POLITECNICO DI TORINO

Master Degree in
Computer Engineering - Software

Master Degree's Thesis

# Advanced C++14 Multithreading Modelling of Electronics Systems


Supervisors:
prof.re Edgar Ernesto Sanchez Sanchez
prof.re Alessandro Savino prof.re
Michele Portolan

Candidate:
Hesamoddin Fathollahi

# Acknowledgements

Thanks to my mother and father for encouraging me to undertake the university experience in a country other than that my country, gives another vision to another interest and a wonderful world to me, furthermore, thanks for subsequently supporting all my choices.

Special thanks to my supervisor Prof. Alessandro Savino and Prof. Ernesto Sanchez and Prof. Michele Portolan who followed me in the various work steps, directing and advising me to reach the final goal.

# Abstract

By increasing chips complexity, the demand for designing automation on a higher abstraction level is increased, making the functionality simply comprehended and a more influential trade-off. Automating part of the whole design procedure and updating it to upper levels includes numerous benefits. First, a much shorter design cycle is assured by automation. Then, it could be possible to examine various styles in designing as a result of quick generation and evaluation of different designs. The aim of this thesis adds Thread C++14 standard library to reach a multi-thread program and parallelism by Babmu which is a High-Level Synthesis framework.

# Contents

**Acknowledgements** i

**Abstract** ii

**1 Introduction** 1
- 1.1 Purpose . . . 1
- 1.2 Work introduction . . . 2
- 1.3 Thesis Overview . . . 3

**2 High-Level Synthesis and Panda Bambu** 5
- 2.1 Introduction . . . 5
- 2.2 The synthesis process . . . 5
- 2.3 Bambu . . . 8
  - 2.3.1 Work flow . . . 8
  - 2.3.2 Input and compile source code . . . 11
  - 2.3.3 FRONTEND ANALYSIS and Memories Allocation . . . 12
  - 2.3.4 HIGH-LEVEL SYNTHESIS OF THE FUNCTIONS . . . 14
  - 2.3.5 Netlist Generation . . . 17
  - 2.3.6 Generation of Synthesis and Simulation Scripts . . . 17
  - 2.3.7 Testbenches . . . 18
  - 2.3.8 A Datapath Structure . . . 18
- 2.4 Function call mechanism . . . 19

**3 Thread** 22
- 3.1 What Is a Thread? . . . 22
  - 3.1.1 Kernel threads . . . 25
  - 3.1.2 User threads . . . 25
  - 3.1.3 Hybrid or mixed thread(M:N) . . . 26
- 3.2 Standard C++ Thread Library . . . 26

**4 Modification** **29**
4.1 Bambu Libraries . . . . . 31
4.2 Thread library with context switching . . . . . 32
4.3 Thread Library . . . . . 37
4.4 Parallel Programming . . . . . 38
4.5 Proposed Architecture . . . . . 45
4.5.1 Distributed Controller Architecture . . . . . 46
4.5.2 Memory Interface . . . . . 48
4.5.3 HLS example . . . . . 50
4.5.4 Challenges and Solutions . . . . . 52

**5 Resuls** **53**

**6 Conclusion** **58**

**A Modification** **64**

**B Resuls** **72**

# List of Figures

1.1 Evaluation of VLSI . . . . . 2

2.1 HLS Workflow . . . . . 6
2.2 Front End . . . . . 9
2.3 Middle-End . . . . . 10
2.4 Back-End . . . . . 10
2.5 Bambu Workflow . . . . . 11
2.6 Convert source code to IR . . . . . 12
2.7 FRONTEND ANALYSIS . . . . . 13
2.8 Internal Memories . . . . . 13
2.9 Externla Memories . . . . . 14
2.10 Shows the HLS process in Bambu flow. . . . . 15
2.11 Call Graph . . . . . 19
2.12 Function Call with and Without Function Proxy . . . . . 20
2.13 Indirect function call mechanism . . . . . 21

3.1 Sequential Program Work Flow . . . . . 22
3.2 Multi and Single Thread program . . . . . 23
3.3 Thread Shared Resources . . . . . 23
3.4 User-Level and Kernel-Level Thread . . . . . 24
3.5 Thread Join . . . . . 27

4.1 HLS Flow In Previous Thesis . . . . . 29
4.2 HLS Flow In Thesis . . . . . 30
4.3 Thread Library with Context Switching . . . . . 32
4.4 Thread Life Cycle . . . . . 36
4.5 Multi Threading Schema In OpenMP . . . . . 39
4.6 Sequential Code With Control Flow Graph(A) . . . . . 41
4.7 The Control Dependency Graph . . . . . 43

4.8 Parallel code With Task Graph(A) . . . . . 44
4.9 Extended Program Dependencies Graph . . . . . 46
4.10 Distributed Controller Architecture Correspond to Figure 4.9 . . . . 47
4.11 Memory Interface Controller(MIC) Schema . . . . . 50

5.1 Simulated Model Latency with 4 Parallel Accelerator . . . . . 56
5.2 Simulated Model Latency with 2 Parallel Accelerator . . . . . 56

# Listings


4.1 Default configuration for installing Bambu . . . . . 32
4.2 Sample Code For Context Switching . . . . . 33
4.3 Constructors for User-Level thread . . . . . 34
4.4 Sample code . . . . . 36
4.5 New Thread Library . . . . . 37
4.6 Sample code by OpenMP . . . . . 39
4.7 New Solution for Multi-Threading . . . . . 45
4.8 Sample Code to Synthesized With OpenMP . . . . . 51
4.9 Sample Code to Synthesized With OpenMP After High Level Synthesis transformations . . . . . 51
A.1 Thread Library . . . . . 64
A.2 Deque Library . . . . . 68
A.3 Structures in Thread Library . . . . . 71
B.1 Multi Threaded Sample Code . . . . . 72

# List of Tables

4.1 API for Deque Library . . . . . 35

5.1 Simulation for parallel and Sequential tests with 2 and 4 Accelerator 53
5.2 Summary of resources . . . . . 55

# Chapter 1

# Introduction

## 1.1 Purpose

In 1965, Intel co-founder Gordon Moore was asked by Electronics Magazine to summarize the electronics industry state and a prediction about the semiconductor components industry in the future. Because of this, Moore published a paper[**1**] entitled "Cramming more components onto integrated circuits".

While writing the paper, Moore mentioned a double increase in the number of devices inside the chips (including resistors and transistors) each year, when the size of transistors could be shrunk by the engineers. It was indicated that the capabilities and performance of semiconductors were exponentially and continually growing. Moore modified the law to represent the double increase in the number of transistors every 24 months, in 1975.

In the early 1970s, Very-large-scale integration (VLSI)[1][**2**] was appeared when developing communication technologies and complex semiconductors.

VLSI technology makes it possible to have compression of millions of multiple gates of logic for a chip that is extremely complex limitations for what can and cannot be made. More than 600 rules may be included in an integrated circuit process from 2006. Such complexity requires a very difficult chip, if possible, for designing through the traditional capture and simulated design practice. The evaluation of VLSI where the number of transistors in are increased every year is shown in the figure1.1.

[1] Very-large-scale integration (VLSI) is the procedure to create IC(integrated circuits) by integration of several circuits based on the transistor into a single chip.

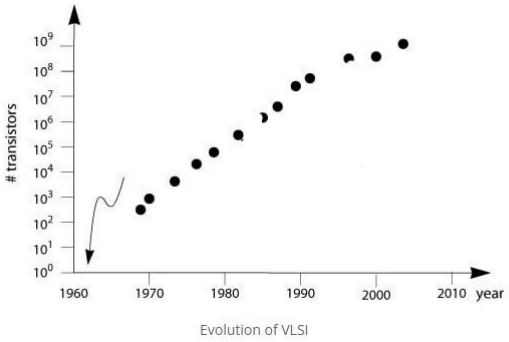



FIGURE 1.1: Evaluation of VLSI

The product development cycle was comprehensively considered by the industry to decrease the time in designing and reach a competitive edge in the time-to-market race. By increasing chips' complexity, the demand for designing automation on a higher abstraction level is increased, making the functionality simply comprehended and a more influential trade-off. Automating part of the whole design procedure and updating it to upper levels includes numerous benefits. First, a much shorter design cycle is assured by automation. Then, it could be possible to examine various styles in designing as a result of quick generation and evaluation of different designs. Ultimately, average human designers may be outperformed by design automation tools to meet most design requirements and constraints.

## 1.2 Work introduction

Chips become more complex so it is so important to have design automation in the upper level of abstraction, where the trade-off is more effective and functionality is simpler for recognizing. This process (automating all or a part of the design process and shifting automation to the upper levels) has many advantages when it least to have different styles in design style and it makes the faster designing process. Furthermore, we could reach a better output by design automation in comparison to previous methods for designing.

Computer-aided tools are developed with success in different parts of the development cycle, they are used when in the design of microelectronic circuits they become useful. These tools, for instance, are used in designing of chemical process**[3]**, synthesis of heat exchanger network and simulation**[4]**.

The most important advantage of the synthesis techniques could be mentioned is increasing the speed of the designing cycle, while human attendance decreases. The quality of the design cycle could improve by optimization techniques when these days synthesis and optimization techniques are used for almost all digital circuit designs.

Register transfer language like VHDL[**5**] or Verilog[**6**] in computer science is an IR (Intermediate Representation) which is similar to the assembly language and used for simulating a data flow at the register transfer level of architecture.

However, nowadays, designing an electronic system is made with a Register Transfer Language that is not a fast and easy process to validity in the model for having symmetrical interplay among components in RTL and guarantee in they work correctly, the most important part is having concurrency in control flow and data, between components and their interaction in both High-Level Synthesis(HLS)[**7**] and Verification.

The goal of this thesis is with considering an existing strong HLS framework, Bambu[**8**] which is made by c++ language, adding The standard C++14 thread library to the library of Bambu framework (libBambu), next performing HLS with this library for having parallel executing blocks by thread library in the Bambu output.

## 1.3 Thesis Overview

The thesis contains 6 chapters describing in detail the HLS and Bambu and appendix for C++ source codes.

Chapter 1 is introduction about HLS and its history and its role in nowadays technologies.

Chapter 2 is a describing the basic concepts that are essential for workflow and the functionality of Bambu as HLS compiler in the thesis.

Chapter 3 describes threads and their different types, also about their functionalities, and introduces the C++ standard thread library.

Chapter 4 describes the proposed methodology and approach to reach the desired result and another considered method to achieve the desired result.

Chapter 5 shows and the obtained results , comparing results in terms of latency, area, and summary of resources that obtained by different simulation configs.

Chapter 6 is discuss about the thesis conclusion and future works.

# Chapter 2

# High-Level Synthesis and Panda Bambu

## 2.1 Introduction

By the synthesis methods, the design cycle is accelerated by reducing human efforts. The design quality is enhanced by optimization methods. Presently, optimization and synthesis procedures are utilized for most digital circuit designs. Nonetheless, their power is not yet completely exploited and most of the work is still manually made.

## 2.2 The synthesis process

Synthesis is the creation of a circuit model, initiating from a less detailed one. Abstraction levels and views are considered to classify the model. Three main abstractions are considered in the present work :

- Architectural: Circuit performs a set of operations, such as data computation or transfer at the architectural level
- Logic: A set of logic functions is evaluated by a digital circuit at the logic level.
- Geometrical: A circuit is a set of geometrical entities at the geometrical level.

The creation of a structural view is included in an architectural-level synthesis for an architectural-level model. Thus, an allocation of the circuit functions to operators is obtained to determine the resources, and their interconnection while timing

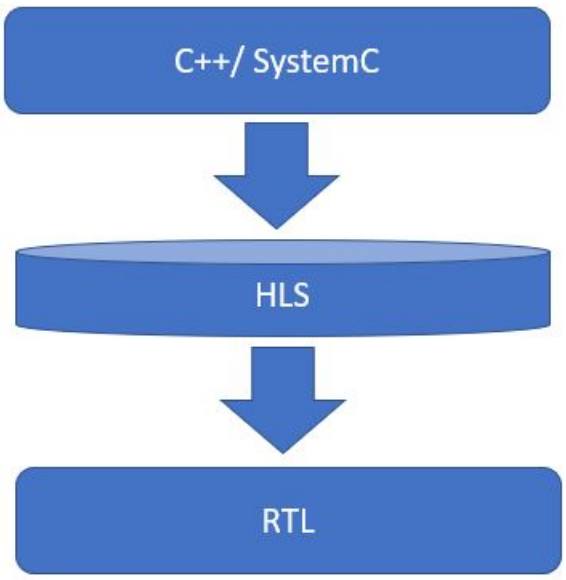



FIGURE 2.1: HLS Workflow

their execution. It is also known as high-level synthesis or structural synthesis, as a result of determining the circuit's macroscopic structure.

As a set of dependencies and operations, a behavioral architectural-level model can be distracted. Architectural synthesis includes the identification of the hardware resources for implementing the operations, arrangement of the operations' execution time, and connecting them to the resources. Hence, a structural model of a data path is defined by synthesis to interconnect the resources, and a logic-level model of a control unit representing the control signals to the data path based on the schedule.

The logic-level synthesis step should be performed followed by the architectural-level synthesis. Logic-level synthesis is the task of the generation of a structural view of a logic-level model. Logic synthesis manipulates the logic specifications for creating the logic models as interconnecting the logic primitives.

Hence, the microscopic structure of a circuit is determined by logic synthesis. To transform a logic model into an interconnection of instances of library cells, the logic synthesis back end, is often represented as technology mapping or library binding.

A state transition diagram of a finite-state machine[1] can provide a logic-level

[1] A FSM or Finite State Machine, is a model for calculation for showing and controlling execution flow and simulating sequential logic. The FSM is also useful to model problems in different forms such as linguistics, artificial intelligence, mathematics, or games.

model of a circuit equivalence by an HDL model or by a circuit schematic. It may be synthesized from an architectural-level model or specified by a designer. Considering the circuit's nature (combinational or sequential) and the initial representation (the schematic or state diagram), the logic synthesis tasks may be different. Since there are numerous possible configurations of a circuit, optimization has a key role, in connection with synthesis to determine the microscopic figures of the implementation merit. A completely structural representation is obtained such as a gate-level netlist as the ultimate consequence of logic synthesis.

The last step is the geometrical-level synthesis, which includes the creation of a physical view at the geometric level. It involves specifying all geometric patterns to define the chip's physical layout and position. This is often known as physical design. The physical design includes the generation of the layout of the chip.

The layout's layers are based on the masks utilized to fabricate the chip. Thus, the final target of microelectronic circuit design is the geometrical layout. Physical design is further based on the style of design. On one end of the spectrum, physical design is handcrafted for custom design via layout editors. It means that the use of automated synthesis tools is renounced by the designer in the search to optimize the circuit geometries by fine hand-tuning. Regarding prewired circuits, physical design is conducted on the opposite end of the spectrum, virtually, since the chips are manufactured fully in advance.

Wiring and placement are the major tasks in a physical design known as routing as well. In the particular case of macro-cell design, cell generation is essential, in which the cells are synthesized not driven from a library.

Physic and logic-level synthesis steps have already been automatized consistently; for example, Altera and Xilinx performed the logic synthesis well via their synthesis instruments for FPGA design.

So far, the main problem is that an RTL design is required by these tools explained through a hardware description language for synthesizing. Thus, a further stage in design automation is to establish tools for bridging the gap between RTL design and behavioral specification. Such tools should create RTL design in a quite short time, regarding design constraints. Moreover, for exploring better and better solutions in terms of the design objectives, they must discover larger and larger design space regions. Thus, over the past 20 years, high-level synthesis was a very hot research topic.

Though, any optimal technique is inefficient to handle the problem since the

design space is too large. Genetic algorithms are a good option for tackling such complex explorations, with their simile-random search. This feature was run in the mixed high-level synthesis for exploiting genetic algorithms and performing design space exploration to decrease the objectives.

## 2.3 Bambu

In 2004 the development of the Panada framework[**14**] for the Polytechnic University of Milan for supporting research infrastructures started and in 2012 the Bambu which is a high-level synthesis tool in the Panada framework released.

Bambu which is developed by c++11 for the Linux OS is a free framework that could be downloaded freely[2] under GPL licence[**15**] is able for helping designers in high-level synthesis in an intricate application which also supports most constructs in C like:

- Function calls and share the correspond module.
- Arithmetical pointers with the dynamic resolution for memory addresses.
- Accessibility to arrays, structs or any mixture of them.
- Sending the variable and struct via their reference or with function copy.

