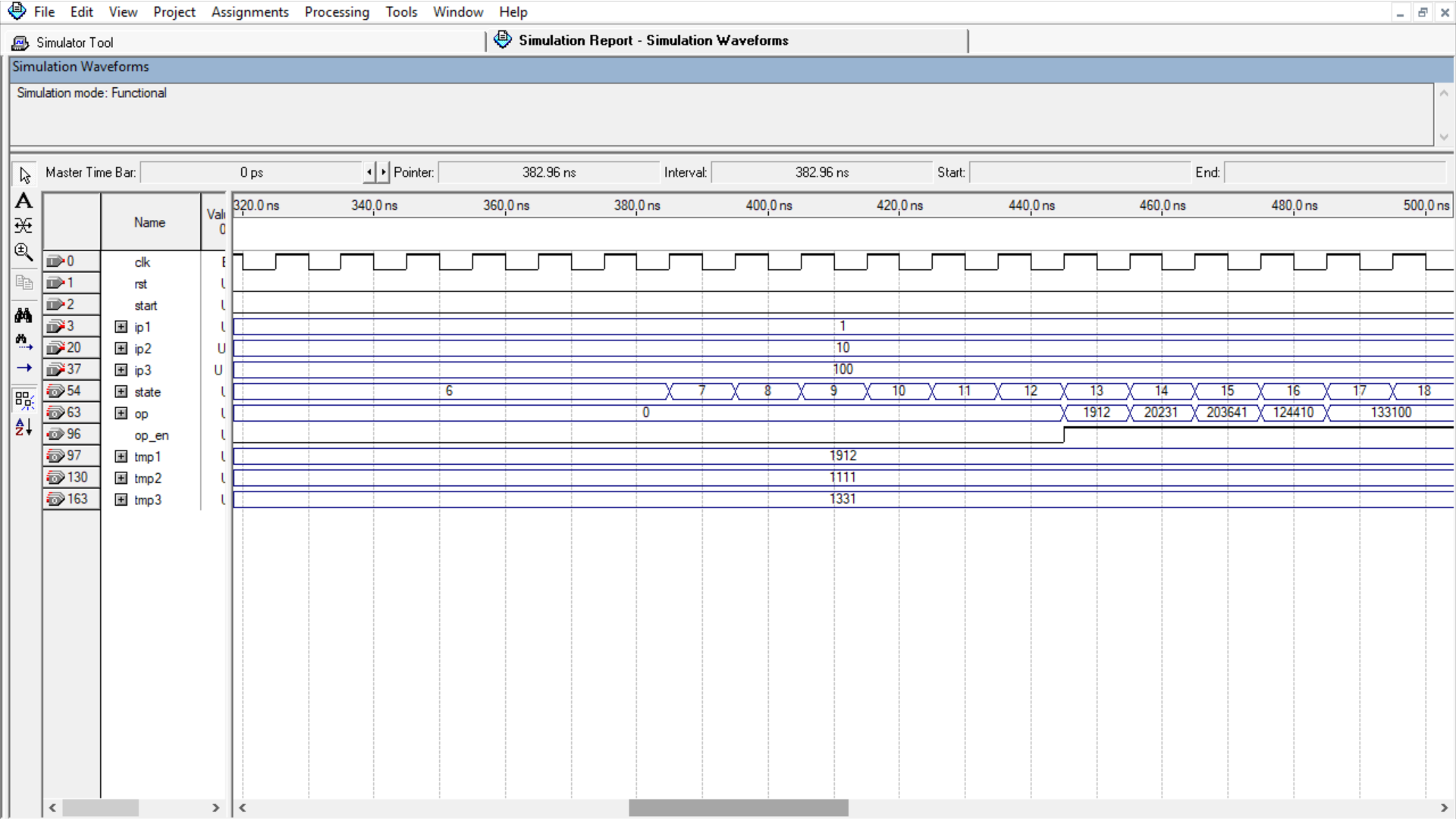


Fig 6.6: simulation Report of FIR

The simulation report of FIR is shown in above figure 6.6. The filters are usually frequency selective networks, which is capable of modifying an input signal in order to facilitate further processing. By using RPAG algorithm we are using in FIR applications. The three inputs are given as ip1, ip2, ip3. Three outputs are stored in tmp1, tmp2, tmp3.

