HIGH LEVEL SYNTHESIS FOR RAPID DESIGN OF VIDEO PROCESSING PIPES

A Thesis

by

Aydın Emre Güzel

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2017

HIGH LEVEL SYNTHESIS FOR RAPID DESIGN OF VIDEO PROCESSING PIPES

Approved by:

Assoc. Prof. H. Fatih Uğurdağ, Advisor
Department of Electrical and Electronics Engineering
Özyeğin University

Asst. Prof. Barış Aktemur
Department of Computer Science
Özyeğin University

Assoc. Prof. Sezer Gören Uğurdağ
Department of Computer Engineering
Yeditepe University

Date Approved: 12 January 2017

To Everyone

ABSTRACT

Pipelining concept is a fundamental technique in digital hardware design, which maximizes the clock frequency or minimizes the resources. Designing a pipelined Field Programmable Gate Array (FPGA) module using pipelined arithmetic modules brings us challenging allocation, scheduling, and binding issues, especially when the Initiation Interval is more than one. In the case of algorithms with high computational cost, for ex., in video processing, we need to automate these error prone and time consuming processes. In this thesis, we share our experience in using High-Level Synthesis (HLS) for rapid development of an optical flow design on FPGA. We have performed HLS using Vivado HLS as well as a HLS tool we have developed for the optical flow design at hand and similar video processing problems. The thesis describes the design problem we have and then discusses our own HLS tool. The tool we developed is general-purpose except for the inability to handle cyclic inter-iteration dependencies. It also introduces novel concepts to HLS, such as pipelined multiplexers. The synthesis results show that we can achieve better timing or better area results compared to Vivado HLS. Furthermore, the Verilog RTL our HLS tool outputs is better than Vivado HLS in terms of readability. Also, the time-resource tables we produce for both arithmetic units and registers make it easier for the designer to debug and modify the RTL.

ÖZETÇE

Boru hattı konsepti saat frekansını yükselten ve kaynak kullanımını azaltan temel bir sayısal donanım tasarımı tekniğidir. Boru hatlı aritmetik modülleri kullanan bir boru hatlı Sahada Programlanabilir Kapı Dizileri (FPGA) modülünün tasarlanması, özellikle Başlatma Aralığı birden fazla olduğunda, tahsisat, çizelgeleme ve bağlama konularında zorlu bir geliştirme sürecine sebep oluyor. Yüksek işlem yüküne sahip algoritmalar söz konusu olduğunda, örneğin video işlemede, bu hata yapmaya eğilimli ve zaman alan süreçleri otomatikleştirmek son derece gereklidir. Bu tezde, FPGA'de bir optik akış tasarımının hızlı geliştirilmesi sırasında yaşadığımız Yüksek Düzeyli Sentez (HLS) deneyimimizi paylaştık. Elimizdeki optik akış tasarımı ve benzeri görüntü işleme problemleri için geliştirdiğimiz HLS aracını ve Vivado HLS'yi kullanarak ayrı ayrı aynı tasarımı gerçekleştirdik. Bu tez, sahip olduğumuz tasarım problemini açıklıyor ve daha sonra kendi HLS aracımızı detaylı bir şekilde anlatıyor. Geliştirdiğimiz araç, döngüsel ara iterasyon bağımlılıklarını işleyememe dışında oldukça genel amaçlı bir araçtır. Ayrıca, "boruhatlı çoklayıcılar" gibi HLS'ye yeni kavramlar getiriyor. Sentez sonuçları, Vivado HLS'ye kıyasla daha iyi zamanlama veya daha iyi alan sonuçları elde edebildiğimizi gösteriyor. Dahası, HLS aracımızın Verilog RTL'si Vivado HLS'den daha okunabilirdir. Üniteler ve yazmaçlar için üretilen kaynak zaman tabloları da düşünüldüğünde, tasarımcının RTL'de hata ayıklamasını ve elle değişiklik yapabilmesini daha kolay hale gelmektedir.

ACKNOWLEDGMENTS

This thesis work has been done as part of a project jointly supported by TÜBİTAK ARDEB-EEEAG and European Union's Artemis Joint Undertaking. On TÜBİTAK side, the project is called "M-RIVA: Methodology development for Real-time Implementation of Video Algorithms on FPGAs" and has been designated project no. 114E343. On Artemis side, the project is part of a bigger project called "ALMARVI: Algorithms, Design Methods, and Many Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing" and has been designated project no. 621439. ALMARVI project (http://almarvi.eu) is run by a consortium of 16 universities and companies from Netherlands, Finland, Czech Republic, and Turkey, which includes Philips, Nokia, Aselsan, and Özyeğin University. I have been partly supported by TÜBİTAK (under project no. 114E343) since November, 2016.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis introduces a new academic semi-generalized high level synthesis (HLS) tool which we call MAFURES, and a design methodology for a complex video processing application utilizing MAFURES in the scope of ALMARVI project.

1.1 Problem Statement

ALMARVI [1] is an ARTEMIS project, which is funded by both TUBITAK and European Union. Name of the project comes from the abbreviation of Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing. It has 16 partners from 4 different countries, which are Netherlands, Finland, Czech Republic and Turkey. Ozyegin University (OZYEGIN) and ASELSAN are the only Turkish partners.

In the scope of ALMARVI project, we trying to perform efficient FPGA implementation of Anisotropic Huber-L Dense Optical Flow [2] algorithm. At the end results will be compared and contrasted with the GPU implementation of the same algorithm which was designed by ASELSAN.

In general, the main aim is to develop a general-purpose architecture for multi-pass video applications targeted to FPGA. We call this architecture FIPPA after Frame, Iteration, Pixel level Pipeline Architecture. OZYEGIN created this architecture in the process of working on a critical building block (i.e., dense optical flow) of the large area video surveillance demonstrator defined by ASELSAN. Dense optical flow is a very compute-intensive algorithm and presents a challenge in terms of low-power implementation. Lessons learned and requirements gathered from this application can be applied to any algorithm that does multiple passes over video frames. The

requirements of such applications when implemented on an FPGA driven by a host CPU can be looked at in three categories: (i) satisfying performance targets, (ii) satisfying resource constraints, and (iii) minimizing power. A design methodology we developed around High Level Synthesis (HLS) has been used in the scope of this project.

1.2 Why HLS?

HLS is a kind of design automation concept. This section answers the questions what is HLS and why we need it.

1.2.1 What is HLS?

Hardware design process includes several layers. Abstractions based on these layers can be listed in three main headings:

- Algorithmic level
- Register transfer level
- Gate level

HLS which is sometimes called behavioral synthesis, electronic system-level synthesis, or algorithmic synthesis, is the automated conversion of algorithmic level abstraction to register transfer level abstraction. The process usually starts with a high level description of the algorithm (generally C-like languages are used as input) without specifying any exact (or clock level) timing information. So, the main point here is to let the HLS tool decide and handle all low level timing issues with the algorithmic knowledge of sequence and dependency of arithmetic operations. In early days, different input specification languages were designed [3], however current research and tools are based on input of ANSI C, C++, SystemC, MATLAB codes. Nevertheless, they cannot cover every possible algorithmic description with the input language. In

all HLS applications, input description must be written considering the rules that come with it. Basically, they promise beneficial results only when the defined synthesizable subset of the high level input language is used.

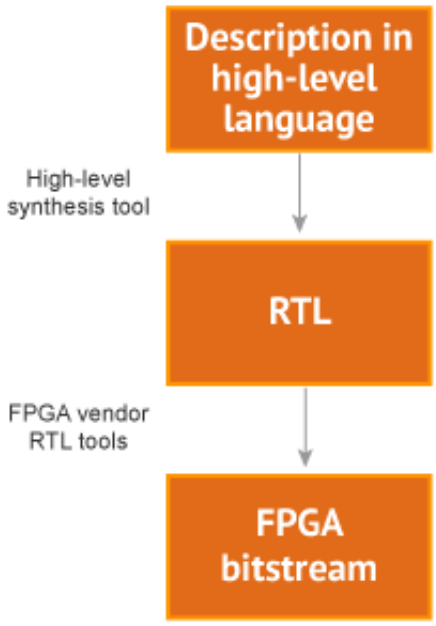


Figure 1: Design Flow Using High-Level Synthesis

Producing register transfer level code from high level languages consists of several procedures. A general HLS application applies these procedures [4]:

- Lexical processing
- Algorithm optimization
- Control/dataflow analysis
- Library processing
- Resource allocation

- Scheduling
- Functional unit binding
- Register binding
- Output processing
- Input bundling

Activities may use different algorithms or tools combine them according to approach. However, steps of a high level synthesis tool could be classified and evaluated with help of these categorizations.

C-like high level languages are not always enough to describe technical design specifications, especially in terms of low level organizations and optimizations. So high level tools accept constraints input for the architectural decisions that we want to decide manually [5]. Interfaces between other systems, memory designs, low level timing constraints, loop transformations, iterations, and hierarchical order are the main topics that are user configurable with constraints input.

The produced RTL code, from high level algorithm description and user constraints with HLS tool, feeds logic synthesis tool. Then algorithm turns into gate level description. The main aim of using HLS is to escape burden of procedural development process that we can generalized to passing an automation tool. This technique gives better control over general architecture of system by using higher level of abstraction approach. Also they are offers development time efficiency, less error prone process and easier verification strategy.

1.2.2 Necessity of HLS

The concept that we want to describe in this project with high level hardware design rhetoric is a kind of automation. The point is production of HDL through a computer

software. This computer software can make some design decisions by making calculations through inputs and design the hardware system according to the abstracted and simple (high level) inputs.

We decided that the production of some parts of our project would be much more convenient in terms of time and effort than manual writing. The difficulty of writing hardware designs, primarily those of special and general purpose influences, is a new paradigm and project specific reasons.

1.2.2.1 Difficulties of Writing RTL instead of High Level Software

Hardware design is a sensitive business that requires technical knowledge, attention, labor and long time at a high level. Compared to writing a computer software that runs on a processor, it is much more difficult. As the complexity and diversity of the products planned to be designed increases, this difficulty is further evolved. We can examine these difficulties under 4 headings. These are design, implementation, verification and updating.

- Design

  We can use the following two approaches to explain the design challenge. The main is to own all the control and responsibility when designing the hardware in the important issues that we do not pay attention when writing software and we cannot easily affect it or our effect is limited. When these decisions are made, fine calculations should be made taking into account the aims, factors and preconditions. For example, the following items can be listed:

  - Optimal use of resources on hardware
  - Fixing the clock speed
  - Data delivery timings

- Optimal use of memory,
- Recursive and repetitive editing complexity
- Limitation of power consumption

The second approach lies in the purpose of hardware design of any algorithm rather than running it in software. To achieve this goal is the design of low-level optimizations that cannot be done with a very high-level approach in the software language due to the architecture of the existing hardware, ie the processor. These designs are capable of adding serious challenges to design. We can exemplify the following objectives:

- Accelerate in terms of delay time
- Accelerate by the amount of work done in the unit
- Any end result can be sharp
- Working with low power

- Implementation

  High level code written in a software environment is compiled by a compiler, and the results of this compilation is converted a specialized processor design. The high level software may contain abstraction, static or dynamic objects may be created, and cyclic and recursive operations may be performed serially, with the order being correct, without being easily examined in any time dimension. The missing points are completed by compiler and processor design. In the hardware environment, almost everything is written in the hardware description language. However, RTL description has to describe and cover everything. Registers, cables, modules, status machines, timings, etc. When it comes to computer software, hardware description software consists of texts that are much harder to read, understand and understand than software texts. Because of this, it is

more likely to make mistakes when writing firmware.

