POLITECNICO DI TORINO

Master Degree in
Computer Science - Software Engineering

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:
Pierpaolo Iannicelli

Luglio 2020

Acknowledgments

Ringrazio infinitamente mia madre e mio padre per avermi spronato ad intraprendere l'esperienza universitaria in una città diversa da quella di residenza, aprendomi le porte verso un mondo nuovo e meraviglioso, infinitamente grazie per aver sostenuto in seguito ogni mia scelta, in particolare l'esperienza erasmus in Cile, che sicuramente non dimenticherò mai.

Un grazie di cuore ai miei fratelli e agli amici di una vita, che hanno sempre reso piacevole il mio ritorno a casa dopo estenuanti sessioni di esami.

Un ringraziamento speciale al mio relatore Ernesto Sanchez e al mio correlatore Alessandro Savino che mi hanno seguito nei vari step del lavoro, consigliandomi e indirizzandomi verso l'obiettivo finale.

Un particolare ringraziamento va al mio secondo correlatore Michele Portolan che mi ha accolto calorosamente a Grenoble dove ho svolto per un periodo la tesi, e che mi ha fatto sentire fin da subito a mio agio in una città nuova, senza nulla togliere al supporto che anche lui ha fornito per il completamento dell'elaborato.

Infine ma non meno importanti ringrazio le nuove amicizie che ho maturato durante il periodo universitario che hanno alleviato notevolmente le difficoltà che ne sono derivate dal trasferimento nella città universitaria e dallo studio.

Abstract

Modern embedded system design has a greater complexity than in the past, and it is increasing more and more. High Level Synthesis (HLS) increases the design abstraction level and thanks to it, it's possible to generate with less effort, optimized register transfer level (RTL) hardware in terms of performance, area and power requirements. This thesis propose an application of the new multithreading synchronization paradigms introduced in the C++14 standard, such as futures and promises, to the design of digital electronics components, with a special attention to High Level Synthesis.

Table of contents

List of figures

List of tables

Listings

Chapter 1

Introduction

1.1 Premise

HLS is about synthesizing, generally a C or C++ function into an RTL implementation in the replacement of traditional RTL design concepts, using a behavioral design language such as Verilog or VHDL. High level design or high level synthesis provides different components of functionality. First point is to develop an IP component in a software environment but also to functionally verify that component in the same software environment and integrate that IP into an hardware simulation environment where is needed to verify at a signal level its functionality. Then starts the optimization phase of that design and so there is a lot of static reports that gets generated, as well as performance results from those simulations run. Lastly the tool allows to easily generate an IP to integrate it in the traditional FPGA design tool as part of the FPGA design. The motivation for high level synthesis is simple when looking at a traditional FPGA design process, it is in general a fairly time consuming effort so it always start with a hardware description language such as Verilog or VHDL. Then it is needed to write the testbench, and then run it in a hardware simulator such as modelsim. Once it's functionally verified at the RTL level, run it through a logic synthesis[1] tool and then run the placement and route inside a software. This is actually a fairly time consuming process, so the goal of HLS is to increase the productivity of designing hardware architectures, getting the benefit of the performance of running the IP on an FPGA without going through the lengthy development times and optimization times. With HLS it is possible to develop at much higher level, therefore increase productivity, being able to debug software much faster than hardware because it's possible to functional debugging

[1] In electronics, logic synthesis is a process by which an abstract specification of desired circuit behavior, typically at register transfer level (RTL), is turned into a design implementation in terms of logic gates [4]

staying with a software debugging tools and utilities rather than using hardware simulator or on chip debugging techniques which are much more time consuming. It is important to specify the functions to accelerate the software so it is needed to write a testbench as well as a component in C or C++ environment, then easily indicate to the HLS software the exact component to implement in hardware.

The idea of this thesis starts from the introduction of the quite new features of the C++ language, futures and promises, whose functionality are later explained. This work proposes a possible application of this new features in the field of the HLS process. Starting from some functionality of the HLS tool Bambu, a new mechanism that is able to translate futures and promises in an RTL implementation is proposed.

1.2 Work introduction

One significant limitation in using high level synthesis tools is that the high level language accepted, in order to be able to obtain the RTL description, is only a subset of the whole language and this limits the programmer that cannot exploit the real power of such programming language.

Nowadays is not only important to write programs, but to write optimized programs in terms of resources utilization and times. Thanks to compilers some optimization phases are performed automatically, but they are not able to identify the sections that can be executed in parallel. For these purpose the threads are used but is totally up to the programmers writing an optimized and functional code. In order to synchronize them, various techniques are adopted.

In the field of high level synthesis there are already some works introducing the threads in the description language that a tool can synthesize but no one is talking about the possibility of using the relatively new features of the C++. This Master thesis work, aims to first of all analyze the available open source high level synthesis tools and to choose the one that fits best to our case. Second, using such a tool to give a possible application of futures and promises together with the threads in the context of high level synthesis. This work demonstrates that it is possible to apply such a features in this context, the methodology proposed is not implemented directly in the tool itself but starting from the verilog code generated by the latter it was modified to obtain the desired result. This is the starting point to implement the methodology inside a tool.

Generally, high level synthesis tools synthesize the input source code trying to parallelize as mush as possible the execution of the tasks, but the result is not always optimal and even if something could be executed in parallel, it is scheduled in sequence. Giving the possibility to the programmer of using these features, the expected improvement is not only in the execution time of the obtained system that should be a way lesser, but also in the creation of the synchronization graphs by

the tool that is expected to be faster. Moreover the set of the accepted language is enlarged.

The next section gives an idea of how this work is organized in its chapters.

1.3 Overview of the thesis

This thesis is divided in 7 chapters describing in detail the application of futures and promises analyzed in the previous section and an appendix providing C++ and verilog codes.

Chapter 2 contains a presentation of some basic concepts needed to understand the rest of the work.

Chapter 3 presents the tool that was used to perform high level synthesis, in particular talks about the peculiarity that are useful for our goal.

Chapter 4 presents the proposed methodology and the strategy followed to obtain the desired result.

Chapter 5 gives details about the implementation of thread, futures and promises in the architecture generated by the high level synthesis tool.

Chapter 6 describes the results and the case studies used to obtain that, focusing more on the timing obtaining through simulation.

Chapter 7 is the conclusion of the thesis with a final discussion and presents some future works.

Chapter 2

High Level Synthesis and C++

2.1 Introduction to High Level Synthesis

In the HLS model use, starting point is the high level language code, where there is the main function which could call lots of other functions and it's possible to run it in a traditional compiler. Suppose we are talking about C/C++ code, we can run it using GCC or G++ which will generate the executable file. If we want to convert certain of these functions into an hardware component, it's enough to use compiled specific directives, mark those functions and then run the entire C or C++ code with the HLS compiler instead of the GCC or G++ compiler. An HLS compiler will generate the specific IPs for each of the functions that are marked and then it's possible to integrate that particular IP for example into an FPGA design. Some of the HLS compiler are also able to generate emulation environment to allow to functionally debug the entire design. A designer will tipically approach to HLS writing the behavioral specification using an high level language to describe the functionality of a module that is to be implemented, like a FIR, a controller or a custom hardware. This step is an abstraction level above the RTL specification in which everything is untimed, there is no delay and the data types of the variables are not related to the hardware. HLS tools transform this untimed and hardware unrelated specification in a custom architecture that correctly implement the described behavior [1].

2.2 High Level Synthesis flow

The HLS tools flow showed in figure 2.1 is similar to the organization of a compiler. It starts with a front-end which is responsible of performing lexical and syntactical analysis for then producing an intermediate representation (IR) that is optimized using the usual techniques such as the dead-code optimization, loop unrolling or function inlining. Starting from this IR, a formal model is produced in which are

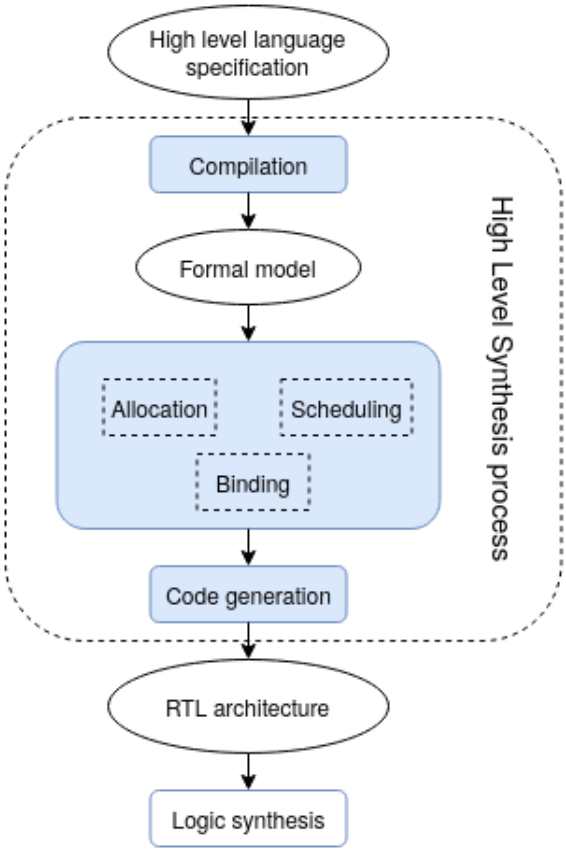


Figure 2.1: HLS flow

shown data and control dependencies. All these information are represented by the tool using for example a data flow graph and a control flow graph in which a node refers to a basic block operation and the edges represents the execution flow of this basic blocks. Depending on the tool this information could be showed to the user or not, but the tool internally use this in order to perform the following steps in the flow. Next there is the back-end that perform some steps that generate the RTL representation. These steps could be summarized in allocation, scheduling, binding and code generation.

The allocation phase deals with, based on the design constraints, the choice of the components and functional units used to obtain the desired behavior.There are libraries contained by the tools that implement these modules. The choice is made not only depending on the functional behavior but also by some time or area performance goal of the design and so the libraries also contain the metrics for each

module. The final result from the allocation phase is the selection of at least one component contained in the libraries for each operation described in the IR.

Scheduling is about mapping the single operation in clock cycle, so for example a simple operation like $a\ op\ b$ it is expected that variable a and b are read and bring to the functional unit that was chosen in the previous phase to perform this operation. Depending on the unit that execute it, the operation can be scheduled in one or more clock cycles. Notice that if there are no data dependencies between two operations and there is enough resource availability, it should be possible that they are executed in the same clock cycle. So the final objective of the scheduling is to assign operation to clock cycles satisfying the constraints.

In binding phase both operations and values are respectively mapped to functional units and storage resources. The objective of this phase is to optimize the mapping. It means that if it's possible share the hardware between more than one operation or value and if more than one unit could be bound to an operation, the choice should be made on the metrics this unit has. Finally is performed the connectivity binding that is interconnect resources introducing other logics to perform the transfer between components.

Before continuing with the last step, a note on the three previous phases must be added, these are interdependent and so optimizing first one of the three could be incompatible with the optimization of another, so the choice made in one phase could influence the others. In this kind of problem the execution order of these is very important.

The code generation is just the application of the previous design choices and generates an RTL model using an hardware description language such as verilog or VHDL that is synthesizable [1] [5].

The generated architecture is usually organized in two blocks showed in figure 2.2, a data path and a controller implemented as a final state machine. The first contains all that components that are used to move and manipulate the data of the system, like registers, multipliers, shifters and multiplexers these components are linked and executed in clock cycles thanks to the previous phases. The controller is the mind that lead the data flow using control signals. Usually it has a status register containing the current state in which the controller is. Based on it and on the control input it has a next state register saying which is the state changing for the next clock cycle, so follows the behavior of a final state machine. Obviously the system has to communicate with the external world to receive inputs and give outputs. These are divided in data and control input and output connected respectively to the data

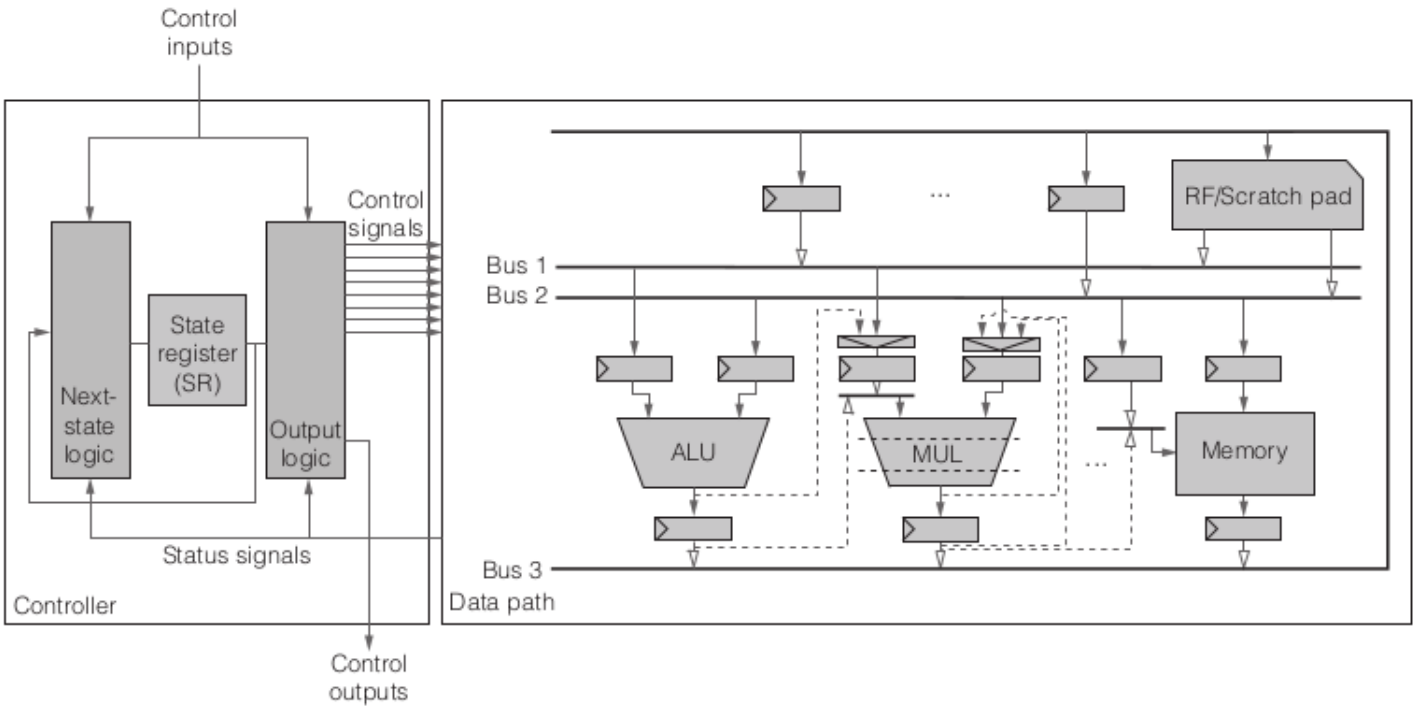


Figure 2.2: Typical architecture [1]

path and controller. This architecture is not fixed and there could be something different from what previously described [1].

2.3 High Level Synthesis tools

Due to a growing increase in the complexity of digital systems design and verification, tools are needed to make this process simpler and faster. HLS tools are one of this design automation that given as input an high level language so a behavioral program that describe the system you want to produce, it generates the equivalent hardware. It's possible to list some of the advantages of having an HLS tool. For example with this kind of tools it is not necessary anymore to manual translate a behavior in an RTL that is very time consuming, the program size of such input languages are smaller and more readable respect to an hardware description language (HDL), HLS generate code that is optimized and so probably use less components that the handwritten HDL and moreover it supports verification useful when a very reliable system is needed [6]. To explore the HLS tools it was necessary to select some criteria of evaluation.

First of all it's important to understand which source language could be used and which restrictions are made on the language. This is one of the point of this thesis, that try to reduce the gap between algorithms and hardware design introducing in the accepted C++ language the relatively new features, futures and promises. The more of the language can be used the more comfortable the programmer feels.

Another aspect is how hard is to use the tool, if it has or not a user interface, how complex is the tool and if it's enough documented or not.

As said before, verification is an important task in the design process, some tools are able to automatically generate testbenches for the generated design giving again the opportunity to be more accurate and saving time.

Other two aspects not ignored for this work are the operating system on which the tool could run and if it is open source or a proprietary one. More detail on the motivation of this choice are described in chapter 3.

A table with some tools is provided in table 2.1 showing the property listed before. More tools with different evaluation criteria are provided here [7].

The following subsections give an overview on some of the HLS tool taking into account.

2.3.1 GAUT

It is an open source tool developed at Université de Bretagne-Sud and is supported by Windows and Linux operating systems. The input language accepted by this tool is the C/C++ and generates an equivalent RTL code written in VHDL. It is one of the tools that automatically generate a testbench that can be simulated using third party tools like modelsim. A great strength of GAUT is that it uses an eclipse IDE so is quite easy to use it, and inside the IDE are shown some useful information regarding the synthesis such as which components (adder or multiplexer) are used and in which number, the data control flow (CFG) and the schema of the finite states machine. On its website[1] there is a small guide and a video to help to start using the tool but there is only a little documentation for the user level so it could be difficult to understand how does it works. Another aspect is that it works fine with the given examples but has some problem when trying to change them. Unfortunately the project is now with a very low priority and so it is not updated anymore.

2.3.2 MyHDL

MyHDL is not really a tool, is an open-source package of Python thanks to which it is possible to write Python code mixed with this library that has constructs very similar to an hardware description language and it's possible to use it for

[1] http://www.gaut.fr/

Tool	Tested on	Input	Output	Open source	IDE	OS	Testbench
ROCCC	No	C	VHDL	Yes	Eclipse	Linux	No
Bambu	Ubuntu 18.10	C/C++	Verilog VHDL	Yes	No	Linux	Yes
GAUT	Ubuntu 12.04 Ubuntu 18.10	C/C++	VHDL	Yes	Eclipse	Linux Windows	Yes
Icarus Verilog	Ubuntu 12.04 Ubuntu 18.10	iVerilog	VHDL	Yes	No	Linux Windows	No
LegUp	No	C	Verilog	Yes	No	Linux	Yes
MyHDL	Ubuntu 16.04	Python	Verilog VHDL	Yes	No	Linux Windows	Yes
xPilot	Ubuntu 12.04 Ubuntu 18.10	C SystemC	VHDL	Yes	No	Linux	No
Kiwi	Ubuntu 18.10	C#	Verilog SystemC	Yes	No	Linux Windows	Yes
Catapult C	No	C/C++ SystemC	Verilog VHDL	No	Yes	Linux	Yes
Agility compiler	No	SystemC	Verilog VHDL	No	No	Linux	No
Vivado HLS	Ubuntu 18.10	C/C++ SystemC	Verilog VHDL	No	Yes	Linux Windows	Yes
BlueSpec	No	BlueSpec System Verilog	System Verilog	No	BlueSVEP Eclipse-based	Linux Windows	No
C to silicon	No	C/C++ SystemC	Verilog SystemC	No	Stratus by cadence	Linux	No
Synphony C compiler	No	C/C++	Verilog VHDL	No	No	Linux	Yes
Cynthesizer	No	SystemC	Verilog	No	No	Linux	Yes

Table 2.1: HLS tools

simulation and verification of a design. The MyHDL code can also be converted into Verilog or VHDL code and then continue the synthesis using a third-party tool. Python is a very powerful and high level programming language and so using it for simulation and design purpose could be very simple and useful. As usual there are few limitations in the Python language to be converted. There is available a complete documentation on the website and are also provided some examples that

makes it easier to start with and is also supported by a community. Listing 2.1 shows a simulation example of usage of MyHDL in Python taken from the available examples. Inside the always block we can write Python code with some limitations while outside it's possible to exploit all the power provided by the language [8].

```
from myhdl import block, delay, always, now

@block
def HelloWorld():

@always(delay(10))
def say_hello():
print("%s Hello World!" % now())

return say_hello


inst = HelloWorld()
inst.run_sim(30)
```

Listing 2.1: MyDHL example

2.3.3 Kiwi

This is an open source project developed at university of Cambridge, it is basically a compiler that converts Csharp bytecode into Verilog or SystemC. The tool has an IDE and is supported by Windows and Linux operating systems. Compared with others tools, it has some particular point to underling, first is the input language that uses Csharp, supports dynamic allocation of objects and floating point manipulation, it's possible to control clock cycle from the Csharp file and also supports some recursive programs. The user guide provided is exhaustive, gives information on how to correctly install and setup the working environment, the language subset limitations giving also some examples and a guide for developers that explain which are the internal operations and how the tool works [9].

2.3.4 Vivado HLS

Vivado HLS is a proprietary tool by Xilinx that extends the already existing tool Vivado HLx for synthesis design. So Vivado is a complete tool chain that starting from the high level language that can be C/C++ and SystemC, produces the bitstream that can be inserted in the targeted hardware. It has its own IDE that simplify the synthesis and verification of the design, moreover it has an additional component

for simulation and there are a lot of information provided at the end of synthesis such as power consumption, the number of components used and information on the timing. There is a well done documentation [10] supported also with video tutorial.

2.4 Futures and promises

Futures and promises are used to handle concurrency, these concept allow a program to pass values between threads without using any typical synchronization mechanism used when sharing data between threads like mutexes. Some troubles are centered around the terminology, various terms have been used by different researchers developers and computer languages to mean very similar things, it can be challenging which definition is being used in a given context. Futures and promises are useful when you have some operation or set of operations which produce a result could take a significant amount of time or may not need to happen in any particular order a good example of when you might use future and promise to improve performance is when you are reading or writing data like opening a file in an editor, ideally it would be good to allow the user to continue interacting with the program after the first portion of the file has been loaded, the remainder of the file could then be loaded in the background. Or there could be a system which is making a call to a web service using a raw TCP socket and you don't want to wait for this operation to complete before moving on to another task. Another example is when doing database queries which often have long running tasks.