22

|  |  |  |  |
| --- | --- | --- | --- |
|  | **LUT** | **DSP** | **FREQUENCY/delay** |
| FA32 | 90 | 0 | 20.191ns |
| RPAG | 377 | 0 | 143.10MHz |
| FIR | 578 | 0 | 131.61MHz |

Table 5.1: Summary of Compilation report

23

**5.1 APPLICATIONS:**

1.FIR filters implementation:

As fir filter has many numbers of useful properties compared to IIR filter. It requires no feedback and are inherently stable since the output is a sum of a finite number of finite multiples of the input values. Also, in fir multiplication is the basic operation. constant coefficients are usually multiplied with input terms. By the use of FPGA, we can implement high speed multipliers using constant coefficients. speed will be faster.

2.Digital signal processors:

Multiplication is the predominant operation in digital signal processors. we know that DSP is the fastest processor. The major operation performed in DSP is convolution. Convolution involves multiplication of impulse response with delayed version of input. By the use of FPGA and reconfigurable multiplier, we can make convolution perform faster compared to normal operation.

**5.2 ADVANTAGES:**

* High level optimisation
* Less power consumption
* Less complexity
* Faster configuration
* Multiplier-less constant multiplication
* Easy to implement FIR filter

24

**CHAPTER 6**

**CONCLUSION AND FUTURE SCOPE**

**CONCLUSION AND FUTURE SCOPE**

* The new algorithms and reconfigurable processing element architectures are developed on reconfigurability, low-complexity, low area, low cost, low power consumption, and high speed perspective.
* A run-time reconfigurable constant multiplication circuit based on reconfigurable LUTs was presented.
* It could be shown that the proposed realization can be beneficially used instead of a generic multiplier in terms of required hardware resources and provides a solution with short reconfiguration times compared to using constant coefficient multiplier IP cores and Partial Reconfiguration via the ICAP.

25

**BIBLIOGRAPHY**

[1] M. Garrido, F. Qureshi, and O. Gustafsson, “Low-complexity multiplierless constant rotators based on combined coefficient selection and shift-and-add implementation (CCSSI),” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 7, pp. 2002–2012, Jul. 2014.

[2] M. Faust, O. Gustafsson, and C.-H. Chang, “Reconfigurable multiple constant multiplication using minimum adder depth,” in *Proc. 44th Asilomar Conf. Signals Syst. Comput. Conf. Rec.*, Pacific Grove, CA, USA, Nov. 2010, pp. 1297–1301.

[3] Deepak, S. and B.J. Kailath, 2012. Optimized MAC unit design. Proceedings of the 2012 IEEE International Conference on Electron Devices and Solid State Circuit (EDSSC), December 3-5, 2012, IEEE, Bangkok, Thailand, ISBN: 978-1-4673-5694-7, pp: 1-4.

[4] J. Chen and C.-H. Chang, “High-level synthesis algorithm for the design of reconfigurable constant multiplier,” *IEEE Trans. Comput.- Aided Design Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1844–1856, Dec. 2009.

[5] More, T.V.; Kshirsagar, R.V. (2011) “Design of low power column bypass multiplier using FPGA, “in Electronics Computer Technology (ICECT), 2011 3rd International Conference on, vol.3, no., pp.431435, 810.

[6] J. Becker, M. Huebner, and M. Ullmann, BReal-time dynamically run-time reconfiguration for power-/cost-optimized Virtex FPGA realizations. [ presented at the IFIP Int. conf. very Large Scale Integration 2003, Darmstadt, Germany

[7] Bonamy et al.(2011) Bonamy, chillet, sentieys and Bilavam R. Bonamy, D. Chillet, O. Sentieys, S. Bilavam, Parallelism Level Impact on energy Consumption in Reconfigurable Devices, ACMSIGARCH Computer Architechture News 39 (2011) 104-105.