- Verification

  Humans tend to make mistakes. While designing algorithms, hardware designs or writing software, many human errors can occur. Finding and correcting mistakes can be a difficult and time-consuming task.

  When we arrive at the design and post-implementation verification part, we are faced with a very different and difficult environment according to the software environment. We can examine this situation under two headings. First is the difficulty of finding an error. The second is the difficulty of correcting the error found. Simulation and error correction tools in the software environment are highly developed. If the hardware description languages are tested, it is also necessary to write test environment software. In addition, while computer software can be analyzed line by line, the results of hardware designs can be examined by following multiple waveforms from the waveform display. The error correction issue can be seen as a significant disadvantage to the writing of hardware design. Because reusability in the software is extremely high, the slightest design changes in the hardware description language are likely to cause a significant portion of the software to be rewritten.

- Update

  The biggest disadvantage of hardware design is living under this title. The design code may have to be discarded and rewritten at any point in the hardware description code in the algorithm, the hardware environment, any changes in requirements. Computer software is not greatly affected by the hardware environment and does not have the complexity of parallax in time dimension (except multi-core programming). Algorithm-related changes can be accommodated much more quickly due to high level, reusability and abstract structures.

Requirement changes (e.g. resolution) can easily be made parameter-based. In the hardware description, these parameter changes often cause other large changes in the software.

1.2.2.2 Changing Electronic and Software Paradigm

Technology, engineering and science are also making steady progress in many areas. In parallel, the complexity and variety of electronic equipment is constantly increasing. In the long development process, it is about to be outdated to technology and needs, to be crushed by the costs of time and human resources. In addition, even if everything is planned in the right way, the likelihood of making simple mistakes arising from the fact that engineers working on hardware realization are human is also increasing. The predisposition to this flaw further increases the time and effort burden with exhausting and lengthy verification efforts. The resulting progress can be more important than being able to respond quickly, optimally and with the right design. The paradigm of designing high fundamental low possession is rapidly shifting to high possession, rapid reaction. In addition, it will be a better thing to approach the design process proactively, not reactive, by foreseeing changes that may occur in the future.

1.2.2.3 The Need for High-Level Design Specific to the Project

The algorithm we use to design the hardware in the project we are involved contains serious complexities. Our goal is to run this algorithm on the FPGA as efficiently as possible and with low power consumption. Besides, we face many unknowns. We list the main causes of this uncertainty:

- Uncertainty of Development Card and FPGA to be Used

  The final development environment to be used during the start-up of the project was unclear. It was planned to acquire a development environment that would meet the optimum needs over the experience and knowledge to be formed over time. The work to be done during the decision making process in this direction

should have been at a level where it could show compatibility with different hardware options.

- Uncertainty of Resource Needs of Different Modules

  The hardware design of the project consists of different parts such as iteration, interpolation and warping in terms of task. These structures, which are jointly designed by our project team, are realized by sharing tasks in different modules. This process is proceeding in parallel. In this case, it is not possible to predict exactly how many resources will be used by the module, how fast it will be, how fast it will be, who will use the bandwidth and when. The designed design should be able to provide flexibility to such incompletely predictable situations.

- Alteration Between Algorithm Power and Hardware Performance

  On the algorithm we will design and synthesize hardware, there are parameters that change the power of the algorithm. The number of iterations to be done, the number of warp to be executed, or how many bit operations arithmetic operations are some of these. After design and implementation, the hardware description code will have to be rewritten many times according to different algorithms and hardware speed performances according to needs. In addition to this, the preparation of a platform that can easily adapt to improvements or reforms to be applied with similar reasons for the content of the algorithm should also be included in this production procedure.

1.3 Contributions of the Thesis

The contributions to the literature are the following:

- A HLS tool that generates readable and hence more easily debuggable Register Transfer Level (RTL) code

- The concept of multiplexer pipelining in scheduling time and implementation

on a HLS tool

- Better timing results with multiplexer pipelining and better area results without it based on an optical flow algorithm compared to state-of-the-art tool Vivado HLS
- Pipelining exploiting and HLS centered design strategy for a complex optical flow algorithm

CHAPTER II

RELATED WORK

HLS is a huge topic and consist of lots of layer. There are lots of previous work and academic research for each. In addition, commercial and academic HLS tools are developed and some of them still developing today. This chapter tells history of HLS and active tools we used in the project.

2.1 History of HLS

There is a productivity gap between algorithm development and hardware development. It may also be stated as a productivity gap between development of simulation models in software and their efficient real-time implementation on custom hardware platforms. High-Level Synthesis (HLS) has been looked at as a panacea to this problem.

In 1970s, the early days of VLSI design, designers took algorithms and hand-drew layout. Tediousness of hand-drawing of layout resulted in design automation work. As VLSI design community raised level of abstraction of the input of the design description to their layout generators, their appetite for higher and higher levels of input abstraction grew bigger and bigger. Eventually by 1979, silicon compilers became a utopic goal. What was meant by a silicon compiler starting in 1979 and then in early 1980s was a tool that could take in a behavioral description in a programming language and output layout with no intervention. Soon (by 1981) it was clear that this was indeed a utopia and that rather a layered approach was needed, where point tools would have to be used in a sequence (if needed in multiple design iterations) with intervention at cleanly defined design flow interfaces. There were, by the way, already efforts in the direction of a layered approach [6]. The first layer in this new

layered approach would take in a high-level behavioral (i.e., functional) description of the design. It would then directly output gates or output RTL, which would then go into a logic synthesis tool. The design flow would continue with physical design automation tools. By 1981, researchers called this most front-end step in their layered approach architectural synthesis [**?**] or data path synthesis [7]. Starting around 1984, these two terms were most of the replaced with the term HLS. In 1985, people started calling it behavioral synthesis. Since then both terms, HLS and behavioral synthesis, survived, with HLS being used more commonly in the academia.

Roughly, 1985 to 1995 saw a huge interest in HLS in the academia. The euphoria in the academia resulted in commercial HLS tool releases in the industry as well as some in-house tools [8] by 1995. Rise and fall of HLS in the industry was quick. In about 10 years at the most (by mid 2000s), it was clear that HLS was not going anywhere. It was very surprising, because its sister, logic synthesis, had become a commodity tool long ago.

Starting around the same time (mid 2000s), HLS were seen as a means to make FPGAs easily adoptable into applications with few or no hardware experts in the team. And since 2011-12, HLS has been successfully used by the FPGA community and its users. HLS is again in a euphoric state; however, this time industry success has spawned academic interest, whereas before it was the (presumed) academic success spawned industry interest.

Why has HLS failed first, in the years 1980-2005, but then seems to be succeeding since 2011? As in many markets, a complete answer requires the grasp of the full history of how things progressed, which can be found in survey articles. We recommend [9] for a survey relatively recent work (mostly since 2000). For a more complete but older survey, please see [10].

We have a relatively simple answer to the above question. In the first phase of

HLS (1980-2005), we all expected too much from HLS. We wanted it to be general-purpose, work for the design of commodity ICs, and produce results that even beat expert designers. Imagine a software that plays chess, checkers, and backgammon and beats world champions in all three. And imagine a problem that is even harder than that, because in the case of HLS, it was not even clear what it was designed for in the first phase of its history. Yes, the same is possible for software compilers, but then, they are not designed to produce real-time implementations, and they do not beat a human assembly coder in terms of code footprint. How come HLS is successful now? That is because FPGA community expects only acceleration from it, and the execution model is more well-defined than before [10]. Or as in the case of synthesis from OpenCL [11], it is domain specific.

Having said that, we claim for HLS to be successful for commodity IC designs and/or for truly high throughput systems (i.e., for beyond acceleration), it cannot be a one size fits all type of tool. We look at it in the spirit of semi-automation in terms of the bigger picture [12]. Even then, it may have to be domain specific for truly high-throughput systems. There is a plethora of work on domain specific languages in the software domain [e.g., [13]); and thankfully, we are also seeing an influx of work on domain specific HLS such as [14],[15],[16]. Furthermore, the problem of HLS problem becomes more tractable within a particular design framework. A framework is a flow or methodology, which usually comes up with a top-level design as well as libraries. Framework based approach is common in hardware verification but not as much in hardware design itself so far.

2.2 State-of-the-Art

There are several in-use HLS tools. We look them considering the following criteria:

- Floating point arithmetic operation support
- Loop pipelining support with adjustable II constraint

- General purpose domain
- C or C-subset input
- Verilog output

After the elemination following tools are remained:

- CHC (Commercial) (2008)
- Symphony C (Commercial) (2010)
- LegUp (Academic) (2011)
- VivadoHLS (Commercial) (2013)

We started to work with the newest 2 which are LegUp HLS and Vivado HLS to implement the algorithms were used in ALMARVI.

2.2.1 Vivado HLS

Vivado HLS [17] is a commercial high-level synthesis tool developed by the Xilinx company. Vivado HLS, which is widely used in the industry, targets FPGA cards of the same company. It includes advanced professional interface, communication, signal processing, video processing and other important libraries. It is receiving C / C ++ software as input, and can produce Verilog and VHDL hardware description software.

2.2.2 LegUp HLS

The LegUp HLS [18] tool is a high-level Verilog software synthesis tool developed by a crowded team to dissect a doctorate at the University of Toronto. It has been developed since 2010. They're open source. They are C / C ++ software as input. They produce Verilog software as output. The devices they are targeting are Altera FPGAs. It is planned as an infrastructure that can be tried on high-level synthesis

algorithms developed initially. Their long-term goal is to make FPGA programming easier for software developers.

CHAPTER III

MAFURES

3.1 Problem Definition

Pipelining concept is a fundamental technique in digital hardware design, which maximizes the clock frequency or minimizes the resources. Designing a pipelined FPGA module using pipelined arithmetic modules brings us challenging allocation, scheduling, and binding issues, especially when the Initiation Interval is more than one. In the case of algorithms with high computational cost, for ex. in video processing, we need to bypass these error prone and time consuming processes.

In ALMARVI project we design hardware system top level to down. Parts of algorithm of optical flow considered as different sub-systems such as iteration part [19], warping part[20], downscaler part [20] etc. Organization of high level system (numbers and placements of sub systems module, data delivery and communication protocols between them, FIFOs between them, DRAM communication and etc.) and low level sub-systems evaluated differently. MAFURES entered the game for designing and implementation of these sub systems with arithmetic functional units.

In this work, we propose high level optimization strategies for an optical flow algorithm, which lead to a semi-generalized HLS back-end tool that aims to maximize resource sharing and clock frequency. Our tool got better timing and area results for the mentioned optical flow algorithm, when compared to the state-of-the-art commercial tool (Vivado HLS) when using the same function units. In addition, our tool generates a user friendly Verilog output as well as time and resource tables for both arithmetic units and registers make debug and manual modification easier.

3.2 MAFURES Overview

MAFURES is a HLS tool that aims to automatically generate FPU level design of ALMARVI project. It produces synthesizable Verilog HDL output from algorithmic inputs that have no information about cyclic timing level. We call our HLS tool MAFURES after Multiplex-ing Aware FUnction and Register Scheduler.