In the late 70s the term promise was first introduced and then a year later the idea of future was introduced. Over the next 20 years they were implemented in a few lesser-known computer languages, these construct were considered mostly theoretical at the time. They were not originally developed to solve the modern-day problems we have with networks web servers and distributed systems. 30 to 40 years later were finally realizing the value of this technology. Future and promise were not widely known until early 2002 when Python introduced the notion of what they called deferred objects in a library called twisted. The boost library introduced one of the first thread libraries for C++ in 2001, it wasn't until 2009 that the boost thread library added functionality to support future and promise. With the release of C++11 future and promise became part of the standard library. There have been several new proposals to expand the functionality of future and promise in the C++ standard and these are available as experimental in some compilers, it is expected that C++20 will contain most of these enhancements [11].

When you are looking at computer language or browsing through code samples you need to be very careful, not every computer language has both constructs namely futures and promises, and the definitions and implementations may be different. If you are using future and promise in a language other than C++ these constructs

may not be in the core language or standard library in most all cases future and promise will be supported in various third-party libraries.

For the purpose of this thesis we will focus on C++ features. In C++ futures and promises are instances of a template class, defined in listing 2.2.

Line code 2 7 base template [2].
Line code 3 8 non-void specialization, used to communicate objects between threads.
Line code 4 9 void specialization, used to communicate stateless events.

```
1 \\Future class
2 template< class T > class future;
3 template< class T > class future<T&>;
4 template<>          class future<void>;
5
6 \\Promise class
7 template< class R > class promise;
8 template< class R > class promise<R&>;
9 template<>          class promise<void>;
```

Listing 2.2: Definition of Future and Promise [3]

A future object is read-only, its purpose is to encapsulate a data that may not be available yet but will be provided at some point. The template parameter for the future template indicates the type of data the future will hold. The primary way you work with a future object is to call the get method, calling get will retrieve the data stored in the future, if the future is not yet ready, meaning the data has not been provided the call to get will block. If for some reason an error occurred while computing the value, the get method will throw an exception.

The template parameter of the promise object indicates the datatype of the stored value. The T for the promise and the T for the future must match. This may seem a bit odd, but if you create or instantiate your own future object it will always be invalid and it cannot be used for anything, there is no way to make this future valid. The only public constructor for a future creates an invalid empty future, to create a usable future you first create a promise, when the promise is instantiated it automatically creates an invalid but usable future object which you can then extract

[2] A base template class is related to the idea of inheritance in object oriented programming, it's on an higher level of the hierarchy respect to the derived classes, which can inherit depending on some types, attributes and member functions [12]

from the promised object if this future will become valid when the promise fills in the data, this is called fulfilling a promise.

The purpose of the promise object is to guarantee that some function will compute some value and make it available in the corresponding future object. A typical usage involves first creating a promise object, from the promise object, future is extracted and hold on to it then the promise is moved to another thread or function. Once the function has finished it is responsible for setting the value in the promise and the future becomes available.

At any point the main thread can call a get method of the future object to wait for the data in the future to be available. If the function has already finished, the call to get will return immediately, if it is not available the call get will wait until the future is finished. Figure 2.3 shows a typical work flow.

If the function fails what happens is that, if the promise contains an exception, when the main thread calls the get method the exception in the promise will be thrown, there are two ways the promise might contain an exception. If the function encounters an exception instead of actually throwing it should place the exception object in the promise, this allow the exception to be thrown later when get is called and not in the function itself. It would be a very bad practice to throw an exception in a thread since it is unclear who should catch it and according to the C++ standard throwing an exception in a thread calls STD terminate. If the function does not set a specific exception in the promise then the promise will go out of scope without being fulfilled. When this occurs the promise automatically sets a generic exception in the future, this is called a broken promise exception. This is accurate, since the promise was not fulfilled and the function has therefore broken its promise to compute a value. The problem with relying on the broken promise exception to be set is that actually does not indicate why the promise was broken and it's not a very meaningful error.

Listing 2.3 shows a simple example that uses a future to store a computed value generated as a result of some function. The main function creates a promise and extract the corresponding future, a new thread is then started to run the doWork function and the promise object is moved into the thread. The main thread then calls get and prints the result, the thread are joined to complete the process. Notice the T in STD promise and STD future are both int, if you were to change the T for STD future to a double you will get a compile error saying something like conversion from future int to future double requested. The doWork function will run at some point after the thread has started and before the call to get returns, since the main thread cannot continue until the value is available it waits. If the main thread runs first it may be blocked until doWork has completed. The reason the promise and future are separate objects is to encapsulate the two different sets of functionality, the promise is used by the function which is responsible for computing a value in order to store the return value or exception in the corresponding future. The future

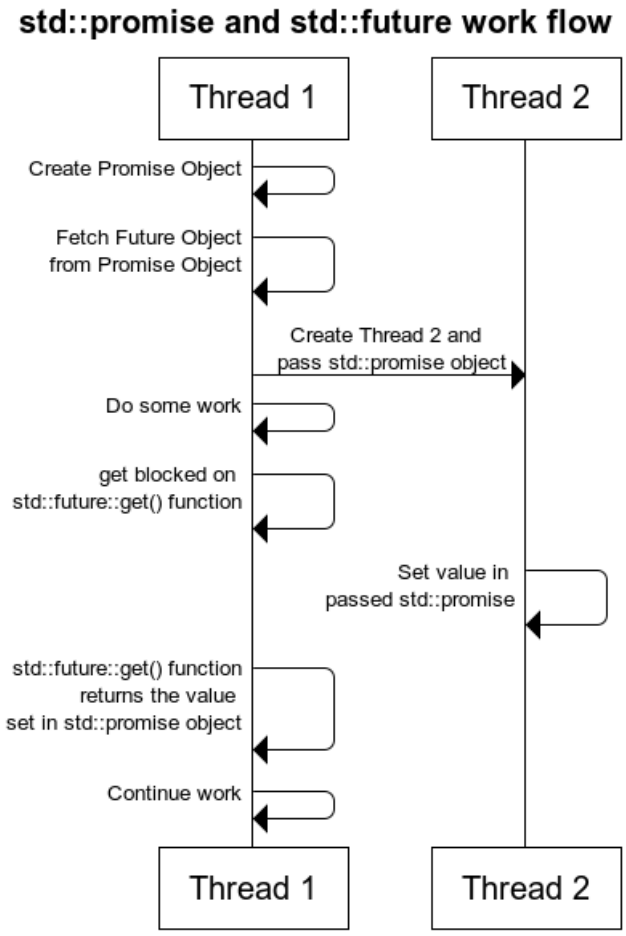


Figure 2.3: Typical future and promise work flow [2]

is used by the caller to retrieve the information which was computed, this is why the get method exists in only the future and the set_value method exists in only the promise.

It's also important to notice that for each promise there is only a future object linked to it and you can use them once. This means that is not possible to link for example two different future to one promise or vice versa and is not possible to call two times the get method on a future neither the set_value on the promise.

The standard library provides a function called std::async which encapsulates a portion of the complexity of setting up a promise and a future. Async provides the abstraction of calling a regular function in another thread, the return value of the thread will automatically be returned in the future. Internally async manages the promise and calls the set values method in the promise when the function returns. If

there is a failure async will call set exception to indicate the problem. Basically async provides a higher level mechanism to call an ordinary function in another thread and retrieve the value when it's ready. It is valuable to note that the function which is called does not have to handle futures or promises in any way, it can be a normal function which takes parameters and returns a value. This means async allows to call existing function in a separate thread without modifying the original function. However this functionality is not a subject of the thesis.

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

void doWork(promise<int> * promObj){
int x;
cout<<"Do some work here"<<endl;
promObj->set_value(x);
}

int main(){
int x;
promise<int> promiseObj;
future<int> futureObj = promiseObj.get_future();

thread th(doWork, &promiseObj);
//retrieve the value from thread function
x = futureObj.get();
th.join();
return 0;
}
```

Listing 2.3: Future and Promise sample [3]

Chapter 3

PandA Bambu

3.1 Features

PandA [13] project is a framework developed at Politecnico di Milano, as mentioned on the website "*The primary objective of the PandA project is to develop a usable framework that will enable the research of new ideas in the HW-SW Co-Design field*".

Bambu is the name of the open source tool that implements High Level Synthesis, the language accepted as behavioral description is a subset of C and C++ and generates as output a Verilog or VHDL code that is correctly synthesized by some commercial tool and so can be used as starting point for a complete synthesis process.

The tool is developed for Linux systems and is written in C++, an overview about the flow and how does it works is given in section 3.2. Its internal design is modular, in fact implements the tasks of the HLS process in different C++ classes, using different internal representations according to the stage involved.

There is no a user interface but everything is managed by the command line, several options are provided to control some parameter in the various synthesis phases, it's possible to include option for the compiler, to get some outputs such as verbose printing for debugging purpose, it's possible to specify the algorithm used for scheduling, binding and memory allocation as well as constraints. A lot of graphs are used to extract information on the input code and these graphs are built using the boost::graph library. Including an option the graphs are given in a .dot format file that is a very interesting output.

Most of the tool restrictions on the language deal with the usage of pointers, that should be statically allocated at compile time to be synthesized, Bambu override this restriction and is able to manage these situations producing a working architecture. This work focuses on this aspect and in particular on the method of synthesis of function pointers, that is modified allowing the high level user the possibility to write in his code the C++ futures and promises.

The same mechanism used to translate function pointers can be found in the tool in another situation. A state-of-the-art for function calls translation is to instantiate in the data path of the caller the module implementing the function, the module may receive input and produce output according to the function definition. If a function is called more times from different callers, it's possible that the corresponding module is also instantiated more times, increasing the area of the design but improving the speed. According to some criteria and thresholds Bambu can avoid this creating only one module for each function call and sharing it, reducing the area against a bit of speed. Think about it as function inlining if we talk about software compilers. More details useful to then understand the introduced new approach can be found in section 3.4.

3.2 Tool flow

Like traditional tools the execution flow is similar to software compilation, and can be divided in three part.

The Bambu front-end can exploit, depending on the user choice, the GNU Compiler Collection or LLVM/Clang compiler to parse the source language and thanks to a plugin generate a GIMPLEpssa intermediate representation that is similar but not completely equal to the one generated by GCC. An advantage of exploiting these compiler is that code optimization are performed and the traditional compiler option are accepted by the tool, in fact as shown in figure 3.1 the intermediate representation file is extracted after the middle-end optimization of GCC. This file is then given in input for the Bambu middle-end.

In the middle-end others transformations are applied to the IR to cleans up the code, simplifies memory access and do other stuffs simplifying the next steps. There are also some FPGA oriented transformations that transform some multiplications or divisions into simpler expressions, for example a multiplication like $3 * x$ becomes $(x << 1) + x$. Then the file GIMPLEpssa is parsed by Bambu and from this creates a data structure like a graph in which each node contains the instruction with all the information related to it. Once finished, the graphs that describe the dependencies between the instructions and between the basic blocks are constructed and will be used to obtain the correct design.

Finally in the back-end the HLS is performed. The synthesis process is applied on each function and so the architecture obtained is modular and follow the structure of the call graph. Each function is composed by a data-path and a controller. Here are performed all the steps already discussed in section 2.2, but now let's see

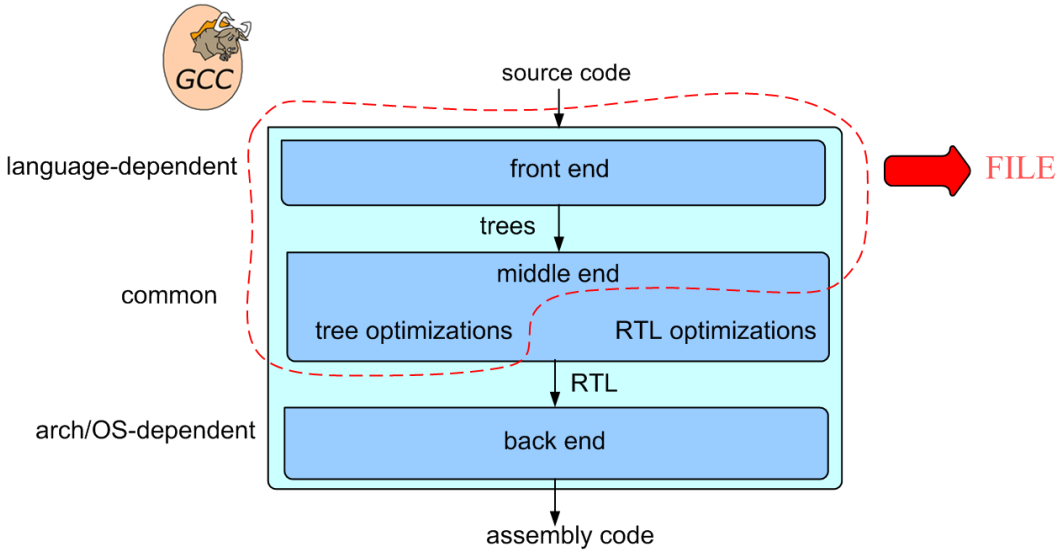


Figure 3.1: GCC compiler

some peculiarity implemented in Bambu.

The allocation phase is divided in three, functions allocation defines the hierarchy of the modules, memory allocation specify the way the variables are stored and how the dynamic memory is implemented, thanks to the options it's possible to choose a bunches of setting for memory allocation such as the algorithm used, the policy or the base address. Finally the resource allocation maps operation to functional units, the library used is rich and includes various units for each operation with different characteristics i.e. latency or resource occupation.

The scheduling algorithm used here is by default a list-based one that associates a priority to operations following some metrics. At the end of the steps each operation is associated to a clock cycle. There are options to specify other algorithms to use or provide a fixed scheduling in an XML file.

The binding is dependent from the previous phase because operations that was scheduled to be executed in concurrency cannot share the same units in order to avoid conflicts. This step is performed as a weighted graph coloring problem and the graph is built looking at the scheduling according to some criteria.

Figure 3.2 shows the complete Bambu process flow.

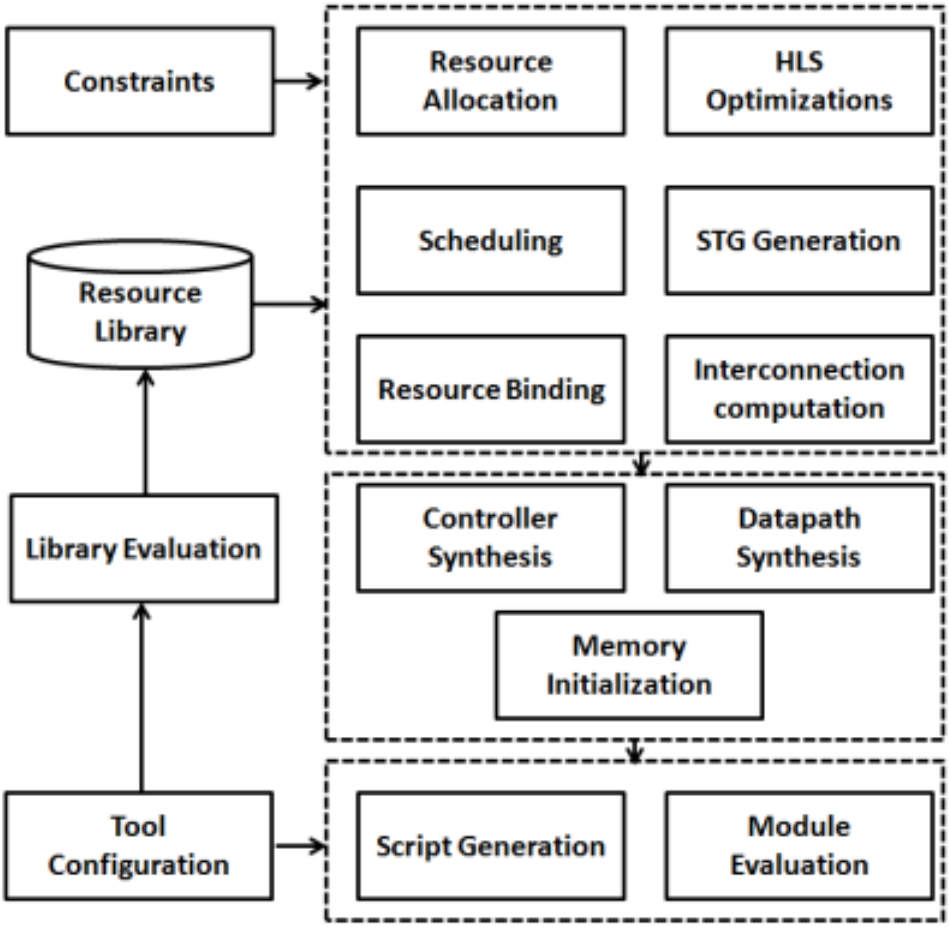


Figure 3.2: Bambu tool flow

3.3 Motivations

The aim of this section is to motivate the choice of Bambu over the others to introduce future and promise in the HLS context. It is important to specify that this section is not intended to represent Bambu as a better tool than others but only to provide the reasons why Bambu is the best choice among the tools analyzed for this specific work, moreover not all the tools available have been analyzed in detail.

The first point to analyze, which obviously excludes about half of the possible choices, is that the tool must be open source for various reasons. The source code can be modified introducing new algorithms or optimization techniques in some specific steps of the process to obtain the required new behavior. This thesis doesn't aim to modify the source code of the tool but is a starting point for doing that in the future.

Having the code is possible to compile it on different operating system environments and online forums created by programmers that works on open projects can be used as support for a beginner.

The idea of this work is based on the concepts of futures and promises, which not all high-level languages have and therefore this is a discriminant. From the table 2.1 showed in the previous chapter is possible to see that almost the majority of the proprietary tools supports C++ but very few of the open ones does it. It's important to notice that futures and promises are not only implemented in C++, but there are some construct also in Python. Regarding Python a similar mechanism is provided by using the sub classes of the Executor, ThreadPoolExecutor and ProcessPoolExecutor abstract class. We focus on ProcessPoolExecutor which therefore takes advantage of multi-processes rather than multi-threading because Python does not support a real parallel execution of threads cause they are scheduled and blocked by the Global Interpreter Lock which executes them only one at a time but supports real multi-processes. It is well known that comparing C++ with Python in terms of speed C++ is much faster, so speed as in development time, Python hands down, simulation speed is less straightforward.

Compared with others tools Bambu provide a lot of information both in graphical and written form, that helps understanding the internal mechanism. Some tools are now abandoned or without documentation, some others accepts only provided example while introducing an own one does not gives a correct output. Moreover the design architecture provided by Bambu is quite standard and so could be easier introduce new modules understanding how it works. The Verilog code given by Bambu can be introduced in other third-party tools to perform the logic synthesis obtaining a more detailed results.

Bambu already provide a mechanism that supports function call using function pointers that is needed to call the thread method necessary to introduce futures and promises. This mechanism is a key point for the new methodology introduced in this work and is explained in the following section.

3.4 Function call mechanism

A usual approach of HLS flow to generate a design from the hierarchical point of view is to follow the schema of the call graph of the input specification, and typically a function in this graph corresponds to a module instantiated in the data path of the caller. In some tools what happens is that, if different modules call the same function, the latter is instantiated in each of the data path of the callers, so they can't share this resource. Another weakness of usual HLS tools is the impossibility of usage of function pointers to call a function due to the non static resolution of it [14]. Inside Bambu there is a function call mechanism that solve both the

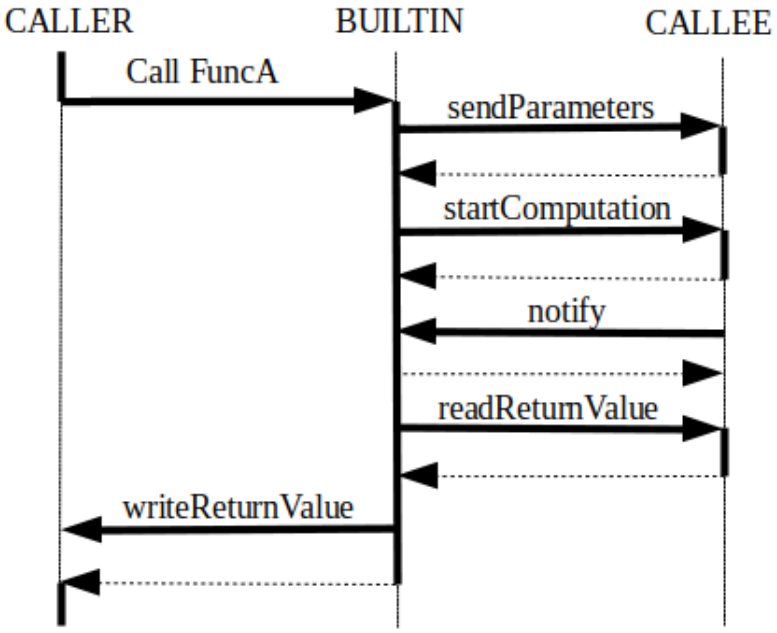


Figure 3.3: Diagram of the notification mechanism

problems described called builtin wait call and this is the starting point for the new architecture proposed in this work introducing futures and promises in the C++ input language synthesizable by the tool, so this section describes the mechanism.

The methodology extends the modules using a memory mapped interface, a module in between the caller and the callee, let's call it builtin and a communication protocol based on master and slave signals, having as final result an architecture interconnected through a bus.

Figure 3.3 shows the sequence diagram of the mechanism in which the caller asks through the builtin the usage of the callee, so transfer the control to the builtin and waits until the execution of the called module is terminated. The builtin send the parameters and make the computation starts and retrieve the return value from the function, everything is done using the memory mapped interface.

Looking at the prototype of the function is possible to obtain the memory mapped interface, in fact the inputs of the function are translated in input registers, the same for the output if present, and then independently from the prototype is instantiated a status register and a notify caller module. The status register is in charge of save the state of the function, starts the computation and intercepts when the computation is completed and the notify caller module will warns the caller about the completion of the task. In the memory allocation phase Bambu select an ID for each function and this ID is a unique base address associated to each

interface so thanks to the communication protocol if more modules want to call the same function will use the same base address.

An example of how the code in listing 3.1 is translated in a design architecture is shown in figure 3.4. It's important to notice some features of the generated architecture, when the wait call mechanism is used. One waitcall module is instantiated inside the data path of the caller for each call plus one register to store the return value if it is present, there is one bus merger in each data path involved in the call, this module is discussed in the next section. The callee is always instantiated in the data path of the top level module, and the callee architecture is always the same regardless the number of calls it receive.