**APPENDIX**

**APPENDIX**

**FA32:**

library ieee ;

use ieee.std\_logic\_1164.all ;

use ieee.std\_logic\_arith.all ;

use ieee.std\_logic\_signed.all ;

entity FA32 is

port(A,B : in std\_logic\_vector(31 downto 0) ;

Cin : in std\_logic ;

S : out std\_logic\_vector(31 downto 0) ;

Co : out std\_logic );

end FA32 ;

architecture FA32 of FA32 is

component FA is

port(A,B,C : in std\_logic ;

S,Co : out std\_logic);

end component ;

signal z1,z2,z3,z4,z5,z6,z7,z8,z9,z10,z11,z12,z13,z14,z15 : std\_logic ;

signal z16,z17,z18,z19,z20,z21,z22,z23,z24,z25,z26,z27,z28,z29,z30,z31 : std\_logic ;

begin

C0: FA port map (A(0),B(0), Cin, S(0),z1) ;

C1: FA port map (A(1),B(1),z1,S(1),z2) ;

C2: FA port map (A(2),B(2),z2,S(2),z3) ;

C3: FA port map (A(3),B(3),z3,S(3),z4) ;

C4: FA port map (A(4),B(4),z4,S(4),z5) ;

C5: FA port map (A(5),B(5),z5,S(5),z6) ;

C6: FA port map (A(6),B(6),z6,S(6),z7) ;

C7: FA port map (A(7),B(7),z7,S(7),z8) ;

C8: FA port map (A( 8),B( 8),z8,S( 8),z9) ;

C9: FA port map (A( 9),B( 9),z9,S( 9),z10) ;

CA: FA port map (A(10),B(10),z10,S(10),z11) ;

CB: FA port map (A(11),B(11),z11,S(11),z12) ;

CC: FA port map (A(12),B(12),z12,S(12),z13) ;

CD: FA port map (A(13),B(13),z13,S(13),z14) ;

CE: FA port map (A(14),B(14),z14,S(14),z15) ;

CF: FA port map (A(15),B(15),z15,S(15),z16) ;

D0: FA port map (A(16),B(16),z16,S(16),z17) ;

D1: FA port map (A(17),B(17),z17,S(17),z18) ;

D2: FA port map (A(18),B(18),z18,S(18),z19) ;

D3: FA port map (A(19),B(19),z19,S(19),z20) ;

D4: FA port map (A(20),B(20),z20,S(20),z21) ;

D5: FA port map (A(21),B(21),z21,S(21),z22) ;

D6: FA port map (A(22),B(22),z22,S(22),z23) ;

D7: FA port map (A(23),B(23),z23,S(23),z24) ;

D8: FA port map (A(24),B(24),z24,S(24),z25) ;

D9: FA port map (A(25),B(25),z25,S(25),z26) ;

DA: FA port map (A(26),B(26),z26,S(26),z27) ;

DB: FA port map (A(27),B(27),z27,S(27),z28) ;

DC: FA port map (A(28),B(28),z28,S(28),z29) ;

DD: FA port map (A(29),B(29),z29,S(29),z30) ;

DE: FA port map (A(30),B(30),z30,S(30),z31) ;

DF: FA port map (A(31),B(31),z31,S(31),Co) ;

end FA32;

**RPAG:**

library ieee ;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_arith.all;

use ieee.std\_logic\_signed.all;

entity RPAG1 is

port ( clk, rst : in std\_logic;

start : in std\_logic;

X : in std\_logic\_vector (15 downto 0);

Z1, Z2, Z3 : out std\_logic\_vector (31 downto 0);

state : buffer std\_logic\_vector (7 downto 0);

tmp1, tmp2, tmp3 : buffer std\_logic\_vector(31 downto 0) ;

op\_en : out std\_logic);

end RPAG1;

architecture RPAG1 of RPAG1 is

component FA32\_ver1 is

port (A, B : in std\_logic\_vector (31 downto 0);

Cin : in std\_logic;

Add\_sub : in std\_logic;

S : out std\_logic\_vector (31 downto 0);

Co : out std\_logic);

end component;

signal const1, const2, const3 : std\_logic\_vector (15 downto 0);

signal A1, B, S1 : std\_logic\_vector (31 downto 0);

signal C1, AS1, CO1 : std\_logic;

signal A2, B2, S2 : std\_logic\_vector (31 downto 0);

signal C2, AS2, CO2 : std\_logic;

signal A3, B3, S3 : std\_logic\_vector (31 downto 0);

signal C3,AS3,CO3 : std\_logic ;

signal p1,p2,p3 : std\_logic\_vector(31 downto 0) ;

signal p4,p5,p6 : std\_logic\_vector(31 downto 0) ;

signal p7,p8,p9,p10,p11,p12 : std\_logic\_vector(31 downto 0) ;

signal p13,p14,p15,p16,p17,p18 : std\_logic\_vector(31 downto 0) ;

signal p19,p20,p21,p22,p23,p24 : std\_logic\_vector(31 downto 0) ;

signal p25,p26,p27 : std\_logic\_vector(31 downto 0) ;

begin

Comp0 : FA32\_ver1 port map (A1,B1,C1,AS1,S1,CO1) ;