MAFURES can calculate functional requirements, schedule functional assignments to functional hardware blocks, arrange data delivery between these blocks, inputs and outputs with using registers. Also it is modular and open to imply new algorithms for any step of this HLS process. Now MAFURES behaves a back-end HLS tool which accepts intermediate representations and constraints. Front end conversions and optimizations are done manually (see future work). Technically MAFURES can support and combine these two pipeline techniques:

- Functional pipelining

  Term of functional pipelining is about developing system with pipelined building blocks. In this work we implemented algorithms using pipelined floating point units with different number of pipeline stage. MAFURES accepts function descriptions with different parameters we will discuss later such as latency number of cycles. MAFURES capable to design and produce HDL implementation using high level inputs of algorithms that written with descripted function units.

- Loop pipelining

  Loop pipelining enhance the hardware performance by exploiting parallelism between loop iterations. In sequential systems, new iteration of a loop starts when last operation of previous loop is finished. This technique allows performing different steps of loop concurrently. So next iteration of loop can start after the first pipeline stage of previous one. This is valid for all other stages. Such

as a loop iteration divided into 5 stages means 5 sequential loop iteration can perform concurrently in steady-state.

Here a significant term Initiation Interval (II) comes into game. It is describing time between two consecutive loop iterations in terms of cycles. MAFURES can evaluate a pipelined module with given II using pipelined or non-pipelined blocks. It is essential technique to exploit parallelism in computer vision and image processing algorithms because they contain lots of loops to open such as frame loops, pixel loops, iterative loops etc. In ALMARVI project, we prepared algorithms to loop pipelining pixel based in high level. Our tool fed by them and did all back-end work with implemented scheduling, allocation and binding algorithms automatically in transition of algorithm to RTL. MAFURES is pretty similar to most HLS tools in terms of its sub steps and involves the following:

- Read in the library of FPUs and initiation interval
- Determine the number of FPUs
- DFG creation
- Computation of operation (op) priorities
- Scheduling and allocation of ops to FPUs
- Register allocation
- Creation of muxes

MAFURES first reads a library pipelined FPUs (throughput = 1 op/cycle and any latency), which is supposed to have one FPU for each op type. The tool synthesizes algorithms written in a very basic C syntax. Based on the number of ops and the initiation interval, the number of FPUs is determined. MAFURES does not handle cyclic inter-iteration dependencies. They do not occur in video processing much. As

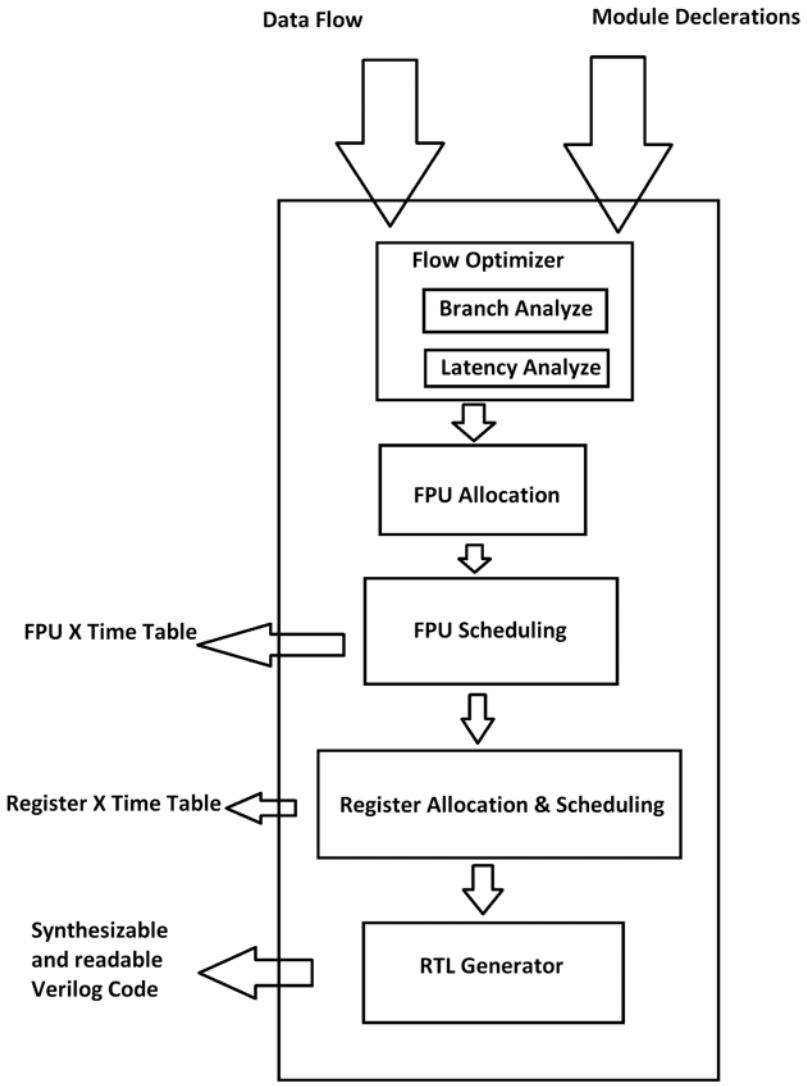


Figure 2: MAFURES Flowchart

a result, any desired initiation interval can be achieved by folding the loop body as many times as needed.

A DFG is created for the input algorithm, and then the longest execution path through every op is computed and becomes its priority. Ops whose predecessors have already been scheduled are put into a list. Each time, the op with the highest priority is picked and is scheduled in the earliest cycle possible, and then the list is updated. The earliest cycle is determined by when the predecessors end and which

cycles are unoccupied. When there are multiple FPUs available in the same slot we are considering, the FPU selection is done to balance the amount of muxing at the FPU inputs. During scheduling, when we go beyond the schedule table in the time axis, we wrap around to the first cycle and decrement the iteration index of the op being scheduled. As for register allocation, we can handle values with lifetimes longer than the initiation interval through multiple registers that connect to each other in a shift-register fashion.

3.3 Main Steps of MAFURES

3.3.1 Calculating FPU Requirements

Required number of FPUs are calculating by dividing number of operations to initiation interval for every operation and FPU type. Ceiling of these numbers give us count of utilized FPU modules.

$$\#\_of\_FPU(II = k) = \#\_of\_FPU(II = 1)/k \tag{1}$$

3.3.2 Rearranging Schedule Priority of Operations

Arithmetic operations will be scheduled one by one from top to down. So order of operation also decides priority. One of the important aim of the tool is reducing total latency of system. At best case total latency can be equal the sum of operation latencies of FPUs on longest path. This is possible with connecting outputs to inputs successively without any gap. However, in some cases tool may not schedule the earliest cycle when all related FPUs are using at that cycle. Then operation scheduled one of the next cycles and this situation increase total latencies. To eliminate or minimized this kind of lack of resource delay, our tool applies following steps before scheduling:

- calculate longest paths in terms of latency
- give schedule priority the first element of longest paths

- remove it from the graph
- repeat these steps until reaching empty graph

3.3.3 Scheduling of Pipelined Blocks

Scheduling of blocks algorithm can be explained with the following steps:

- Calculate earliest schedule cycle we can by finding maximum of input result latencies and related condition result latencies
- Try to schedule one of the slot at earliest cycle modulo initiation interval
- If all of them are not available at that cycle, increase schedule cycle by one and try previous step again. Repeat it until find an empty slot
- Set this operation in schedule table and make it unavailable
- Calculate result latency by adding operation latency and start latency and save it for next operations
- Update discard cycle of input variables for the life time calculations

3.3.4 Multiplexer Balancing

Due to high initiation interval, multiple operations can be assigned into single FPU with multiplexers. When assigned operations have conditional assignments for specific FPU, size of the setup multiplexer going to increase. This situation may increase longest path. To avoid from accumulation of conditional multiplexing, tool can swap operations between equivalent FPUs for the same cycle. In this way, after schedule all operations in schedule table, tool counts conditional assignments quota per FPU than compare biggest and lowest and if difference bigger than 1, its going to replace scheduled operation among them for one row which first one the maximum one has multiplexing and minimum one dont have. Repeating this procedure until max difference one or zero will balance the conditional multiplexing.

```
for all aritmetic operations -> unit
    temp = max(unit.inputs.arrivalLatency,unit.relatedConditions.arrivalLatency);
    unit.setStartLatency(temp)
    lackOfResourceDelay = 0;
    resourceNum = 0;
    while (true) {
        if (slotAvailability[resourceNum][(unit.getStartLatency)%throughput]==true){
            slotAvailability[resourceNum][(unit.getStartLatency)%throughput]=false;
            unit.setModuleNum(resourceNum);
            scheduleTable[resourceNum][(unit.getStartLatency) % throughput] = unit;
            unit.setOutputLatency(unit.getStartLatency + unit.operationLatence);
            unit.setResourceDelay(lackOfResourceDelay);
            temp = max(unit.getStartLatency,unit.inputs.getDiscardCycle);
            unit.inputs.setDiscardCycle(temp);
            temp = max(unit.getStartLatency,unit.relatedConditions.getDiscardCycle);
            unit.relatedConditions.setDiscardCycle(temp);
            break;
        }
        i++;
        if (i == scheduleTable.slotAvailability.length) {
            i = 0;
            lackOfResourceDelay++;
            unit.startLatency++;
        }
    }
}
```

Figure 3: Pseudocode of Scheduler

3.3.5 Register Scheduling and Allocation

Main steps of the register scheduling and allocation algorithm I used:

- Calculate life time of inputs, outputs and intermediate values by subtracting arrival cycle from last using cycle
- If it is smaller than initiation interval, search a register which is available in those cycles from register pool. If cannot find, add new empty register into pool and allocate required cycles
- If lifetime bigger than initiation interval, divide registration process into parts which are equal or smaller than initiation interval. Then previous step is applied for every parts.

In case of life time of a variable bigger than II, there should be ceil of life time over II copies of registers. Here there are two options to save them correctly. First method is shifting values to next register when new data arrived which equals to

```
for all Variables -> unit
    releaseCycle = unit.getDiscardCycle() + 1
    currentPosition =  unit.getArrivalLatency() + 1
    startCycle =  unit.getArrivalLatency() + 1
    while (true) {
        if (currentPosition == releaseCycle) {
            locateBetween(startCycle % throughput, releaseCycle % throughput, unit)
            break
        }
        if (currentPosition % throughput == throughput - 1) {
            locateBetween(startCycle % throughput, throughput, unit)
            startCycle = currentPosition + 1
        }
        currentPosition++
    }

void locateBetween(startPoint,endPoint,unit) {
    usingRegister = findAvailableRegBetween(startPoint,endPoint)
    if(regIndex == false){
        usingRegister = new Register(throughput)
        registerPool.add(usingRegister)
    }
    for (i = startPoint i < endPoint i++) {
        usingRegister[i] = unit
        unit.updateRegisterLocationArrray(usingRegister)
    }
}

int findAvailableRegBetween(a,b) {
    reg = registerPool.getFirst
    for (int i = a; i < b; i++) {
        if (reg.availability[i] == false) {
            reg == reg.getNextRegister
            if(reg == null)
                return false
            i = a-1
            continue
        }
    }
    return reg
}
```

Figure 4: Pseudocode of Register Allocator and Scheduler

II. Second method is putting multiplexers of copies and sending every new data to different register sequentially. In this method variable stores in same register without shifting but multiplexers may decrease maximum clock frequency of circuit. We are using method one due to timing reason.