An important point in Bambu design is that, since it's extremely modular it is able to perform different activities in the HLS process and special algorithms, in different classes of c++ that are working on various IRs related to the level of the synthesis. Hence the flow in HLS has some similarities to synthesis flow in software that begins in a high-level specification and after different steps in optimizations and analysis, the low-level code will be made

### 2.3.1 Work flow

The Bambu as the input gets source code written by C/C++ language (behavioral description of a specification) and as output, it will make HDL description of the corresponding RTL that is matchable by tools in commercial RTL synthesis and a test-bench because of simulation purpose and behavior validity. The process in Babmu is divided into three parts as below:

[2]https://github.com/ferrandi/PandA-bambu

- Front-end: As it is shown in image 2.2, the entered source code will be converted to intermediate representations which will be used in the next step of the flow.in this phase, to parse the source code the Bambu use GCC[**16**] or CLANG-LLVM[**17**].

  It also supports fully ANSI C (built-in C functions, pointers, etc...) and some analysis for optimization in the source code. the options in the compiler are allowed by tools, the intermediate representation file is obtained after the GCC optimization in the middle-end and will be considered as input for the next phase.

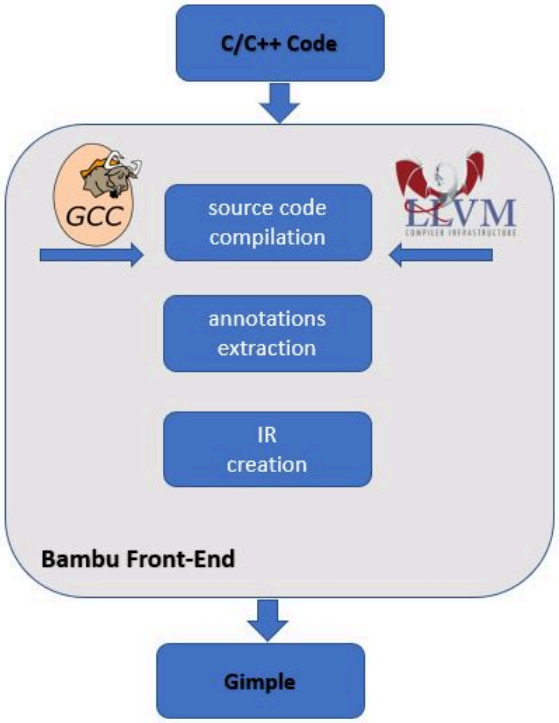



FIGURE 2.2: Front End

- Middle-end: the task which should be performed in this step is about optimizations and target-independent analyses. In another word, To have clean code, better memory access, etc... in this phase also some other changes could be performed in the IR (figure 2.3).

- Back-end: The synthesis process is performed to functions and as result, the HLS is done. In the image (figure 2.4) the whole process for the HLS is shown

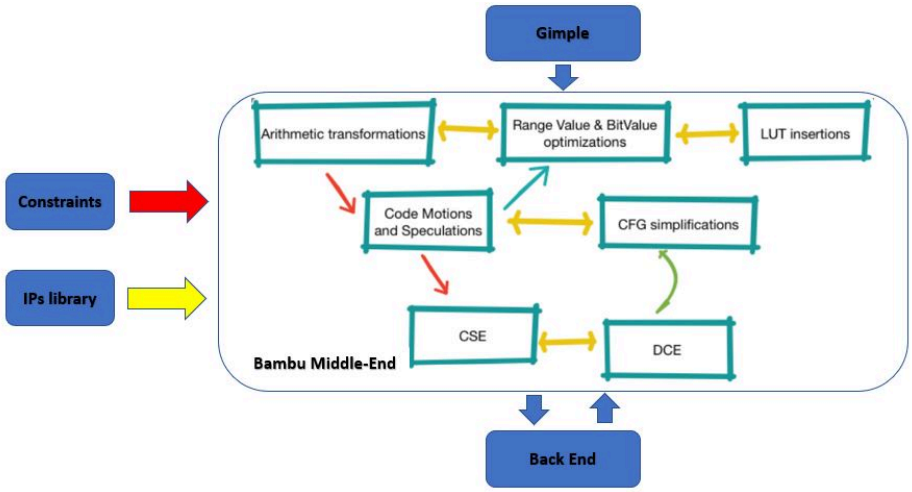



FIGURE 2.3: Middle-End

visually where after seven main phases from A to G which will describe in the following, the entered source code will be converted to the HDL.

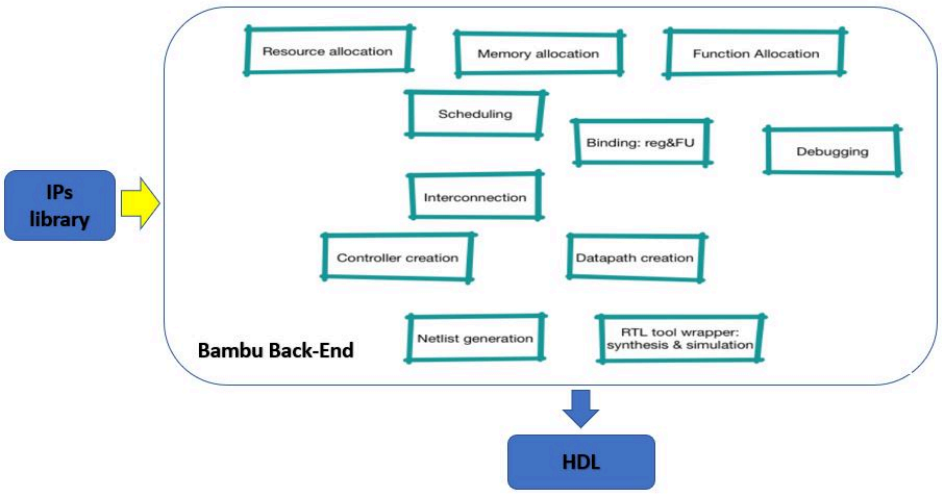



FIGURE 2.4: Back-End

The full process of the Babmbu from entering the source code to generating the output(Like HDL) is shown in the image 2.5 which is divided into seven main parts. In the following, each part with its functionality will describe separately.

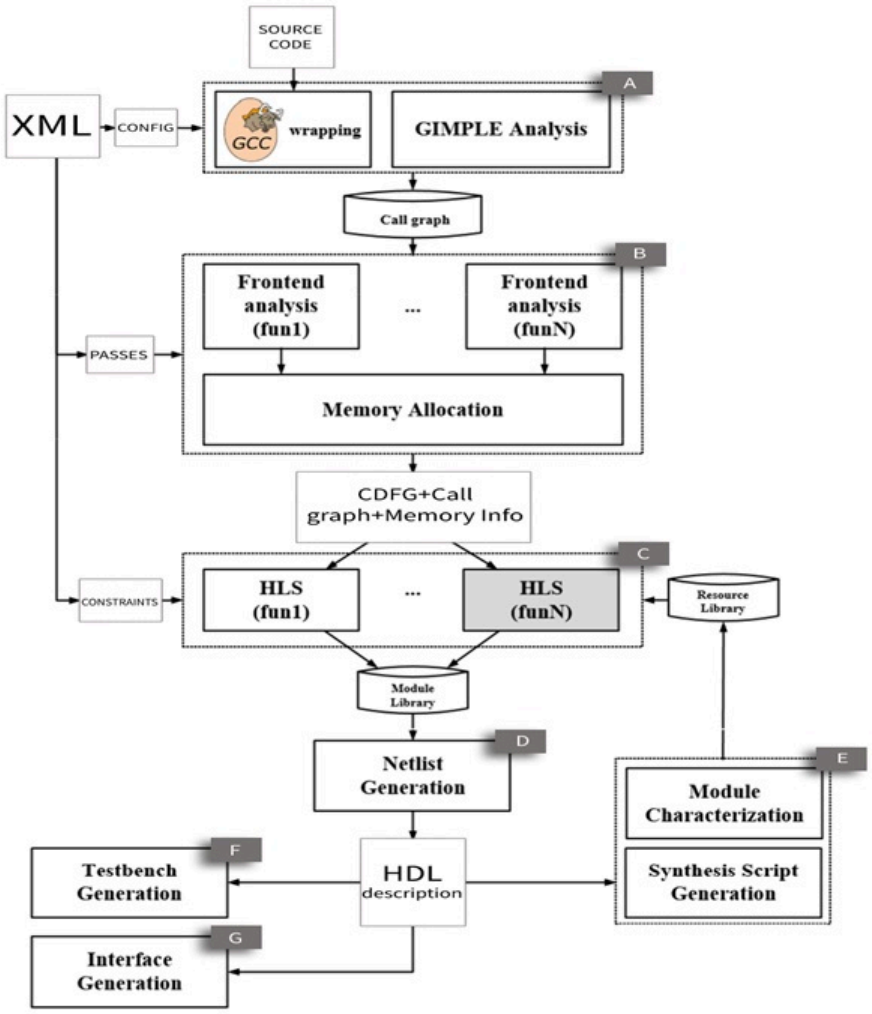



FIGURE 2.5: Bambu Workflow

### 2.3.2 Input and compile source code

In the first step (section A in figure 2.5 ) of the process, the Babmu framework gets the input source file( C /C++ source code) and an XML file for the configuration. Bambu is intake advantage of the customized interface of GCC for the front end which makes enables the framework possibility of delivering the IR in SSA form[3] of the initial code after some optimizations.

By inputting configuration file or command lines options it could be feasible to make some optimizations by the compiler[4] like constant propagation, dead code

[3]https://gcc.gnu.org/onlinedocs/gccint/SSA.html
[4]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

elimination, loop unrolling.

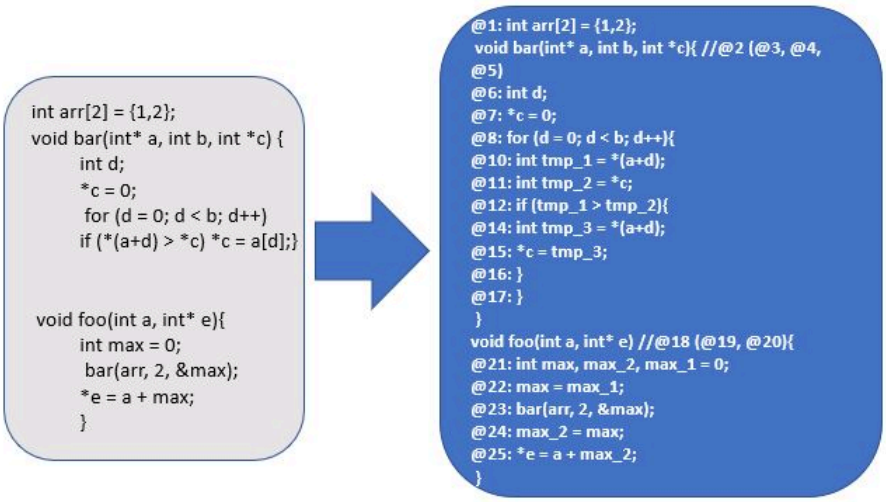



FIGURE 2.6: Convert source code to IR

As an example, in the figure above (figure 2.6 )we could see a restructuring of the code on the left side to explicit memory accesses on the right side of the figure.

### 2.3.3 FRONTEND ANALYSIS and Memories Allocation

As it could be seen in the step B in figure 2.5 Front-End analysis for each function and next memory allocation will perform in next step.

Firstly in the Front-End analysis of the IR a call graph of the application will be generate(left part in figure 2.7) and next after GCC optimization for each function a graph base representaion with identifying of data types, variables, function parameters and output will be generate (right part in figure 2.7).

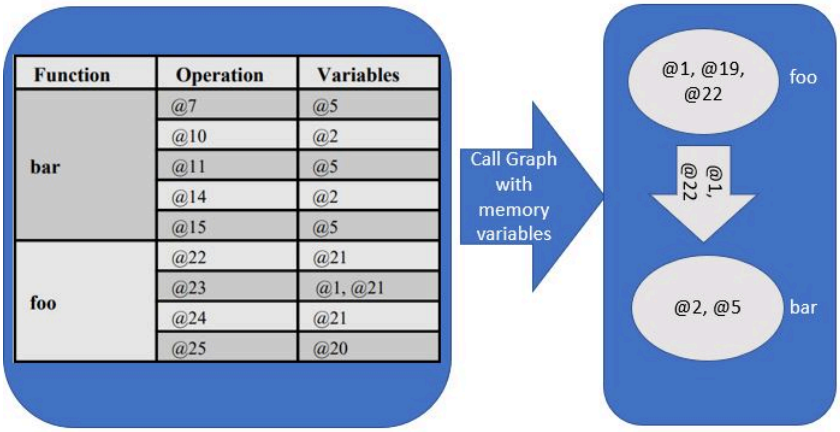



FIGURE 2.7: FRONTEND ANALYSIS

Memory allocation consider the memory space for structures, arrays, global variables and implementing dynamic memory allocation. For memory access in Bambu it makes a data-path hierarchy that is connected to dual-port BRAM directly.

On one hand, whenever a global scalar data type or a local aggregated begin in use with the code determined,when the accesses could be specified during the compile time, that will be resulted to parallel accesses to multiple memory. on another hand, the memories are interconnected which is feasible for supporting the dynamic resolution of the addresses.

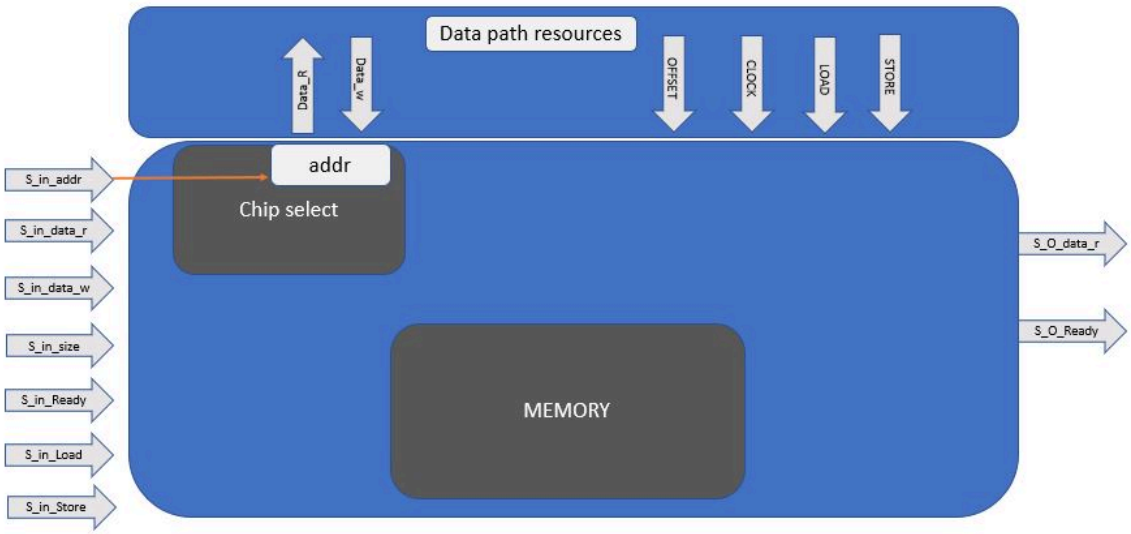



FIGURE 2.8: Internal Memories

As it could be seen in image a chip-select logic for accessing internal memories is performed which tells whether that entered address is for that memory or not while interfaces for reaching the external memories are performed.

with a memory interface in memory allocation, external variables are reachable while internal variables are allocated to dedicated heterogeneous memories that could be reached with functional units.

The image 2.9 represents a schema of external memories. the images show that it is connected to another interface for generating a chain and gives the possibility of a dynamic resolution to address.

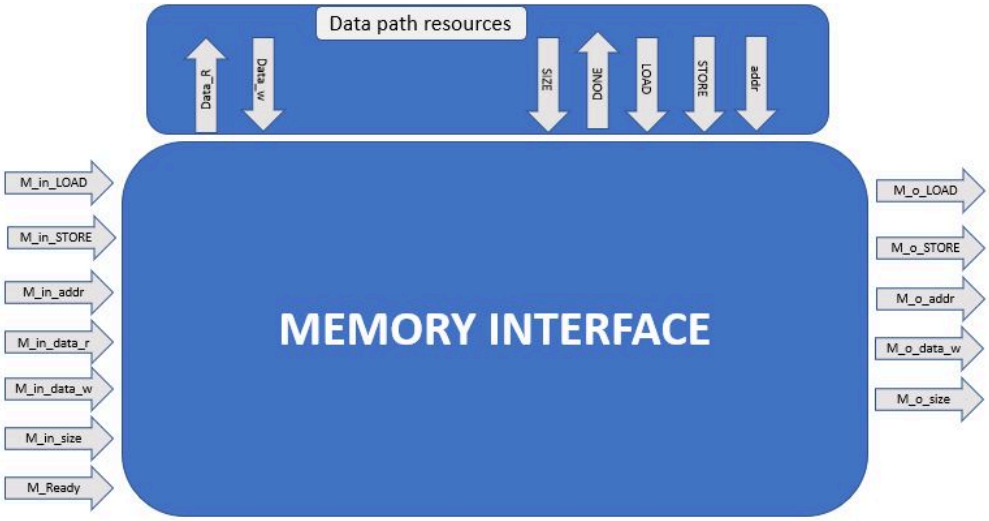



FIGURE 2.9: Externla Memories

### 2.3.4 HIGH-LEVEL SYNTHESIS OF THE FUNCTIONS

The HLS part (part C) is the step which the framework starts to make essential models for implanting the specification, according to the FSM paradigm the controller modules and the memory interface for them. The Bambu also to resource binding and scheduling considered several algorithms which could be controlled by the user (by XML file for configuration or options in command line).