Comp1 : FA32\_ver1 port map (A2,B2,C2,AS2,S2,CO2) ;

Comp2 : FA32\_ver1 port map (A3,B3,C3,AS3,S3,CO3) ;

process(clk)

begin

if(rst = '1') then

const1 <= x"0778" ;

const2 <= x"0457" ;

const3 <= x"0533" ;

state <= x"00" ;

op\_en <= '0' ;

elsif(rising\_edge(clk))then

case state is

when x"00" =>

if(start = '1') then

state <= x"01" ;

if(x(15) = '0') then

p1 <= x"0000" & x ;

p2 <= x"0000" & x ;

p3 <= x"0000" & x ;

else

p1 <= x"1111" & x ;

p2 <= x"1111" & x ;

p3 <= x"1111" & x ;

end if ;

end if ;

op\_en <= '0' ;when x"01" =>

A1 <= p1(27 downto 0) & "0000" ;

B1 <= p1 ;

C1 <= '0' ;

AS1 <= '1' ;

A2 <= p2(27 downto 0) & "0000" ;

B2 <= p2 ;

C2 <= '0' ;

AS2 <= '1' ;

A3 <= p3(27 downto 0) & "0000" ;

B3 <= p3 ;

C3 <= '0' ;

AS3 <= '1' ;

state <= x"02" ;

when x"02" =>

p4 <= S1 ;

p5 <= S2 ;

p6 <= S3 ;

p7 <= p1(24 downto 0) & "0000000" ;

p8 <= p2 ;

p9 <= p3(24 downto 0) & "0000000" ;

state <= x"04" ;

when x"04" => p10<= p7(30 downto 0) & '0' ;

p11 <= p8(30 downto 0) & '0' ;

p12 <= p9(30 downto 0) & '0' ;

p13 <= X"00000000" ;

p14 <= p5(27 downto 0) & x"0" ;

p15 <= p6(27 downto 0) & x"0" ;

state <= x"06" ;

when x"06" =>

A1<= p4 ;

B1 <= p10 ;

C1 <= '0' ;

AS1 <= '0' ;

A2<= p5 ;

B2 <= p11 ;

C2 <= '0' ;

AS2 <= '1' ;

A3<= p6 ;

B3 <= p12 ;

C3 <= '0' ;

AS3 <= '0' ;

state <= x"07" ;

when x"07" =>p16 <= S1 ;

p17 <= S2 ;

p18 <= S3 ;

A2<= p14 ;

B2 <= p2 ;

C2 <= '0' ;

AS2 <= '1' ;

A3<= p15 ;

B3 <= p3 ;

C3 <= '0' ;

AS3 <= '1' ;

state <= x"09";

when x"09" => p19 <= x"00000000" ;

p20 <= S2 ;

p21 <= S3 ;

state <= x"0A" ;

when x"0A" => p22 <= p16(28 downto 0) & "000" ;

p23 <= p17 ;

p24 <= p18 ;

p25 <= p19(29 downto 0) & "00" ;

p26 <= p20(29 downto 0) & "00" ;

p27 <= p21(29 downto 0) & "00" ;

state <= x"0C" ;

when x"0C" => A1<= p22 ;

B1 <= p25 ;

C1 <= '0' ;

AS1 <= '1' ;

A2<= p23 ;

B2 <= p26 ;

C2 <= '0' ;

AS2 <= '1' ;

A3<= p24 ;

B3 <= p27 ;

C3 <= '0' ;

AS3 <= '1' ;

state <= x"0D" ;

when x"0D" => Z1 <= S1 ;

Z2 <= S2 ;

Z3 <= S3 ;

state <= x"00" ;

op\_en <= '1' ;

when others => null ;

end case ;

end if ;

end process ;

tmp1 <= p19 ;

tmp2 <= p20 ;

tmp3 <= p21; end RPAG1;

**FIR:**

library ieee ;

use ieee.std\_logic\_1164.all ;

use ieee.std\_logic\_arith.all ;

use ieee.std\_logic\_signed.all ;

entity FIR is

port (clk, rst : in std\_logic;

start : in std\_logic;

ip1,ip2,ip3 : in std\_logic\_vector (15 downto 0);

state : buffer std\_logic\_vector (7 downto 0);

op\_en : out std\_logic ;

op : out std\_logic\_vector(31 downto 0) ;

tmp1,tmp2,tmp3 : buffer std\_logic\_vector(31 downto 0) ) ;

end FIR ;

architecture FIR of FIR is

component RPAG1 is

port (clk, rst : in std\_logic ;

start : in std\_logic ;