This kind of expression allows us to apply any kind of improvement on area by changing the order of assignment blocks. Different strategies will be analyzed in the future.

3.4 MUX Pipelining Feature of MAFURES

After the first synthesis results of working designs we saw that maximum frequency of synthesized circuit can run is significantly higher than slowest FPU we used. That means connections described in RTL between FPUs and registers involved longest path make slower the circuit in terms of maximum frequency.

Next step multiplexing is carried to different cycle in scheduling. To do this, scheduler assumed latency of our operations plus one in scheduling time. All conditional and non-conditional assignments done with that way. This approach gave better result but still there is a significant gap between results and target.

At that point pipelined multiplexers option is implemented for the MAFURES. This is slightly new idea in HLS world. Before this, there is a one work about it [21]. They did multi-cycle and false positive path analysis efficiently. Then founded slacks near the huge multiplexers utilizing with replacing these with pipelined ones without increasing total latency of data flow. They modified LegUp HLS and get better results. Our approach is inserting pipelined multiplexers in scheduling time. Then all multiplexers can be divided into smaller ones with our way. This will increase the latency of flow however increasing the throughput by increasing maximum clock frequency with adding latency is good trade-off most of the time.

Now MAFURES can produce m input pipelined MUX threes with n input combinational MUXs in behavioral level. This n and m implemented as a variable in MAFURES. Pipeline stage number of created MUX calculated from below formula:

$$\log_n m = \#MUX\_PIPE\_STAGE \tag{2}$$

In the process of execution m can be calculated from II and maximum of number of conditional input of scheduled operation module. Here I faced off chicken and egg problem. Program cannot know exact number of inputs an instantiated module without scheduling and also cannot schedule the operation blocks without knowing

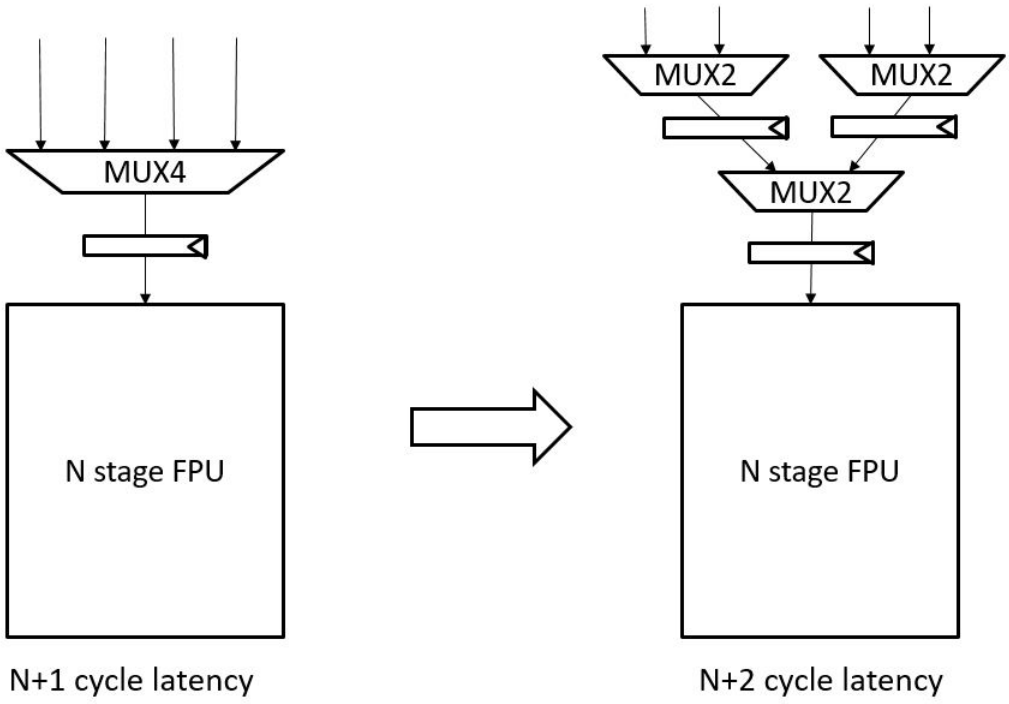


Figure 5: MUX Tree Example

number of pipeline stage of MUX which calculated from input count.

In our case studies, the design parameter n is determined manually. We synthesized different sized MUX and selected closest one to the slowest floating point unit we used. Another design parameter m can be calculated automatically from the II and number of conditional inputs for each operation. The summation of II and ceiling of conditional input number over allocated module number gives value m for each operation.

3.5 Software Classes of MAFURES

In this section I will describe MAFURES software that generates RTL in the direction of inputs in terms of the basic classes it contains.

3.5.1 Arithmetic Unit Class

We considered the types of arithmetic operations as the classes with different characteristics from the beginning. As a result of this thinking, we have implemented software classes as a data structure in which these classes hold data related to the related arithmetic operation. We have decided that these software classes that we have specified should store the following information related to the operation process,

- The name of the operation
- Process Order
- Number of Inputs
- Object Signs of Inputs
- Start Delay
- End Delay
- Lack of Source Delay
- Life Out
- Register storage time
- ID of storage registers it used

Among arithmetic operations, addition, subtraction, multiplication, division and comparison take two inputs, a root take and an absolute value take one input. The direct input of the core of the entries' iteration module may be a fixed number or the result of another operation module. The arithmetic operation should know where the input of the object came from, namely the input signal. This pointer, which is owned by the class, must be a superclass that hosts these three as a subclass, since it can be

a number input, a general input, and a module ending. To be able to subtract the timing chart, the arithmetic units must know the corresponding delays in terms of clock cycles. These delays will be used to overwrite the results of the chart calculated by the calculator classes over this data structure onto the object.

With the generated timing chart, the register requirement of the system can be calculated. The registers are used to move the data present in the modules. The exact lifetimes of the inputs and outputs of the modules need to be known in order to be able to identify and locate the required registers. The tool that will do the registering and locating and optimizing will do the processing by reading the delay and input output information from the arithmetic operation classes and then the same class will tell which register in which time slot the output of the input is in. Generally, the arithmetic operation class is considered as input data at the beginning and data structures which will store all the design decisions and calculations related to the automation of the system in the hardware description language with the results calculated by the auxiliary classes with the aid of this data. This class will be an abstract superclass where everything is class hierarchy, code complexity and code repetition are reduced. It will be the real class, such as addition, subtraction, multiplication, which will have the ability to write itself in the hardware language.

3.5.2 Arithmetic Input Class

As mentioned in the arithmetic unit class, the input of any arithmetic unit can be the result of another arithmetic unit, or it can be a constant number or a direct iteration input. It was decided that the arithmetic input class would be an inherited structure that could include different input types that could be added later than the three existing inputs.

3.5.3 Arithmetic Module Classes

The actual arithmetic operation classes that will be used to identify the objects will be made up of real classes that will be referred to by their own symbols. In order to reduce the complexity and repetition of the software in these classes, all common variables and data structures will come from the arithmetic unit class, which is the common ancestor. This gives the system considerable flexibility. Units with different types or the same type that may need to be added in the future and which have different properties, such as the number of inputs, etc., can be easily written and added using common mandatory constructs so that no changes need to be made in other software classes and their own constructs can be added to their class. In addition, the methods contained within these classes will be used by the firmware software printer class. Auxiliary class hardware module will write the hardware description software which we try to automate by understanding snapshots, information about input and output layouts through these methods.

3.5.4 Modulo Scheduler Class

Scheduling plan The main task of the implementing class is to extract the schedule of the system. In this timing chart, there will be information about when and what data will work with the modules, where these data will be read and where they will be written. It is complex, but has serious optimization problems. The most important input on the chart is the production volume data. In this data context, the arithmetic operations on the data structures in the current timeline are scheduled.

The first important point in the software of this timeline builder is that the total delay of a result (the motion vector and derivative variables of the resultant pixel in that project) will be determined here. To minimize this time, the operations on the longest path in the data stream should be prioritized. If we start all the inputs of all variables belonging to the longest path at the time of release of the clock at which all

inputs are ready, we always reach the earliest results and reach the best delay time. In the event that none of the arithmetic units associated with anyone are empty, we will have to wait for the transaction to occur until the nearest transaction unit. In this case, we expect to extend the total delay by the amount of time the oscillator waits. In order to reduce the possibility of this situation as much as possible, we must prevent non-priority transactions that are not on the longest route from blocking the chart. This indicates that we should schedule this process as late as possible. If these delays are not as large as they would be if they were not on the longest path and because they could not find idle sources, the whole system delay would not increase depending on them. If it comes to situations where the opposite is true, this can be a serious optimization issue. Since our goal is to create a demo that will work quickly in the first instance, such improvements will be addressed later on.

3.5.5 Register Allocation and Scheduling Class

In the hardware environment, the data is stored on multi-bit registers. When the stored data arrives, they are moved to the registers of the modules they are going to go with the cables. If the value to be used by a module is produced in that exact clock cycle, the result is taken care of by rusting only one module from the output of the other without spending any additional registers. That is, only the registers in the arithmetic operation modules are used. If there is a time difference between them, the relevant data must be stored in the registers during this time. After the scheduling plan places the implementing class processing units in the timeline,

- until the last use of the iteration inputs
- from the time the iteration output is generated until the end of iteration
- until the last use of intermediates from arithmetic processing modules

registration is required. The problem can be solved optimally by separating a register

for each hold cycle of each variable if the production volume is a clock cycle (ie the core outputs at every hour cycle). If the production volume is more than one, the register used in any clock oscillation can store other values in other oscillations, that is, it can be reused for other variables. Increasing the reusability of the registers and decreasing the idle wait time of the registers will reduce the number of registers used. In this context, if any value is stored for more than one clock oscillation, it is a method we will not prefer as a complexity at the same time by putting the same register as a block and distributing it to a different multiplier with different empty registers. Long storage has a serious optimization problem in thinking as a block. In industrial engineering, a similar problem is called bin packing.

In addition, if the storage period of a data is greater than the production volume, then it is necessary to use more than one register for that register because in such cases the system will overwrite the old value every heartbeat, and the old value will not yet expire. This can be solved by setting up the multipliers in the entrances and modifying the mode of the dispenser at each heartbeat. However, this method is a non-elegant solution that reduces the clock cycle speed of the system and increases complexity. But if we build the register series to be the production volume of the maximum length system, we can put another write in each heartbeat without using minus any minus. In this case, the number of slip registers can also be calculated as a surplus from the number of oscillations of the number of oscillations per oscillation for a variable.

3.5.6 Hardware Description Printer Class

When the main method starts to run the object of this class, all calculations and decisions required to write the hardware description software will be completed. This class prints the software that automation software will output in the direction of all saved design information and data structures. This article is written in Verilog

language. In the desired case, a new printer class can be written and a different hardware description language such as VHDL, software can be produced which will use the same data structures. Our automation software is modular at this point as well as in many stages.