The HLS part for each synthesis step could be extended by different algorithms.

- Resource Allocation:

  Resource allocation makes connection between functional units in the resource library and specification's operations.

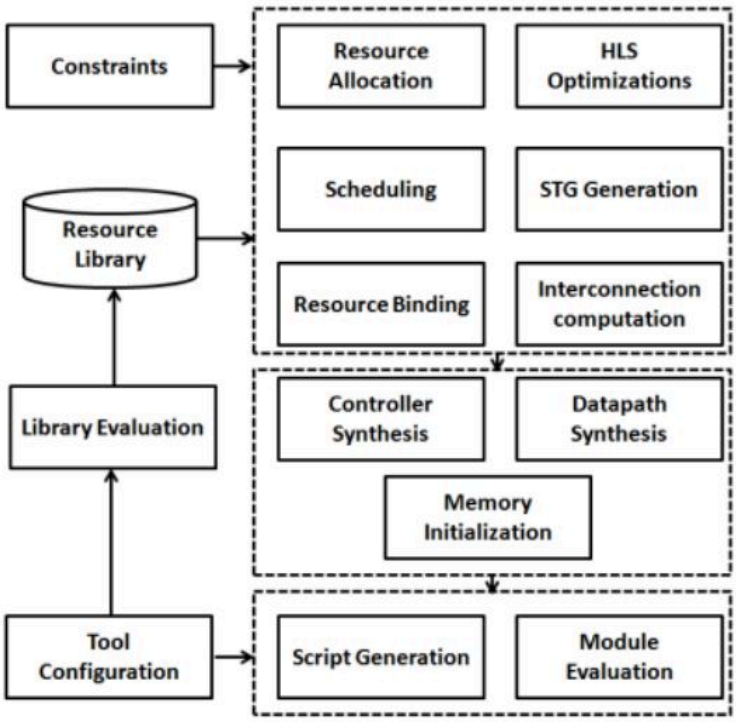



FIGURE 2.10: Shows the HLS process in Bambu flow.

In middle-end step the specification is performed while operations characteristics such as operation's types (multiplication, addition, etc) and value types for inputs and outputs are recognized.

Operations for floating-point are made in the High-Level Synthesis of a soft-float library which contain operations of fundamental soft-float or via FloPoCo[**9**].

In the next step(allocation) they will map to the set of available Functional Units. They contain some data like number of pipeline stages, latency and area. Almost all parings are possible between operation and Functional Unit so the choice of a right Functional Unit is performed with design constraints and while local data could be bounded to local memories, memory resources are allocated to Functional Units.

- Scheduling:

  In scheduling the LIST-based algorithm is the default one, that is constrained, with availability of resource based on common metrics. LIST algorithm gives a priority to any operation.

By considering the critical path, priority could influence operations mobility. critical path's operations get zero-mobility.Increase of the totally circuit latency because of delay in their execution. mobilities and Critical path could made by ASAP and ALAP schedules.

The approach of LIST proceeds repeatedly associating any control step, operations which could be executed. With considering availability of resources, ready operations are scheduled in the current control step. For a resource one with the higher priority is scheduled, when different ready operations are existed.

Another solution is , speculative scheduling algorithm that is working on a system with a variety of constraints. For the scheduling problem it builds an Integer Linear Programming(ILP) formulation, let to the speculations of operations and code motions to the diverse basic blocks. This method is working by ILP solver, that is working by performing the code motions and the speculation introduced by the ILP solution, and next the High-Level Synthesis flow can be implemented.

It is also possible, after the scheduling task, to make the STG(State Transition Graph), in which the STG will reach another analysis for making the FSM of the controller.

- Module Binding:

  Sharing the similar instance of FU is not permitted in operations which based on computed schedule are executing concurrently, so no conflicts in resources.

  After the schedule analysis, the graph compatibility could be generated. Operations scheduled are compatible with various control steps. The positive aspect of sharing resources by two operations could be obtained by their weights. As a result of sharing, they are computed by considering area/time exchanges.[**11**].

  the parts that need a big area such as FUs usually become shared. For computing, the weight, and the number of interconnections for introducing steering logic in the aspect of area and frequency are considered. Different methods in Babmbu for converting issues in generic compatibility/conflict graphs are provided.

- Register and Interconnection Binding:

Register binding is saving the values in registers which required LA(Liveness Analysis) that is a priory analysis phase that should identifies the life intervals of each variable and analyse scheduled function.

Storage values with non-overlapping life intervals could be resulted to sharing a register. By default in Bambu setting, Bambu flow computes liveness data by a non-iterative SSA liveness analysis algorithm. Register assignment next allocated to the matter of colouring a conflict graph in which storage values are graph's nodes and conflict relation are shown by edges. It also considered solution for the issue of weighted clique covering compatibility graph to solve the register binding.

About the Interconnection binding this phase also follow as before phases, where for a share resource, on its inputs the algorithm has steering logic. furthermore, It also recognized the relation between control signals and different operations.

### 2.3.5 Netlist Generation

Finally, thanks to the novel architecture [**10**], by attaching all generated modules and memory interfaces in the master and slave chains the global RTL structural description could be generated:

- The request generate/published by master .
- The slave's responsibility is for gives the related data to the corresponding module for executing the request
- In the case of referring the address to the internal or external memory of the core, the chains will be blocked by the global external interface.

### 2.3.6 Generation of Synthesis and Simulation Scripts

Simulation and automatic generation of synthesis are two critical parts of every HLS flow, In Babmbu this process could be performed by XML configuration files. This step is presented by E in figure 2.5. Resource library could be automatically characterized in Bambu, with the possibility of technology-aware details. the list below represents the compatible tools for RTL-synthesis:

- Xilinx ISE,
- Xilinx VIVADO
- Altera Quartus
- Lattice Diamond

while the supported simulators are:

- Mentor Modelsim
- Xilinx ISIM
- Xilinx XSIM
- Verilator
- Verilog Icarus

### 2.3.7 Testbenches

The next step in Bambu is the testbenches where with the initial C specification and XML dataset, it is possible for having test benches as shown in step F in the image 2.6. It works by HDL which is made as the result of the testbench. in fact, for verification of the execution, it will compare the result with the corresponding software counterpart. Please consider that it is possible that with different configuration to synthesize all CHStone benchmarks.

### 2.3.8 A Datapath Structure

At the register-transfer level (RTL) along with a specification of the finite state, machines are contained in the output from a high-level synthesizer for controlling the datapath. A data path includes functional units, interconnection, and storage elements at the RTL level. Every set of microoperations is specified by the finite state machine for performing the datapath within each control step. Then, the output can be synthesized via the related instruments.

The objective is to decrease one or more design targets for minimizing total area occupied, power consumption, or latency.

Behaviour is specified by behavioural description based on assignment statements, operations, and control constructs within a common high-level language (C language).

It is provided by the class behavioural- manager in the Bambu tool. Several alternatives may exist in the resource library, among which the best one matching the design constraints and maximizing the optimization is selected by the objective synthesizer. It is characterized by the class technology-manager in the Bambu tool.

## 2.4 Function call mechanism

Every time synthesis process is performing on a function that begins from leaves of call graph that as an example it could be seen in the image 2.11. Finite state machines with data path(FSMDs) of callee methods pushed inside the caller methods' data path by compiler.

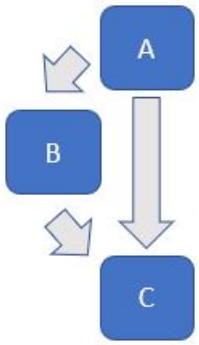



FIGURE 2.11: Call Graph

In figure 2.12 in the left side part the HLS performed without using proxy in function call mechanism which shows in this method at least one callee is needed for every caller module. This approach will be resulted to sub-optimal resource utilization.

As it could be clearly seen in image below because of schedules of callers in this architecture, the funC in existed in both funA and funB modules. In these HLS compilers different callee modules allocated, when should work execute in mutual exclusion (funB depends on the call of funC within funA). One solution for solve the mentioned issues before to makes it possible for sharing modules in the data paths is function proxy, it is knowns as lightweight control elements on all levels of design

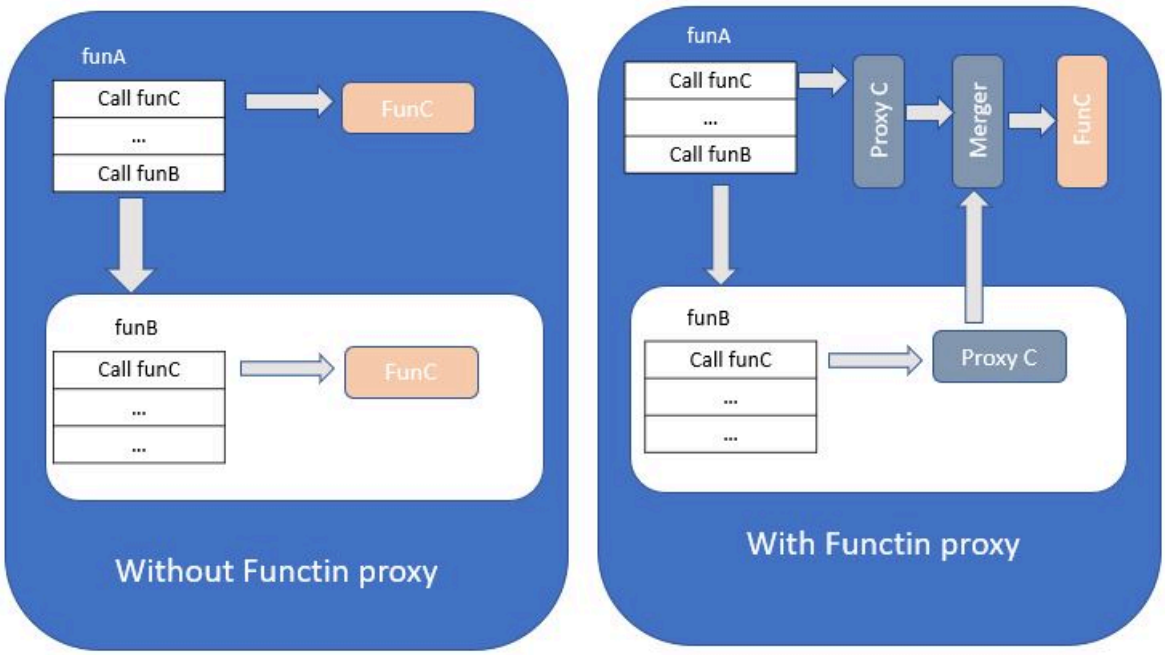



FIGURE 2.12: Function Call with and Without Function Proxy

hierarchy. thanks to the Function it is also possible that without effect on the design and controls complication to have manage on the shared resources.

The proxy contains an interface for begin and finish signals for input and return parameters. The caller has a controller for managing the function modules and its data path includes the module that by using function proxy becomes replacement of the instance of shared module in caller datapath that without any modification in the functionality of the controller of the caller, sends the data signals and control toward the right instance of the module. In the top of the design hierarchy of the generated architecture for all shared function module a module's sample inserted in the caller's data path and to any function call there is a proxy inside the data path of all caller modules.

The right side of the image 2.12 shows the hierarchy of using proxy in function call mechanism. In Bambu only one proxy is working in a clock cycle that could be consider that share module calling is usually happened in mutual exclusion. The merger thanks to sending null signals by inactive proxies needs to manage the incoming signals by simple OR and send back the output to the connected proxies by broadcasting where only active proxy accepts the broadcasted signal.

Common approach in HLS flow is about generate a design by the hierarchical

point of view for following the schema of the call graph of the input specification, while in this graph a function is corresponds to a module instantiated in the caller's data path. in some other HLS compiler function call have different mechanism , because by calling a same function by different modules, next is instantiated in each of the data path of the callers which finally will resulted to lack of sharing resources.

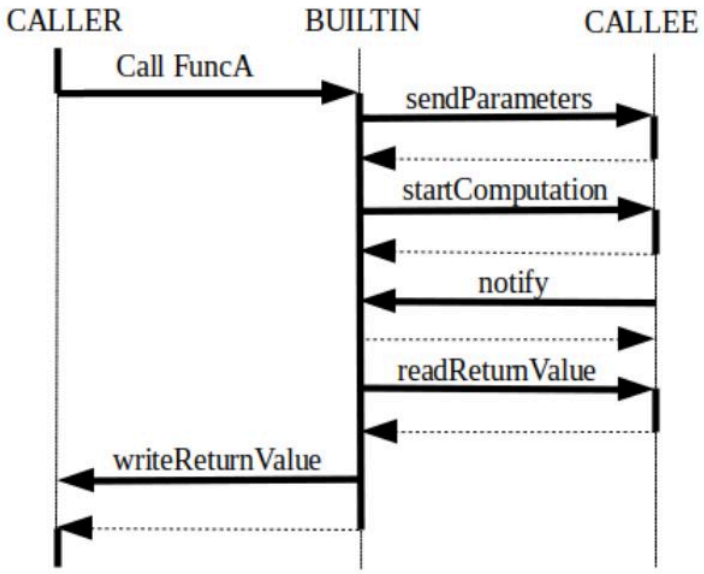



FIGURE 2.13: Indirect function call mechanism

In the the sequence diagram for a function call is shown where the request starts by Caller and it have indirect communication with desired function which in this image it called CALLE, through the builtin.

Next builtin will sends parameters to the desired function(CALLE). So, Caller should wait for the response until Calle finish it's computing and sends it's answer to the builtin when builtin will write retuned value for the Caller.

# Chapter 3

# Thread

## 3.1 What Is a Thread?

Sequential programs (figure 3.1) could be considered as well known for most programmers such as "Hello World" which is the first program that is written as the first program by all programmers or sorting numbers or other sequential programs which are contains three sections beginning, execution sequence and the end.

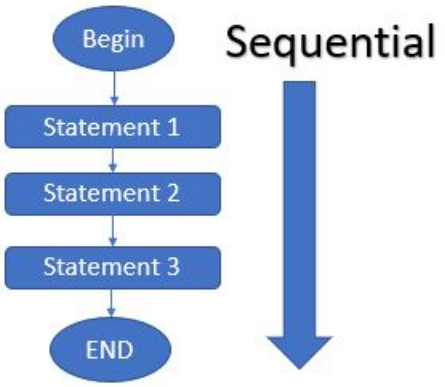



FIGURE 3.1: Sequential Program Work Flow

As described before, the thread is also similar to the sequential program but it could not run autonomously, which also has begin, sequence and the end, which at the run time of the thread the single point of execution is existed.

A program could has more than one thread which are working in parallel. The real excitement surrounding threads is not about a single sequential thread. Instead, it's for using of multiple threads are active at the same time to perform different tasks in a single program. In the figure below(Figure 3.2) we could see in a program we could have a single thread or multi thread which are working in parallel.

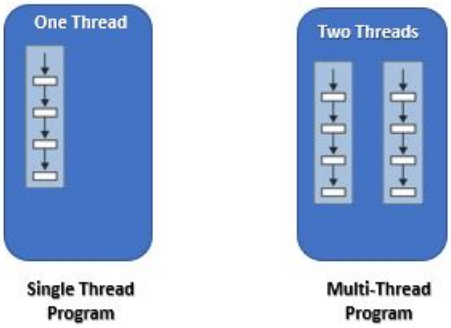



FIGURE 3.2: Multi and Single Thread program

Threads are also using shared code, data and file and operating system resources (e.g., signals) but the register and stack is not shared and it's unique for each thread (figure 3.3).Each thread has private program counter, hardware registers and stack.

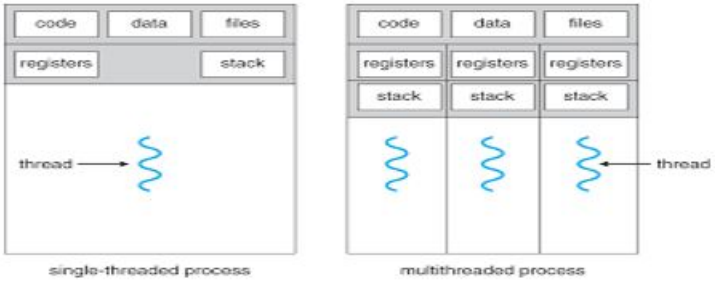



FIGURE 3.3: Thread Shared Resources

The advantages of using thread could considered as following :

- Faster response time.
- Increased scalability.
- Shared resources.
- Lower costs for resource management.