```
1  int funcA( int (*func_pointer)()) {
2  int res=0;
3  res = (*func_pointer)();
4  res += (*func_pointer)();
5
6  res += funcB(18);
7
8  return res;
9  }
10
11 int start_point() {
12 int res;
13 int varC = 10;
14
15 res = funcA(funcC);
16 return res;
17 }
18
19
```

Listing 3.1: Code sample

3.4.1 Master and slave chain

Master and slave signals are used in Bambu to start memory operations and are a key point for the communication protocol presented in the next section. Master output signals of the chain are composed as follow:

Mout_data_ram_size represents the size of the data as the number of bits for both read or write operation.

Mout_Wdata_ram has a mean only if the operation is a write otherwise it is

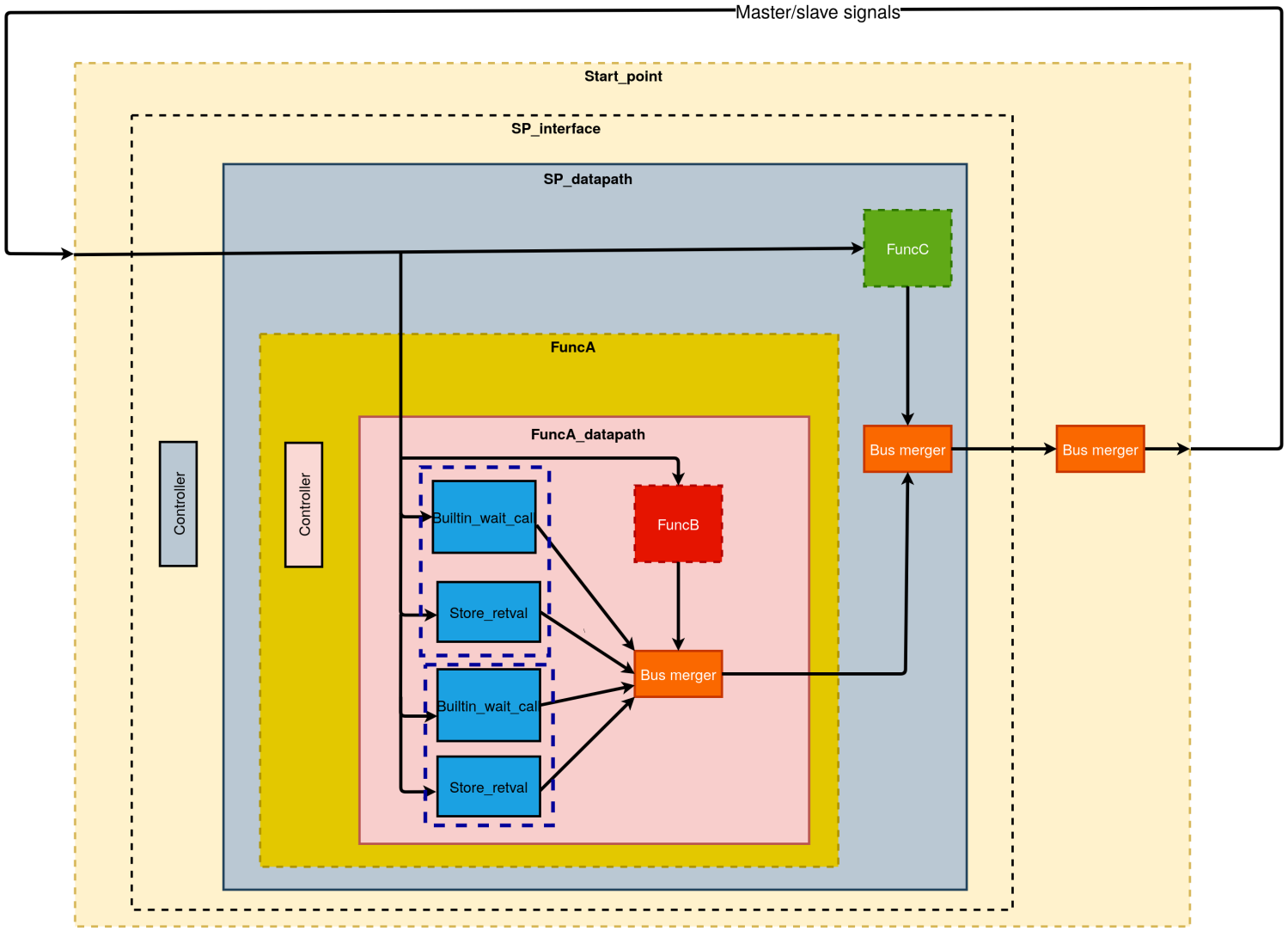


Figure 3.4: Architecture example

zero and represents the data to be written.

Mout_addr_ram is the address of the module or in general is a memory location where I have to read or write a data.

Mout_we_ram this signal is 1 if the master wants to perform a write operation.

Mout_oe_ram this signal is 1 if the master wants to perform a read operation.

Some information are returned from the slave to the master after completing the request and are contained in the following signals:

M_data_rdy after a request the master waits for the termination of the task and this signal notify that the operation is completed.

M_Rdata_ram if a master has request a read operation this signal contains the data read from the target address.

There are two kind of slave signals, the first one is needed to make the request coming from the master being reached to the slave, so the master output signals are propagated through the slaves thanks to these signals that are listed here and which have the same meaning as the master signals:

S_data_ram_size, **S_Wdata_ram**, **S_addr_ram**, **S_we_ram**, **S_oe_ram**.

The other kind of slave signals are used to respond to the master request, and are again with the same meaning:

Sout_data_rdy and **Sout_Rdata_ram**.

So a bus cycle starts when the master request a read or write operation and terminate when the slave respond to it, and during the entire clock cycle the master signals remain set.

3.4.2 Communication protocol

This protocol is used to allow the exchange of information between the builtin and the callee and in general all the modules that uses addresses. It is based on a master and slave chain, a master makes a request (read or write operation) then the signals go out of the system and come in as slave signals. A module knows that it is the target of the request thanks to an address. A module can have both master and slave signals or only one of them. Suppose we have a top level module that instantiate module A and B and they want to communicate. A master that wants to make a request will use Mout_ signals that enter in the system as S_ signals used to propagate the request to the target slave that will answer with Sout_ signals that again will come in as M_ signals. The slave knows he is the target thanks to S_addr_ram signal that contains an address. Every read or write request from the master, terminate with a DataRdy signal notifying the completion of the operation. The signals chain contain the information to know if the operation is a read or write,

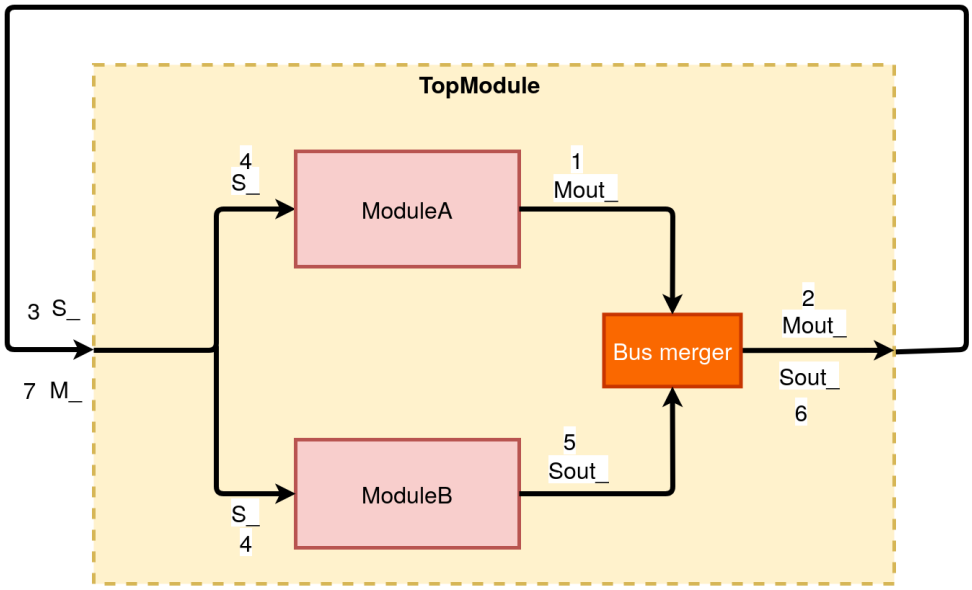


Figure 3.5: Communication protocol

the data to read or write, the size of the data and the address. Figure 3.5 shows this example evidencing the operation order.

The bus merger takes in input the master and slave signals coming from all the modules that are in the data path with the merger and output only one of those signals. It only compute the logic or between the inputs because is already known that only one input at a time will have a meaning while the others will output 0's. This is possible because thanks to the scheduling is sure that in a certain time, only one module will use the bus since the function call exploiting this mechanism are executed in sequence.

Suppose function A calls function B using the wait call method, what happens is that the controller of function A lets the builtin start and then it performs one write operation to the input register located in function B, that contains the parameter it needs to perform its computation, for each input of the function. The builtin then starts the computation of the function B doing a write operation on the status register and then waits for the notify caller module saying the computation is finished. Finally it reads from the return value register of function B the result of the computation and give back the control to function A.

Chapter 4

Methodology introduced

4.1 Application of Futures and Promises in HLS

The idea of application of futures and promises in the HLS context, starts from the function call mechanism already implemented inside the tool Bambu that allow the usage of function pointers in the C and C++ input specifications file. This is an important feature of the tool that helps the goal of the work, in fact talking about futures and promises it is necessary to introduce also the concept of threads. In C++ std::thread is the thread class representing a thread whose definition is shown in listing 4.1.

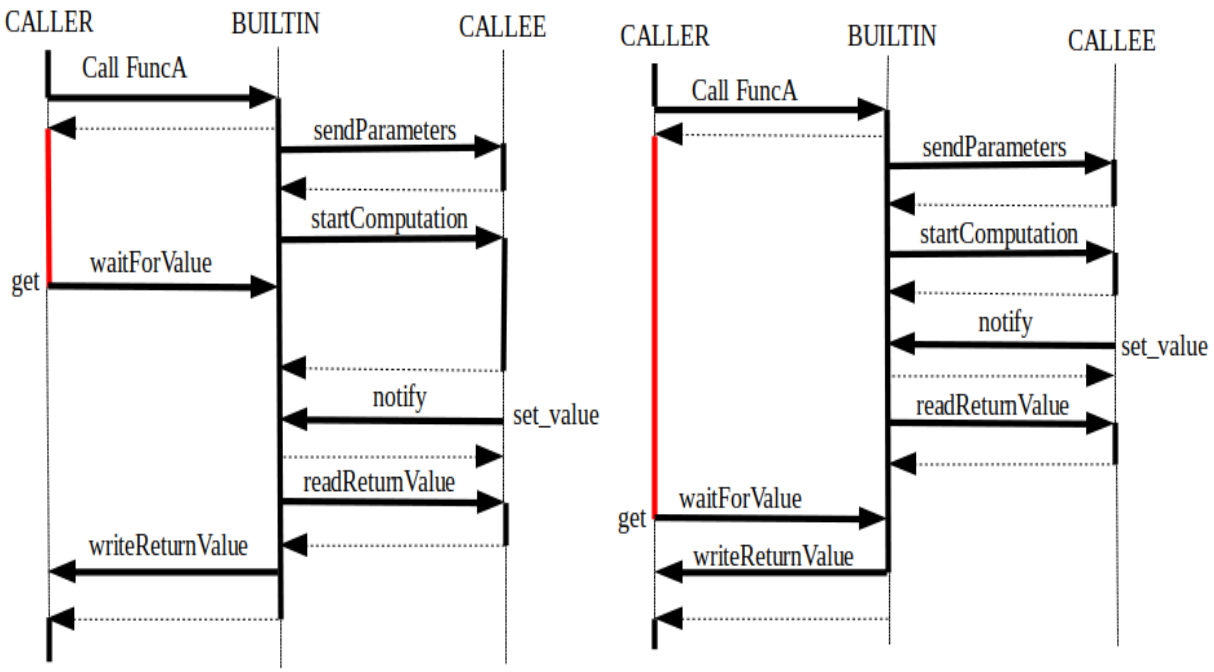
```
import <thread>
std::thread thread_object(callable, param_list)
```

Listing 4.1: Thread definition

Simply creating a thread object and passing parameters to it will release a thread that executes the code specified by the callable. This callable code can be either a function object, a lambda expression or a function pointer. The first step to do in order to make Bambu understand this new keywords such as thread, future, promise and the corresponding member functions is to include in its front-end the library implementing this functionality so that GCC or LLVM can translate them in an intermediate representation and then follow the usual flow. Once this is done, we know that using the thread object, Bambu will activate the function call mechanism for function pointers, but in the standard version the execution of the functions is shown in the notification mechanism of figure 3.3 and so the caller before continue its execution waits for the callee termination. The purpose of the threads is to allow

the caller to continue its execution immediately after releasing the thread which will execute another code in parallel, while the goal of futures and promises, in addition to that of allowing the return of a value from a callee function, is to create a synchronization point between caller and callee thanks to the get method provided by the future class. This is exactly the behavior expected for the synthesized design after introducing this features.

Since we have two functions which in a certain sense are connected to each other because one, at a certain moment, will need the result of the other in order to continue the execution, and the two are running in parallel, two different situations must be handled. The case in which the caller terminate first and this means the get method is executed first against the set_value and so the caller waits the callee because need its return value and the case in which callee terminate first, so set the return value and terminate while, when the caller needs that value and call the get, immediately receive it and so never waits. Figure 4.1 compares these cases in terms of sequence diagrams, 4.1a is the case in which the caller terminate first and the red line shows the time in which the caller is working in parallel with the callee while 4.1b is the case the callee terminate first.



(a) Caller terminate first

(b) Callee terminate first

Figure 4.1: Comparison of sequence diagrams

There could be a third situation that in fact is a critical case. The software

implementation of future and promise rely on locking mechanism to avoid the possibility of having the get and set_value functions executing at the same time resulting in an unpredictable behavior. The mechanism implemented in hardware avoid this critical case thanks to the bus, the communication protocol uses a bus to allow the modules communicate and since only one at a time can use the bus even if the modules try to get and set the value at the same time the bus manager will manage this situation.

About the bus, in the implementation described in the previous chapter, it is very simple and make the logic or between the inputs. In the proposed new methodology the problem is that maybe more than one module wants to access the bus to communicate so it's necessary to introduce a bus manager to manage the multiple requests. This feature is discussed in the next chapter.

To obtain the desired behavior is not enough to make Bambu accepting the keywords but this is just the first step, then the tool must understand the meaning of them and as a consequence generate the correct synchronization graph. So also the middle-end of the tool should be updated to obtain the correct result. The generated graphs are then given to the back-end that should be able to obtain the design architecture without making changes.

```
FuncB(promise){

do_something
promise.set_value()
}

FuncA(){

declaration of F&P
call FuncB(promise)