The software being printed is produced according to the rules of the software template we have specified. If you need to examine the Verilog software printed in stages,

- Module names, names of parameters and input outputs, and bit widths
- Register definitions and bit widths
- Module registers, module cable definitions, module sampling
- All assignments to be made on the rising edge of the clock oscillation (ie the creation of registers)
- State machine that holds all module assignments, register assignments and in conditional operations
- Shift registers

During the writing of the state machine, each line is commented on, as well as the transaction in which the entry or exit is assigned. This provides convenience during development and possible error correction. At the same time, readability is increased. The other point that increases the readability is that the bloat is driven by the clock signal of all the registers during the writing, and all the remaining jobs have consecutive blocks.

3.6 User's Manual

In this section current version of user manual of MAFURES will be given.

3.6.1 Function Declaration Input File

FPU modules and other pipelined custom modules that will be used in flow should be introduced to tool. This introduction should include name of operation, name of different modes, its latency, module name, inputs/output port names and parameter settings in the module declarations file. Here is the sample declaration:

```
Function Name = add:0,sub:1;
// function names used in the data flow with operation number*
Latency = 16;
// latency of used module
Module Name = iter_v2_faddfsub_32ns_32ns_32_14_no_dsp;
// module name of function
Passed Parameters =
    .ID( 1 ),
    .NUM_STAGE( 14 ),
    .din0_WIDTH( 32 ),
    .din1_WIDTH( 32 ),
    .dout_WIDTH( 32 );
// passed parameters can be declared if they exist
Port Names =
// port names declared here
    .clk( clk ),
// inside of the parenthesis is pointed what is it
    .reset( rst ),
// keywords are "clk", "rst", "input:", "opcode:", and "output"
    .din0( input : 0 ),
```

```
// If there is more than one input, order of inputs should
    .din1( input : 1 ),
// be specified as ": 0", ": 1" inside the parenthesis
    .opcode( opcode: ),
    .ce( 1'b1 ),
    .dout( output: );
```

*some modules can be used as different functions (such as adder and subtructer) with different operation code, this situation supported by our tool. **at this version all ports declared 32 bit by default except clock and reset signals

3.6.2 Function Flow Input File

High level algorithm should be described with using described functions. Here is the sample function flow input:

```
initiation interval: 10
m = 588;
cons0 = 32'h3e4ccccd;
cons1 = 32'h3f000000;
a374 = mult( inputS , cons0 );
if(ii == m) {
        a374 = mult( C_input , cons1 );
}
c080 = lt( inputK , cons0 );
as335 = sub( inputW , m283 );
if (c080) {
        as335 = add( asd , cons1 );
}
```

```
if (c080 || jj == m) {
        as335 = sub( asd , cons3 );
}
u_Out = a374;
v_Out = as335;
```

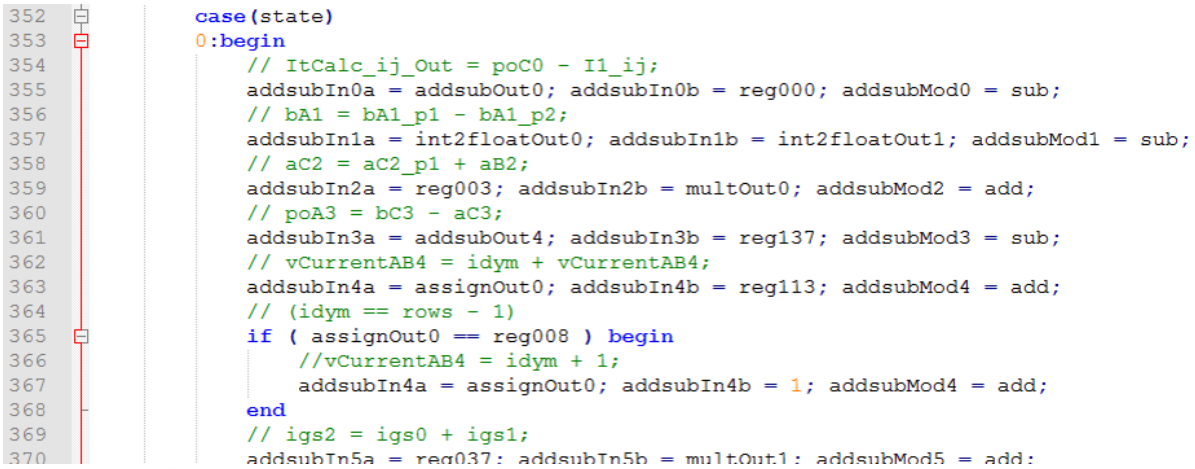
Above code sample represents an example of operation flow feeding way of our tool. Initiation interval should be declared at the first line. Introduced modules (declared in the module declaration file) can be used with their function names. Undefined names, such as inputS, C_input or asd interpreted as an input port of target module. Numbers like 32h3E4CCCCD be saved as parameter and will be described as local parameter in RTL code. Branches can be written as if statements. Variable names that include _Out (like u_Out or v_Out) interpreted as output automatically and wired outputs of Verilog module.

3.6.3 Functional Verilog Module Output

Produced RTL code is simply transformation of scheduling tables of FPUs and register.

When you run the code with valid inputs, program generate two output files which are functional Verilog module and a report file about implementation. The module is ready to instantiate into your design. Connecting inputs and output ports is enough. All inputs should be fed at cycle zero (you can start to count after reset). Outputs will be ready after calculated latency which can be found into the report file and you must collect them at that cycle.

Readability and configurability of produced RTL code is one of the important aspect of the work. With the help of simple writing style, clear naming and support of explanatory comments, understanding every single line and modifying code will be easier. Every comment describes the line below by saying assignments of which



```
352         case(state)
353         0:begin
354             // ItCalc_ij_Out = poC0 - I1_ij;
355             addsubIn0a = addsubOut0; addsubIn0b = reg000; addsubMod0 = sub;
356             // bA1 = bA1_p1 - bA1_p2;
357             addsubIn1a = int2floatOut0; addsubIn1b = int2floatOut1; addsubMod1 = sub;
358             // aC2 = aC2_p1 + aB2;
359             addsubIn2a = reg003; addsubIn2b = multOut0; addsubMod2 = add;
360             // poA3 = bC3 - aC3;
361             addsubIn3a = addsubOut4; addsubIn3b = reg137; addsubMod3 = sub;
362             // vCurrentAB4 = idym + vCurrentAB4;
363             addsubIn4a = assignOut0; addsubIn4b = reg113; addsubMod4 = add;
364             // (idym == rows - 1)
365             if ( assignOut0 == reg008 ) begin
366                 //vCurrentAB4 = idym + 1;
367                 addsubIn4a = assignOut0; addsubIn4b = 1; addsubMod4 = add;
368             end
369             // igs2 = igs0 + igs1;
370             addsubIn5a = reg037; addsubIn5b = multOut1; addsubMod5 = add;
```

Figure 6: MAFURES RTL Output Sample

operation.

3.6.4 Report File Output

Report file includes:

- Utilization of functional units
- Folded schedule table of functional units
- Folded schedule table of registers

3.6.4.1 Folded Schedule Table of Functional Units

In case of initiation interval more than one cycle, same number of arithmetic operation can be assigned a single FPU. So scheduling of FPUs for multi cycles are saved and controlled over a two-dimensional array. This array basically folded version of all schedule. Every cell stores an operation. Column location indicates assigned FPU, row location indicates start cycle modulo initiation interval. In RTL code, start cycle modulo initiation interval will correspond to state number. At that state, related inputs will be assigned to decided FPU. For the next stages of the process, life time and register scheduling of variables and inputs calculated from this table. In addition,

multiplexing assignments of used arithmetic operations modules generated from this table. Our tool allows to modify or manual creation of table by bypassing FPU scheduling part.

```
56                          <<<<<<<<<<ADD_SUB_MODULE>>>>>>>>>>
57 ********MODULE:0****************MODULE:1****************MODULE:2********
58 cycle 0-ItCalc_ij_Out***********bA1*********************aC2*************
59 cycle 1-idxmC******************idxpC******************uCurrentAB0*****
60 cycle 2-idymC******************idypC******************vCurrentAB0*****
61 cycle 3-aA0********************bA0********************vCurrentAB2*****
62 cycle 4-idxxC******************idyyC******************aC0*************
63 cycle 5-poA2*******************uCurrentAD3************vCurrentAD3*****
64 cycle 6-poA0*******************poC1*******************aA3*************
65 cycle 7-poC2*******************aC3********************vCurrentAD4*****
66 cycle 8-poC0*******************uCurrentAB2************uCurrentAD2*****
67 cycle 9-uCurrentAB1************uCurrentAD1************aA1*************
```

Figure 7: MAFURES Functional Unit Time Table Sample

3.6.4.2 Folded Schedule Table of Registers

Inputs and produced variables should be stored until their discard times. Save capacity of every register in time manner equals to initiation interval, it means registers have initiation interval slot for variable location. So registers can be used partially and partially used register can be used for their empty slots again. Register scheduling table also represented with similar two-dimensional array. Each cell in the table says name of saved input value or saved result of an operation at that cycle.

```
332 register:60
333 cycle:0->aA1_p2 cycle:1->aA1_p2 cycle:2->idle cycle:3->bA0p2 cycle:4->bA0p2
334 cycle:5->bA0p2 cycle:6->bA0p2 cycle:7->bA0p2 cycle:8->bA0p2 cycle:9->bA0p2
335 register:61
336 cycle:0->bA0p2 cycle:1->bA0p2 cycle:2->bA0p2 cycle:3->bA0p2 cycle:4->bA0p2
337 cycle:5->bA0p2 cycle:6->bA0p2 cycle:7->bA0p2 cycle:8->bA0p2 cycle:9->bA0p2
338 register:62
339 cycle:0->bA0p2 cycle:1->bA0p2 cycle:2->bA0p2 cycle:3->bA0p2 cycle:4->bA0p2
340 cycle:5->bA0p2 cycle:6->bA0p2 cycle:7->bA0p2 cycle:8->bA0p2 cycle:9->bA0p2
341 register:63
342 cycle:0->xAB1 cycle:1->xAB1 cycle:2->xAB1 cycle:3->xAB1 cycle:4->xAB1
343 cycle:5->xAB1 cycle:6->bA0 cycle:7->bA0 cycle:8->bA0 cycle:9->bA0
344 register:64
```

Figure 8: MAFURES Register Time Table Sample

3.6.5 How to Run MAFURES

- First download current version of MAFURES from following link: "https://github.com/nemesyslab/MAFURES"
- Compile java code with JRE System Library [JavaSE-1.8] or higher one.
- Create a folder and give project name to it
- Put function description (extension should be .ml) and function flow (extension should be .maf) file into your folder
- Run the executer java file with giving path of folder you create: $\gg$ java executer.java path_of_project_folder

CHAPTER IV

EVALUATION

In this chapter, there is evaluation of MAFURES tool based on a case study. ALMARVI project required an optimum FPGA implementation of optical flow algorithm. We tried to apply HLS centered design method by using MAFURES, Vivado HLS and LegUp separately.

4.1 Case Study: Optical Flow

Anisotropic Huber-L Optical Flow [2] implemented. In the implementation process we construct a high level structure and produced core module of calculations via HLS tool. At the and compare and discuss results we find.