There is different example of using multiple threads in software which call multi-thread programming. For example, when we download a file by web browser, we can watch a video by that browser concurrently. As another example could refer to the MS word. The process can check spelling, auto-save, and read files from the hard-drive, all while you are working on a document.

On the another hand the drawbacks of using thread could consider as following.

- Without protection for threads
- Without parent-child hierarchy between thread.

There different kinds of thread are exsited which are as following:

- Kernel-level thread : Thread executed at kernel-level The direct support of the thread concept by the kernel
- User-level thread : Thread executed at user-level The kernel is unaware of existence of threads
- Hybrid or mixed solution : Both user-level and kernel threads are provided by the operating system

In image 3.4 the difference between User-Level and Kernel-Level Thread could be clearly seen.They have different area. Multiple User-Level threads by managing with a scheduler will be allocated to a Kernel-Level.

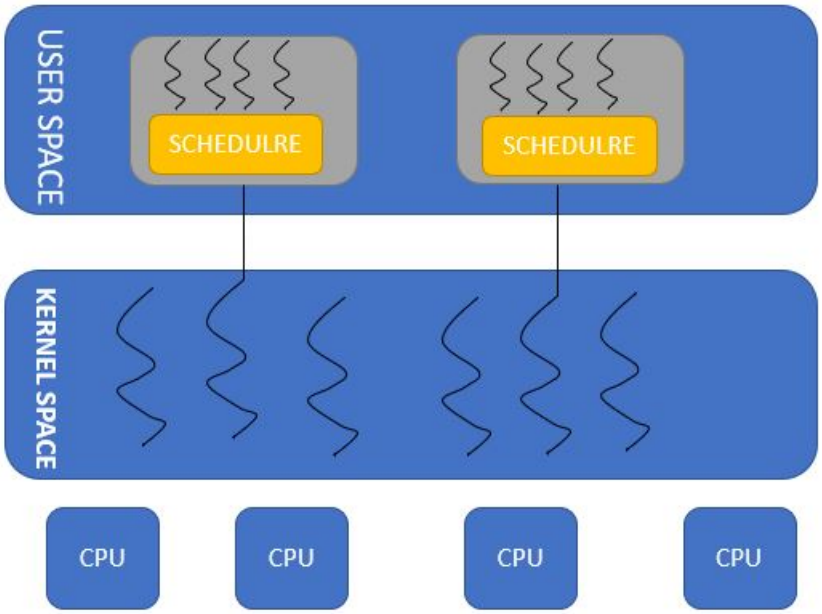



FIGURE 3.4: User-Level and Kernel-Level Thread

### 3.1.1 Kernel threads

The kernel thread is defined as a "lightweight" part of kernel scheduling. Each process contains minimum one kernel thread. In case of presence of multiple kernel threads in a process, the same file resources and memory are shared by them. If process scheduler of operating system is pre-emptive, then kernel threads are multi-tasked pre-emptively. They own a stack, thread-local storage, the register copies, with the program counter. Hence, creation and destruction of the is quite cheap. Besides, thread switching is fairly cheap, which needs a context switching (stack pointer and restoring and saving registers). However, virtual memory is not changed, and it is cache-friendly (affording validity to TLB). The task of the kernel is assignment of one thread to logical cores of the system (since each processor is divided into multiple logical cores if multi-threading is supported by it, or just one logical core is supported in each physical core if it does not), and it is able to swap out the blocked threads. Nevertheless, swapping kernel threads is more time-consuming compared to user threads.

### 3.1.2 User threads

Sometimes threads are conducted in user-space libraries, hence, they are known as user threads. The kernel does not have awareness about them. Thus, their management and scheduling is done in user-space. User threads of some implementations is based on top part of kernel threads for taking advantage of multi-processor machines (M:N model). Green threads are user threads that are executed by virtual machines.

Considering execution of user thread totally in user-space, it is highly efficient to have context switch in user threads in the a process since there is no need for interacting with the kernel. It is possible to run a context switch through local saving of the CPU registers utilized by the presently implementing user fiber or user thread and loading the registers needed by the user fiber or user thread to be performed. Because of occurrence of scheduling in user space, tailoring the scheduling policy to the program's workload requirements would be easier. Nevertheless, it would be challenging in user threads to use blocking system calls (in contrast to kernel threads). If a system call that blocks is implemented by a user fiber or user thread, the other user fibers and user threads in the process would not be able run until return of the system call.

### 3.1.3 Hybrid or mixed thread(M:N)

M is the number of application threads are mapped by M:N into N number of kernel entities or "virtual processors." It is a trade-off between user-level ("N:1") and kernel-level ("1:1") threading. Generally, complexity of implementation of "M:N" threading systems is higher than user or kernel threads since it is required to apply some alterations to kernel and user-space code. The threading libraries are in charge of scheduling user threads on the present schedulable entities in the M:N implementation, which allows to conduct context switch of threads quickly, preventing system calls. Nevertheless, it results in increased complexity and the possibility of priority inversion, and suboptimal scheduling without any extensive (and costly) coordination between the kernel scheduler and userland scheduler.

## 3.2 Standard C++ Thread Library

The class thread as an abstraction for a thread of execution is provided by C++11. This standard library in C++ is a group of classes and functions developed in the core language and part of the C++ ISO standard that use Standard Template Library (STL) and effected by research in generic programming.

The standard thread library is not limited to performing specific implementations and it's free to perform the implementation.

std::thread [1]. is a portable library to be compatible in different operating systems that are supporting C++14[2]. for example, it use Win32 threads in Win32 and pthreads in POSIX.

#include <thread> is the header in the standard C++ Library to declare the threads' functions and classes. the std namespace in C++ Standard Library is responsible for declaring features.

A single thread is represented by the thread class. muti-threads able to concurrently execute multiple functions. The thread execution begins as soon as the object of the related thread is generated.Each thread has it's own id that is public member in the thread class. get_id returns the id of correspond thread.

For constructing a new thread object, the library provides different constructors that could be use as below.

[1]https://www.bogotobogo.com/cplusplus/C11/1_C11_creating_thread.php
[2]https://en.cppreference.com/w/cpp/14

- Default: thread() noexcept;
- initialization: template <class Fn, class... Args>explicit thread (Fn&& fn, Args&&... args);

  fn represents as a pointer to a function or a member and args... is arguments that could be pass for calling the fn.
- copy [deleted] : thread (const thread&) = delete;
- move: thread (thread&& x) noexcept;

  x represents a thread object.

The generated thread object could be destroys with thread destructor. After creating a new thread it starts to execution related function. Meanwhile, the main thread is waiting for terminating the generated thread.this waiting for synchronization by blocking the calling thread till completing the thread is perform by join method(Figure 3.5).

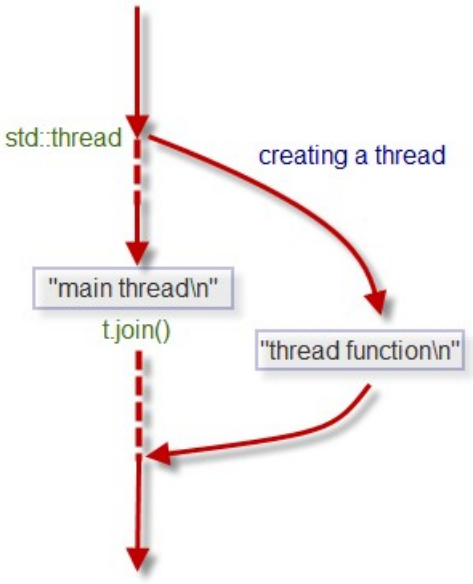



FIGURE 3.5: Thread Join

std::thread::detach provides the execute a thread apart from other threads. in fact the thread object is able to use detach method to continue its execution without blocking and synchronizing and releasing the resources when finished.

hence threads could join or detach prior to their execution flows reach to the destructor.

# Chapter 4

# Modification

Obtaining parallelism in HLS is different than parallelism in software. compilers' abilities for performing parallelism by high-level language is not sufficient hence developers are responsible for determining the parallel programs, so the developer can use a standard library like POSIX Threads or language extensions such as OpenMP.

High-level parallelism in software is performed by runtime libraries but in HLS this duty is up to the compiler to provide the same behavior in a fundamental computational model.

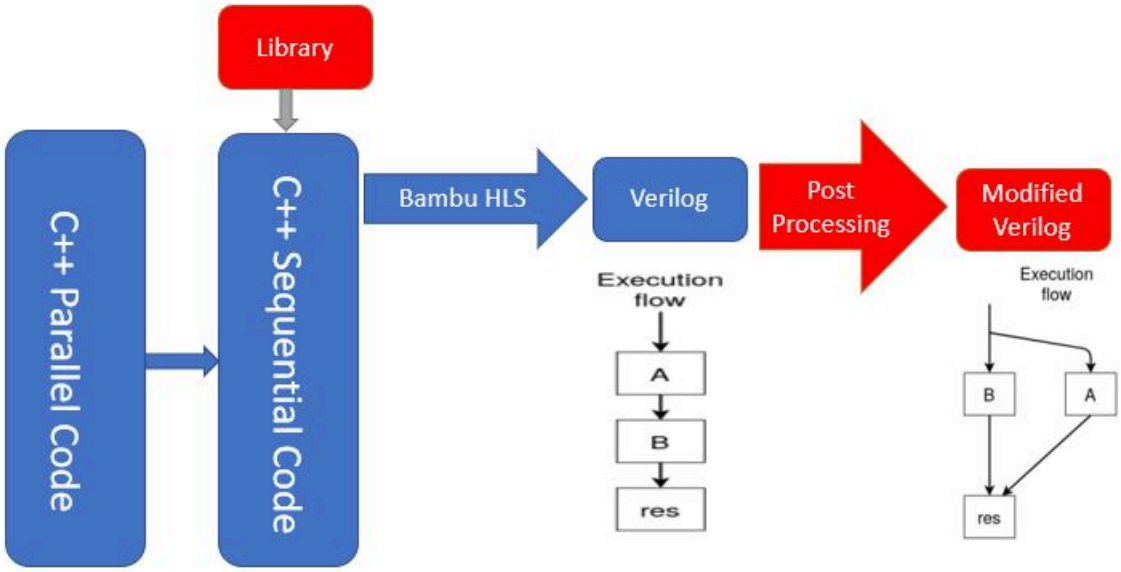



FIGURE 4.1: HLS Flow In Previous Thesis

The previous thesis "Advanced C++14 Multi-threading Modelling of Electronics Systems"[**12**] needs some post-processing steps. The workflow of the previous thesis is shown in figure 4.1 where after entering the C++ source code and using

the thread library, the generated HDL code needs a post-process step manually to convert generated sequential FSMDs to parallel FSMDs. The generated parts in the previous thesis are shown in the image with red color.

The post-process part is not easy to obtain, especially when we want to work on a bigger or more complex project.

Furthermore, in the previous thesis, the thread library only accepts a limited number of constructors and for supporting new threads with different numbers of arguments in generated thread library, we should add a new constructor to the existing library.

In this section, the aim is to improve some parts of the previous thesis and perform different techniques for performing HLS of the multi-thread program by the newly generated thread library by automatic generation and without post-processing.

The aim in this thesis is represented in the list below.

- Add the new thread library to the libambu as a part of Bambu.
- Enable new library for working by a variable number of arguments and different data types.
- HLS with parallel execution for the new thread library without post-processing.

  So, as the final output, we expected to have a new workflow as it is shown in Image4.2.

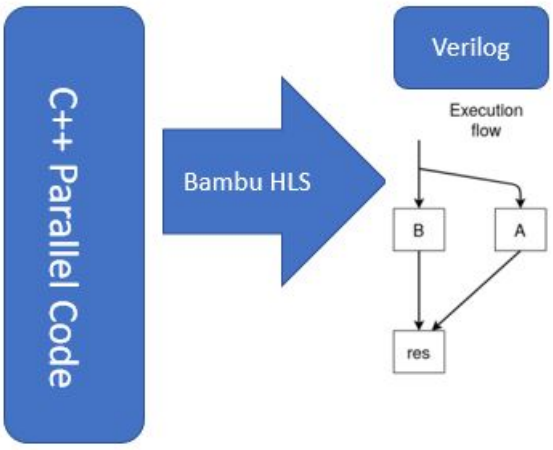



FIGURE 4.2: HLS Flow In Thesis

## 4.1 Bambu Libraries

The Bambu libraries exist in the libbambu folder in /etc/libbambu.

Some functions and libraries are already added in this folder that some of them is mentioned below:

- assert, puts, putchar, read, open, close, write, printf, exit, abort, bswap32,memcmp, memcpy, memmove, memset, malloc, free, memalign, calloc, bcopy, bzero, memchr, mempcpy, libm functions: acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, ceil, cexpi, copysign, cos, cosh, drem, erf, exp, exp10, expm1,tanh, trunc.

For adding a new library like thread library which is the purpose of this thesis, this folder should be considered as a host for the new library. After adding the new library in libBambu the library address like other libraries should be added in the MAkefile.am.

Since Bambu is able to work with different versions of GCC and LLVM compilers like:

- GCC5
- GCC6
- GCC7
- GCC8
- CLANG4
- CLANG5
- CLANG6
- CLANG7

In the Makefile.am the target compliers should specify.

After each modification in the Bambu libraries, we should install the Bambu again to update the compiler and check the new output. For installation, we could set the desired config first. The suggested configuration by Bambu could be seen in below(listing 4.1). More information about configure could be seen in the link [1].

[1] https://panda.dei.polimi.it/?page_id=88

```
../configure --prefix=/opt/panda --enable-flopoco
--enable-icarus --enable-verilator --enable-xilinx
--enable-modelsim --enable-altera --enable-opt
--enable-lattice --enable-release
--with-mentor-license=<license-string>
```

LISTING 4.1: Default configuration for installing Bambu

Next, we should perform the subsequent steps, firstly compile the tool.

```
$make
```

and next install the tool

```
$make install
```

in case of success install, by the following command, the help message should be shown in the output.

```
$/opt/panda/bin/bambu
```

## 4.2 Thread library with context switching

The first idea for adding the thread library in Bambu is to make a user-level thread library in C++. The goal is to develop a thread library same to the standard C++ thread library, that makes multi-instance of the threads, manages the memory allocation and scheduling puts them in a list, and executes them at a right time at the user level. It means the list of threads should be controlled for adding a new one and removed after execution with considering their priority. To make this library in Bambu, some other libraries which are not developed in Babmbu should be made too( Image4.3).

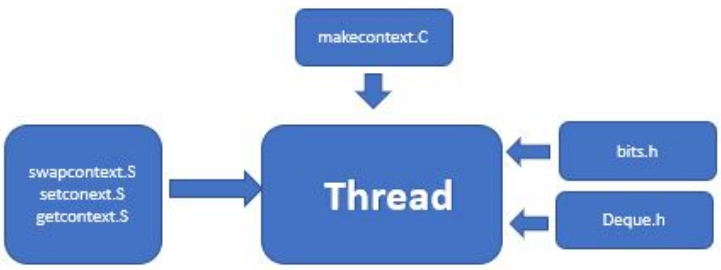



FIGURE 4.3: Thread Library with Context Switching

We implement FIFO[2] dynamic queue for keeping threads in the dynamic list and managing their priority for execution. For this purpose, the Deque.h library generate. The responsibility of the FIFO queue in our solution is when a thread in the head of the queue is running, a new thread will be added to the end of the queue. Every time a thread becomes in the head of the queue and wants to be executed after finishing the previous one, first remove the executed thread,and next read the thread from the ready queue and swap the context between the new current thread and scheduler, by libucontext_swapcontext function(Listing 4.2).

```
1
2   while (ready.GetSize() > 0) {
3       if (current->finished == true)
4           delete_current_thread();
5       Thread* next = ready.GetFront();
6       ready.DeleteFront();
7       current = next;
8       libucontext_swapcontext(scheduler, current->ucontext_ptr);
9     }
```

LISTING 4.2: Sample Code For Context Switching

In this solution, a thread library by two different constructors is generated. The first one is the default constructor without any argument and the second, thanks to the template [3] in C++ supports the variable number of arguments[4]. The constructors are returned in the listing 4.3.

The thread structure is made with four properties:

- unsigned int id: it is the thread Id, 0 for the first thread, and increases after adding a new thread.
- char* stack: stack pointer which refers to the top of the stack.
- libucontext_ucontext_t* ucontext_ptr : thread context pointer. a thread context refer to the subset of thread state(stack environment,registers,...).
- bool finished: flag to show thread is executed or not.

[2]https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/

[3]Template pass the data type as the parameters instead so by this technique we are able to avoid making different methods for different kinds of data. Also by a parameter pack (since c++11) it is possible to consider the variable number of the function arguments.

[4]https://en.cppreference.com/w/cpp/language/template_parameter

```
1   public:
2    //Constructor without argument
3    thread() { }
4
5    //Constructor with variable number of arguments
6    template<class C, class... R>
7    thread(C(*f), R...r)
8  {
9    ...
10 }
```

LISTING 4.3: Constructors for User-Level thread

When for the first time, a thread with arguments is created, the init value becomes 1 which is 0 by default. Also, make a new instance of the context and assign that to the scheduler, next move the first thread in the ready queue to the scheduler by swapping.