do_something
future.get()
do_something_else
}
```

Listing 4.2: Pseudo CPP code

```
call FuncB(promise){

sendParameters()
startComputation()
}

future.get(){

waitForValue()
}

promise.set_value(){

notify()
}
```

Listing 4.3: Pseudo code

Listing 4.2 shows the pseudo code of a program using futures and promises and listing 4.3 is the translation of that code in terms of actions in the desired architecture. Here the action of waiting for the returned value of the function called is disconnected from the call itself but is activated when the main function needs it so when the get is called, in the meanwhile before calling the get the caller can perform other stuff instead of being blocked like in the standard version of the wait

call mechanism. In the proposed architecture the set_value notify a module that the value has been set so when the get is called it is known that the value is already available and the main has not to wait.

The purpose of this thesis is not to make the right changes to the tool to achieve this result but to show that it is possible to apply futures and promises in the context of high level synthesis and how it can be performed. Next section shows the methodology adopted to get the result without introduce modification in the tool.

4.2 Bypassing tool modification

The tools that perform high level synthesis are projects of not indifferent dimensions and which take care of performing quite complicated and intertwined tasks that makes its understanding not easy neither fast to learn, especially if there is no documentation for programmers. In this case, therefore, before attempting to modify the tool itself a good alternative way is to test if the idea works, thus avoiding realizing that you have lost a lot of time for something that is actually not feasible.

Since Bambu uses, to pars the input file, GCC or CLANG if we include the standard library that implements threads, futures and promises it is not a problem for them to translate the high level language into an intermediate representation implementing the correct functionality. The problem raises because the tool plugin that reads this intermediate representation doesn't understand the whole set of instructions generated by the compiler.

In the plugin what happens is that at every operation, if I have a simple one like *and* or *add* operation there is a match one to one, if the operation is not simple the plugin builds the expression that perform the same operation or if possible decompose the operation in simple ones. At the end the intermediate representation is very similar to the gimple GCC IR. Once the instructions are all simple operation or expression understandable by the tool, it is able to translate it in Verilog code in the next phases.

The plugin starts from the clang or GCC intermediate representation. When I compile, clang or GCC they assume that the library is there, then it is the linker that puts the program together with the library, if the library is not found it is the linker that does not find it, Bambu has its own linker. Bambu when it starts puts the object code dumped by the plugin and combines it with the libraries that are in the same gimple format as the object code. So from the compiler's point of view it doesn't matter that the code is written in C or C++.

All the libraries inside Bambu are written in C, and it does not use the standard C library but uses libraries adapted for this purpose, that is not to support all relevant C standards around a wide range of hardware and kernel platforms but are focused on embedded systems and are much smaller than the standard one.

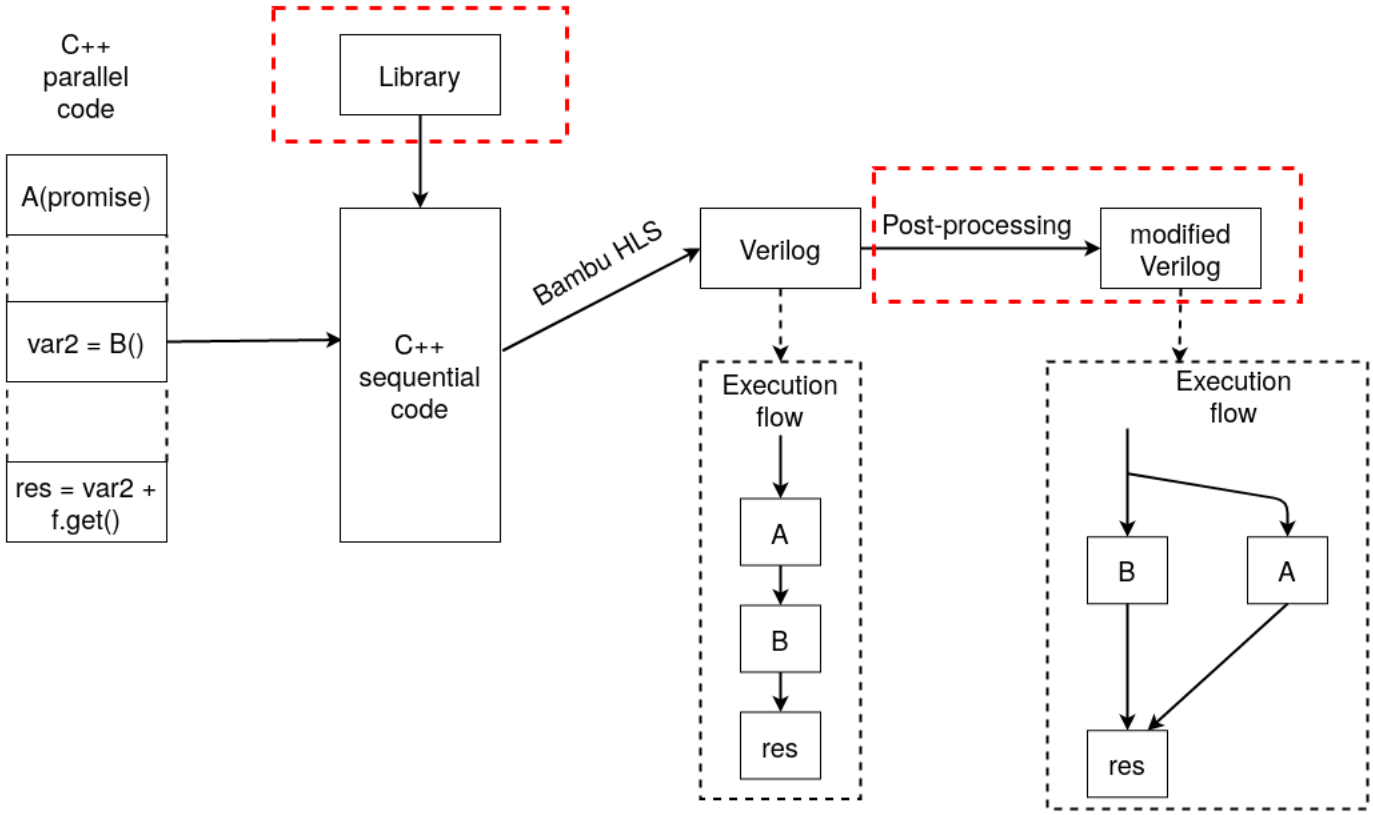


Figure 4.2: Bypassing Bambu modification

For what has been said so far, if you try to introduce the standard C++ library for threads, futures and promises it would not be a problem from the point of view of the compiler which translates it into intermediate representation, but it is a problem for the plugin that cannot understand some of the instructions generated by the compiler.

For the reasons just explained, a minimal library has been created that implements the functionality of threads, futures and promises but sequentially, without introducing parallelism, so that there are no problems for plugins to understand the instructions generated by GCC or clang and another strategy is used to make the final architecture follow a parallel flow of execution but we will talk about it in a moment. The library is written in C++ and the code is provided in appendix A and the analysis of the latter is done in the next chapter.

The library was not introduced directly into the tool, but to obtain the same result an example from those available in Bambu was exploited and adapted, this

example shows how it is possible to build an Autotools based project for the high-level synthesis with Bambu. An exhaustive documentation about Autotools can be found here [15], in our case is used to automatically create the .o files from a library and a source file, this files .o are not object files in GIMPLE format but are in the GIMPLEpssa format that is the one used by Bambu to start with. This project then take this .o files and act as a linker to create a unique intermediate representation and then the tool is called, with an option that allow to start the synthesis process not from an high level specification but from a GIMPLEpssa file format. In practice this allow to add libraries different from the ones already in the tool exploiting some scripts and the Autotools suite. The provided example was written to support libraries written in C so it was necessary to modify it according to the Autotools specification.

As said previously the library introduced is sequential, and so inside the tool it should be necessary to identify the blocks running in parallel and schedule it in that way. Again this can be a future work to implement these functionality in the tool, while in this work another approach is used. A post-processing of the verilog code generated by Bambu modify the code introducing the modules needed to support the features and make the necessary changes to the module already existing. The whole process is shown in figure 4.2 and the introduced methodology are marked in red.

In the next chapter an analysis of the library is done and is presented the post-processing that introduces the modules and the main changes needed to make everything work.

Chapter 5

Modification

5.1 The adapted library

HLS tools often start the synthesis from an intermediate representation generated by compilers as Bambu does. Compilers do not always automatically understand the sections of parallelizable code and for this reason it is up to the programmer to specify parallelism directly in the high level code. Standard approaches to do this is to use open libraries that support multi-threading or the standard libraries introduced in some of the programming languages. Of course it is possible to specify parallelism also in HLS tools, and depending on the tool there can be different methodologies to do that. For example specify loop unrolling through vendor-specific directives inside the code or doing a much more manual job you could use HLS to get a module and then duplicate it and get the correct links by hand.

Therefore it would be a great advantage to have a tool that automatically manages to synthesize a high level code containing parallelism, in this way even the programmer would be free to write good code having fewer limitations. Usually using threads is difficult to do without synchronization methods, in our case futures and promises are exploited to obtain parallelism with a synchronization point.

As already said using the standard C++ library for threads, futures and promises will generate an intermediate representation that is not understandable by the actual Bambu implementation and moreover this representation is very big because to manage parallelism in software there is need a lot of low level instructions and so for example only a call to get method can generate hundreds lines of code.

Using a custom library that makes the C++ code to be sequential avoid to modify the front-end of Bambu otherwise it would be necessary to take all the instruction generated by the compiler that the tool doesn't already understand and gives a meaning to it in order to obtain a corresponding hardware behavior. Another advantage in using the custom library is that the intermediate representation

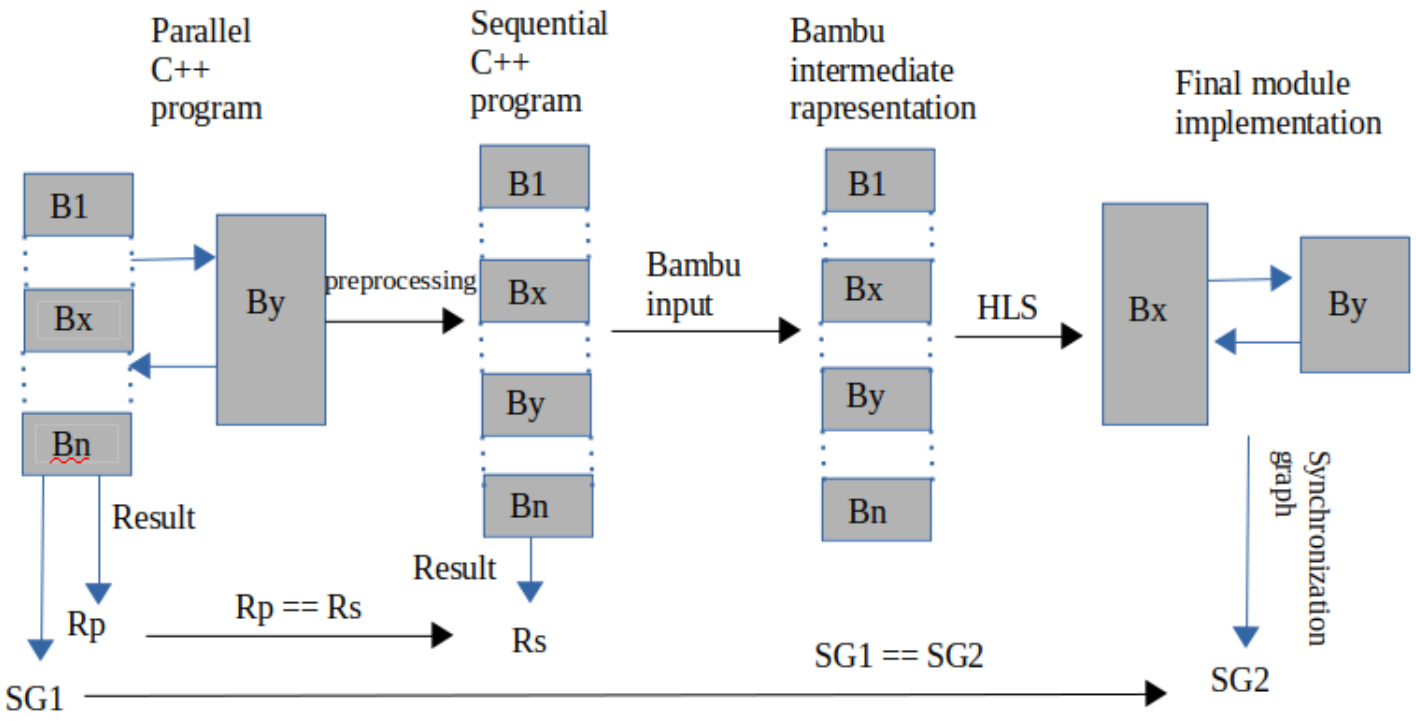


Figure 5.1: bypassing front-end modification

generated by the compiler this time is very small, because no parallelism is introduced. Obviously from the functional point of view, the library must give the same software result as using the standard one. Since no parallelism is introduced at this step then Bambu should recognize and schedule the block to be executed in parallel and as a consequence generate the correct synchronization graph to be given to the back-end obtaining the correct architecture. So at the end of this process, the functional result generated by the standard library is equal to the one generated by the custom library and the synchronization graphs are the same if we extract it from the software parallel program and from the generated architecture. Figure 5.1 shows the entire process.

As already mentioned the goal of this thesis is not to modify the tool itself and so instead of modify the creation of synchronization graphs, we just let the tool generate the Verilog code introducing the library so that the code is sequential and then post-process the Verilog introducing the modules needed to achieve parallelism.

The code of the library is provided in appendix A.1, A.2 and A.3 and is organized as follow. Listing A.1 is the header file that contains both the declaration of future and promise classes with the relative member functions and the thread that is not a class but simply a template function. This because here the only purpose of the

thread is to call the function received through parameter, that is a function pointer, in this way the wait call method discussed in chapter 3 is activated. We don't need the join method of the thread because our synchronization point is represented by the get method and moreover since the library makes the program to be sequential, when the main call the join we already know that the called thread is terminated. The thread function is implemented as a variadic function so that can receive a variable numbers of arguments.

The promise class template contains the public member functions set_value and get_future so that can be called by outside the class and the private attribute retVal that is the value contained by the promise object, the one that is obtained in case a get is called on a future object. The future class template only has the public get method and a private pointer that is a pointer of kind promise used to keep track of which promise this future object is linked to.

Everything is put in the std namespace to make the library as similar as possible to the standard one.

There is another reason why the threads are not a class but just a method, and for the same reason there are no constructors in the future and promise classes. The library in addition to providing the same functional result as the standard one, must also be synthesizable by Bambu, and this is a limitation in the code construct we can use in it. Bambu does not accepts constructors in the source code specification, so instead of launching a thread as in the standard way, it's enough to use it as a function call. Listings 5.1 and 5.2 show the differences. One way to overcome this problem would be to create a thread class without constructor and add a method that starts the thread, while in the standard library the thread object starts as soon as is created.

```
std::thread threadObj(func,
 param);
```

Listing 5.1: Standard thread

```
std::thread(func, param);
```

Listing 5.2: Custom thread

In the .cpp files there are the implementation of the functions specified in the header file. In listing A.3 there is the implementation of the only thread function that just call the received function passing the arguments to it. Listing A.2 shows the implementation of future and promise member functions. Starting from the promise class the get_future method perform the linking between the promise object that call the method and a future object which reference is then returned by the function. So from now a future object has the attribute fPromise pointing to a promise. In the standard library the get_future method doesn't return a reference to a future but it returns a future object by value. Again due to a code limitation is not possible to return objects by value but only by reference. In terms of calling

it from a function the difference is to use the asterisk to retrieve the object pointed by the pointer, this is shown in listing 5.3.

```
//Standard get_future()
std::promise<int> prom;
std::future<int> fut = prom.get_future();