X : in std\_logic\_vector(15 downto 0) ;

Z1,Z2,Z3 : out std\_logic\_vector(31 downto 0) ;

state : buffer std\_logic\_vector( 7 downto 0) ;

tmp1,tmp2,tmp3 : buffer std\_logic\_vector(31 downto 0) ;

op\_en : out std\_logic ) ;

end component ;

component FA32\_ver1 is

port (A,B : in std\_logic\_vector(31 downto 0) ;

Cin : in std\_logic ;

Add\_sub : in std\_logic ;

S : out std\_logic\_vector(31 downto 0) ;

Co : out std\_logic );

end component ;

signal X : std\_logic\_vector(15 downto 0) ;

signal op1,op2,op3 : std\_logic\_vector(31 downto 0) ;

signal z1,z2,z3 : std\_logic\_vector(31 downto 0) ;

signal z4,z5,z6 : std\_logic\_vector(31 downto 0) ;

signal z7,z8,z9 : std\_logic\_vector(31 downto 0) ;

signal op\_en1 : std\_logic ;

signal start1 : std\_logic ;

signal A1,B1,S1 : std\_logic\_vector(31 downto 0) ;

signal C1,AS1,CO1 : std\_logic ;

signal y1,y2,y3,y4,y5 : std\_logic\_vector(31 downto 0) ;

begin

Comp0 : RPAG1 port map (clk,rst,start1,X,op1,op2,op3,open,open,open,open,op\_en1) ;

Comp1 : FA32\_ver1 port map (A1,B1,C1,AS1,S1,CO1) ;

process(clk)

begin

if(rst = '1') then

state<= x"00" ;

elsif(rising\_edge(clk)) then

case state is

when x"00" => if(start = '1') then

state <= x"01";

end if;

when x"01" => start1 <= '1';

X<= ip1 ;

state <= x"02" ;

when x"02" => start1 <= '0' ;

if(op\_en1 = '1') then

z1 <= op1 ;

z2 <= op2 ;

z3 <= op3 ;

state <= x"03" ;

end if ;

when x"03" => start1 <= '1' ;

X<= ip2 ;

state <= x"04" ;

when x"04" =>start1 <= '0' ;

if(op\_en1 = '1') then

z4 <= op1 ;

z5 <= op2 ;

z6 <= op3 ;

state <= x"05" ;

end if ;

when x"05" => start1 <= '1' ;

X<= ip3 ;

state <= x"06" ;

when x"06" => start1 <= '0' ;

if(op\_en1 = '1') then

z7 <= op1 ;

z8 <= op2 ;

z9 <= op3 ;

state <= x"07" ;

end if ;

when x"07" => y1 <= z1 ;

A1 <= z2 ;

B1 <= z4 ;

C1 <= '0' ;

AS1 <= '1' ;

state <= x"08" ;

when x"08" => y2 <= S1 ;

A1 <= z3 ;

B1 <= z5 ;

C1 <= '0' ;

AS1 <= '1' ;

state <= x"09" ;

when x"09" => A1 <= S1 ;

B1 <= z7 ;

C1 <= '0' ;

AS1 <= '1' ;

state <= x"0A" ;

when x"0A" => y3 <= S1 ;

A1 <= z6 ;

B1 <= z8 ;

C1 <= '0' ;

AS1 <= '1' ;

state <= x"0B" ;

when x"0B" => y4 <= S1 ;

y5 <= z9 ;

state <= x"0C" ;

when x"0C" => op\_en <= '1' ;

op <= y1 ;

state <= x"0D" ;

when x"0D" => op <= y2 ;

state <= x"0E" ;

when x"0E" => op <= y3 ;

state <= x"0F" ;

when x"0F" => op <= y4 ;

state <= x"10" ;

when x"10" => op <= y5 ;

state <= x"11" ;

when x"11" => op\_en <= '1' ;

state <= x"12" ;

when others => null ;

end case ;

end if ;

end process ;

tmp1 <= z1 ;

tmp2 <= z2 ;

tmp3 <= z3 ;

end FIR ;