- thread(): It's the default constructor for making a thread without argument.
- template<class C,class... R> thread(C(*f), R... r): Second thread constructor with a variable number of arguments. the first argument should be the function name and the next variable number of arguments with different types.
- template<class C,class ... Args> int thread_create(C(*f), Args... r): Makes a new thread and put it at the end of the ready queue. The main thread is generated the first time we call the thread library and after making the first thread init flag becomes true.
- static int exec_func(thread_startfunc_t func,void* arg ): this function after executing the thread update the thread status as finished and remove that context from the scheduler.
- void delete_current_thread(): Delete the current thread and its context.
- int thread_wait(unsigned int lock, unsigned int cond).
- int thread_unlock(unsigned int lock) Remove the thread id from lock_map queue.
- int thread_lock(unsigned int lock) add the thread id to lock_map queue.

The functions mentioned below are generated for controlling and switching user-level context between multiple threads. Since for using context using registers is essential , assembly is the right choice in this part. Some external assembly libraries used, that are shown image4.11(swapcontext.s,setcontext.s and getcontext.s).

- int libucontext_getcontext(libucontext_ucontext_t *): this method is responsible for overwriting the entire context on the current context.
- void libucontext_makecontext(libucontext_ucontext_t *, void (*)(), int, ...): writes the function and data in a context[5]
- int libucontext_swapcontext(libucontext_ucontext_t *, const libucontext_ucontext_t *): Records the current context in the first entered context pointer and switches to the second context pointer5
- int libucontext_setcontext(const libucontext_ucontext_t *): Assign the current context to the desired context.[6]

The image shows that for making a thread library, further existed libraries in Bambu, also other libraries should developed which could be supported by Babmu for HLS purposes.

Deque.h which is a Dynamic queue library that is compatible with Babmu is developed for this solution. In the table below (table 4.1)the library functions are shown.

| | |
|---|---|
| bool IsEmpty() | Returns true if the queue is empty |
| bool IsFull() | Returns true if the queue is full |
| void InsertFront(T Value) | Add new item in the front of the queue |
| void InsertRear(T Value) | Add new item in the end of the queue |
| void DeleteFront() | Delete the item in the front of the queue |
| void DeleteRear() | Delete the item in the end of the queue |
| T GetFront() | Returns the item in the front of the queue |
| T GetRear() | Returns the item in the end of the queue |
| int GetSize() | Returns the size of the queue |
| ~Deque() | Decountroctor of the class |
| void GrowArray() | Increase the size of the queue |

TABLE 4.1: API for Deque Library

[5]make context and swap context https://linux.die.net/man/3/swapcontext
[6]setcontext https://linux.die.net/man/3/swapcontext https://linux.die.net/man/2/setcontext

Figure 4.4 shows the desired architecture for the generated thread library. The life cycle begins from the program on the disk and finishes after execution by the CPU. After adding the new Thread library which is described more in the next sec-

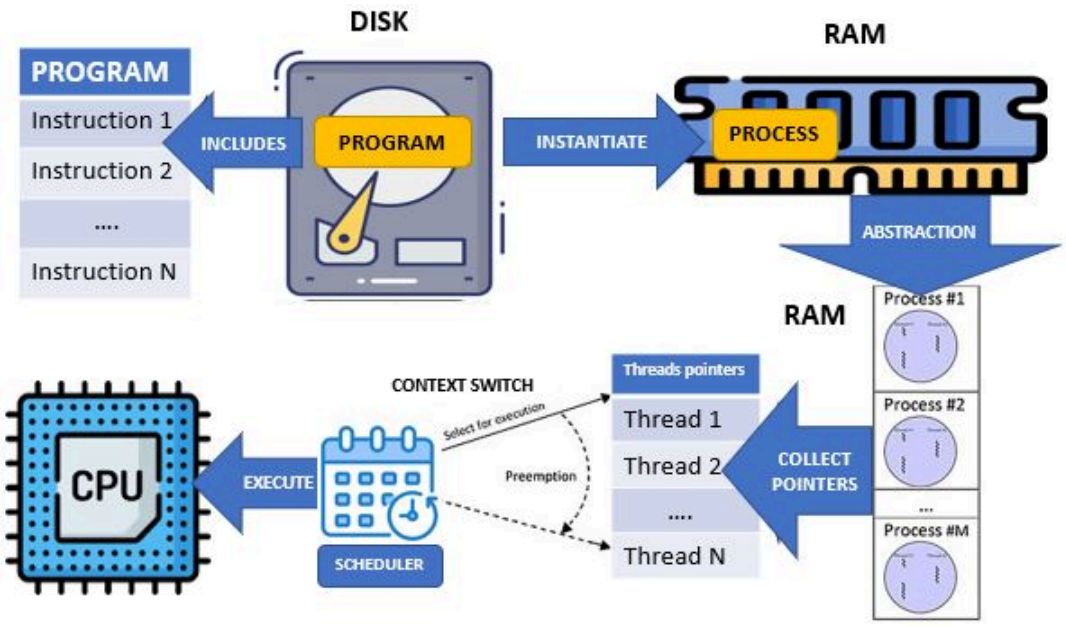



FIGURE 4.4: Thread Life Cycle

tion we are able to use Thread like format below for HLS by Bambu.

```
#include <thread>
using namespace std;
int var;
void funcA ( int counter){
  ...
}
void funcB ( int first,int second){
  ...
}
void main(){
thread(funcA,100);
thread(funcB,10,15);
}
```

LISTING 4.4: Sample code

Listing 4.4 is a sample code that works with generated thread library and could be compiled by GCC correctly but not in Bambu. Since in this solution we should use

some assembly library while HLS starts from high-level languages, another solution for parallel execution should be considered. However, the generated thread template with some partial modifications becomes effective in other methods.

## 4.3 Thread Library

The C++ thread library designed as returned in listing 4.5 to have a multi-threaded program in Bambu. This decision for developing and choosing this library is obtained after some tests and considering the possibility in HLS workflow.

```
#include <iostream>
using namespace std;
struct Thread {
  unsigned int id;
};
static int id = 0;
class thread {
public:
//default constructor
thread(){}
//other constructors
template<class C,class... R>
thread(C(*f), R... r){
  thread_create( (*f), r... );
    }

 template<class C,class ... Args> \__attribute\__((noinline))
 int thread_create(C(*f), Args... r) {
  Thread* t;
    t = new Thread;
     (*f )(r...);
    t->id = id;
    id++;
  return 0;
}
};
```

LISTING 4.5: New Thread Library

This thread library works by two different constructors one which works without any argument and another one which gives as input a function firstly and next

variable number of other arguments. These two features are(template and function pointer) supported in Babmbu and thanks to them we are able to make this library.

Another useful feature in Babmu for making Thread library with this template in C++ is about supporting function pointers [**34**] while in some other HLS tools, usage of function pointers to call a function is not possible due to the non-static resolution.

By this mechanism that supports function calls by function pointers, we are able to use this thread library which is working by the function pointers. For example, as it could be seen in the source code above, in the thread_create the first argument is a function pointer.

Another attribute that is supported by Babmu is noinline function. we could avoid considering our threads to be inline by this attribute and don't inline this function under any circumstances. So, if the function has not any side-effect, some optimizations apart from inlining that use to function calls become optimized away, however, the function call exists. By this attribute, Bambu generates an independent module for every noinline function.

## 4.4 Parallel Programming

In this part, the aim is to describe how it is possible to change the current execution flow from sequential serial Finite State Machines with Data-path(FSMDs) to the parallel Distributed Controller(DC) with access to the shared memory.

While most HLS are focused to generate Instruction Level Parallelism (ILP) which means they are focused to generate FSMDs which should be scheduled statically, we want to generate Task Level Parallelism(TLP) by DCs that are able to work by dynamic scheduling.

Fortunately, Bambu is compatible with OpenMP[**25**]that is a programming interface for supporting multiprocessing programming and a multi-platform shared memory in both C and C++. In other words, it is the solution for creating code to run on multiple cores/threads.

The source code description by the parallel specification also existed in other HLS software. For example, LegUp[**26**] also supports OpenMP specifications, while for scheduling it needs an instance of an extra general-purpose processor.

However, it usually uses in loop-level parallelism, but it is possible to reach function-level parallelism too. An advantage of using OpenMP is that we are not

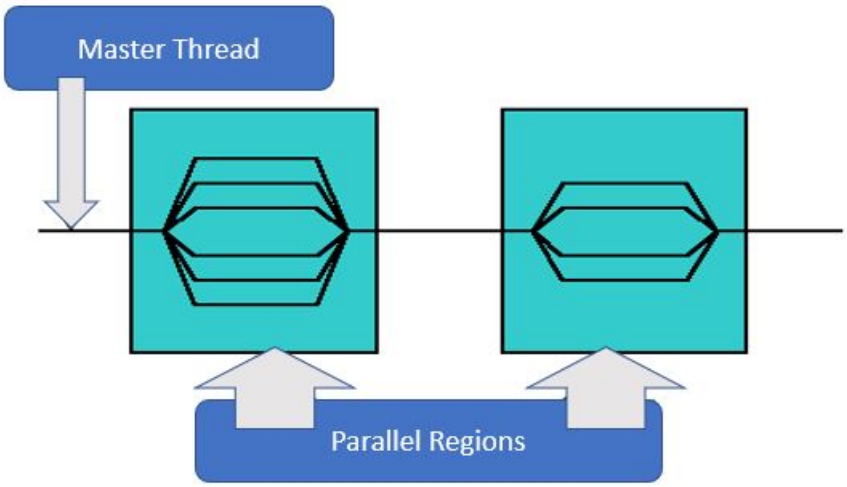



FIGURE 4.5: Multi Threading Schema In OpenMP

required to add external libraries to compile this code by OpenMP because it is built inside the compiler.

The execution model of OpenMP for multi-threading is shown in the image4.5. At the start of the parallel area, the master thread (sequential) makes a group of threads combined by itself and a collection of threads. Next, at the end of the parallel area, the thread group finishes the execution while only the master thread continues the execution of the program.

```
#pragma omp parallel  // create a group of threads
 {
#pragma omp single //where a single thread
   {
  P= head of_ list();
  while (lend_of_ list (p)){
  #pragma omp task  //submit a Task
  process(p);
  p= next_ element(p);
                              }
   }
 }
```

LISTING 4.6: Sample code by OpenMP

The source code in listing 4.6 shows how to generate parallel tasks by OpenMP.The parallel region makes a group of threads while a single thread creates tasks, next

adding them to the queue which belongs to the group. The task is a block of instructions with data that is scheduled for execution with a thread. The task in the queue is assigned to the thread in their group by the task scheduler.

The functionality of OpenMP in Bambu like most HLS software is to make a parallel Intermediate Representation(IR) different than a sequential one. For clarification and explain the OpenMP functionality and generating parallel IR in Bambu the workflow is represented.

The synthesis receives different graph-based IR base on the GIMPLE[**27**] as input. An advantage of using GIMPLE is possibility of decreasing the instrumentation overhead. It starts to synthesize one function every time according to the call graph's structure to generate a modular, hierarchical design. The following most common IR in compilers are described:

- Control Flow Graph (CFG): the data structure at top of the IR, abstracting the control flow behavior of a function that could be traversed in function execution. In this directed graph, vertices represent basic blocks and edges represent could transfer of control flow from one basic block to another.

- Control Dependence Graph (CDG): a directed graph for controlling the dependencies of a basic block that controls the execution of another basic block or not.

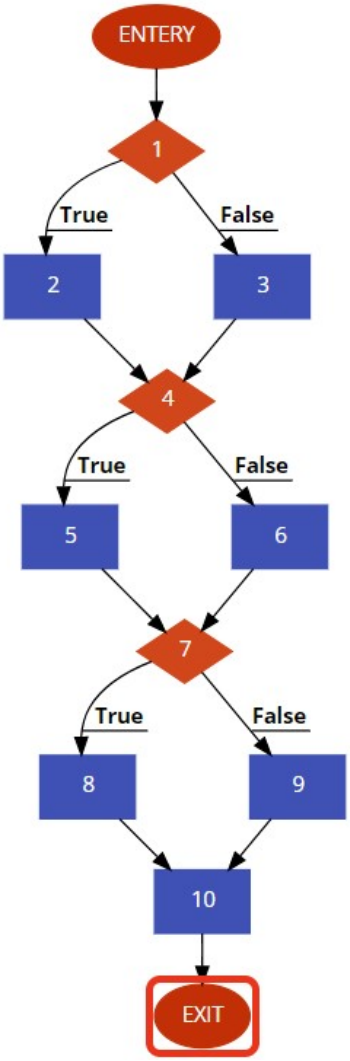
- Control Dependence Regions (CDR): basic block partitions in the equality classes that region could be the same with another basic block if they have similar control dependencies in the control dependency graph.

- Loop Forest:the loops hierarchy inside a control flow graph.

Control Flow Profiles are used in many compilers by collecting code instrumentation or making static samples from the program counter. After generating a graph of the program, vertices (basic blocks) or edges (branch transitions) are represented by profiles, for counting the number of executions of these elements.

For Example, figure 4.6 shows a sequential version of the mainFunc on the left side and its CFG on the right side. The program contains 20 lines and 10 basic blocks. the source code is on the left side and the related basic block in the right side with BB. This basic block in the right side, in CFG is represented as a sequential program that starts with begin and ends with exist.

```
— Sample Sequential Code —
int mainFunc(int c1,int c2,int c3){
    int  a, b, c;//BB1
     a = c2 + c1;       //BB1
     if (c1)            //BB1
         fun_1(&a);     //BB2
    else
         a *= 2;         //BB3
     a += b;            //BB4
     b = a + c1;         //BB4
    c1= a;              //BB4
    if (c1)             //BB4
        fun_2(&a);      //BB5
    else
       a *= 2;          //BB6
    c = 1;              //BB7
   if (c2)              //BB7
        fun_3(&c);      //BB8
    else
         c *= 2;        //BB9
   return a + b + c;    //BB10
```




(A)

FIGURE 4.6: Sequential Code With Control Flow Graph(A)

The parts of the program with high time consumption could be optimized during the optimization process. However, there are no completely correct estimates of the frequently executed paths (sequences of branch transitions) when these profiles are used broadly.

OpenMP represents path profiling[**28**] as a method to reach paths' frequency for making better profiling information while the amount of instrumentation upward is limited.

The approach is to divide the program into different tasks. Therefore, profiling

information without considering the relations among tasks is used to estimate the performance of single tasks. However, lack of the considering the relation, tools for analyzing the performance cannot collect essential information on code hot spots and load balancing, to find the greatest, average, or worst performance estimation of all the task graphs, the combination of estimations is considered. Next, each task is represented as a thread and should perform a fork task, and at the end of the execution of all threads join the task for all of them. This concurrency is represented by OpenMP.

Architecture properties are essential for a valid performance evaluation of path profiling and specification. These properties are also considered in the process of mapping from Gimple nodes to the target assembler statements. The number of cycles for the target processor could be obtained by using an analytical model with the list of assembler statements correlated with a GIMMPLE node same approach to [**29**] and [**30**]. This estimation is not exact completely and it has a few errors, but it's accepted because it has high accuracy usually since the goal is to focus on high-speed techniques for task optimization.

The implementation of common parallel embedded applications that are works by cycles is possible through the hierarchy. Hierarchical Task Graph (HTG) [**31**] is the IR of a parallel program that could be obtained from the Control Flow Graph of the sequential program by recognizing the edges when data and control dependency analyses are used for recognizing the edges in a graph. HTG with considering control dependences and data generates an intermediate program that makes partitions an application into tasks and specifies parallelism at all levels.

In a hierarchy, a vertex could be:

- Compound: connected to another HTG. These kinds of relations resulted in stronger graphs than Direct Acyclic Graphs (DAGs), which do not possibly have feedback edges.
- Simple: only a single task without another task
- Loop: Repetition its body that it's a separate HTG by itself

The outcome graph is non-cyclic the task could be classified into three categories:

- Fork: multi successors tasks
- Join: multi predecessors tasks

- Normal: other tasks

The CDG for mainFunc is represented in image (4.7). The graph shows execution of basic block 2 is depended to the basic block 1 if return true and if it returns false the block 3 could execute. the story is same for basic blocks 5,6,8 and 9. on the other hand basic blocks 1,4 and 7 could execute in parallel because there is no dependency between them.