//Custom get_future()
promise<int> myp;
future<int> myf = *(myp.get_future());
```

Listing 5.3: get_future()

The set_value method receive an input that is the value to set in the retVal attribute of the promise class. This value set by the promise is then retrieved calling the get from the future object that was previously linked to it.

Before introducing the modification on the Verilog code it is necessary to understand which is the architecture generated by Bambu when using this library.

5.1.1 Generated design

Appendix A.4 shows the main function that was synthesized using Bambu for the purpose of showing how the library is translated in hardware components.

Futures and Promises objects are translated in array-1D, that stores the return value in the case of a Promise object and an address in case of a Future. This address is the address of the Promise the Future is linked to, thanks to get_future(). Array-1D modules are used to store something and they have load and store operation. Bmemory-ctrl are modules used to control the array-1D, so thanks to master and slave signals they can load something from the array setting the oe-ram signal equal 1 (read operation) or can store something inside the array setting we-ram signal equal 1 (write operation). Array-1D has an address associated to it to allow the bmemory-ctrl targeting one of them.

All the member functions of the classes corresponds to a module in the generated architecture. The get_future module contains an array-1D, in which is stored the address of another array corresponding to a Promise in the top level module called start_point in this example. This array is copied thanks to the internal_bambu_memcpy from the get_future module to another array that represents a Future. So at the end when the get_future is executed in the data path of the top level module there are two array-1D, a Future pointing to a Promise.

The set_value module is located inside the funcA data path, where there are also other modules to perform the wait call mechanism as discussed in chapter 3 such as

the input register and the status register. Set_value contains a bmemory-ctrl module that simply perform a write operation to a Promise array object, this means a store to the array that corresponds to setting the promise value.

Remembering that the get is called from a future object it performs two read operations because the value to being retrieved is not in the future but in the promise, so the first read is to obtain the address of the promise that is located in the array corresponding to the future and the second read, reads the value contained in the promise.

All the modules know the address of the others or because they read it from arrays or because are given as constant value.

The thread is only used to call the function thorough function pointers and so activate the wait call using the builtin mechanism.

The complete architecture schema is showed in figure 5.2. The bus merger here performs the logic or between the input because thanks to the scheduling is assured that only one module at a time uses the master and slave chain.

There is an option in Bambu that if active generate some dot files that are a graph description language files containing the representation of all the graphs used by the tool to perform the synthesis. Figure 5.3 shows the scheduling of the start_point function and is important to notice that the tool schedules the thread and the funcB to be executed one after the other while our goal is to execute the thread in parallel with other functions.

The previous example was the simplest one, in which there was only one couple of future and promise and one thread. Synthesizing a more complicated one where there are two couples of future and promises with two threads we obtain the following result. The get_future module now contains a bmemory-ctrl that control a temporary array in the top level module data path and the operations performed calling the module are a write in this temporary array writing the address of a promise array and then the internal_Bambu_memcpy copy this value in the corresponding future array, calling for the second time the get_future the same operations are performed unless that the address written in the future is now the address of the second promise. So a general rule for the number of array-1D in the data-path of the top level module is 2*F&P_couples + 1 where one is the temporary array.

For what concern the set_value, since now is used by two functions Bambu instead of duplicate it, instantiate in the data-path of the top level module one copy of the set_value and the functions that use it, have a proxy that calls the instantiated module providing all the information needed to complete its work. In fact the proxy send to the set_value module the address of the promise, the value he wants to set and a start signal. The proxy_merger performs the same action as the bus merger. A schema of this architecture is showed in figure 5.4.

Finally the get and the thread modules do not change respect to the first example.

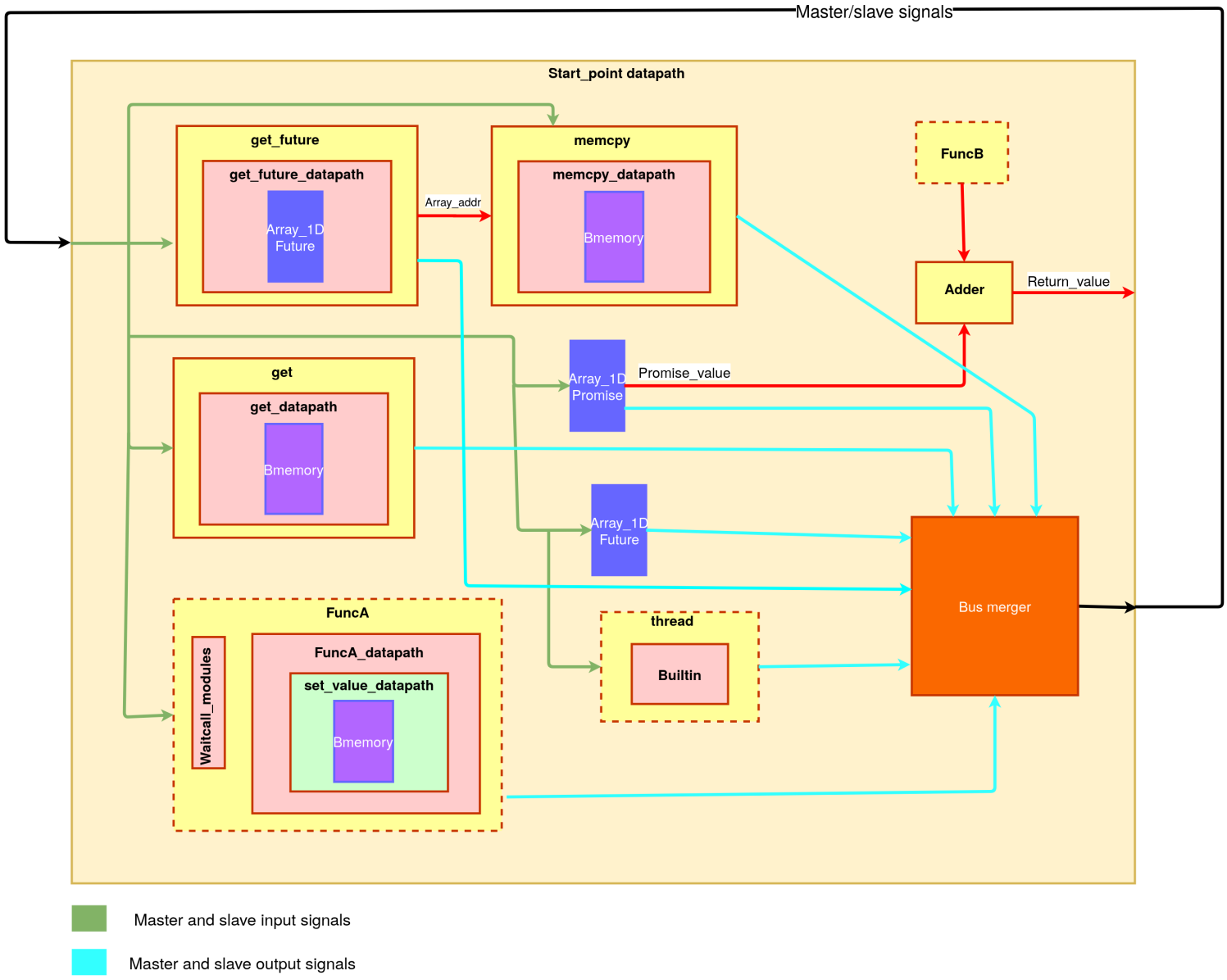


Figure 5.2: Architecture example

This conclude the description about the architecture generated when the provided library is used. The next sections describe the modules introduced in the design explained here to obtain the desired behavior.

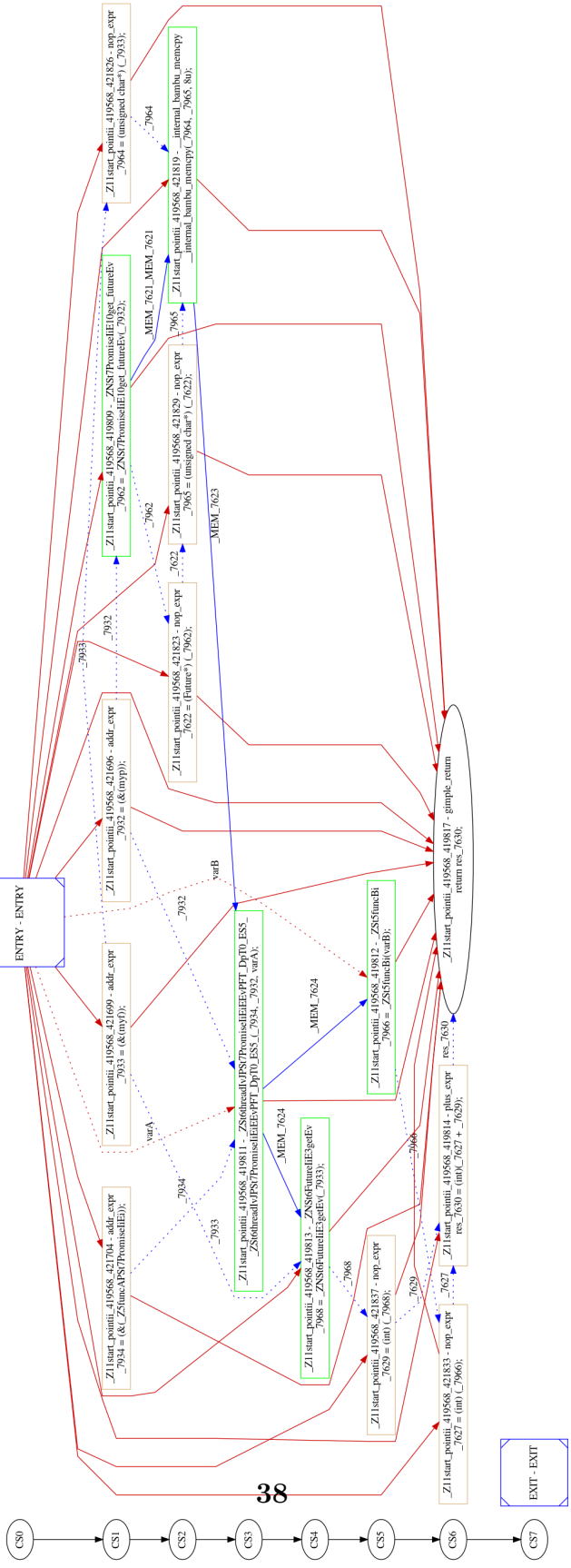


Figure 5.3: Start_point schedule

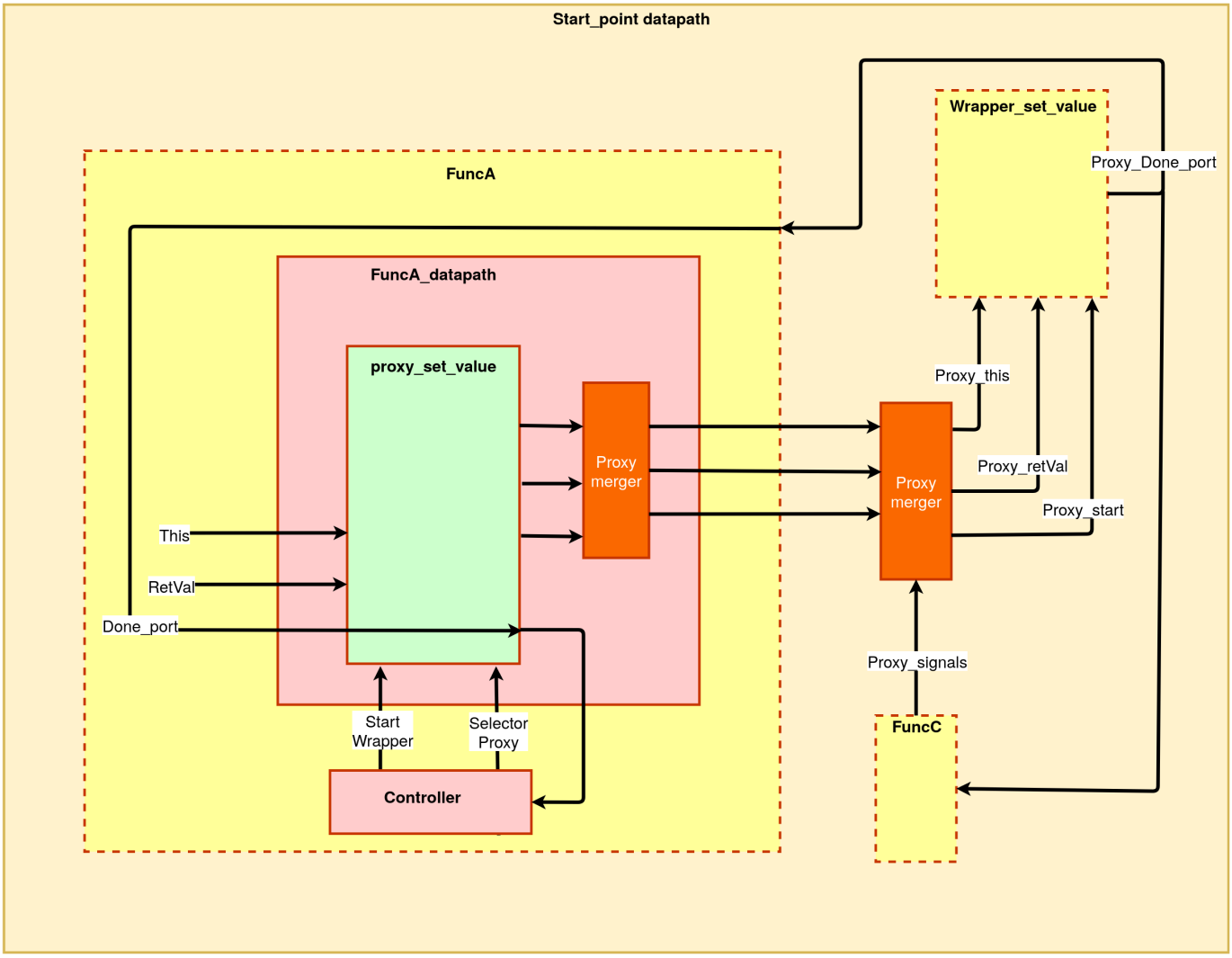


Figure 5.4: Architecture with proxy

5.2 New modules

Starting from the already implemented wait call method of Bambu, different strategies could be applied to parallelize the execution of some functions according to the high level specification. Obviously the less the modules generated by Bambu are modified the better is so that in future it should be easier to implement this methodology inside the tool. An important observation is that the goal is not to optimize as much as possible the parallelization of the tasks, but to parallelize the functions that the programmer writing the high level language file wants to execute in parallel through the thread, future and promise features of C++. So it will be up to the programmer to exploit such features to optimize the code.

Having said that it is clear that the scheduling and the actions performed before and after the calls to thread methods are not modified, so for example the get_future module is not touched respect to the one generated by the tool. The gets are scheduled by Bambu in sequences so only one get module is enough to manages all the get calls executed, moreover since this is the synchronization point it is necessary to not execute them in a different order it's specified in the description. Imagine more than one thread executing different functions are used in the source code, depending on how many clock cycles are needed to execute the functions there will be cases in which two or more set_value are executed at the same time so differently from the get more than one set_value module should be needed. As already said Bambu uses some proxies and only one set_value module as showed in figure 5.4 so the bus manager proxy that is introduced in the next chapter manages multiple calls at the same time.

In Bambu controllers are organized as finite state machines, so they have states and for each state perform some actions that then are executed by the data-path. To make the threads run in parallel it is enough to modify the controller of the top level module, that has not to wait for the termination of one to start the other, and so instead of waiting on a variable in the same state, it will go directly to the next state.

Now we want that if the value inside the promise object is already set, the get has to receive this value otherwise blocks the execution of the main. A promise object is translated in an array_1D and to obtain the desired behavior should be modified for example introducing a flag that is one if the value is set or zero if not so that the get can know if wait or read the value and let the main continue the execution. But in order to not modify the modules that Bambu generates the following strategy is used. The promise objects is now a module (promise_FU) containing the array-1D used as usual to store a value and a new module called notify_caller_p, both the modules uses the master and slave signals so inside the promise_FU also a bus_merger is instantiated. The derived architecture is showed in figure 5.5.

The notify_caller_p is build as a finite state machine and the code is provided in

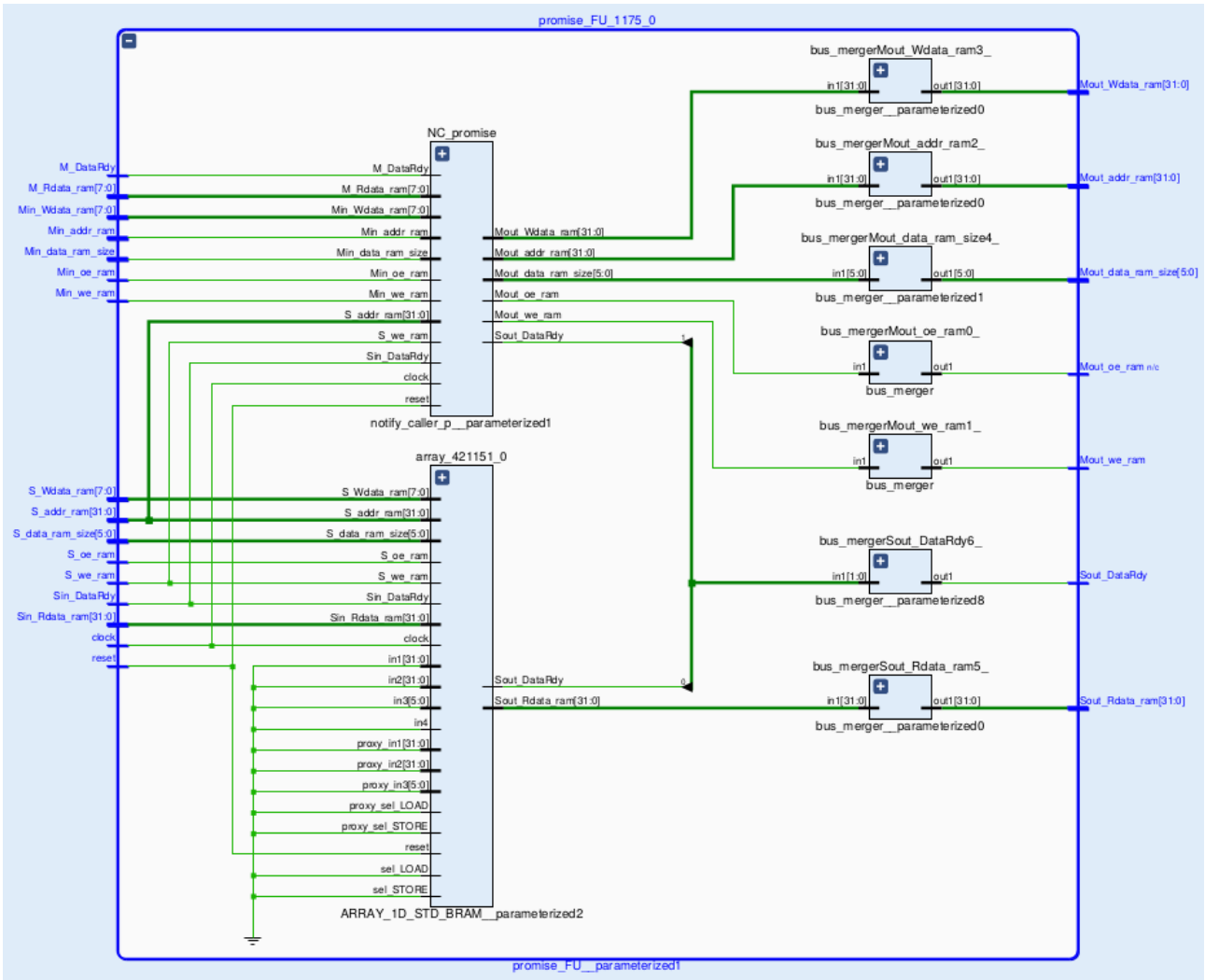


Figure 5.5: Promise_FU module

appendix B.1. This module's goal is to keep track of the state of the value in the array-1D and to let the get module know if the value has already been set or not yet. Here two situations can happens, one between set_value and get is executed before the other, so the initial state of this module check this possibility, and if intercepts first the set_value, the notify_caller_p goes in a state while if the get is executed first it goes in another state. Doing this the module knows which operation has been executed first and so in the case of the set_value the second operation is the get and this means that the get is not waiting because the value is set, while in case the get is executed first the second will be a set_value and this means to wake up the get

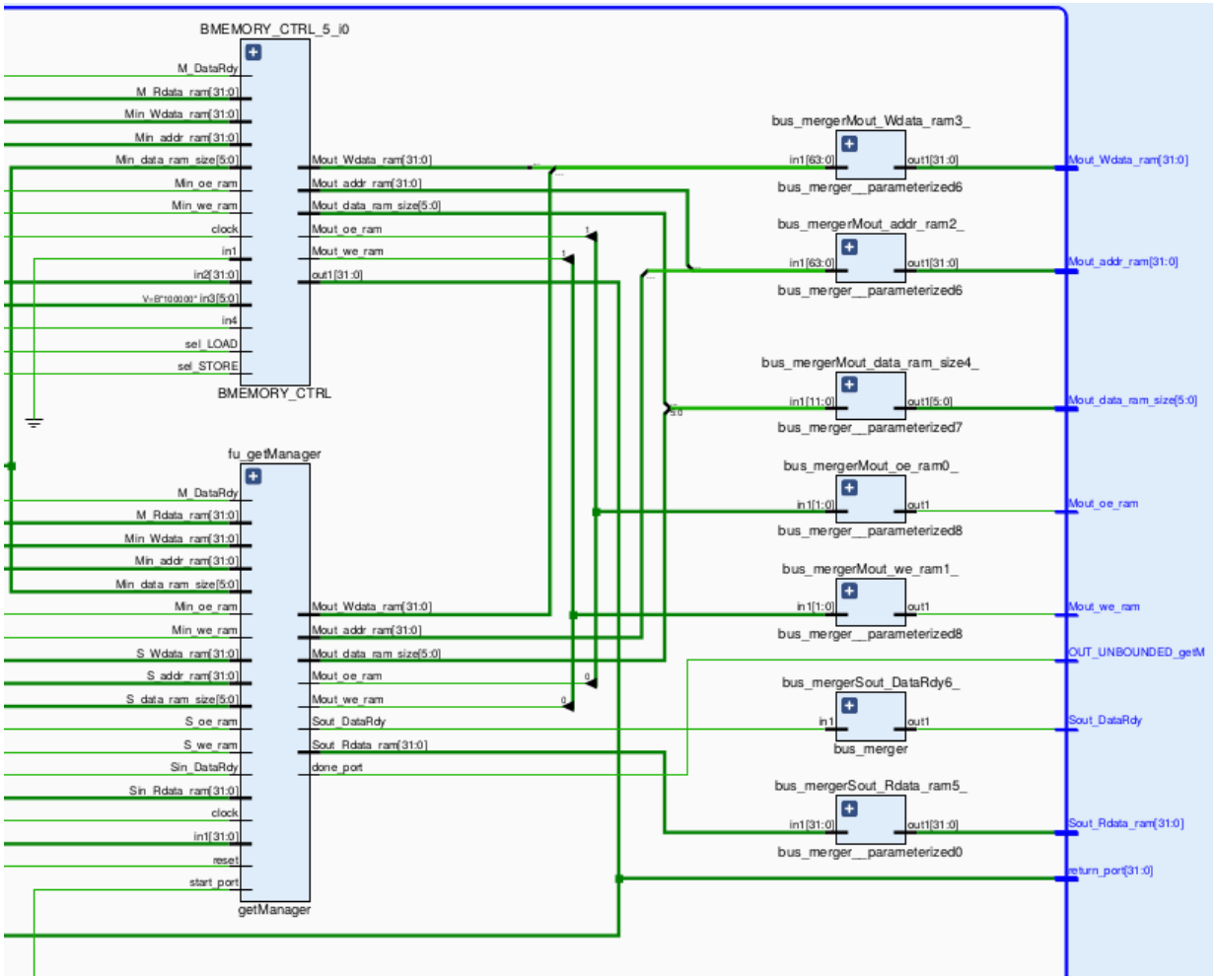


Figure 5.6: getManager module

module and provide the correct value to it.

The second module that was introduced in the architecture is called getManager and is instantiated inside the data-path of the get in which there is the bmemory-ctrl, the bus_merger and other modules needed, the design is showed in figure 5.6.

Again this module is implemented as a finite state machine and the code is in appendix B.2. Is controlled by the controller of the get module and is started after the first read operation of the get. As already discussed the get performs two read operations, the first one can be executed without controls because it just retrieve the address of the promise from a future array while the second can be performed only if the value in the promise has already been set. So before executing the second read this module starts and performs a write operation, through the master and

slave signals, to the notify_caller_p meaning that the get is executed, then waits for a message from the notify_caller_p that can arrive in the same instant if the set has already been executed or in the future if not.

A pseudo code of the getManager and notify_caller_p is showed respectively in listing 5.4 and listing 5.5.

```
write(notify_caller_p_address);

step2:
  if(notify){
//this signal will start the get
    done;
  }
  else
    goto step2;
```

Listing 5.4: getManager pseudocode

```
step1:
  if(set_value)
    goto step2;
  else if (getManager_write)
    goto step3;
  else
    goto step1;

step2:
  if(getManager_write)
    goto step4;
  else
    goto step2;

step3:
  if(set_value)
    goto step4;
  else
    goto step3;