4.1.1 Top Level Structure

The need to go over a single frame multiple times requires DRAM storage of intermediate values and an efficient interface to DRAM (i.e., the outer layer in Fig. 9). The optical flow algorithm we are using applies 500 iterations of the same procedure to every frame. Our FIPPA architecture uses multiple instances of an iteration processor. An Iter block applies one of the 500 iterations. Since 500 instantiations of Iter would not fit in most FPGAs, we need to recirculate pixels back into the pipeline (see Fig. 9). For example, p=20 pipe stages (i.e., Iter Pipeline in Fig. 9 and Fig. 10) would require 25 rounds of recirculation. At the end of every recirculation, a pixel goes into DRAM and comes back out and enters the pipe again (see Fig. 10)

Actually, at the very top of the algorithm we have frame-level pipelining (see Frame 1 versus Frame 2 in Fig. 11), then we have Iter-level pipelining (see i=1..p in Fig. 11), then we have pixel-level pipelining in each Iter block as the Iter block

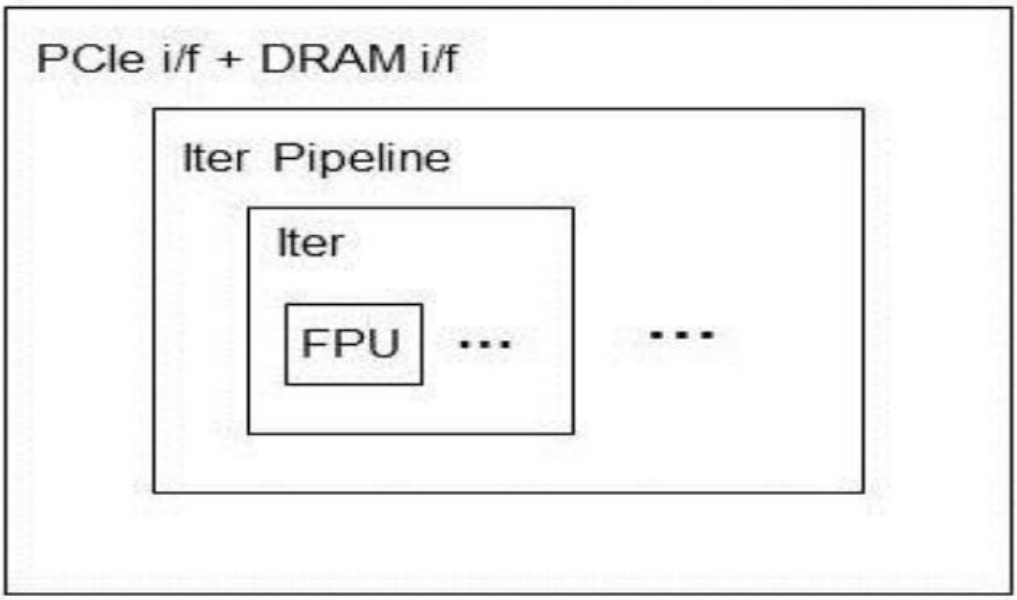


Figure 9: Layers of the FIPPA architecture

can accept a new pixel before the previous pixels processing is finished (see Fig. 12). Note that the FPUs are also pipelined.

Each Iter block uses traditional fine-grain parallelism, while we schedule computations to the associated Floating-point Units (FPUs) through Functional Loop Pipelining over c clock cycles. The pipe of Iter blocks (p stages) uses traditional (simple) pipelining in a sort of asynchronous (decoupled) manner through the line buffers in between. The ratio of p/c is determined by the amount of logic available on the FPGA. Hence, if p is increased, c also has to be increased. That is to say that if we allocate more pipe stages, then pipe stages have to have fewer FPUs, and thus, a pipe stage will have a longer processing time (cycle-time) per pixel. The smaller the cycle-time (c), the higher the utilization of FPUs will be. Hence, it is better to lower both c and p proportionally to get the highest fps from the FPGA at hand and to get that fps with the least internal FPGA power consumption. However, as c comes down, p comes down, and hence DRAM bandwidth increases. This increases DRAM

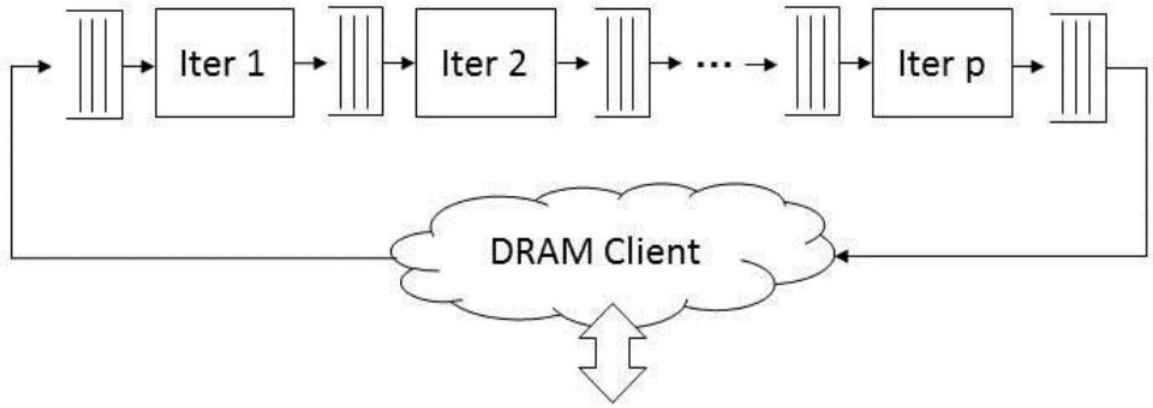


Figure 10: Iter Pipeline

cost of the system as well as DRAM power consumption, which can be a substantial percentage of the total power consumption. Therefore, as long as there is a target fps, and as long as it can be achieved with the selected FPGA, and if there is a slack in terms of FPGA utilization, the number of Iter pipe stages (p) has to be maximized to lower DRAM cost and power.

4.1.2 Using MAFURES

Above is the mindset with which we started the work presented in this thesis. That is, we neither expected HLS to solve all our problems nor expected our design to be completely push-button. We came up with an architecture for our design and looked at it to see where and how HLS could benefit us.

We were working on a video processing design, more specifically, a real-time implementation of an optical flow algorithm [22] on an Altera Arria 10 FPGA. The algorithm to be implemented is a quite long and also complex algorithm. We did not have a specific resolution and fps target. That is, we had an IP generation problem. HLS can fit well within IP generation tools.

Altera promotes synthesis from OpenCL to a GPU type of implementation [11].

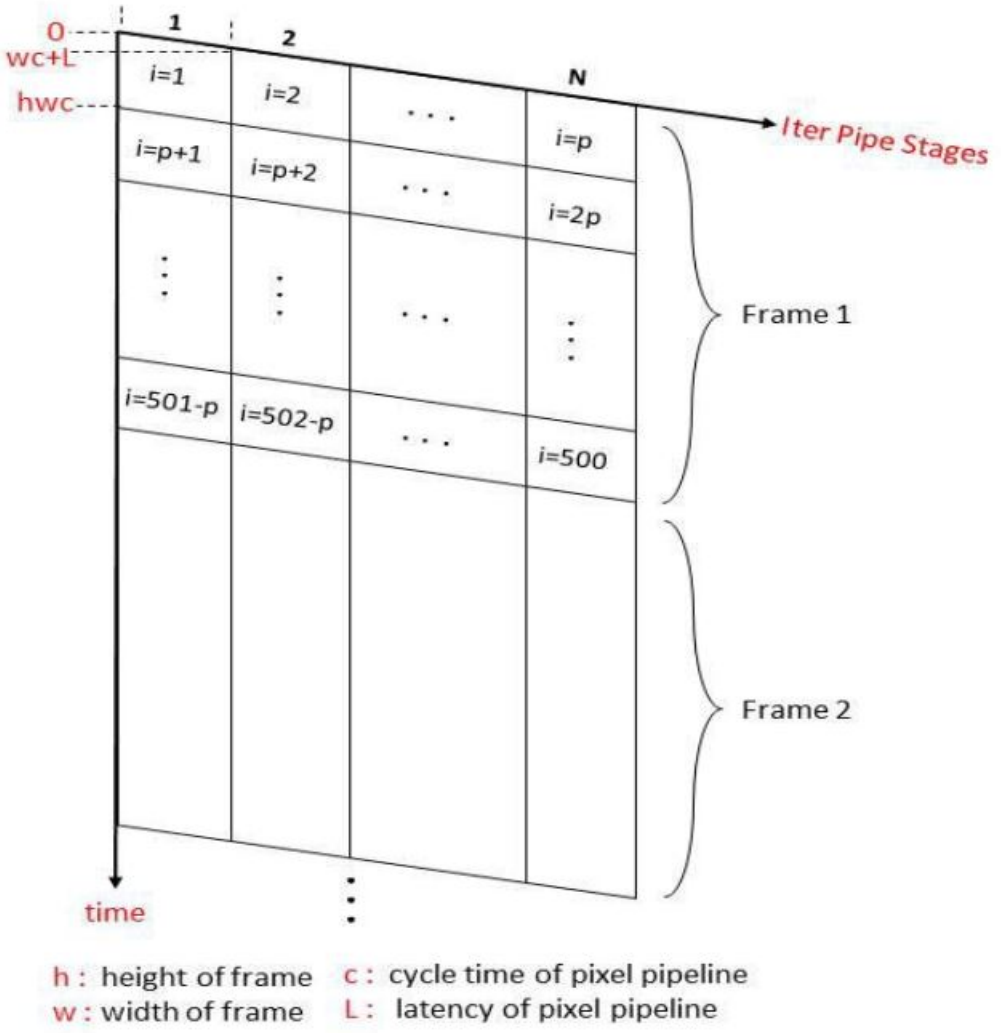


Figure 11: Iter Pipe Table

Although OpenCL and GPUs are pretty general-purpose at least within the context of video processing applications, logic design for ASIC/SoC or FPGAs offers the best efficiency and throughput, if deep pipelining is used with on-the-fly processing of streamed raster-order pixels. For that purpose, a more general-purpose HLS tool is best. Unfortunately, Altera does not offer such tool. We tried LegUp [18] but had extreme difficulty in getting it to do what we wanted. Xilinx offers such general-purpose HLS tool, namely, Vivado HLS [17]. Nevertheless, Xilinx allows Vivado HLS to be used only for Xilinx FPGAs. Therefore, we came up with our own domain

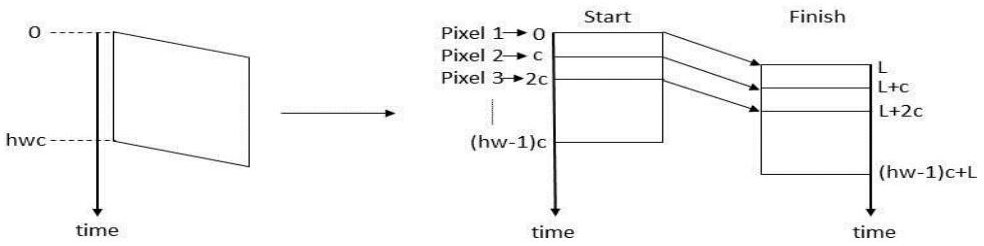


Figure 12: Timing of Pixel Pipelining

specific HLS tool, which we call MAFURES. By the time, we were finished, we had realized that MAFURES has a pretty broad scope although it is not totally general-purpose. Later on, we decided to implement our design also on Xilinx FPGAs so that we could benchmark MAFURES against Vivado HLS or vice versa. We had much easier time with Vivado HLS when trying to get it to do what we were looking for.