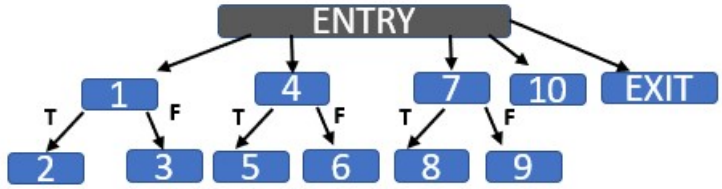


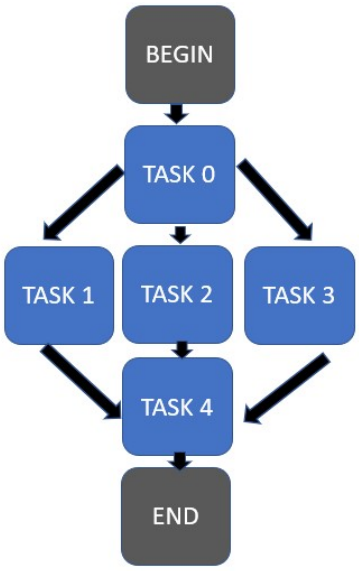

FIGURE 4.7: The Control Dependency Graph

The source code below (Figure 4.8) shows the sample code for generating parallel tasks by OpenMP. This program starts with a Main task (Task 0) and finally ends with Task 4. During this workflow Task, 1,2 and 3 are executed in parallel.

The generated task graph for this source code is represented in figure 4.8 The image shows the execution graph for the parallel program in left side, where task 1, task 2, and task 3 can execute in parallel. Possibility of the parallel execution and concurrency of the model are obtained by explicit fork and next the join by OpenMP in a shared memory area.

Sample Parallel Code

```
int mainFunc(int c1, int c2, int c3){
    int  a, b, c;
    //TASK0
#pragma omp parallel sections{
//TASK1
#pragma omp section {
     a = c2 + c1;
     if (c1)
         fun_1(&a);
    else
         a *= 2;
     a += b;
     b = a + c1;
    }
    //TASK2
#pragma omp section {
    c1= a;
    if (c1)
        fun_2(&a);
    else
       a *= 2;
    }
    //TASK3
#pragma omp section {
    c = 1;
   if (c2)
        fun_3(&c);
    else
         c *= 2;
    }
    }
    //TASK4
     return a + b + c;
```




(A)

FIGURE 4.8: Parallel code With Task Graph(A)

The aim of this thesis is to put each task in a function and call them in the program by our generated thread library(Listing 4.5).In this term, another version of the program in figure 4.8 could be as below code(Listing 4.7).

```
#include <iostream>
#include <thread>
void task1 .......
void task2 .......
void task3 .......
int mainFunc(int c1, int c2, int c3){
int  a, b, c;
#pragma omp parallel sections{
thread(task1 ,....)
thread(task2 ,....)
thread(task3 ,....)
 return a + b + c;
}
}
```

LISTING 4.7: New Solution for Multi-Threading

## 4.5 Proposed Architecture

The Bambu generates FSMD in HLS that is serial inherently and could be useful for ILP, but TLP that is a coarser granularity another architecture design for supporting multiple execution flows by a distributed controller.

The generated architecture for the DC apart from previous methods, a different and customized approach is considered[**35**]. The significant advantage of using DC is, they execute operations dynamically while most HLS approaches work by execution schedule. In addition, DC is able to manage multi-concurrent execution flow because any operation and function are controlled apart, while in centralized FSMD its management is complex. in other words, the number of states and transitions in an FSM is related to the number of flows. Hence, designing becomes complex and or infeasible although for a tiny degree of TLP while in DC the design is simpler because its design complexity increases parallel with the number of operations without considering the number of concurrent flows and operations' latency.

This architecture requires an extra step in comparison with FSMD, where the front-end analysis requires calculating the ACs and generating EPDG. in this term, Bambu uses a specific algorithm for register [**36**] and module binding [**37**].

For concurrency and synchronization in memory resources, the solution is Memory Interface Controller (MIC)[**38**] that works without specific modification in synthesis algorithms.

### 4.5.1 Distributed Controller Architecture

The architecture by adaptive DC [**39**] performs the dynamic scheduling and parallel execution with a collection of communication modules each of them is corresponds to an operation. So by dynamic scheduling, the scheduling (execution order ) in design time is skipped with the possibility of run-time exploitation of parallelism.

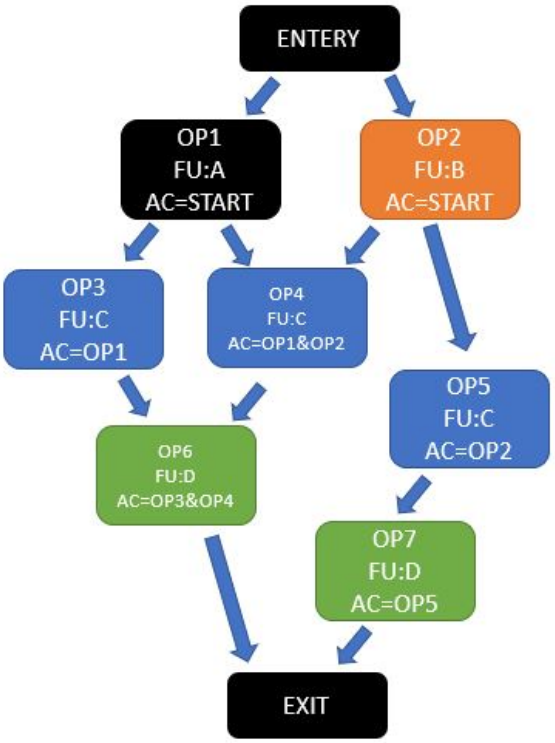



FIGURE 4.9: Extended Program Dependencies Graph

The responsibility of execution of the related operation is up to Execution Managers (EMs) that are module controllers. Execution of the correlated operations is

performed by EMs immediately after satisfying all of their dependencies and fixing resource conflicts.

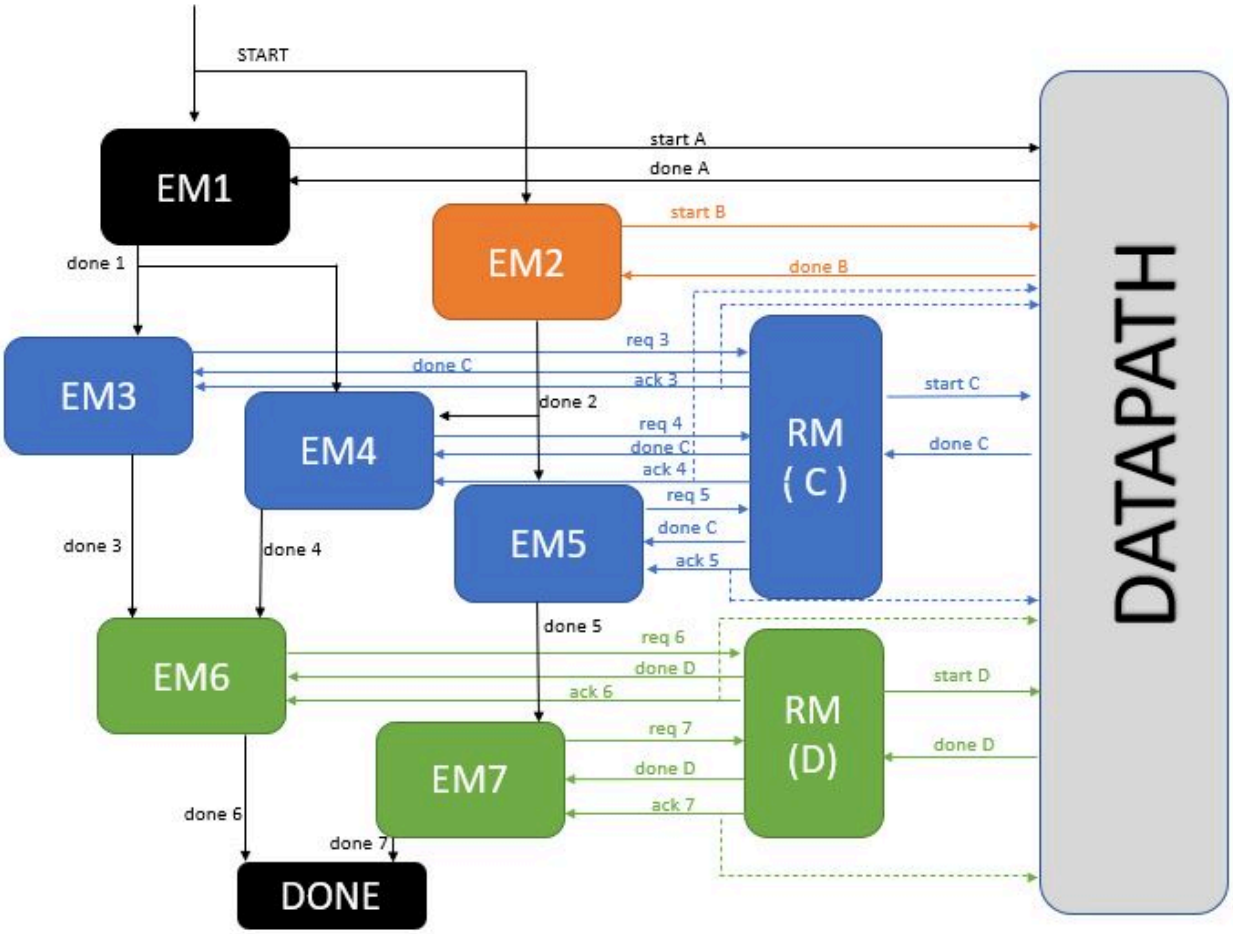



FIGURE 4.10: Distributed Controller Architecture Correspond to Figure 4.9

After analyzing the Extended Program Dependencies Graph (EPDG) of the algorithm, which represents a common Program Dependence Graph (PDG)[7] by control-flow information(like loops' back edges), Activating Conditions (ACs) could be computed for obtaining the least dependencies for every operation.

The communication protocol in EMs is based on a light token-based schema. In this protocol, EM gets a token signal after satisfying the dependency. As soon as collecting all AC tokens by the controller(i.e. satisfying all dependencies ) resource

[7]program dependence graph (PDG) is a graph notation for represents the data and control dependencies explicitly [**32**] which are for analyzing dependencies during compiler optimization for performing changes. These changes are performed for improving parallelism and working by multi-cores[**33**].

availability should be checked. The execution starts in case of finding empty resources for related operations. Since this method doesn't use sophisticated protocol, there is no overhead in communication.

ACs work as logic functions that are synthesized especially for every EM. furthermore, Resource Managers (RMs) manage resource conflicts in shared resources. For example, if different operations want to reach a resource, the RM lets access the operation with a higher priority, and so on.

Figure 4.9 shows an example of an EPDG and its DC is represented in Figure 4.10. The EPDG represents the bonded information and ACs for the corresponding architecture in the parallel controller. OP1 and OP2 can start to work in parallel because there is no dependency for executing. Other blocks could execute after performing their dependencies. The EPDG will use for generating DC in image 4.10. The architecture shows operations 6,7 are connected to D while Operations 3,4,5 are connected to the resource C to prevent structural conflict the corresponding EMs interface by RMs.

The timing can be managed by EMs directly when the latency of execution is known. Since, in this sample the latency for every operation such as function calls, and external memory accesses,... is unknown, the EMs were informed about finishing their execution by a done signal from the datapath.

### 4.5.2 Memory Interface

Calculating resources replication is known as the conventional view for hardware synthesis in TLP. Threads and tasks are performed via special hardware components so many of these modules will be used in the final design. Bambu's policy for parallel working is for different hardware components to bind concurrent function calls while all resources could not be replicated directly because of memory resources.

In parallel programming, memory could be a shared resource because generating tasks, that sharing data is convenient among them, and the concurrency of memory operations should be controlled in parallel execution. In this solution, the performance could be decreased because of the memory limitation and this effect is more in memory-bound programs. As consequence the memory could not be ignored in these programs since its not enough calculation intensity.

Most proper architectures are working with partitioning and /or distributing memory. With some extra challenges, these methods provide concurrent access to memory with different operations.

- During the run time the location of the destination could be recognized because usually statically memory addresses are not known.
- Structural conflicts should be skipped on shared memory resources
- Synchronization should perform when multiple tasks parallelly want to reach memory.

Explained issues are solved in Babmu by introducing a Memory Interface Controller (MIC) during HLS. In fact, it provides the solution for concurrency and synchronization in memory resources **[40]** by mapping memory operations among distributed and/or multi-ported memories. where N ports in MIC get requests for accessing to the memory with corresponding address, data, and operation type line that could be a load or a store.

MIC immediately gets a request when the related port becomes ready, consequently considering the address of the request, the MIC directs the request for an output port of M. The result sends back to the operation when it receives a done signal of M with the corresponding result in case of load request. Memory consists of M autonomous banks that every output port could have access to a bank. it should mention that non-overlapping addresses are allocated for every memory bank.

For the routing, the considered solution is via a customizable control logic to perform the synthesis that could be matched by the common scrambling function for broadcasting the data in a memory system. Concurrency management is obtained by preventing each structural conflict in shared resources without extra delay with a light arbitration plan.

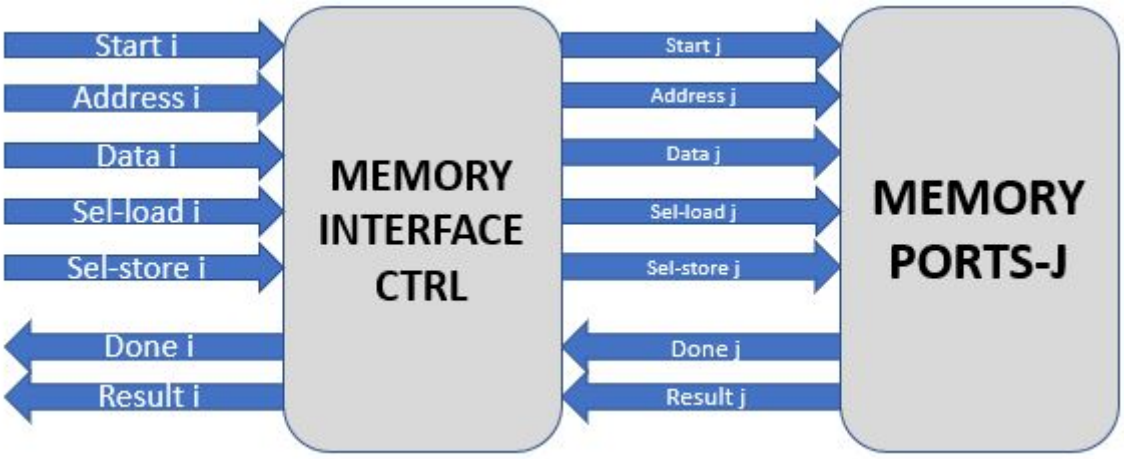



FIGURE 4.11: Memory Interface Controller(MIC) Schema

Like DC, RMs also represented in MIC to arbitration. For obtaining better memory bandwidth usage in the system by MIC, resource availability and access routing are checked in runtime, next issuing concurrent memory operations if both are not looking for an equal memory address. Atomic memory operations are guaranteed by RMs when only atomic memory operation can send the request to the memory location of the atomic memory operation. Synchronization is possible by atomic memory operations because of supporting special operations like compare-and-swap and fetch-and-add.

### 4.5.3 HLS example

As described previously, we employ OpenMP pragmas for parallelism. According to different tests and searches in Bambu, as a possible solution for having parallelism is using #pragma omp parallel. the template presented in the sample code below(Listing 4.8) is an example of using OpenMP pragmas. Since the design uses accelerators and is not allocated to the special language we could implement the solution in different languages too. In the example, a parallel loop is implemented that needs concurrent access to the memory. Because of this atomic operations #pragma omp atomic in line 9 for atomic operation considered. With this solution, the critical section in parallel execution will be operated in mutual exclusion.

```
1 void topfunction(...) {
2    ...  // code block
3   #pragma omp parallel for
4     for ( i = 0; i < N; i++) {
5     {
6      // parallel body
7       #pragma omp atomic
8      atomic(...);
9     }
10  }
11   ... // code block
12 }
```

LISTING 4.8: Sample Code to Synthesized With OpenMP

for list 4.8, the compilers should distinguish parallel functions that could require specific HLS steps, hence next to code factoring and wrapping done for transforming from Listing 4.8 to Listing 4.9 functions will be categorized as non-standard or standard HLS functions. the standard HLS functions are functions that are able to synthesize by current HLS and are without specific parallelism or context switching. for example, the topfunction in line 17 of Listing 4.9 doesn't need any particular pattern for synthesis that needs a special HLS step, but the generated parallel loop using OpenMP parallel will be synthesized differently according to describe parts previously.

```
1 void atomic(...) {
2 body
3 }
4 void kernel(int i, ...) { // loop body
5    atomic(...);
6    // loop body         }
7 void parallel(...) {
8    for (i = 0; i < N; i++)
9      thread(kernel,i, ...);
10 }
11  void topfunction(...) {
12    {...} //source code
13    parallel();
14 }
```

LISTING 4.9: Sample Code to Synthesized With OpenMP After High Level Synthesis transformations

### 4.5.4 Challenges and Solutions

Some fundamental aspects of parallel programming are listed below with the provided solution in this thesis, some of them related to parallelism in architecture, and some to memory management.