step4:
  notify_getManager;
```

Listing 5.5: notify_caller_p pseudocode

5.3 The bus architecture

The modifications introduced until now are enough to manage parallelism in Bambu unless for the bus system. A bus is already implemented in the tool and everything works well if the calls are executed in sequence, because it is sure that only one module at a time will use the master and slave signals so implementing a bus that perform the logic *OR* between the inputs is enough.

Figure 5.7 shows the bus architecture implemented in Bambu. There is one bus_merger for each signal and the input that receives is the union of all the signals coming from the different modules, so for this example the Sout_Rdata_ram is 384 bits because there are 12 modules using the bus each with a 32 bits signal. This architecture cannot work in a parallel scenario because how could they be synchronized to output the input number one rather than input number two? Imagine this situation, a module wants to perform a read operation, so the Mout_oe_ram signal is

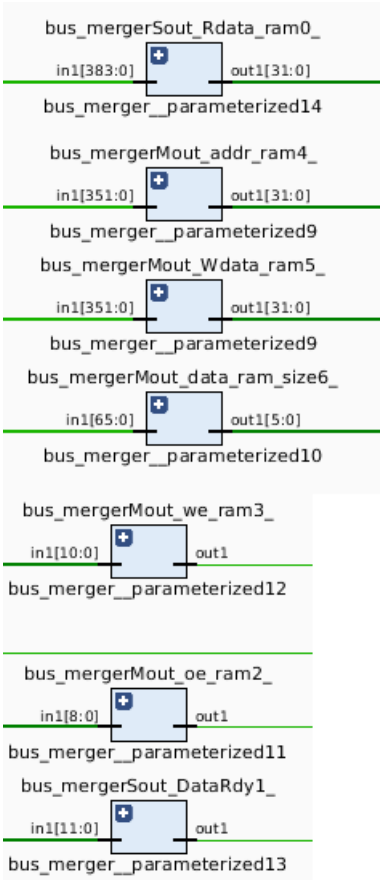


Figure 5.7: Bambu bus architecture

set to one and the other needed signals have some values that we don't care, another module at the same time is asking for a write operation so this time its Mout_we_ram is set. At this point what happens is that the bus managing the read signal request will output one because looking at the input there is only one module that whats to read, and the same thing happens to the write bus and so at the end the output master and slave signals are meaningless because is like a module is asking for a read and write operation at the same time.

To avoid this kind of situations the following strategy was implemented. Instead of having one bus merger for each signals, there is only one, and for each module that uses the master and slave signals there is a new one called compacting_FU that takes in input the signals used by a module and output the union of these following a precise order showed in figure 5.8 so that when the bus will output a signal, another module called unzip_FU takes this and again unzips the signal that was compacted by the compacting_FU module. Doing this it is not necessary to modify any of the modules unless for the bus_merger that obviously has to be modified to manage multiple requests but the details are provided in the next section.

A schema of the new architecture is showed in figure 5.9. Notice that now the bus_merger takes in input the union of all the signals coming from the compacting_FU modules that are associated to each of the module that uses the master and slave signals. The compacting_FU module creates a fixed length output and in case a module has for example only slave signals, the master are set to 0 and the output of the compacting has the same length as a module that uses both masters and slaves. So the problem explained before doesn't exists anymore because think about the same example, if the bus decide to output first the read operation it will output

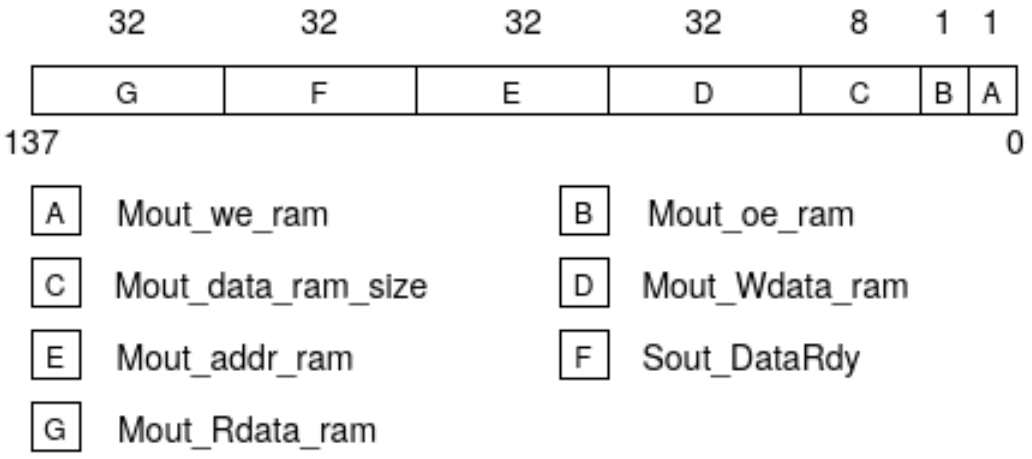


Figure 5.8: Compacting_FU order

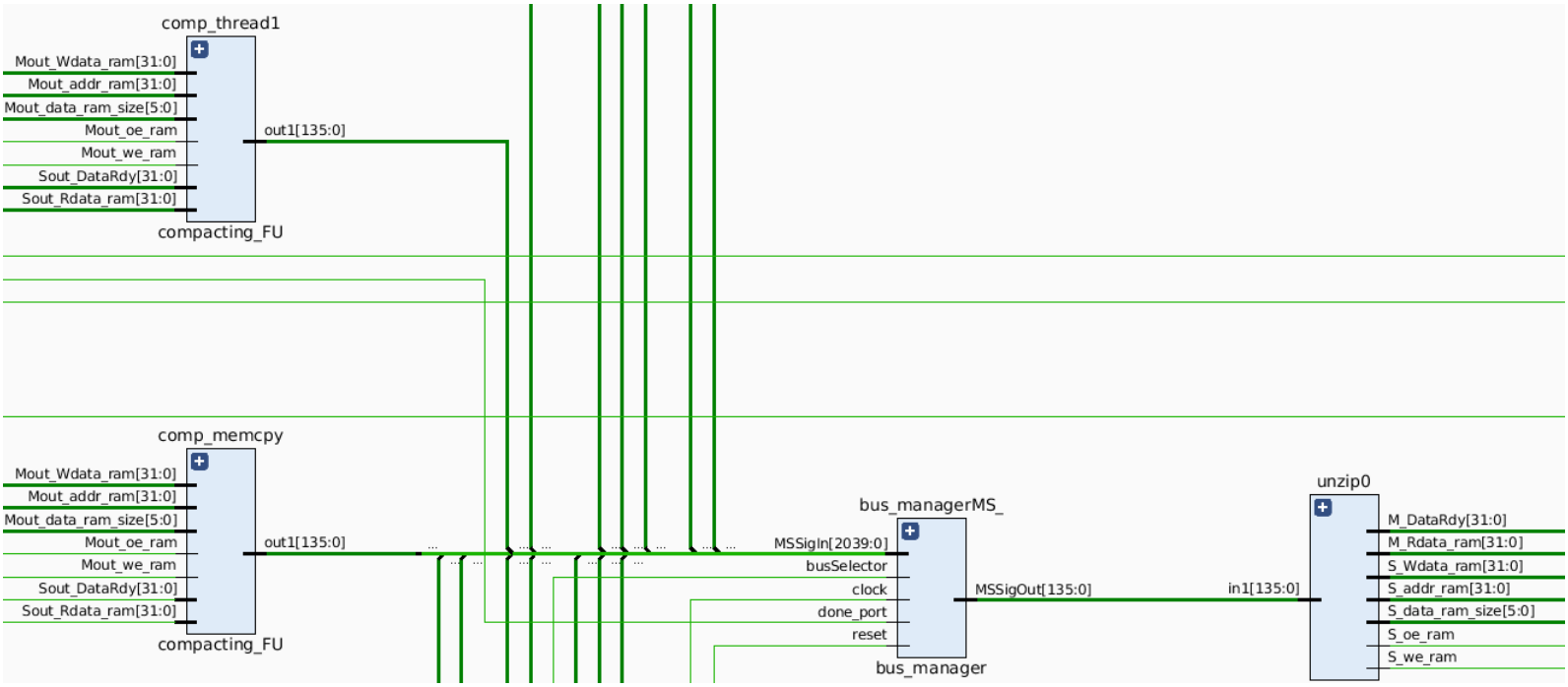


Figure 5.9: New bus architecture

a signal with Mout_oe_ram equal to one and Mout_we_ram equal to zero thanks to the work of the compacting_FU.

The code of these two modules is provided in appendix B.3 and B.4. This was the preliminary phase to now in the next section provide an overview of the bus manager that is the last step to obtain a working architecture supporting parallelism thorough thread, future and promise.

5.3.1 Bus manager

Since we want to parallelize the functions called by the thread method and then synchronize them through futures and promises the actions executed by the architecture generated by Bambu before the call to the thread functions are not modified. So even if a bus manager is introduced in the architecture, the old bus merger is kept and thanks to a signal is possible to switch from the old to the new. The new one is activated just before the call to a thread, so before the parallel execution starts. The wrapper module is called bus_manager and it contains both implementations and the signal to select which bus to use is called busSelector. The bus merger is implemented using just a function in Verilog, while to implement the bus manager a controller and a FIFO buffer are used and instantiated inside the bus_manager

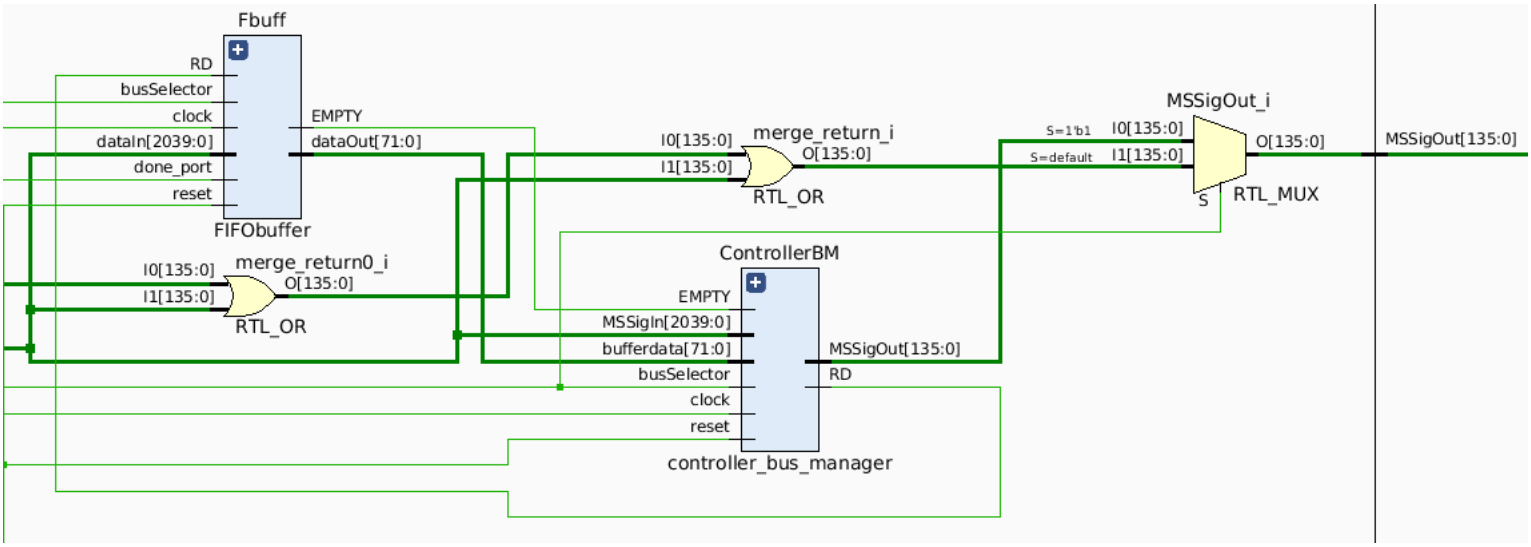


Figure 5.10: Bus manager architecture

module. The complete architecture excluding the bus merger is showed in figure 5.10.

In the already existing bus system a bus cycle consist in a master doing a request, which keep the signals value up until it receives a response of operation complete from the slave. So also with the bus manager the goal is to keep in output the master signals until the slave responds.

The controller_bus_manager checks if the buffer is empty or not, when it is not empty it reads from the FIFO a value that corresponds to the first master request registered in the buffer. Then it is looking for the slave response checking at all the slave signals entering in the bus module if is not zero. Found the slave response it is looking again in the FIFO buffer for the next master request.

The FIFO buffer is able to fill itself with new values, in fact it receive the input coming from all the modules that uses master and slave signal and filter the master signals that are different from zero and put it in the buffer. In the meanwhile it can react to a read request done by the controller. The FIFO dimension is twice the number of the modules that use the bus and this guarantees with a margin that the buffer will never been full, since each module doing a request through the bus stays blocked until the slave response, even in the worst cases is not possible to exceed the FIFO dimension.

A clarification must be made regarding the DataRdy signal, the one which is in charge of notify the master that the operation of reading or writing was successful. In the architecture implemented by Bambu the dimension of this signal is one bit, because by construction is assured that only one master at a time is doing a request

and so the DataRdy signal for sure is for that master. In our case is not like that but maybe more than one master is waiting for a slave response, if the signal is again one bit long both the master receive it and the two may understand that the request they done is successful and so both continue the execution, while a response from a slave is associated to one master request and not more.

So it was necessary to modify the meaning of this signal enlarging it to 32 bits, now it is a signal specifying the address of the slave that is notifying the master that the operation is successful. To clarify it, if a master with address X made a request, the slave with address Y when it has completed, writes Y in the DataRdy signal to target that particular master that made the request, if other masters are waiting for a slave response will ignore that. For construction there will never be two masters waiting for the same slave response.

The Bambu architecture is very modular, and regarding the bus merger it is instantiated in each data-path involved in the function call. Since we add parallelism only in the top level module the bus manager is instantiated in the data-path of that module while for the others the bus merger is kept.

The complete code of the bus_manager can be found in appendix B.5.

5.3.2 Proxy manager

As already discussed in section 5.2 this component is needed to make the system properly works because there can be situations in which more than one set_value is performed at the same time so like the bus manager also here is needed a sort of manager that let pass one request at a time.

Since the mechanism of the proxy is different from the communication protocol adopted by the bus a different, but with some common point, strategy is used to create the arbiter. The compacting and unzip modules are adapted to this situation, so an architecture similar to the previous one is obtained and is showed in figure 5.11.

Each function that uses the set_value module is linked to a modified version of the compacting_FU and then all these modules are linked to the proxy manager that again choose which function can use first the set_value module. So the output of the manager goes to an unzip_FU module which output is the input for the set_value. When the operation of setting is complete the done goes, in addition to the calling functions, through the manager so that it is aware that another request can be satisfied.

Here the arbiter is simpler than the bus manager, it has only one state in which looks for requests and when one is found is checking the done signal coming from the set_value and when it is set starts again founding a request.

This functionality must be implemented cause the architecture generated by the tool but since the set_value module is not so complicated and not so area consuming

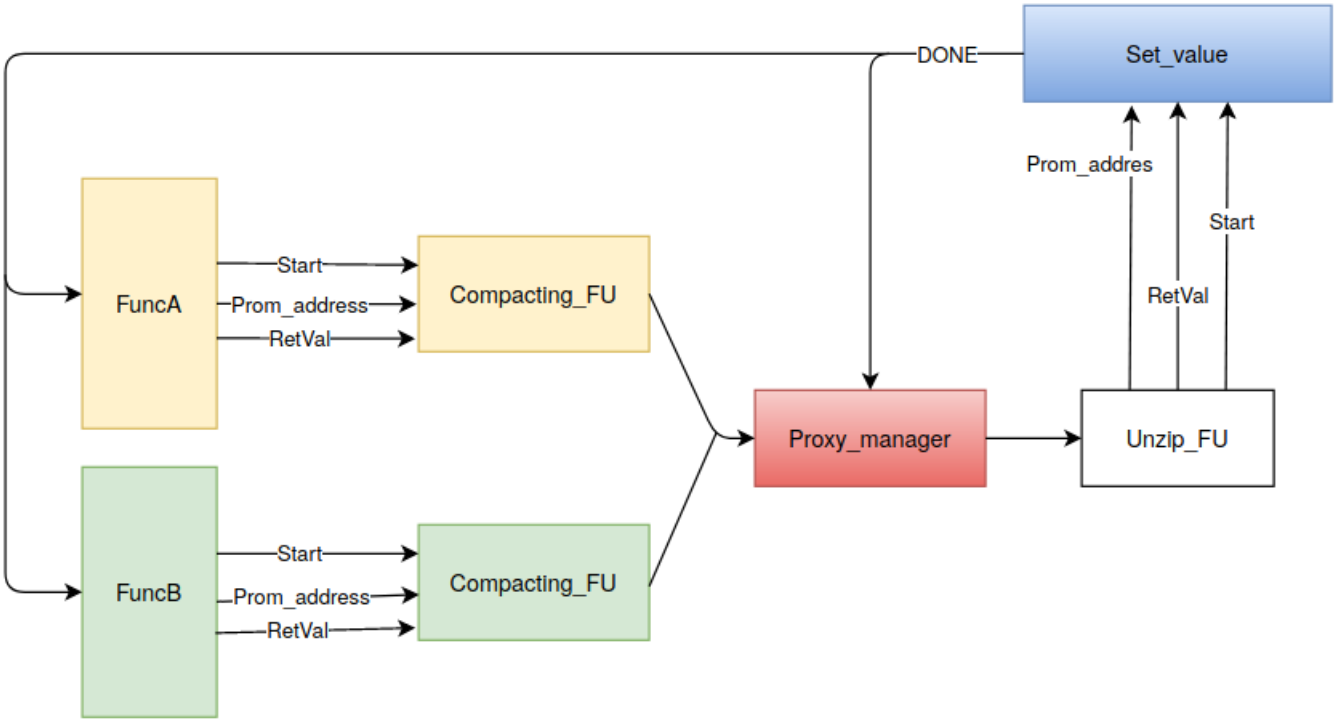


Figure 5.11: Architecture with proxy manager

probably it should be better to avoid using proxies but directly implement the set_value module inside each function using it.

Again the code of this module is showed in appendix B.6.

This section concludes the changes made to the verilog code generated by the tool, and in the next chapter the results obtained using this system are discussed.

Chapter 6

Discussion

6.1 Case studies

One of the case studies chosen is not a well-known example used in many case study works concerning high level synthesis, but is a custom example specifically created to show in a simple way how the introduced methodology works. This choice is made for two reasons, first of all to demonstrate that futures and promises can be applied in an high level synthesis context. If the methodology is applied to parallelize a *FIR* filter more than an histogram equalization process is not important; the most relevant aspect is the execution flow of our system. Second the methodology is not implemented automatically by the tool. Starting from the Verilog code generated, it is implemented by hand to demonstrate that it works, so using a complex system could be very difficult and time consuming to apply all the needed modification to the code.

The C++ code of this first case study is showed in appendix A.5. The idea is to have some functions that are not dependent on each other so that the execution flow can be parallelized applying the methodology. Bambu rely on the GCC or LLVM compiler concerning code optimization. If the compiler is able to generate an intermediate representation where section of code that can be executed in parallel is already identified, there would be no need of these constructs to parallelize the execution. Given to Bambu the provided example, it is not able to understand that the functions can be executed in parallel and so it make sense to apply future and promise in this case. The functions chosen implement a *for* loop and this is an important point because in this way it's possible to control the execution time of any function and changing this time is possible to made some considerations that will be done in the next section. Moreover the loop is made in a way that the compiler cannot optimize, otherwise there are cases in which a loop of hundreds of iterations are executed in one clock cycle. Listing 6.1 shows an example of loop that can be

optimized by the compiler and for any value of y the execution time will always be one clock cycle.

```
for(y = 0; y < iterations; y++)
  result += varA;