Fig. 13 shows a high-level view of the optical flow algorithm we were expected to implement. Optical flow finds the motion in a video frame at every pixel. Every incoming frame goes through P and then V (thus Frame Loop). P is pyramid formation, i.e., creation of k downscaled versions of the frame, with kth being the smallest. V is motion vector computation, which is internally an iterative algorithm with 4 nested loops. V finds the motion vectors for each downscaled frame (hence VLk..0) and then upscales it (US in Pyramid Loop) and uses it as a starting point for the next (bigger) downscaled version (thus Pyramid Loop). The pyramid loop in V has one more iteration as it eventually operates on the original frame (VL0). For every downscaled version of the incoming frame, we have an iterative optimization algorithm with mq iterations. After every q iterations, we warp (the W in Iteration Loop) the previous frame and hence zero out the motion vector. This enhances the precision of the algorithm. And finally, every iteration goes through all r pixels in a frame (thus Pixel Loop).

Fig. 14 and 15 show how we schedule the algorithm in Fig. 13. We pipeline

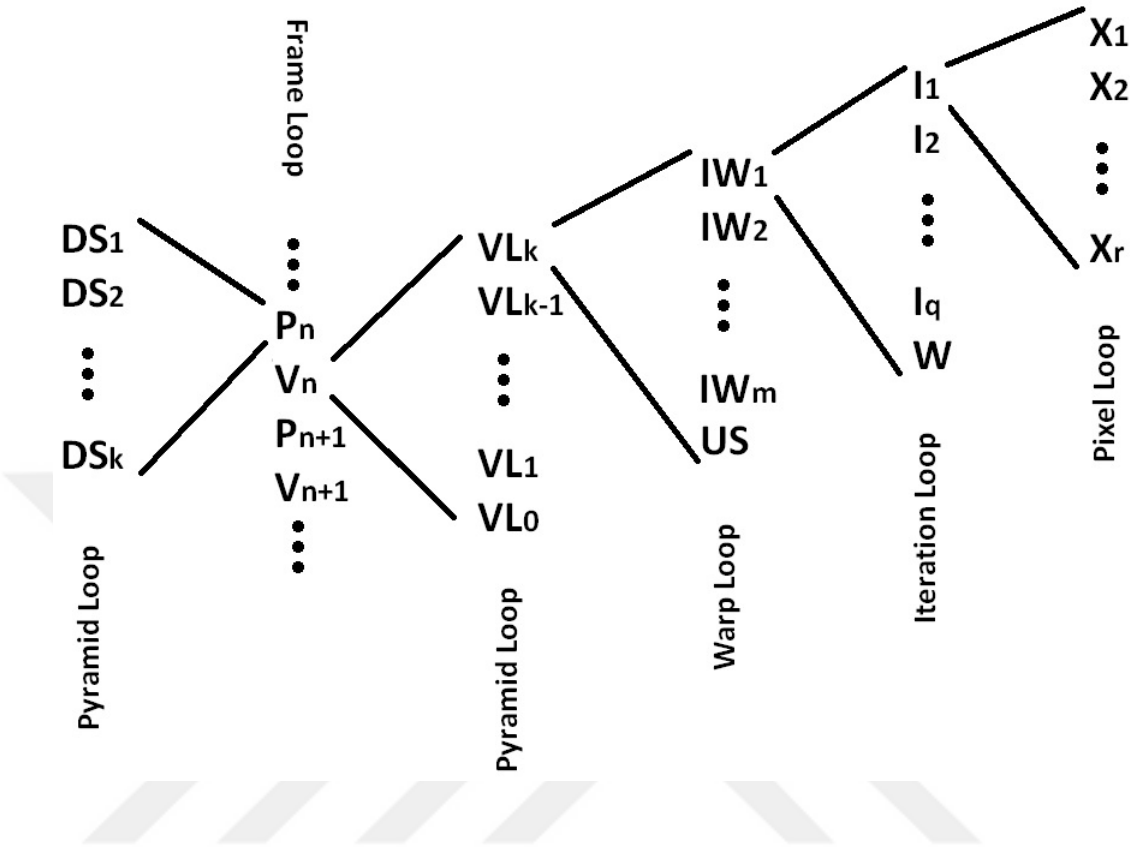


Figure 13: Optical Flow Algorithm

the Frame Loop, Iteration Loop, and Pixel Loop but not the Pyramid Loop nor the Warp Loop. For the Iteration Loop, we use a simple pipeline of identical blocks (which we call iter-block) cascaded through line buffers. For the Pixel Loop, hence the design of iter-block, we use functional pipelining. That is, using HLS we schedule a particular instance of X (which has 84 floating point operations) with a particular initiation interval (i.e., cycle time) and synthesize a datapath that implements it. This is equivalent to unrolling and rerolling the loop or folding it.

Our design and verification flow uses as the golden reference a MATLAB model coming from the image processing engineers of our extended team. The model uses 64-bit floating point. However, we decided to do the FPGA implementation with 32-bit floating point. We wrote a Java model, which can be configured to be 64 or 32 bits. Its 64-bit version was tested against the MATLAB model, and its 32-bit

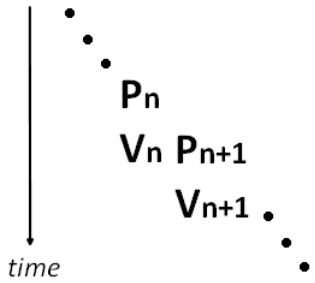


Figure 14: Pipelining Frame Loops

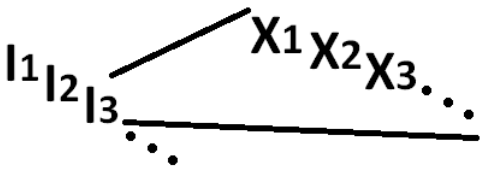


Figure 15: Pipelining Pixel Loops

version became our new golden reference. We have written some Perl scripts for architectural planning through which we can experiment with different scenarios in terms of Floating Point Units (FPUs) we use, number of iter-blocks in the pipeline, their initiation interval, FPUs throughputs and latencies.

Its main advantage compared to Vivado HLS is that it produces much more readable and compact (1453 vs 2766 lines) RTL (input to logic synthesis), which is hence easier to debug and customize if needed, in shorter runtimes. On top of that, it can pipeline the muxes resulting from resource sharing as well as data select operations to achieve greater clock frequencies. When muxes are allowed only one cycle of their own

instead of two cycles, MAFURES usually uses fewer FPGA slices for a similar timing. Unlike Vivado HLS, one can use MAFURES to target Altera FPGAs and is pretty indifferent to where its FPU implementations come from. In summary, MAFURES offers the following:

- Around 2 times more compact RTL output
- Shorter runtimes (1-2 vs 10-15 sec)
- MUX Pipelining for improved timing or area
- Targets both Xilinx and Altera even ASIC
- FPUs can be imported from various sources

4.1.3 Synthesis Results

We have obtained two types of synthesis results for our optical flow design with both MAFURES and Vivado HLS. One is for a timing-oriented design (i.e., fastest clock frequency possible). The other is where a mediocre timing target of a 4 ns clock period is set and an area-efficient result is sought. To be able to compare oranges with oranges, we made MAFURES use the same FPUs Vivado HLS picked. We also used Vivado Logic Synthesis on the RTL both tools produced (with Retiming turned on).

Table 1: Time Driven Synthesis Results of Optical Flow

$HLSTool$	$ClkPeriodAchieved$	$ClkPerioddduetoFPUs$	$\#Slices$
MAFURES	2.8	2.2	2899
Vivado HLS	3.1	2.2	2719

For the results above we targeted a Xilinx Kintex-7 FPGA (XC7K160TFBG484-1). We obtained the above results with some search involved with Vivado Logic

Table 2: Area Driven Synthesis Results of Optical Flow

HLSTool	*ClkPeriodAchieved*	*ClkPeriodduetoFPUs*	*#Slices*
MAFURES	3.7	3.0	1893
Vivado HLS	3.7	3.0	2203

Synthesis in terms of target clock period (at 10 min for each run). The results show that we are able to offer 10% better timing with 6.6% more slices when over 300 MHz is desired. When an easily achievable speed of 250 MHz is targeted for our design, then MAFURES offers 14% improvement in the number of FPGA slices used. The reason we cannot achieve the clock periods dictated by the FPUs is because our critical path are the integer compare ops that drive our data select ops, which are not pipelined.

4.2 Comparison

This automation tool we have produced in the context of the project has come to a level comparable to the general purpose high level synthesis tools when viewed from the big picture. Although we are specialized in functional and loop pipeline design structures that can be intertwined due to the iteration design we are working on automation, it also has a general purpose as well as being able to receive input as high level software and support any simple arithmetic operation. The fact that our high-level tool is academically open-sourced is a key value in the project, as it is a basic structure that allows for change and experimentation as desired, open to developing and developing practical training for developers. When we look at the engineering perspective, it is not a matter to judge the necessity of our tool only by what we do and by looking at academic researches. Having a conviction in this regard should take into account the sector, how the current commercial or open source tools can do what they can, how they can do what we do, their performance in various

ways, their ease of use and development. The development of our tool can only be rated by comparing the available tools with the ones we have just considered.

For this comparison, we tried to produce the iteration design which is the basis of our project in Verilog software with the tools we will discuss below. We compared the generated Verilog software functionally and performance with our own. We will also talk about the experiences and difficulties we experience during the synthesis of the iteration software in these tools.

4.2.1 Comparison with LegUp HLS

The LegUp HLS tool supports loop pipeline construction and functional pipeline construction on our own automation tool. Besides that, it can produce FPGA and processor mix hybrid designs as well as design for entire FPGA. It is more like an accelerator in this regard.

The current structure in our project is designed to talk over the FIFO memory structures between each other. That is, the intermediate module that we will set up with the iteration function must feed the FIFOs and the values it produces again. The first important issue that we encountered when giving the code of the Iteration code to LegUp in C code as a high level is to manage the input and output of Verilog modules. As his documents say, input and output were written as arguments to the written functions. If an argument is called by name, it is said to be an input, and if it is called by its mark, it is treated as an output.

In order to understand how to add the Verilog module to our own projem, we tried to do a simple design and synthesize a Verilog before it was created. We tried to compile a simple function by writing input arguments as function arguments. Our goal was to let LegUp give us only one module that does the job. At this point LegUp sent us a warning message and demanded the 'main' function from us. As you can see, LegUp was just as a self-contained, full-functioning system. It did not compile

for a single C function that did all the work.

We recompiled our function by calling it through the 'main' function. LegUp had a lot of different and mixed structures inside the Verilog code that it extracted. We have reviewed the code that is extremely difficult for a stranger to be readable and reorganized. However, we could not find a corresponding Verilog module for the function we built as input, or a structure corresponding to it as input and output. When we bend over a bit more and find out that everything is addressed to a memory unit made up of registers built in the main module without being made into a separate Verilog module (as opposed to documentation), created in the main module. In this case it was almost impossible to pull out a module from the inside. It wrote that various interfaces could be used to transfer data from related documents, but adapting them to the current project was a problem that would be a problem. In addition, LegUp allows you to add Verilog modules from the outside, but it also says that designs with external modules cannot be turned into a loop pipeline.