- Design a parallel architecture instead of existed sequential FSMDs : for this part, the solution is designing a distributed controller instead of serial FSMDs. The DC contains a set of EMs and RMs that executes tasks as soon as possible.
- Management of concurrency: MIC provides concurrent access to memory in distributed/multi-ported shared memories. the MIC also supports memory scrambling.
- Management of synchronization: MIC also supports atomic operations.because the atomic operation us able to lock the associated memory port. The atomic operation also provides area for critical sections of a parallel code that should be operate in mutual exclusion.

# Chapter 5

# Resuls

In this chapter, we validate and evaluate the generated method. We synthesize and simulate the source code which is added in appendix B by Bambu.

The source code is responsible for making a search inside the graphs with about 26,454 triples. It searches to find items with the same properties as those entered for searching. The database is made in an XML file which is compatible with the Bambu simulator as the input file. we performed 4 different configurations for HLS to make 2 parallel samples with 2 and 4 accelerators and 2 sequential samples with 2 and 4 accelerators in the first test.In the first test 4 memory banks for all cases considered but for the next test, we want to change it to see the effect in latency. The simulation was performed in Babmu by choosing VERILATOR as the simulator.

In this part, the output performance usually cosider by Area and Latency.

- Area: the number of hardware resources needed for implementing the design based on the resources available such as block RAMs, look-up tables (LUT), and registers, ...
- Latency: Amount of required clock cycles to perform a function and obtain the output.

In the first table (Table 5.1) we want to understand the difference between latency and area in different models.

| Summary of resources | | | |
|---|---|---|---|
| | Parallel-4 Accelerators | Parallel-2 Accelerators | Sequential |
| Latency(Cycles): | 20687 | 32957 | 49988 |
| Total estimated area: | 5,438 | 5,438 | 7,586 |
| max frequency (MHz): | 409.53 | 409.53 | 348.57 |

TABLE 5.1: Simulation for parallel and Sequential tests with 2 and 4 Accelerator

According to the results in table 5.1 we could reach a lower latency by parallelism and a better outcome with fewer clock cycles when the number of accelerators becomes more. In the Bambu configuration, the number of accelerators should be multiple of 2. By increasing the number of accelerators from 2 to 4 the number of clock cycles decreased from 32957 to 20687 which is could be considered a good outcome.

This number remains the same in the sequential version, where increasing the number of accelerators is without effect in this version.

The total estimated area is the same for both parallel versions and lower than the Sequential version.

The max frequency (MHz) is also the same in both parallel versions and a bit more than in the sequential version.

Table(Table 5.2), shows the summary of allocated resources in different simulated models.

Both sequential versions had the same resources, according to the previous table they had the same latency and area too. So, in conclusion, the number of accelerators has no effect in the sequential version.

As could be seen in the table above, some resources in the parallel version are different from the sequential version. For example scheduler or controller_parallel is represented in parallel versions only. It means that special resources are considered in parallel models.

Furthermore, the number of resources in a parallel version with 2 accelerators is lower. For example number of ui_rshift_expr_FU ,ui_pointer_plus_expr_FU, ui_eq_expr_FU and read_cond_FU.

| Summary of resources | | |
|---|---|---|
| Parallel with 4 Accelerators | Parallel with 2 Accelerator | Sequential |
| AND_GATE: 8 | AND_GATE: 4 | ASSIGN_UNSIGNED_FU: 7 |
| ASSIGN_UNSIGNED_FU: 20 | ASSIGN_UNSIGNED_FU: 10 | ASSIGN_VECTOR_BOOL_FU: 2 |
| ASSIGN_VECTOR_BOOL_FU: 5 | ASSIGN_VECTOR_BOOL_FU: 3 | MEMORY_CTRL: 3 |
| MUX_GATE: 98 | MUX_GATE: 52 | MUX_GATE: 35 |
| OR_GATE: 25 | OR_GATE: 13 | OR_GATE: 3 |
| UIdata_converter_FU: 1 | UIdata_converter_FU: 1 | PRINTF_VECTOR_BOOL32 _UINT32: 1 |
| UUdata_converter_FU: 81 | UUdata_converter_FU: 43 | UIdata_converter_FU: 1 |
| __controller_parallel: 1 | __controller_parallel: 1 | UUdata_converter_FU: 26 |
| addr_expr_FU: 6 | addr_expr_FU: 4 | addr_expr_FU: 4 |
| bus_merger: 54 | bus_merger: 30 | bus_merger: 20 |
| constant_value: 54 | constant_value: 32 | constant_value: 24 |
| flipflop_AR: 8 | flipflop_AR: 4 | flipflop_AR: 3 |
| lut_expr_FU: 4 | lut_expr_FU: 2 | lut_expr_FU: 1 |
| mem_ctrl_kernel: 9 | mem_ctrl_kernel: 5 | read_cond_FU: 11 |
| memory_ctrl_parallel: 1 | memory_ctrl_parallel: 1 | register_SE: 42 |
| read_cond_FU: 32 | read_cond_FU: 16 | ui_bit_and_expr_FU: 2 |
| register_SE: 12 | register_SE: 12 | ui_bit_ior_concat_expr_FU: 4 |
| register_file: 156 | register_file: 78 | ui_eq_expr_FU: 6 |
| scheduler: 4 | scheduler: 2 | ui_gt_expr_FU: 1 |
| ui_bit_and_expr_FU: 9 | ui_bit_and_expr_FU: 5 | ui_lshift_expr_FU:13 |
| ui_bit_ior_concat_expr_FU: 10 | ui_bit_ior_concat_expr_FU: 6 | ui_lt_expr_FU: 4 |
| ui_eq_expr_FU: 24 | ui_eq_expr_FU: 12 | ui_minus_expr_FU: 2 |
| ui_lshift_expr_FU: 44 | ui_lshift_expr_FU: 24 | ui_ne_expr_FU: 2 |
| ui_lt_expr_FU: 16 | ui_lt_expr_FU: 8 | ui_plus_expr_FU: 11 |
| ui_minus_expr_FU: 5 | ui_minus_expr_FU: 3 | ui_pointer_plus_expr_FU: 18 |
| ui_plus_expr_FU: 30 | ui_plus_expr_FU: 16 | ui_rshift_expr_FU: 4 |
| ui_pointer_plus_expr_FU: 65 | ui_pointer_plus_expr_FU: 35 | |
| ui_rshift_expr_FU: 10 | ui_rshift_expr_FU: 6 | |

TABLE 5.2: Summary of resources

The next simulation part is performed to see the effect of the memory banks in the parallel version. for this simulation, the first model with 4 parallel accelerators and different numbers of memory banks, and the second model with 2 parallel accelerators and different numbers of memory banks.

In image 5.1 that the test was performed with 4 parallel accelerators and 4,8 and 16 memory banks. The statistics show the number of memory banks has a direct relation with latency, where the number of clock cycles for 4,8 and 16 memory banks decreased from 20,687 to 17,470 and 16,571 respectively.

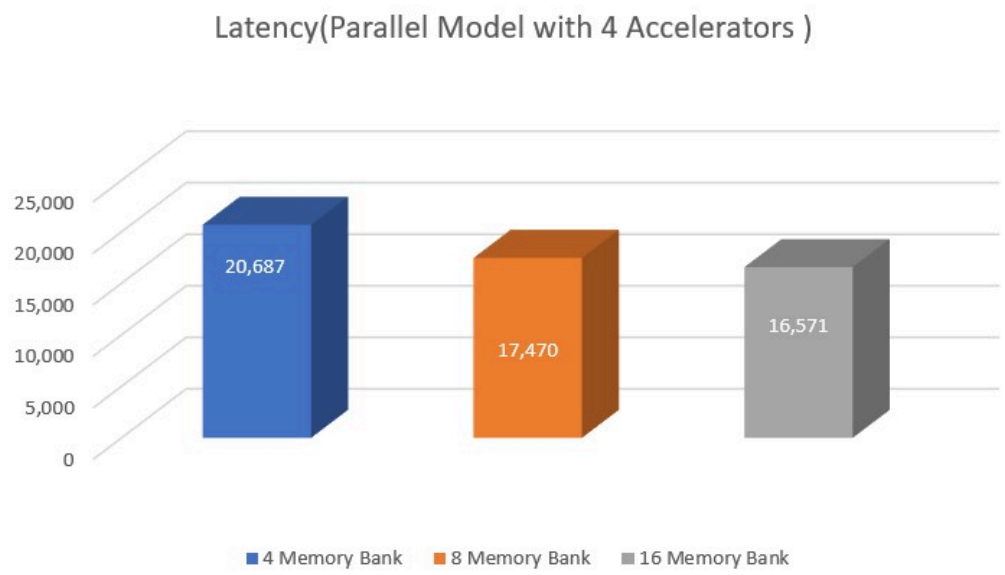



FIGURE 5.1: Simulated Model Latency with 4 Parallel Accelerator

Image 5.2 shows the data about a similar test as the previous, but with 2 accelerators. The changes in the number of clock cycles in this version are not significant like the previous test but still, we could see a slight decrease after increasing the number of memory banks. As could be seen image, the number of clock cycles was reduced slightly from 32,957 to 31,506 after doubling the number of memory banks from 8 to 16.

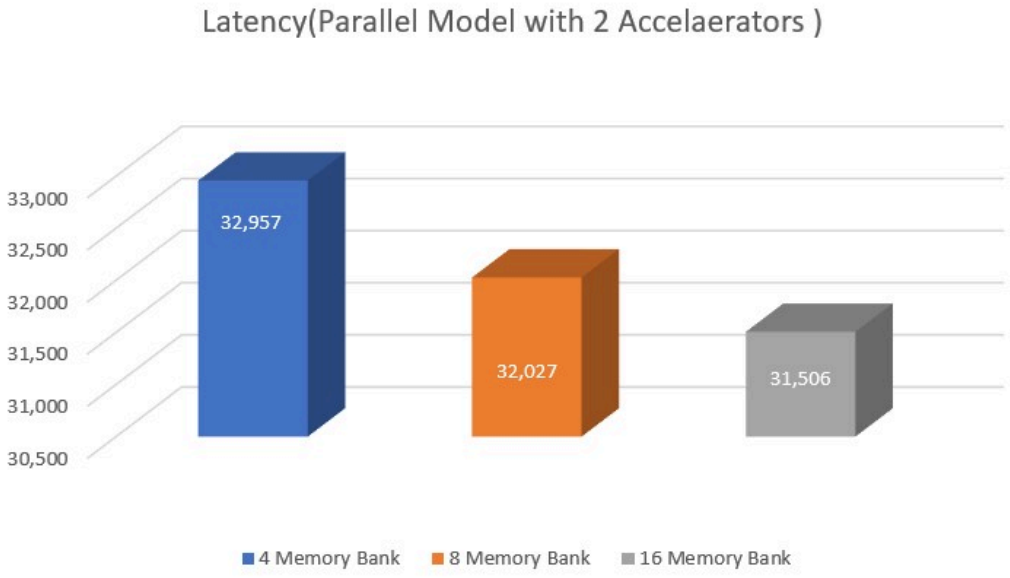



FIGURE 5.2: Simulated Model Latency with 2 Parallel Accelerator

So, increasing the number of memory banks in a version with 4 accelerators is more effective than another one with 2 accelerators.

# Chapter 6

# Conclusion

In this paper, the main goal is to add a standard C++ thread library to the Bambu, an existing high-level synthesis framework to execute a program in parallel. Bambu makes it possible to solve some basic parts in parallelism. for example other HLS frameworks are supporting instruction-level parallelism while we could have task-level parallelism in Bambu.

Most HLS frameworks are supporting FSMDs only that are serial inherently. HLS flows usually generate Finite State Machines with Datapaths, which are inherently serial Concurrency and synchronization management could be solved by supporting atomic operations.

So, after working on different solutions for having a C++ thread library for working in parallel, we used a thread class with a template for calling a thread library with a variable number of arguments while this library is able by using Open MP, run multi accelerators in parallel. For more straightforward implantation, the solution to work on shared memory is introduced. Two main components used in the solution by Bambu are:

- Distributed Controller (DC): to have task-level parallelism.
- Memory Interface Controller (MIC): for handling accessibility in concurrency to a multipored shared memory thought shared kernel with the possibility of atomic operation in memory.

By considering the final result of this solution we could see that in parallel execution in contrast with sequential one, an increase in the number of accelerators will be resulted in to decrease in the number of clock cycles for executing a task so shorter response time.

## Future Work

The purpose of this paper was to add a standard thread library in C++ which could be able to perform tasks in parallel. This topic could be performed by different views in HLS since. In this paper parallel, working is made by using OpenMP. Another solution for reaching different areas and latency could be using diffident FSMDs which are working in parallel by static scheduling in Bambu for existed thread library which is not easy to obtain but the outcome could be interesting.

# Bibliography

[1] Gordon E. Moore, “Cramming More Components onto Integrated Circuits,” Electronics, pp. 114–117, April 19, 1965. Publisher Item Identifier S 0018-9219(98)00753-1.

[2] K. H. Yeap, and H. Nisar, "Introductory Chapter: VLSI", in Very-Large-Scale Integration. London, United Kingdom: IntechOpen, 2018 [Online]. Available: https://www.intechopen.com/chapters/55591 doi: 10.5772/intechopen.69188

[3] Gani, Rafiqul. "Computer-aided methods and tools for chemical product design." Chemical Engineering Research and Design 82.11 (2004): 1494-1504.

[4] Klingstam, Pär, and Per Gullander. "Overview of simulation tools for computer-aided production engineering." Computers in industry 38.2 (1999): 173-186.

[5] 1076-1987 – IEEE Standard VHDL Language Reference Manual. 1988. doi:10.1109/IEEESTD.1988.122645. ISBN 0-7381-4324-3..

[6] 1800-2vhhu017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language

[7] P. Coussy, D. D. Gajski, M. Meredith and A. Takach, "An Introduction to High-Level Synthesis," in IEEE Design and Test of Computers, vol. 26, no. 4, pp. 8-17, July-Aug. 2009, doi: 10.1109/MDT.2009.69.

[8] F. Ferrandi et al., "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications," 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 1327-1330, doi: 10.1109/DAC18074.2021.9586110.

[9] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. IEEE Design and Test of Computers, 28(4):18–27, July 2011.

[10] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," in Proceedings of CODES+ISSS '11, 2011, pp. 49–58.

[11] C. Pilato, A. Tumeo, G. Palermo, F. Ferrandi, P. L. Lanzi, and D. Sciuto, "Improving Evolutionary Exploration to Area-Time Optimization of FPGA Designs," Journal of Systems Architecture - Embedded Systems Design, vol. 54, no. 11, pp. 1046–1057, 2008

[12] P. Iannicelli "Advanced C++14 Multithreading Modelling of Electronics Systems." https://webthesis.biblio.polito.it/15345/

[13] C. Pilato, V. Castellana, S. Lovergine, and F. Ferrandi, "A Run time Adaptive Controller for Supporting Hardware Components with Variable Latency," in AHS 2011: NASA-ESA Conference on Adaptive Hardware and Systems, 2011, pp. 153 – 160.

[14] PandA-bambu http://panda.dei.polimi.it/

[15] GNU General Public License https://www.gnu.org/licenses/

[16] GCC - GNU Compiler Collection https://gcc.gnu.org/onlinedocs/gcc.pdf

[17] Clang C Language Family Frontend for LLVM https://clang.llvm.org/

[18] Simpson, Philip. FPGA design. Springer, 2010.

[19] Churiwala, Sanjay, and I. Hyderabad. "Designing with Xilinx FPGAs." Circuits and Systems. Springer 2017.

[20] Obaid, Zeyad Assi, Nasri Sulaiman, and M. N. Hamidon. "FPGA-based implementation of digital logic design using Altera DE2 board." International journal of computer science and network security 9.8 (2009): 186-194.

[21] Torbey, Elie, and John Knight. "High-level synthesis of digital circuits using genetic algorithms." 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360). IEEE, 1998.

[22] "Assembler language". High Level Assembler for zOS & zVM & zVSE Language Reference Version 1 Release 6. IBM. 2014 [1990]. SC26-4940-06.

[23] Archer, Benjamin (November 2016). Assembly Language For Students. North Charleston, South Carolina, USA: CreateSpace Independent Publishing. ISBN 978-1-5403-7071-6. Assembly language may also be called symbolic machine code.

[24] Streib, James T. (2020). "Guide to Assembly Language". Undergraduate Topics in Computer Science. Cham: Springer International Publishing. doi:10.1007/978-3-030-35639-2. ISBN 978-3-030-35638-5. ISSN 1863-7310. S2CID 195930813. Programming in assembly language has the same benefits as programming in machine language, except it is easier.

[25] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in ISSS, 2002, pp. 109–111.