//the above loop becomes result = iterations*varA
```

Listing 6.1: Loop optimization example

The case study taken in consideration is a quite complete example which explores different application cases of futures and promises, in fact we test the case in which a function receive more than one promise and execute more than one *set_value*. The main function called *start_point* has three future and promise objects and so execute three *get*. Having in mind this example, the others can be obtained by replicating the current architecture, since Bambu is very modular so the system can change in number of elements it has, but not in the way they are created and linked. A comparison schema between the execution flow of this example implemented by Bambu with and without the methodology is provided in figure 6.1 and the next section shows some results obtained in terms of execution time and area.

The second case study taken in consideration is a *FIR* filter, in which the multiplication of coefficients is parallelized by the methodology. Instead of executing a *for* loop performing the multiplication in one thread, it is split in three and is executed by different threads, then synchronized through the *get* method. The code of the *FIR* filter is given in listing A.6. As can be noted, the three *filter_mul* functions have different definitions and names, it is a strategy to make Bambu creating three different modules so that we can have effective parallelism, the same consideration is made for the *set_value* function. Another strategy adopted is to split the array named *insamp*, that holds the input samples in three arrays, so that every function can work in parallel on a different component avoiding to wait for the bus access. In this way there is a splitting in the control and data bus, since every function has its own bus line to access the array. This strategy is needed to not congestion the bus manager and to make more efficient the mechanism of parallelization. The example was taken from the ones provided by bambu and adapted for our case[16]. A discussion about the result obtained with this case study is provided in the next section.

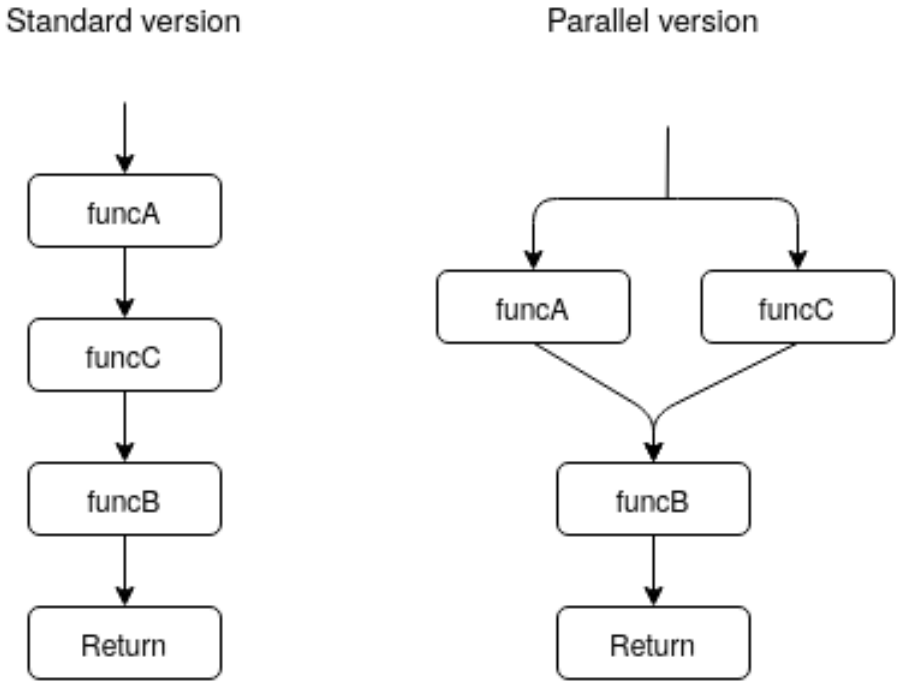


Figure 6.1: Comparison execution flow

6.2 Results

Bambu is a tool that given a C or C++ specification translate it in Verilog or VHDL but cannot be used for simulation and implementation goal. So the Vivado HLx 2019.1 tool [17] was used for this purpose, and thanks to which the results were obtained.

Three variations of the case study are taken in consideration: the one generated by the tool without introducing any parallelism, the second variation is the introduction of the *wait call* mechanism in the standard version, and the last one is with the new methodology introduced in this work.

Observe that the version with the *wait call* is not very significant in this case because an improvement in performance, particularly in the area usage is obtained when testing this mechanism with examples in which there are multiple calls to the same function by different modules, which is not our case. The study of this mechanism has already been carried out [14], for this reason we have not gone into this aspect in detail.

6.2.1 Custom sample

From the functional point of view the new methodology introduced is perfectly working, in fact it was tested with some simulations giving different input and obtaining the correct result for all of them compared with the results of the standard version.

Established that the result provided is correct it was possible to analyze the qualitative aspect. Table 6.1 shows the execution time in terms of clock cycles of the three versions as the number of iterations change, so for example 100i means that each function perform a loop of 100 iterations. The simulations was done using a clock frequency equal to 50MHz obtaining a clock period of 20ns.

	0i	1i	10i	100i	1000i	10000i	100000i	1000000i
Standard	4	8	44	404	4004	40004	400004	4000004
Wait call	126	130	166	526	4126	40126	400126	4000126
Parallel	409	363	372	661	3432	30432	300432	3000432

Table 6.1: Timing results of the three versions in number of clock cycles

Analyzing the results it is possible to say that in terms of time is preferable to use the standard method when the number of iterations, and so in general when the execution time of such functions is lesser than a certain threshold. If the functions are doing something that require more time, the proposed methodology is a way better than the other two. This is something that could be predicted, in fact we already know that introducing this method there is an overhead due to the communication protocol and the bus manager, and to get an improvement in the final execution you have to overcome this overhead making the functions be executed in parallel as long as possible.

To better explain this concept let's take the numbers of the table 6.1, in particular let's consider the iteration 100. In our there are four loops and each loop to complete an iteration uses one clock period. Now in the standard version the four loops are executed in sequence, this means that the final execution time considering only the loops in terms of number of clock cycles is:

$$1 \times 100 \times 4 = 400$$

Since other operations are needed to retrieve the result, the simulation lasts 404 clock cycles and we can say that an overhead of 4 is added to the execution of the standard version. Considering the new methodology again, the number of iterations is 100, here again there are four loops but they are not executed in sequence, two of them are executed in parallel and the other two in sequence so at the end we

have to considering three loops instead of four. We obtain a number of clock cycles considering only the loops of:

$$1 \times 100 \times 3 = 300$$

So if we consider only the execution of the functions, there is already an improvement but due to the overhead that in this case is about $661 - 300 = 361$ our methodology is not convenient in this case.

Using a bit of mathematics it's possible to calculate which is the minimum execution time of the functions to obtain an improvement using the methodology proposed against the standard one. In reality it is not the execution time of the functions to overcome a given threshold but is the time in which two functions are executed in parallel that must overcome the threshold. Suppose you are parallelizing two functions, one lasts 1 clock while the second one lasts 100, the gain in executing them in parallel respect to do them in sequence is just 1. If both last 100 clock cycles the gain is of 100. So if this methodology was implemented in Bambu, the tool to decide if is better to use the standard or the new version should make this consideration first.

The relation that must be satisfied in order to privilege the new methodology against the standard one is $Ov < t_p$ that is the overhead introduced by the method must be lesser than the parallel execution time.

Some numbers that can draw attention are shown again in table 6.1 regarding iterations zero and one, the new methodology requires more time to execute zero loop than one and this could seems a bit odd but thinking about how the method works there is an explication. Studying the trend of the three versions more in detail in the small numbers of iterations, it is possible to conclude that differently from the other two versions, the overhead introduced by the methodology is not fixed but depends on the number of iterations and so on the execution time of the functions. A graphic of this trend is shown in figure 6.2.

Since we have a *FIFO* queue that manages the master requests, it is possible that a master waits zero time or a time that can be more or less long. So we can differentiate between two limit cases, the one in which all the masters waits zero time that is the best scenario in terms of execution time, and the case in which the summation of the waiting time of all the masters is maximum, and this is the worst case. It is evident that in the case with zero iteration the bus is more congested than the case with one iteration.

Figure 6.3 shows that after a certain number of iteration the overhead introduced by the bus manager becomes fixed.

The goal of this work was to demonstrate the feasibility of application of futures and promises in the high level synthesis context and an important aspect related to it, is the time analysis that we already discussed. Since a previous work[14] already

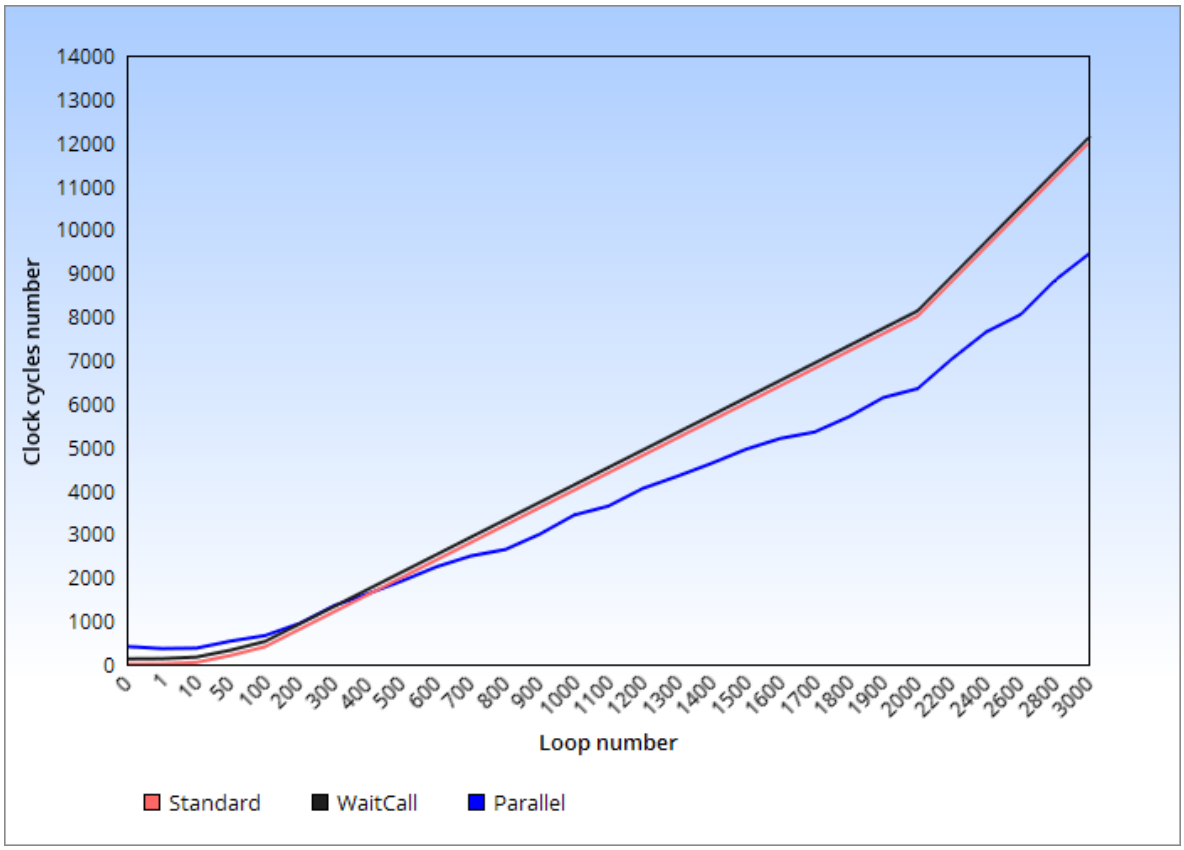


Figure 6.2: Timing for few iterations

talks about the area comparing the standard architecture generated by Bambu, we will not focus to much on that. Moreover this example was done with the idea to collect information on the execution time more than the area aspect and is not appropriate to discuss about the latter. Respect to the *wait call* method the new methodology introduce some other modules to manage the parallelism and so it is obvious that there is an increment in the area utilization. Table 6.2 reports the results for the three versions.

	LUT	FF	BUFG
Standard	549	205	1
Wait call	2394	1883	1
Parallel	3524	1990	2

Table 6.2: Area results of the three versions

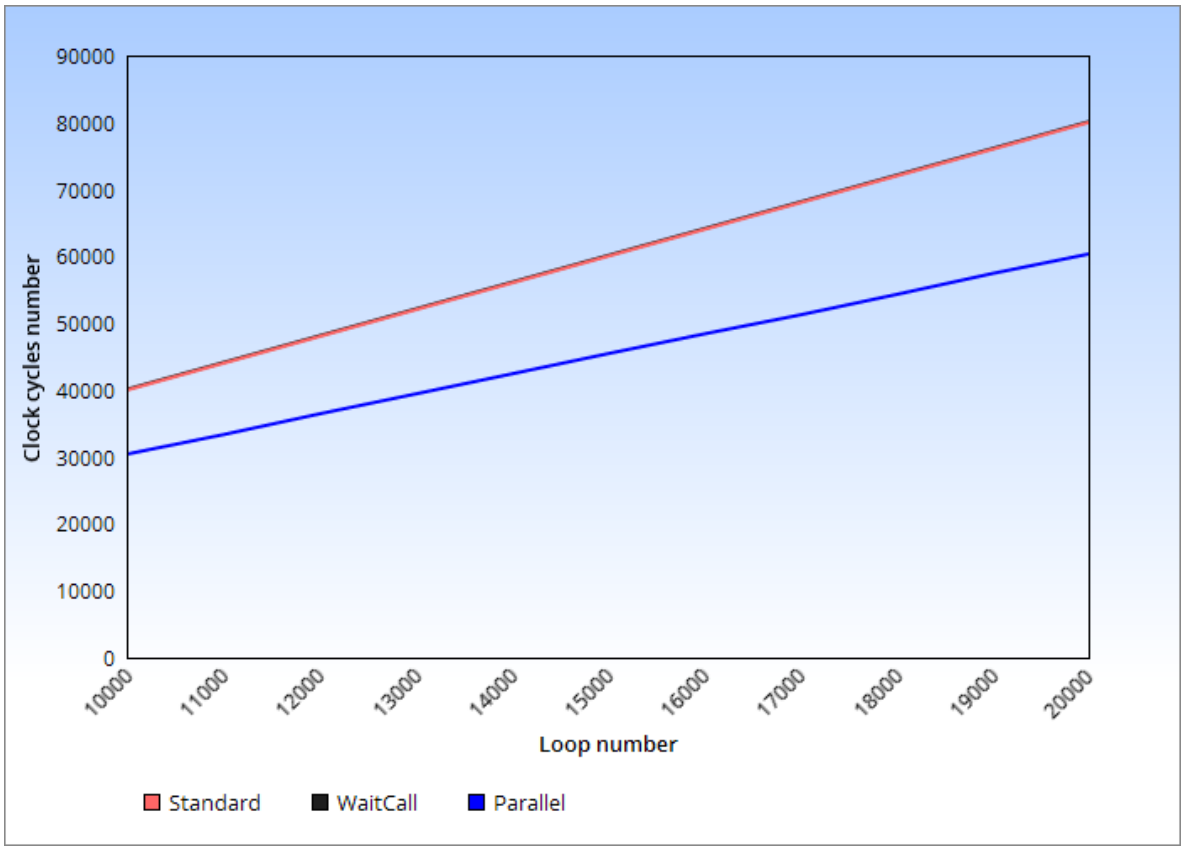


Figure 6.3: Timing for high number of iterations

With this case study, the area utilization is a way lower for the standard version with respect to the others two, due to the compiler choice to *inlining* all functions. In fact, *inlining* functions can generate an architecture that is bigger if the functions are big and are called many times, while can lead to a smaller system in case of small functions called few times because then the schedule and the optimization phases can better optimize the *inlined* function body considering it inside the calling function.

In general thanks to the previous work done on the *wait call* mechanism, it is possible to say that if the same function is called many time from different callers, there is a significant gain in the area using the *wait call* because it will instantiate only one instance of the called module instead of one for each call.

For our methodology is true that we use the same *wait call* mechanism to call the functions but since we want parallelism, it is not possible to instantiate only one module for all the call to it, otherwise we don't obtain a real parallelism. It should be necessary to introduce a method that evaluate the trade off between area and time. Let's suppose having one function called ten times using threads, futures and

promises. The programmer through an option must have the possibility to choose if the tool has to synthesize the high level specification as it is and so having a complete parallelism instantiating ten equal modules and executing them in parallel so using more area but saving time or to let the tool optimize the system evaluating a good trade off between the two.

6.2.2 FIR filter

Ideally the same considerations made for the previous example can be done also for the *FIR* filter, because the case of application is quite the same, there are three functions executing a *for* loop, this time for a fixed number of iterations. A difference is that now the function creating futures, promises and using the threads is called inside a *for* loop in the main, so is like calling the parallel methodology a certain number of times. The final overhead that before was calculated one time, now must be multiplied by the number of times the method is called in the *for* loop, 800 in this case. As already mentioned, the *FIR* example was taken from the one provided by bambu[16] but it was necessary to make some modification in order to obtain a system functionally working and also more efficient. In the original version there is only a loop that performs the multiplication and interacts with a global array containing the input samples. In the parallel version it was necessary to split the global array in three, one for each function, and create separate bus line in order to access these, otherwise since there is no concept of parallelism in bambu, it uses a unique bus line to access the arrays causing an high congestion of the bus. In previous case study this problem did not arise since no global memory was used, so the only purpose of the bus was to manage the control signal used to allow the parallel execution. Here there is a separation of control and data signals to relax the bus overload and have a gain in the execution time. For this example it was necessary to modify the high level language program to make bambu generating an architecture functionally working and optimized. Theoretically, if bambu had these concepts, it should be able to automatically perform this code optimization hiding the details to the programmer that could code without thinking too much at these kind of problems.

Let's focus now on some timing results. The times collected in terms of clock cycles are showed in table 6.3 and are now discussed.

	Clock cycles
Standard	253603
Parallel	413591

Table 6.3: Timing result fir filter

For what concerns the standard version we have the *main* that calls 800 times the *firfilter* function that has a loop of 63 iterations each lasts 3 clock cycles, one is used to load the two values of the global array, one for the multiplication operation and the last for storing the result. The shift of the global array is performed at the end of the function and here again there is a *for* loop of 63 iterations each lasts 2 clock cycles, one load and one store operation. In addition outside the loops, there are other 2 clock cycles. Having this numbers we obtain:

$$800 \times (63 \times 3 + 63 \times 2 + 2) = 253600$$

The same considerations can be done for the parallel version with the difference that now we have to consider the overhead introduced at every iteration of the *main* loop and that there are three functions executing the loop in parallel, so each loop is of 21 iterations instead of 63 as previously said, obtaining the following formula:

$$800 \times (21 \times 3 + 63 \times 2 + 2 + Ov) = 152800 + 800 \times Ov$$

This falls into the cases previously showed in which is not convenient applying the methodology, cause the overhead introduced in a single iteration of the *main* loop is higher than the gain we have in executing the functions in parallel. Each function executed by a thread is too short in terms of execution time to see a gain. it should be possible to see an improvement by increasing the size of the filter window, causing an increment in the execution time of *for* loops and also increasing the number of threads that run in parallel. But again the high level synthesis tool should make some considerations also in terms of area before saying which version to use, taking into account that more threads means more area.

A last consideration should be done on this case study. The main operation performed by the functions that are executed in parallel is the multiplication between the coefficients and the input samples, this operation acts on integers, and bambu assumes that it is computed in one clock cycle. In some real cases, it is possible that for designing reasons this operation can lasts more than one clock cycle, or that instead of having a multiplication between integers, real numbers are involved and so with high probability, is not possible to execute it in only one clock cycle. Figure 6.4 shows the trend of the execution time referred to the standard and parallel version increasing the time required to perform the multiplication. It is possible to see that if the multiplication required more than six clock cycles there would be an improvement in the performance of the *FIR* filter execution time. This is in accordance with the results obtained in the first case study, in which varying the duration of the parallel functions at a certain moment the gain obtained by the parallel execution against the sequential one overcome the overhead introduced by the methodology.

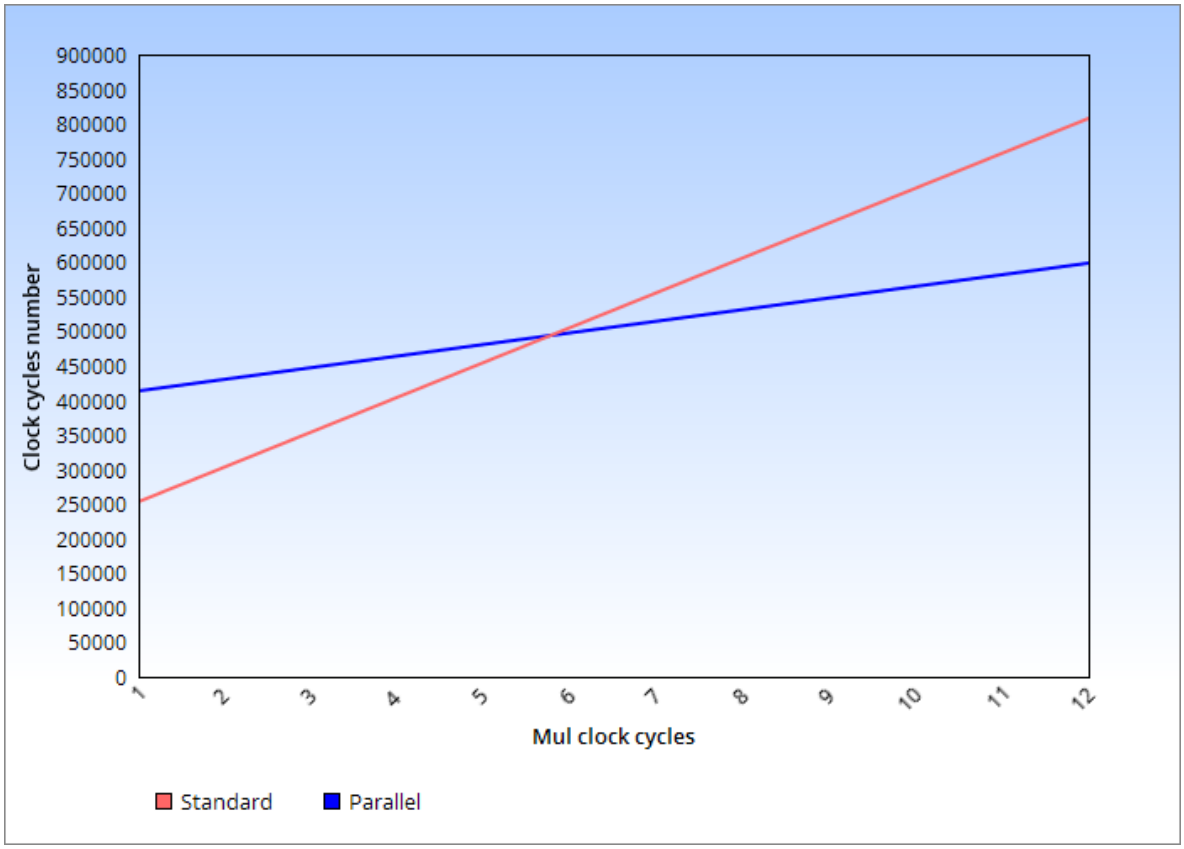


Figure 6.4: Execution time of fir filter, varying the mul clock cycles

Chapter 7

Conclusions

7.1 Final considerations

This work has presented a methodology that introduce the features of the C++ futures and promises in the frame of the high level synthesis. To obtain this result, the functionality of the tool Bambu was exploited, that is able to synthesize function calls through function pointers.

The strategy used to test the method, without entering in the implementation details of such a tool is to use an *autotool* project to link the library implemented to support threads, futures and promises, still following a sequential execution flow and then modify the generated Verilog introducing the needed modules to support the parallel execution.

The benefits produced by introducing this synchronization method for the threads are over than the execution time that is reduced. Other benefits are the widening of the input language that the tools can accept, and so as a consequence the more confidentiality of the high level programmer that can exploit better its capability of writing codes. Moreover if in future this methodology will be implemented inside the tool itself, it could be possible to reduce the time in creating the synchronization graph that are the starting point of the tools to then produce the working architecture. This graphs could be extracted faster because of the well known schedule and synchronization method introduced by futures and promises.

The final conclusion is that the basic concept was introduced and a working implementation of that is provided. Since only two case studies were taken in consideration, even if they care about different situations of usage of futures and promises it is not possible to conclude that the method works for every design until more and more testing will be produced. Since for now the implementation of such a method is not automated, a better choice could be implementing first the method in the tool; then conduce more test on other case studies. However since the general idea

and a first implementation is provided there should be more effort in verification and testing of the methodology to find and solve potential bugs.

7.2 Future work

The library proposed in this thesis is very minimal and could be improved to make it much more similar to the standard one, implementing also some errors checking. This library should be put directly inside the tool Bambu so that it is possible to use it without the need of an *autotool* project. Then for what concern the methodology introduced, it can be the starting point to implement it in a way that the tool can automatically generates the desired architecture and test with more different case studies if the proposed solution can be adapted to all the cases of usage of futures and promises.
Moreover it should be possible to introduce some strategies that get an improvement in the execution time due to the overhead saving some clock cycles. Even saving very few clock cycles in the control sequence that enable the methodology to work, in some contexts can have a big impact, think about the *FIR* example, in which the methodology is called inside a for loop, if that loop has a huge number of iterations the gain in reducing the overhead time is very significant.

7.2.1 Async

Async is a function introduced to simplify the usage of the futures and promises features, in fact using that function it is possible to achieve the same result managing only the future object, while the rest is automated. *Async* runs a function asynchronously and returns a future object, then when the called function terminates and maybe return a value, this value is automatically set in the future object, and can be retrieved by the *main* with the *get* method. So with *Async* there is no need of managing promises and threads but just futures.

```
template< class Function, class... Args >
async( std::launch policy, Function&& f, Args&&... args
```

Listing 7.1: async

Listing 7.1 shows the function template definition, in which policy can be *std::launch::async* or *std::launch::deferred*. The first one means that the function *f* is launched in a separate thread and it is executed independent of the *main* thread so the *main* thread continue to execute. *std::launch::deferred* with this policy the

function f is executed by the *main* thread only when the *get* is called on the future object.

Without going more in depth in the *async* function, it is possible to say that on the bases of what achieved in this work, it should be possible to implement the function in the high level synthesis context, in fact the *async* only mask some behaviors that have already been implemented. It could also be easier with respect to using threads, futures and promises because there will be the need to map only the *async* function in doing what has been done using threads and promises while the same meaning is assigned to future objects.

Appendix A

C++ library code

Listing A.1: Library header

```
#ifndef MYLIB_HPP_
#define MYLIB_HPP_


namespace std {

  //forward declaration
  template<class P>
  class promise;
  template<class F>
  class future;


  template<class T, class... _Args>
  void thread( T (*f)(_Args... __args), _Args... __args);


  template<class P>
  class promise {
    P retVal;

    public:
      void set_value(P val);
      future<P>* get_future();
  };

  template<class F>
  class future {
    promise<F> *fPromise;
```

```
    public:
      F get();
  };

}//namespace std

#endif
```

Listing A.2: future and promise class

```
#include "FP.h"


namespace std {

//////////////////FUTURE////////////////////
  template<class P>
  future<P> tmp = * (new future<P>);

  template<class F>
  F future<F>::get(){
    return this->fPromise->retVal;
  }


//////////////////PROMISE//////////////////

  template<class P>
  void promise<P>::set_value(P val){

    this->retVal = val;
  }

  template<class P>
  future<P>* promise<P>::get_future(){
    tmp<P>.fPromise = this;

    return &tmp<P>;
  }


////////////////////////////////////////

//explicit instantiations
template class promise<bool>;
template class future<bool>;
```

```
template class promise<char>;
template class future<char>;
template class promise<signed char>;
template class future<signed char>;
template class promise<unsigned char>;
template class future<unsigned char>;

template class promise<int>;
template class future<int>;
template class promise<unsigned int>;
template class future<unsigned int>;
template class promise<short int>;
template class future<short int>;
template class promise<unsigned short int>;
template class future<unsigned short int>;
template class promise<long int>;
template class future<long int>;
template class promise<unsigned long int>;
template class future<unsigned long int>;
template class promise<long long int>;
template class future<long long int>;
template class promise<unsigned long long int>;
template class future<unsigned long long int>;

template class promise<float>;
template class future<float>;

template class promise<double>;
template class future<double>;
template class promise<long double>;
template class future<long double>;

}//namespace std
```

Listing A.3: thread class

```
#include "FP.h"

namespace std{


  template<class T, class... _Args>
  void thread( T (*f)(_Args... __args), _Args... __args) {

      (*f)(__args...);
```

```
  }


////////////////////////////////////////
//explicit instantiations
template void thread<void, promise<int>*, int>(void (*f)(promise<
    int> *prom, int a), promise<int>* prom, int a);
template void thread<void, promise<int>*, promise<int>*, int, int>(
    void (*f)(promise<int> *prom, promise<int> *prom1, int a, int b)
    , promise<int>* prom, promise<int> *prom1, int a, int b);

}//namespace std
```

Listing A.4: Example 1

```
#include "FP.h"

using namespace std;

void funcA(Promise<int> *prom, int varA){

    int res = 0;

    for (int i = 0; i < varA; i++){
        res += varA+i;
    }

    prom->set_value(res);
}

int funcB(int varB) {
    int res=0;

    for (int i = 0; i < varB; i++)
    {
        res += varB+i;
        if(res > 12)
            res+= 2;
        else
            res+= 3;
    }

    return res;
}

int start_point(int varA, int varB)
```

```
{
    int res, var;

    Promise<int> myp;
    Future<int> myf = *(myp.get_future());

    thread (funcA, &myp, varA);
    var = funcB(varB);

    res = var + myf.get();

    return res;
}
```

Listing A.5: Example 2

```
#include "FP.h"

using namespace std;


void funcA(promise<int> *prom, promise<int> *prom1, int varA, int
   varAa){

    int res = 0;
    int res1 = 0;

    for (int i = 0; i < varA; i++)    {
        res += varA+i;
    }

    for (int i = 0; i < varAa; i++)    {
        res1 += varAa+i;
    }

    prom->set_value(res);
    prom1->set_value(res1);
}

int funcB(int varB) {
    int res=0;

    for (int i = 0; i < varB; i++){
      res += varB+i;
    }
```

```
    return res;
}

void funcC(promise<int> *prom, int varC){

    int res = 0;

    for (int i = 0; i < varC; i++)    {
        res += varC+i;
        if(res > 12)
            res+= 2;
        else
            res+= 3;
    }

    prom->set_value(res);
}

int start_point(int varA, int varB, int varC, int varAa)
{
    int res, var;

    promise<int> myp;
    future<int> myf = *(myp.get_future());

    promise<int> myp1;
    future<int> myf1 = *(myp1.get_future());

    promise<int> myp2;
    future<int> myf2 = *(myp2.get_future());

    thread (funcA, &myp, &myp2, varA, varAa);
    thread(funcC, &myp1, varC);

    var = funcB(varB);

    res = var + myf.get() + myf1.get() + myf2.get();

    return res;
}
```

Listing A.6: Parallel FIR filter

```
#include <stdio.h>
#include <stdint.h>
#define _USE_MATH_DEFINES
```

```
#include <array>    // array
#include <utility>  // index_sequence, make_index_sequence
#include <cmath>    // sinf,cosf
#include <cstddef>  // size_t
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <cassert>
#include "FP.h"

using namespace std;

// maximum length of filter than can be handled
#define FILTER_LEN     63
#define array_length   21

// arrays to hold input samples
int16_t insamp0[ array_length ];
int16_t insamp1[ array_length ];
int16_t insamp2[ array_length ];

void filter_mul(Promise<int> *prom);
void filter_mul1(Promise<int> *prom, int x);
void filter_mul2(Promise<int> *prom, int x, int y);

template <int filter_len>
class firFixedClass
{
    constexpr static short int compute_coeff(size_t index)
    {
        // parameters and simulation options
        float        fc    = 0.20;  // normalized cutoff frequency
        // generate time vector, centered at zero
        float t = (float)index + 0.5f - 0.5f*(float)filter_len;
        // generate sinc function (time offset in 't' prevents
   divide by zero)
        float s = sinf(2*M_PI*fc*t + 1e-6f) / (2*M_PI*fc*t + 1e-6f)
   ;
        // generate Hamming window
        float w = 0.53836 - 0.46164*cosf((2*M_PI*(float)index)/((
   float)(filter_len-1)));
        // generate composite filter coefficient
        return (s * w)*(1<<14);
    }
    template <std::size_t... I>
    constexpr static std::array<short int, sizeof...(I)> coeff_fill
   (std::index_sequence<I...>)
    {
```