LegUp gave the error that we could not do it because of dependencies when we wanted to put the function containing the iteration software into the loop pipeline. We tried again by deleting the related dependencies from the algorithm (hopefully we can find a different solution for this problem later). This time with the square root extraction algorithm in the loop, the loop gives the warning that the pipeline cannot function. In the next test we will also remove the square root from the algorithm, LegUp collapses during compilation without giving a new error or warning. At this point we have given up and started to work on another high-level synthesis tool Vivado HLS.

4.2.2 Comparison with Vivado HLS

Compared to LegUp HLS, the process is much more comfortable. A single C function written in Vivado HLS can be compiled directly. The compilation result is a Verilog

module with the same name as the name function. The input and output definitions of the generated module are derived from the arguments of the function. The ones called by name are input, and those called by mark are set as output. In addition, for each output, a 'CALL' signal has been generated indicating that the output is ready to print. In addition, the 'READY' and 'START' signals for general use were made available to allow the module to be moved from the outside.

When the iteration code is written as a high level within this C function, we got an error message in the absolute value function. The absolute value was compiled without problem after rewriting the calculation as a manual condition instead of a function. We then selected the loop pipeline adjustment and the demanded production volume. We recompiled after setting a target for the clock frequency. The result seemed positive.

We wrote a simple shell module around the resulting Verilog module and attached it to FIFO structures. We went through the verification software after a simple design that will start and stop the ready signals. The Vivado HLS production module passed the verification software.

There was a comparison of the performance of the Vivado HLS tool with our own tool. The key point of this comparison was the floating-point arithmetic processing units to be used. We have used the same arithmetic processing units in both designs so that there is no injustice at this point and that the tools can compete fairly. To do this, we first synthesized the Vivado HLS tool and let it use its own arithmetic units. After the synthesis of Vivado HLS, the units that came out exported our own tool and made our own synthesis according to the structure of these units. As you see in previous section MAFURES can get better timing results with MUX pipelining. When MUX pipelining feature was disabled, MAFURES can get better area results. Comments and case statement structure make MAFURES better in terms of readibility.

```
1941  always @ (posedge ap_clk) begin
1942      if ((ap_const_logic_1 == ap_sig_cseq_ST_pp0_stg4_fsm_4)) begin
1943          ap_reg_ppstg_d074_reg_1510_pp0_it12 <= d074_reg_1510;
1944          ap_reg_ppstg_d074_reg_1510_pp0_it13 <= ap_reg_ppstg_d074_reg_1510_pp0_it12;
1945          ap_reg_ppstg_d074_reg_1510_pp0_it14 <= ap_reg_ppstg_d074_reg_1510_pp0_it13;
1946          ap_reg_ppstg_d074_reg_1510_pp0_it15 <= ap_reg_ppstg_d074_reg_1510_pp0_it14;
1947          ap_reg_ppstg_d074_reg_1510_pp0_it16 <= ap_reg_ppstg_d074_reg_1510_pp0_it15;
1948          ap_reg_ppstg_d074_reg_1510_pp0_it17 <= ap_reg_ppstg_d074_reg_1510_pp0_it16;
1949          ap_reg_ppstg_d074_reg_1510_pp0_it18 <= ap_reg_ppstg_d074_reg_1510_pp0_it17;
1950          ap_reg_ppstg_d074_reg_1510_pp0_it19 <= ap_reg_ppstg_d074_reg_1510_pp0_it18;
1951          ap_reg_ppstg_d074_reg_1510_pp0_it20 <= ap_reg_ppstg_d074_reg_1510_pp0_it19;
1952          ap_reg_ppstg_d074_reg_1510_pp0_it21 <= ap_reg_ppstg_d074_reg_1510_pp0_it20;
1953          tmp_13_reg_1437 <= tmp_13_fu_696_p2;
1954      end
1955  end
1956
1957  always @ (posedge ap_clk) begin
1958      if (((ap_const_logic_1 == ap_reg_ppiten_pp0_it15) & (ap_const_logic_1 ==
          ap_sig_cseq_ST_pp0_stg7_fsm_7))) begin
1959          as049_1_reg_1596 <= grp_fu_414_p2;
1960      end
1961  end
1962
1963  always @ (posedge ap_clk) begin
1964      if (((ap_const_logic_1 == ap_reg_ppiten_pp0_it15) & (ap_const_logic_1 ==
          ap_sig_cseq_ST_pp0_stg5_fsm_5))) begin
1965          as049_reg_1591 <= grp_fu_438_p2;
1966      end
1967  end
```

Vivado HLS RTL Output

```
352          case(state)
353          0:begin
354              // ItCalc_ij_Out = poC0 - I1_ij;
355              addsubIn0a = addsubOut0; addsubIn0b = reg000; addsubMod0 = sub;
356              // bA1 = bA1_p1 - bA1_p2;
357              addsubIn1a = int2floatOut0; addsubIn1b = int2floatOut1; addsubMod1 = sub;
358              // aC2 = aC2_p1 + aB2;
359              addsubIn2a = reg003; addsubIn2b = multOut0; addsubMod2 = add;
360              // poA3 = bC3 - aC3;
361              addsubIn3a = addsubOut4; addsubIn3b = reg137; addsubMod3 = sub;
362              // vCurrentAB4 = idym + vCurrentAB4;
363              addsubIn4a = assignOut0; addsubIn4b = reg113; addsubMod4 = add;
364              // (idym == rows - 1)
365              if ( assignOut0 == reg008 ) begin
366                  //vCurrentAB4 = idym + 1;
367                  addsubIn4a = assignOut0; addsubIn4b = 1; addsubMod4 = add;
368              end
369              // igs2 = igs0 + igs1;
370              addsubIn5a = reg037; addsubIn5b = multOut1; addsubMod5 = add;
```

MAFURES RTL Output

Figure 16: Vivado HLS RTL vs MAFURES RTL

CHAPTER V

CONCLUSIONS & FUTURE WORK

In this thesis, we have shared our experience in using HLS for a challenging and high-throughput video processing FPGA design. We have obtained competitive results with Vivado HLS and even more competitive results with our own HLS tool (MAFURES), which does the job in our case. MAFURES can also target Altera FPGAs and ASICs and has a few more desirable features. We plan to add features to MAFURES and make it more competitive.

Developments planned to be made on the MAFURES can be viewed under following titles:

- Front-end Compiler Support
- Generating More Synthesis Option Constraints
- Implementations of Different Scheduling and Allocation Algorithms both Registers and Building Modules
- Top Level Wrapper Generator
- Test Bench Generator
- User Friendly Graphical User Interface

REFERENCES

[1] "http://www.almarvi.eu/,"

[2] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, and H. Bischof, "Anisotropic huber-l1 optical flow.," in *BMVC*, vol. 1, p. 3, 2009.

[3] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.

[4] M. Meredith, "A look inside behavioral synthesis," *EEdesign. com*, pp. 04–08, 2004.

[5] B. Bowyer, "The 'why' and 'what' of algorithmic synthesis," May 2005.

[6] E. A. Snow, D. P. Siewiorek, and D. E. Thomas, "A technology-relative computer-aided design system: Abstract representations, transformations, and design tradeoffs," in *Proceedings of the 15th Design Automation Conference*, pp. 220–226, IEEE Press, 1978.

[7] S. Director, A. Parker, D. Siewiorek, and D. Thomas, "A design methodology and computer aids for digital vlsi systems," *IEEE Transactions on Circuits and Systems*, vol. 28, no. 7, pp. 634–645, 1981.

[8] H. F. Ugurdag and T. E. Fuhrman, "Autocircuit: a clock edge general behavioral synthesis system with a direct path to physical datapaths," in *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on*, pp. 514–523, IEEE, 1996.

[9] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, pp. 1591–1604, 2016.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[11] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 531–534, IEEE, 2012.

[12] H. F. Ugurdag, "Experiences on the road from eda developer to designer to educator," in *Design & Test Symposium, 2013 East-West*, pp. 1–4, IEEE, 2013.

[13] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipa cc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2016.

[14] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich, "Fpga-based accelerator design from a domain-specific language," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pp. 1–9, IEEE, 2016.

[15] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, 2014.

[16] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 120–127, IEEE, 2013.

[17] "Vivado design suite user guide: High-level synthesis," Tech. Rep. UG902 v2014.3, Xilinx Inc., Sept. 2014.

[18] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, ACM, 2011.

[19] A. E. Guzel, V. E. Levent, M. Tosun, M. A. Özkan, T. Akgun, D. Büyükaydin, C. Erbas, and H. F. Ugurdag, "Using high-level synthesis for rapid design of video processing pipes," in *East-West Design & Test Symposium (EWDTS), 2016 IEEE*, pp. 1–4, IEEE, 2016.

[20] M. Büyükmıhçı, V. E. Levent, A. E. Guzel, O. Ates, M. Tosun, T. Akgün, C. Erbas, S. Gören, and H. F. Ugurdag, "Output domain downscaler," in *International Symposium on Computer and Information Sciences*, pp. 262–269, Springer, 2016.

[21] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral-level multicycle path analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1832–1845, 2014.

[22] D. Büyükaydin and T. Akgün, "Gpu implementation of an anisotropic huber-l1 dense optical flow algorithm using opencl," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 326–331, IEEE, 2015.

[23] H. W. Lawson Jr, "New directions for micro-and system architectures in the 1980s," in *Proceedings of the May 4-7, 1981, national computer conference*, pp. 57–62, ACM, 1981.

VITA

Name: Aydın Emre Güzel

Date of Birth: 15/02/1991

Birth Place Antalya

Languages: Turkish, English

Education

- MS: Ozyegin University Computer Sciene 2016
- BS: Ozyegin University Electric Electronics Engineering 2014
- High School: Adem Tolunal Anatolian High School

Work Experience

- Vestek Research and Development Corp.
  Summer Internship 2012
- Vestek Research and Development Corp.
  Summer Internship 2013

Publications

1. A.E. Guzel, V.E. Levent, M. Tosun, M.A. Ozkan, T. Akgun, D. Buyukaydin, C. Erbas, H.F. Ugurdag, Using High-Level Synthesis for Rapid Design of Video Processing Pipes, Proc. of East-West Design & Test Symposium (EWDTS), Yerevan, Armenia, October 2016.

2. M. Tosun, M.A. Ozkan, A.E. Guzel, H.F. Ugurdag, Fast One- and Two-Pick Fixed-Priority Selection and Muxing Circuits, Proc. of East-West Design & Test Symposium (EWDTS), Yerevan, Armenia, October 2016.

3. M. Buyukmihci, V.E. Levent, A.E. Guzel, O. Ates, M. Tosun, T. Akgun, C. Erbas, S. Goren, H.F. Ugurdag, Output Domain Downscaler, Proc. of International Symposium on Computer and Information Sciences (ISCIS), submitted, Krakow, Poland, October 2016.

4. A. Kakacak, A.E. Guzel, O. Cihangir, S. Goren, H.F. Ugurdag, Fast Multiplier Generator for FPGAs with LUT based Partial Product Generation and Column/Row Compression, Accepted in Integration, the VLSI Journal.

Honors and Awards

- The top 2nd BS alumnus in the entire graduating class of 2014 at Ozyegin University with a CGPA of 3.76 out of 4.00
- High Honor awarded by Ozyegin University between 2010 and 2013