[26] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for fpgas," in Field-Programmable Technology (FPT), 2013 International Conference on, Dec 2013, pp. 270–277.

[27] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations," in 5th Int'l Workshop on Languages and Compilers for Parallel Computing, 1993, pp. 406–420

[28] T. Ball and J. R. Larus, "Efficient path profiling," in MICRO-29, 1996, pp. 46–57

[29] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu, "Software performance estimation strategies in a system-level design tool," in CODES, 2000, pp. 82–86

[30] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A sw performance estimation framework for early system level-design using fine-grained instrumentation," in DATE, 2006, pp.468–473

[31] Girkar, M., Polychronopoulos, C.D. The hierarchical task graph as a universal intermediate representation. Int J Parallel Prog 22, 519–551 (1994)

[32] Jeanne Ferrante; Ottenstein, Karl J.; Warren, Joe D. (July 1987). "The Program Dependence Graph and its Use in Optimization" (PDF). ACM Transactions on Programming Languages and Systems. 9 (3): 319–349. CiteSeerX 10.1.1.101.27. doi:10.1145/24039.24041

[33] Ferrante, J.; Ottenstein, K. J.; Warren, J. (1987). "The program dependence graph and its use in optimization". TOPL. doi:10.1145/24039.24041.

[34] M. Minutoli, V. G. Castellana, A. Tumeo and F. Ferrandi, "Inter-procedural resource sharing in High Level Synthesis through function proxies," 2015 25th International Conference on Field Programmable Logic and Applications (FPL), 2015, pp. 1-8, doi: 10.1109/FPL.2015.7293958.

[35] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada and F. Ferrandi, "High level synthesis of RDF queries for graph analytics," 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2015, pp. 323-330, doi: 10.1109/ICCAD.2015.7372587.

[36] V. G. Castellana and F. Ferrandi, "Scheduling Independent Liveness Analysis for Register Binding in High-Level Synthesis," in DATE 2013: Design, Automation and Test in Europe, 2013, pp. 1571–1574.

[37] ——, "An automated flow for the high level synthesis of coarse grained parallel applications," in Field-Programmable Technology (FPT), 2013 International Conference on, Dec 2013, pp. 294–301.

[38] V. G. Castellana, A. Tumeo, and F. Ferrandi, "An adaptive memory interface controller for improving bandwidth utilization of hybrid and reconfigurable systems," in DATE 2014: Design, Automation and Test in Europe, 2014, pp. 1–4.

[39] C. Pilato, V. Castellana, S. Lovergine, and F. Ferrandi, "A Run-time Adaptive Controller for Supporting Hardware Components with Variable Latency," in AHS 2011: NASA/ESA Conference on Adaptive Hardware and Systems, 2011, pp. 153 – 160.

[40] V. G. Castellana, A. Tumeo, and F. Ferrandi, "An adaptive memory interface controller for improving bandwidth utilization of hybrid and reconfigurable systems," in DATE 2014: Design, Automation and Test in Europe, 2014, pp. 1–4.

# Appendix A

# Modification

```
#include <iostream>
#include "Deque.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "bits.h"
using namespace std;
typedef void (*thread_startfunc_t) (void*);
extern "C" int libucontext_getcontext(libucontext_ucontext_t*);
extern "C" void libucontext_makecontext(libucontext_ucontext_t*, void (*)
    (), int, ...);
extern "C" int libucontext_setcontext(const libucontext_ucontext_t*);
extern "C" int libucontext_swapcontext(libucontext_ucontext_t*, const
    libucontext_ucontext_t*);
struct Thread
{
    unsigned int id;
    char* stack;
    libucontext_ucontext_t* ucontext_ptr;
    bool finished;
};
struct Lock
{
    Thread* owner;
    Deque<Thread*>* blocked_threads;
};
static Deque<Thread*> ready;
static Thread* current;
static libucontext_ucontext_t* scheduler;
static bool init = false;
static int id = 0;
unsigned int lockM = 1, cond = 1;
static Deque<unsigned int> condition_map1;
static Deque<unsigned int> lock_map;
static Deque<Lock*> lock_map_list;
int thread_yield(void)
{
    if (!init) return -1;
    ready.InsertRear(current);
    libucontext_swapcontext(current->ucontext_ptr, scheduler);
    return 0;
}
class thread
```

```
43 {
44     int thread_lock(unsigned int lock)
45     {
46         if (!init) return -1;
47         //getfront
48         unsigned int lock_iter;
49         Lock* l;
50         if (!lock_map.IsEmpty())
51         {
52             lock_iter = lock_map.GetFront();
53         }
54         if (lock_map.IsEmpty())
55         {
56             l = new Lock;
57             l->owner = current;
58             l->blocked_threads = new Deque<Thread*>;
59             lock_map.InsertRear(lock);
60             lock_map_list.InsertRear(l);
61         }
62         else
63         {
64             l = lock_map_list.GetFront();
65             if (l->owner == NULL)
66                 l->owner = current;
67             else
68             {
69                 if (l->owner->id == current->id)
70                 {
71                     return -1;
72                 }
73                 else
74                 {
75                     l->blocked_threads->InsertRear(current);
76                     libucontext_swapcontext(current->ucontext_ptr,
    scheduler);
77                 }
78             }
79         }
80         return 0;
81     }
82     int unlock_without_interrupts(unsigned int lock)
83     {
84         Lock* l;
85         if (lock == lock_map.GetRear())
86             return -1;
87         else
88         {
89             l = lock_map_list.GetFront();
90             if (l->owner == NULL)
91                 return -1;
92             else
93             {
94                 if (l->owner->id == current->id)
95                 {
96                     if (l->blocked_threads->GetSize() > 0)
97                     {
98                         l->owner = l->blocked_threads->GetFront();
99                         l->blocked_threads->DeleteFront();
100                        ready.InsertRear(l->owner);
101                    }
102                    else
```

```
103                         l->owner = NULL;
104                 }
105                 else
106                     return -1;
107             }
108         }
109         return 0;
110     }
111     int thread_unlock(unsigned int lock)
112     {
113         if (!init) return -1;
114         int result = unlock_without_interrupts(lock);
115         return result;
116     }
117     int thread_wait(unsigned int lock, unsigned int cond)
118     {
119         if (!init) return -1;
120         if (unlock_without_interrupts(lock) == 0) {
121             unsigned int cond_iter1 = condition_map1.GetFront();
122             if (cond_iter1 == condition_map1.GetRear())
123             {
124                 condition_map1.InsertFront(current->id);
125             }
126             else
127                 condition_map1.InsertRear(current->id);
128             libucontext_swapcontext(current->ucontext_ptr, scheduler);
129             return thread_lock(lockM);
130         }
131         return -1;
132     }
133     void delete_current_thread()
134     {
135         delete current->stack;
136         current->ucontext_ptr->uc_stack.ss_sp = NULL;
137         current->ucontext_ptr->uc_stack.ss_size = 0;
138         current->ucontext_ptr->uc_stack.ss_flags = 0;
139         current->ucontext_ptr->uc_link = NULL;
140         delete current->ucontext_ptr;
141         delete current;
142         current = NULL;
143     }
144 public:
145     //default constructor
146     thread() { }
147     //with prominse
148     template<class C, class P, typename...R>
149     thread(C(*f), P(*n), R...r)
150     {
151         //cout << "Thread library Promis.\n";
152     }
153     //other constructors
154     template<class C, class... R>
155     thread(C(*f), R...r)
156     {
157         if (!init)
158         {
159             init = true;
160             thread_lock(lockM);
161             thread_create((*f), r...);
162             thread_wait(1, 1);
```

```
163             Thread* first = ready.GetFront();
164             ready.DeleteFront();
165             current = first;
166             scheduler = new libucontext_ucontext_t;
167             libucontext_getcontext(scheduler);
168             libucontext_swapcontext(scheduler, first->ucontext_ptr);
169             while (ready.GetSize() > 0)
170             {
171                 if (current->finished == true)
172                     delete_current_thread();
173                 Thread* next = ready.GetFront();
174                 ready.DeleteFront();
175                 current = next;
176                 libucontext_swapcontext(scheduler, current->ucontext_ptr)
    ;
177             }
178             if (current != NULL)
179             {
180                 delete_current_thread();
181             }
182         }
183         else
184         {
185             thread_create((*f), r...);
186             thread_unlock(lockM);
187         }
188     }
189     static int exec_func(thread_startfunc_t func, void* arg)
190     {
191         current->finished = true;
192         libucontext_swapcontext(current->ucontext_ptr, scheduler);
193         return 0;
194     }
195     template<class C, class ... Args> int thread_create(C(*f), Args...r)
196     {
197         if (!init) return -1;
198         Thread* t;
199         t = new Thread;
200         t->ucontext_ptr = new libucontext_ucontext_t;
201         libucontext_getcontext(t->ucontext_ptr);
202         t->stack = new char[STACK_SIZE];
203         t->ucontext_ptr->uc_stack.ss_sp = t->stack;
204         t->ucontext_ptr->uc_stack.ss_size = STACK_SIZE;
205         t->ucontext_ptr->uc_stack.ss_flags = 0;
206         t->ucontext_ptr->uc_link = NULL;
207         static const std::size_t value = sizeof...(Args);
208         (*f)(r...);
209         libucontext_makecontext(t->ucontext_ptr, (void(*)())exec_func,
    value + 1, (thread_startfunc_t)(*f), r...);
210         t->id = id;
211         id++;
212         t->finished = false;
213         ready.InsertRear(t);
214         return 0;
215     }
216 };
```

LISTING A.1: Thread Library

```
#include<iostream>
using namespace std;
#pragma once
template<typename T>
class Deque
{
  T *Ptr;
  int Front;
  int Rear;
  int Size;
  void GrowArray()
  {
    T *Temp = new T[Size * 2];
    if (Front == Rear + 1)
    {
      for (int i = 0; i <= Rear; i++)
      {
        Temp[i] = Ptr[i];
      }
      int k = (Size * 2) - 1;
      for (int j = Size - 1; j >= Front; j--, k--)
      {
        Temp[k] = Ptr[j];
      }
      Front = k + 1;
      delete[]Ptr;
      Ptr = Temp;
      Temp = nullptr;
      Size = Size * 2;
    }
    else
    if (Front == 0 && Rear == Size - 1)
    {

      for (int l = 0; l < Size; l++)
      {
        Temp[l] = Ptr[l];
      }
      delete[]Ptr;
      Ptr = Temp;
      Temp = nullptr;
      Size = Size * 2;
    }
  }
public:

  Deque(int Size = 1)
  {
    Front = -1;
    Rear = 0;
    this->Size = Size;
    Ptr = new T[Size];
  }
  bool  IsFull()
  {
    return ((Front == 0 && Rear == Size - 1) || Front == Rear + 1);
  }
  bool  IsEmpty()
  {
    return (Front == -1);
  }
```

```
62  void  InsertFront(T Value)
63  {
64    if (IsFull())
65    {
66      GrowArray();
67    }
68    if (Front == -1)
69    {
70      Front = 0;
71      Rear = 0;
72    }
73    else if (Front == 0)
74      Front = Size - 1;
75
76    else
77      --Front;
78    Ptr[Front] = Value;
79  }
80  void  InsertRear(T Value)
81  {
82    if (IsFull())
83    {
84      GrowArray();
85    }
86
87    if (Front == -1)
88    {
89      Front = 0;
90      Rear = 0;
91    }
92    else if (Rear == Size - 1)
93      Rear = 0;
94    else
95      ++Rear;
96
97    Ptr[Rear] = Value;
98  }
99  void  DeleteFront()
100 {
101   if (IsEmpty())
102   {
103     throw("Queue Underflow\n");
104   }
105   else
106   if (Front == Rear)
107   {
108     Front = -1;
109     Rear = -1;
110   }
111   else
112   if (Front == Size - 1)
113     Front = 0;
114   else
115     Front = Front + 1;
116 }
117 void  DeleteRear()
118 {
119   if (IsEmpty())
120   {
121     throw(" Underflow\n");
122   }
123   else
124   if (Front == Rear)
```

```
125     {
126       Front = -1;
127       Rear = -1;
128     }
129     else if (Rear == 0)
130       Rear = Size - 1;
131     else
132       --Rear;
133   }
134   T  GetFront()
135   {
136     if (IsEmpty())
137     {
138       throw(" Underflow\n");
139     }
140     return Ptr[Front];
141   }
142   T  GetRear()
143   {
144     if (IsEmpty() || Rear < 0)
145     {
146       throw(" Underflow\n");
147     }
148     return Ptr[Rear];
149   }
150   int GetSize(){
151     return Front-Rear;
152   }
153   ~Deque(){};
154 };
```

LISTING A.2: Deque Library

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
typedef int libucontext_greg_t, libucontext_gregset_t[19];

typedef struct libucontext_fpstate {
  unsigned long cw, sw, tag, ipoff, cssel, dataoff, datasel;
  struct {
    unsigned short significand[4], exponent;
  } _st[8];
  unsigned long status;
} *libucontext_fpregset_t;

typedef struct {
  libucontext_gregset_t gregs;
  libucontext_fpregset_t fpregs;
  unsigned long oldmask, cr2;
} libucontext_mcontext_t;

typedef struct {
  void *ss_sp;
  int ss_flags;
  size_t ss_size;
} libucontext_stack_t;

typedef struct libucontext_ucontext {
  unsigned long uc_flags;
  struct libucontext_ucontext *uc_link;
  libucontext_stack_t uc_stack;
  libucontext_mcontext_t uc_mcontext;
} libucontext_ucontext_t;
```

LISTING A.3: Structures in Thread Library

# Appendix B

# Resuls

```
__attribute__((noinline))
void kernel(size_t i3, Graph* graph, NodeId var_2, PropertyId p3,
    PropertyId p4, PropertyId p5, PropertyId p7, PropertyId p9, PropertyId
     p11, size_t in_degree2, Edge* var_2_1_inEdges)
{
    unsigned localCounter = 0;
    PropertyId var_3;
    var_3 = var_2_1_inEdges[i3].property;
    NodeId var_1;
    var_1 = var_2_1_inEdges[i3].node;
    int  cond_level_2 = (var_3 == p3);
    if (cond_level_2)
    {
        size_t out_degree1 = getOutDegree(graph, var_1);
        Edge* var_1_3_outEdges = getOutEdges(graph, var_1);
        size_t i5;
        for (i5 = 0; i5 < out_degree1; i5++)
        {
            PropertyId var_5;
            var_5 = var_1_3_outEdges[i5].property;
            NodeId  var_4;
            var_4 = var_1_3_outEdges[i5].node;
            int  cond_level_4 = ((var_5 == p5) & (var_4 == p4));
            if (cond_level_4)
            {
                Edge* var_1_5_outEdges = getOutEdges(graph, var_1);
                size_t i7;
                for (i7 = 0; i7 < out_degree1; i7++)
                {
                    PropertyId var_7;
                    var_7 = var_1_5_outEdges[i7].property;
                    NodeId  var_6;
                    var_6 = var_1_5_outEdges[i7].node;
                    int  cond_level_6 = (var_7 == p7);
                    if (cond_level_6)
                    {
                        Edge* var_1_7_outEdges = getOutEdges(graph, var_1
    );
                        size_t i9;
                        for (i9 = 0; i9 < out_degree1; i9++)
                        {
                            PropertyId var_9;
                            var_9 = var_1_7_outEdges[i9].property;
                            NodeId  var_8;
```

```
42                                  var_8 = var_1_7_outEdges[i9].node;
43                                  int  cond_level_8 = (var_9 == p9);
44                                  if (cond_level_8)
45                                  {
46                                      Edge* var_1_9_outEdges = getOutEdges(
    graph, var_1);
47                                      size_t i11;
48                                      for (i11 = 0; i11 < out_degree1; i11++)
49                                      {
50                                          PropertyId var_11;
51                                          var_11 = var_1_9_outEdges[i11].
    property;
52                                          NodeId  var_10;
53                                          var_10 = var_1_9_outEdges[i11].node;
54                                          int  cond_level_10 = (var_11 == p11);
55                                          if (cond_level_10)
56                                          {
57                                              localCounter++;
58                                          }
59                                      }
60                                  }
61                              }
62                          }
63                      }
64                  }
65          }
66          atomicOP(&counter), localCounter);
67      }
68  }
69
70  __attribute__((noinline))
71  void parallel(Graph* graph, NodeId var_2, PropertyId p3, PropertyId p4,
    PropertyId p5, PropertyId p7, PropertyId p9, PropertyId p11, size_t
    in_degree2, Edge* var_2_1_inEdges)
72  {
73      size_t i3;
74  #pragma omp parallel
75      for (i3 = 0; i3 < in_degree2; i3++)
76      {
77          thread(kernel,i3, graph, var_2, p3, p4, p5, p7, p9, p11,
    in_degree2, var_2_1_inEdges);
78      }
79  }
```

LISTING B.1: Multi Threaded Sample Code