```
        return std::array<short int, sizeof...(I)>{compute_coeff(I)
    ...};
    }
    template <std::size_t N>
    constexpr static std::array<short int, N> coeff_fill()
    {
        return coeff_fill(std::make_index_sequence<N>{});
    }

    public:
        static const ::std::array<short int,filter_len> coeffs;

    short int operator()(short int inputt)
    {
        int16_t output;
        int32_t acc;     // accumulator for MACs

        Promise<int> myp;
        Future<int> myf = *(myp.get_future());

        Promise<int> myp1;
        Future<int> myf1 = *(myp1.get_future());

        Promise<int> myp2;
        Future<int> myf2 = *(myp2.get_future());

        // put the new samples at the high end of the buffer
        insamp2[array_length - 1] = inputt;

        // load rounding constant
        acc = (1 << 14)+((int32_t)(coeffs[0]) * (int32_t)(inputt));

        thread(filter_mul, &myp);
        thread(filter_mul1, &myp1, 0);
        thread(filter_mul2, &myp2, 0, 0);

        acc = acc + myf.get() + myf1.get() + myf2.get();

        // saturate the result
        if ( acc > 0x3fffffff ) {
            acc = 0x3fffffff;
        } else if ( acc < -0x40000000 ) {
            acc = -0x40000000;
        }
        // convert from Q30 to Q15
        output = (int16_t)(acc >> 15);

        // shift input samples back in time for next time
        for (int k = 0; k < array_length-1; k++){
```

```
            insamp0[k] = insamp0[k+1];
        }

        insamp0[array_length-1]=insamp1[0];

        for (int k = 0; k < array_length-1; k++){
            insamp1[k] = insamp1[k+1];
        }

        insamp1[array_length-1]=insamp2[0];

        for (int k = 0; k < array_length-1; k++){
            insamp2[k] = insamp2[k+1];
        }

        return output;
    }
        firFixedClass() {}
};

template <int filter_len>
const ::std::array<short int,filter_len> firFixedClass<filter_len
   >::coeffs = coeff_fill<filter_len>();


// the FIR filter function
#ifdef WITH_EXTERNC
extern "C"
#endif
short int
__attribute__ ((noinline))
firFixed( short int inputt)
{
    firFixedClass<FILTER_LEN> fir;
    return fir(inputt);
}


int start_point()
{
    int16_t input[800] = fill_input();

    int16_t output[800];

    for (int i = 0; i < 800; i++){
        output[i] = firFixed(input[i]);
    }

    return output[796];
```

```
}

void filter_mul(Promise<int> *prom) {

    int k;
    int32_t acc = 0;

    for ( k = 0; k < array_length; k++ ) {
        acc += (int32_t)(firFixedClass<FILTER_LEN>::coeffs[
   FILTER_LEN - 1 -k]) * (int32_t)(insamp0[k]);
    }

    prom->set_value(acc);
}

void filter_mul1(Promise<int> *prom, int x) {

    int k;
    int32_t acc = 0;

    for ( k = 0; k < array_length; k++ ) {
        acc += (int32_t)(firFixedClass<FILTER_LEN>::coeffs[
   FILTER_LEN - 1 -k - array_length]) * (int32_t)(insamp1[k]);
    }

    prom->set_value1(acc, 0);
}

void filter_mul2(Promise<int> *prom, int x, int y) {

    int k;
    int32_t acc = 0;

    for ( k = 0; k < array_length; k++ ) {
        acc += (int32_t)(firFixedClass<FILTER_LEN>::coeffs[
   FILTER_LEN - 1 -k - 2*array_length]) * (int32_t)(insamp2[k]);
    }

    prom->set_value2(acc, 0, 0);
}
```

Appendix B

Verilog codes

Listing B.1: notify_caller_p

```
module notify_caller_p(clock, reset, Min_oe_ram, Mout_oe_ram,
    Min_we_ram, Mout_we_ram, Min_addr_ram, Mout_addr_ram,
    M_Rdata_ram, Min_Wdata_ram, Mout_Wdata_ram, Min_data_ram_size,
    Mout_data_ram_size, M_DataRdy, S_we_ram, S_addr_ram,
    Sout_DataRdy, Sin_DataRdy);
  parameter BITSIZE_Min_addr_ram=1, BITSIZE_Mout_addr_ram=1,
    BITSIZE_M_Rdata_ram=1, BITSIZE_M_DataRdy=1,
    BITSIZE_Min_Wdata_ram=1, BITSIZE_Mout_Wdata_ram=1,
    BITSIZE_Sin_DataRdy=1, BITSIZE_Sout_DataRdy=1,
    BITSIZE_Min_data_ram_size=1, BITSIZE_S_addr_ram=1,
    BITSIZE_Mout_data_ram_size=1, MY_ADDRESS=0, ARRAY_ADDRESS=0,
    notifyAddress=0;
  // IN
  input clock, reset;
  input Min_oe_ram, Min_we_ram, S_we_ram;
  input [BITSIZE_Min_addr_ram-1:0] Min_addr_ram;
  input [BITSIZE_M_Rdata_ram-1:0] M_Rdata_ram;
  input [BITSIZE_Min_Wdata_ram-1:0] Min_Wdata_ram;
  input [BITSIZE_Min_data_ram_size-1:0] Min_data_ram_size;
  input [BITSIZE_S_addr_ram-1:0] S_addr_ram;
  input [BITSIZE_M_DataRdy-1:0] M_DataRdy;
  input [BITSIZE_Sin_DataRdy-1:0] Sin_DataRdy;
  // OUT
  output Mout_oe_ram, Mout_we_ram;
  output [BITSIZE_Sout_DataRdy-1:0] Sout_DataRdy;
  output [BITSIZE_Mout_addr_ram-1:0] Mout_addr_ram;
  output [BITSIZE_Mout_Wdata_ram-1:0] Mout_Wdata_ram;
  output [BITSIZE_Mout_data_ram_size-1:0] Mout_data_ram_size;

  reg [31:0] state =0;
```

```
reg [31:0] next_state;
reg Mout_oe_ram, Mout_we_ram;
reg [BITSIZE_Mout_addr_ram-1:0] Mout_addr_ram;
reg [BITSIZE_Mout_Wdata_ram-1:0] Mout_Wdata_ram;
reg [BITSIZE_Mout_data_ram_size-1:0] Mout_data_ram_size;
wire condition;

assign condition = S_addr_ram == MY_ADDRESS;
assign Sout_DataRdy = condition ? MY_ADDRESS : Sin_DataRdy;

parameter [31:0] S_0 = 32'd0,
S_1 = 32'd1,
S_2 = 32'd2,
S_3 = 32'd3,
S_4 = 32'd4;

always @ (posedge clock ) begin
  if (reset == 1'b0) begin
    state <= 0;
  end
  else begin
    state <= next_state;
  end
end

always @ (*) begin
  Mout_we_ram = Min_we_ram;
  Mout_Wdata_ram = Min_Wdata_ram;
  Mout_oe_ram = Min_oe_ram;
  Mout_addr_ram = Min_addr_ram;
  Mout_data_ram_size = Min_data_ram_size;

  if (state == S_0) begin
     if (S_we_ram && S_addr_ram == ARRAY_ADDRESS) begin //set
 executed first
        next_state = S_1;
     end
     else if (S_we_ram && condition) begin//get executed first
        next_state = S_2;
     end else begin
        next_state = S_0;
     end
  end
  else if (state == S_1) begin
     if (S_we_ram && condition) begin
        next_state = S_3;
     end else begin
        next_state = S_1;
     end
```

```
    end
    else if (state == S_2) begin
       if (S_we_ram && S_addr_ram == ARRAY_ADDRESS) begin
          next_state = S_3;
       end else begin
          next_state = S_2;
       end
    end
    else if (state == S_3) begin
       Mout_we_ram = 1'b1;
       Mout_addr_ram = notifyAddress; //address of the getManager
       Mout_Wdata_ram = 1'b0;
       Mout_data_ram_size = 32;
       if (M_DataRdy == notifyAddress) begin
         next_state = S_0;
       end else begin
         next_state = S_3;
       end
    end
  end

endmodule
```

Listing B.2: getManager

```
module getManager(clock, reset, start_port, in1, Min_oe_ram,
   Min_we_ram, Min_addr_ram,  Min_Wdata_ram, Min_data_ram_size,
   M_DataRdy, S_oe_ram, S_we_ram, S_addr_ram,  Sin_Rdata_ram,
   Sin_DataRdy, done_port, Mout_oe_ram, Mout_we_ram, Mout_addr_ram,
    Mout_Wdata_ram, Mout_data_ram_size, Sout_Rdata_ram,
   Sout_DataRdy);
parameter MyAddress=0, BITSIZE_Min_addr_ram=1,
   BITSIZE_Mout_addr_ram=1, BITSIZE_Min_Wdata_ram=1,
   BITSIZE_Mout_Wdata_ram=1, BITSIZE_Min_data_ram_size=1,
   BITSIZE_Mout_data_ram_size=1, BITSIZE_S_addr_ram=1,
   BITSIZE_Sin_Rdata_ram=1, BITSIZE_Sout_Rdata_ram=1,
   BITSIZE_M_DataRdy=1, BITSIZE_Sin_DataRdy=1, BITSIZE_Sout_DataRdy
   =1, BITSIZE_in1=1;
  // IN
  input clock, reset, start_port;
  input [BITSIZE_in1-1:0] in1;
  input Min_oe_ram, Min_we_ram;
  input [BITSIZE_Min_addr_ram-1:0] Min_addr_ram;
  input [BITSIZE_Min_Wdata_ram-1:0] Min_Wdata_ram;
  input [BITSIZE_Min_data_ram_size-1:0] Min_data_ram_size;
  input [BITSIZE_M_DataRdy-1:0] M_DataRdy;
```

```
input S_oe_ram, S_we_ram;
input [BITSIZE_S_addr_ram-1:0] S_addr_ram;
input [BITSIZE_Sin_Rdata_ram-1:0] Sin_Rdata_ram;
input [BITSIZE_Sin_DataRdy-1:0] Sin_DataRdy;
// OUT
output done_port;
output Mout_oe_ram, Mout_we_ram;
output [BITSIZE_Mout_addr_ram-1:0] Mout_addr_ram;
output [BITSIZE_Mout_Wdata_ram-1:0] Mout_Wdata_ram;
output [BITSIZE_Mout_data_ram_size-1:0] Mout_data_ram_size;
output [BITSIZE_Sout_Rdata_ram-1:0] Sout_Rdata_ram;
output [BITSIZE_Sout_DataRdy-1:0] Sout_DataRdy;


reg [31:0] step =0;
reg [31:0] next_step;
reg done_port;
reg Mout_oe_ram, Mout_we_ram;
reg [BITSIZE_Mout_addr_ram-1:0] Mout_addr_ram;
reg [BITSIZE_Mout_Wdata_ram-1:0] Mout_Wdata_ram;
reg [BITSIZE_Mout_data_ram_size-1:0] Mout_data_ram_size;
wire condition;

assign condition = S_addr_ram == MyAddress;
assign Sout_DataRdy = condition ? MyAddress : Sin_DataRdy;
assign Sout_Rdata_ram = Sin_Rdata_ram;

parameter [31:0] S_0 = 32'd0,
S_1 = 32'd1,
S_2 = 32'd2;

always @ (posedge clock )
  if (reset == 1'b0)
  begin
    step <= 0;
  end else begin
    step <= next_step;
  end


always @(*)
  begin
  done_port = 1'b0;
  next_step = S_0;
  Mout_we_ram = Min_we_ram;
  Mout_Wdata_ram = Min_Wdata_ram;
  Mout_oe_ram = Min_oe_ram;
  Mout_addr_ram = Min_addr_ram;
  Mout_data_ram_size = Min_data_ram_size;
```

```
    if (step == S_0) begin
      if (start_port == 1'b1) begin
        next_step = S_1;
      end else begin
        next_step = S_0;
      end
    end
    else if (step == S_1) begin
      Mout_we_ram = 1'b1;
      Mout_addr_ram = in1 + 32'd500; //address of notify_caller_p
      Mout_Wdata_ram = 32'd0;
      Mout_data_ram_size = 32;
      if (M_DataRdy == (in1 + 32'd500)) begin
        next_step = S_2;
      end else begin
        next_step = S_1;
      end
    end
    else if (step == S_2) begin
      if (S_we_ram == 1 && condition) begin
  done_port = 1'b1;
  next_step = S_0;
      end else begin
        next_step = S_2;
      end
    end
  end

endmodule
```

Listing B.3: compacting_FU

```
module compacting_FU(Sout_DataRdy, Sout_Rdata_ram, Mout_oe_ram,
    Mout_we_ram, Mout_addr_ram, Mout_Wdata_ram, Mout_data_ram_size,
    out1);
  parameter BITSIZE_Sout_DataRdy=1, BITSIZE_Sout_Rdata_ram=1,
    BITSIZE_Mout_addr_ram=1, BITSIZE_Mout_Wdata_ram=1,
    BITSIZE_Mout_data_ram_size=1;
  // IN
  input [BITSIZE_Sout_Rdata_ram-1:0] Sout_Rdata_ram;
  input Mout_we_ram;
  input Mout_oe_ram;
  input [BITSIZE_Mout_addr_ram-1:0] Mout_addr_ram;
  input [BITSIZE_Sout_DataRdy-1:0]Sout_DataRdy;
  input [BITSIZE_Mout_Wdata_ram-1:0]Mout_Wdata_ram;
  input [BITSIZE_Mout_data_ram_size-1:0]Mout_data_ram_size;
```

```
  // OUT
  output [(BITSIZE_Sout_DataRdy + BITSIZE_Sout_Rdata_ram + 2 +
   BITSIZE_Mout_addr_ram + BITSIZE_Mout_Wdata_ram +
   BITSIZE_Mout_data_ram_size)+(-1):0] out1;


  assign out1 = {Sout_Rdata_ram, Sout_DataRdy, Mout_addr_ram,
   Mout_Wdata_ram, Mout_data_ram_size, Mout_oe_ram, Mout_we_ram};

endmodule
```

Listing B.4: unzip_FU

```
module unzip_FU(M_DataRdy, M_Rdata_ram, S_oe_ram, S_we_ram,
   S_addr_ram, S_Wdata_ram, S_data_ram_size, in1);
  parameter BITSIZE_M_DataRdy=1, BITSIZE_M_Rdata_ram=1,
   BITSIZE_S_addr_ram=1, BITSIZE_S_Wdata_ram=1,
   BITSIZE_S_data_ram_size=1;
  // IN
  input [(BITSIZE_M_DataRdy + BITSIZE_M_Rdata_ram + 2 +
   BITSIZE_S_addr_ram + BITSIZE_S_Wdata_ram +
   BITSIZE_S_data_ram_size)+(-1):0] in1;
  // OUT
  output [BITSIZE_M_Rdata_ram-1:0] M_Rdata_ram;
  output S_we_ram;
  output S_oe_ram;
  output [BITSIZE_S_addr_ram-1:0] S_addr_ram;
  output [BITSIZE_M_DataRdy-1:0]M_DataRdy;
  output [BITSIZE_S_Wdata_ram-1:0]S_Wdata_ram;
  output [BITSIZE_S_data_ram_size-1:0]S_data_ram_size;


  assign M_Rdata_ram = in1[(BITSIZE_M_DataRdy + BITSIZE_M_Rdata_ram
    + 2 + BITSIZE_S_addr_ram + BITSIZE_S_Wdata_ram +
   BITSIZE_S_data_ram_size) +(-1) : (BITSIZE_M_DataRdy + 2 +
   BITSIZE_S_addr_ram + BITSIZE_S_Wdata_ram +
   BITSIZE_S_data_ram_size)];

  assign M_DataRdy = in1[(BITSIZE_M_DataRdy + 2 +
   BITSIZE_S_addr_ram + BITSIZE_S_Wdata_ram +
   BITSIZE_S_data_ram_size) +(-1) : (2 + BITSIZE_S_addr_ram +
   BITSIZE_S_Wdata_ram + BITSIZE_S_data_ram_size)];

  assign S_addr_ram = in1[(2 + BITSIZE_S_addr_ram +
   BITSIZE_S_Wdata_ram + BITSIZE_S_data_ram_size) +(-1) : (2 +
   BITSIZE_S_Wdata_ram + BITSIZE_S_data_ram_size)];
```

```
  assign S_Wdata_ram = in1[(2 + BITSIZE_S_Wdata_ram +
   BITSIZE_S_data_ram_size) +(-1) : (2 + BITSIZE_S_data_ram_size)];

  assign S_data_ram_size = in1[(2 + BITSIZE_S_data_ram_size) +(-1)
   : (2)];

  assign S_oe_ram = in1[1:1];

  assign S_we_ram = in1[0:0];

endmodule
```

Listing B.5: busManager

```
module FIFObuffer( clock, dataIn, RD, dataOut, reset, EMPTY,
    busSelector);
  parameter PORTSIZE_in1=1, BITSIZE_in1=1, BITSIZE_master=1;
  //IN
  input clock, RD, reset;
  input [(PORTSIZE_in1*BITSIZE_in1)+(-1):0] dataIn;
  input busSelector;
  input done_port;
  //OUT
  output reg [BITSIZE_master-1:0] dataOut;
  output EMPTY;

  reg [7:0] Count = 0;
  reg [BITSIZE_master-1:0] FIFO [0:PORTSIZE_in1*2];
  reg [7:0] readCounter = 0, writeCounter = 0;

  integer i1;

  assign EMPTY = (Count==0)? 1'b1:1'b0;

  always @ (dataIn)
  begin
  if(busSelector == 1) begin
    for(i1 = 0; i1 < PORTSIZE_in1; i1 = i1 + 1)
    begin
      if(dataIn[(i1*BITSIZE_in1) +:BITSIZE_master] != 0) begin
        if(Count < PORTSIZE_in1*2) begin
          FIFO[writeCounter] = dataIn[(i1*BITSIZE_in1) +:
   BITSIZE_master];
          writeCounter = writeCounter+1;
          if (writeCounter==PORTSIZE_in1*2) begin
```

```
            writeCounter=0;
          end
        end
      end
    end
   end
  end

  always @ (posedge clock)
  begin
  if (busSelector==1) begin
    if (RD ==1'b1 && Count!=0) begin
      dataOut  = FIFO[readCounter];
      readCounter = readCounter+1;
    end
    if (readCounter==PORTSIZE_in1*2) begin
      readCounter=0;
    end
    if (readCounter > writeCounter) begin
      Count=readCounter-writeCounter;
    end
    else if (writeCounter > readCounter)
      Count=writeCounter-readCounter;
    else;
  end
  end

  always @ (posedge clock)
  begin
    if (reset == 0) begin
      readCounter = 0;
      writeCounter = 0;
      Count = 0;
    end
  end

endmodule


//////////////////////////////////////////////////
module controller_bus_manager(MSSigIn, MSSigOut, clock, reset,
   bufferdata, EMPTY, RD, busSelector);
  parameter BITSIZE_in1=1, PORTSIZE_in1=2, BITSIZE_out1=1,
   BITSIZE_Sout_Rdata_ram=1, BITSIZE_Sout_DataRdy=1, BITSIZE_master
   =1, BITSIZE_slave=1;
  // IN
  input [(PORTSIZE_in1*BITSIZE_in1)+(-1):0] MSSigIn;
  input clock, reset;
  input EMPTY;
```

```
input busSelector;
input [BITSIZE_master-1:0] bufferdata;
// OUT
output [BITSIZE_out1-1:0] MSSigOut;
output RD;
parameter [2:0] S_0 = 3'b001,
  S_1 = 3'b010,
  S_2 = 3'b100;
reg [2:0] _present_state = S_0, _next_state;
reg [BITSIZE_master-1:0] result_M = 0;
reg [BITSIZE_slave-1:0] result_S = 0;
reg RD;
reg [BITSIZE_out1-1:0] MSSigOut;


always @(posedge clock)
  if (reset == 1'b0)
  begin
    _present_state <= S_0;
  end else begin
    _present_state <= _next_state;
  end

always @(negedge clock)
  begin
    if(busSelector == 1) begin
    _next_state = S_0;
    case (_present_state)
      S_0 :
        begin
          if(EMPTY == 0) begin
            RD = 1;
            _next_state = S_1;
          end else begin
            _next_state = S_0;
          end
        end
      S_1 :
        begin
          if(bufferdata != 0) begin
            RD = 0;
            _next_state = S_2;
            MSSigOut = bufferdata;
            result_M = bufferdata;
          end else begin
            _next_state = S_1;
            MSSigOut = 0;
          end
        end
```

```
        S_2 :
          begin
            result_S = findSlave(MSSigIn);
            if(result_S != 0) begin
              MSSigOut = {result_S, result_M};
              _next_state = S_0;
            end else begin
              _next_state = S_2;
            end
          end
        default :
          begin
            MSSigOut = 0;
          end
      endcase
     end
    end


  function [BITSIZE_slave-1:0] findSlave;
    input [BITSIZE_in1*PORTSIZE_in1-1:0] m;
    reg [BITSIZE_slave-1:0] res;
    integer i1;
  begin
    res={BITSIZE_slave{1'b0}};

    for(i1 = 0; i1 < PORTSIZE_in1; i1 = i1 + 1)
    begin
      if(m[((i1 + 1)*BITSIZE_out1 - BITSIZE_Sout_DataRdy -
   BITSIZE_Sout_Rdata_ram) +:(BITSIZE_Sout_DataRdy +
   BITSIZE_Sout_Rdata_ram)] != 0) begin
        res = m[((i1+1)*BITSIZE_out1 - BITSIZE_slave) +:(
   BITSIZE_slave)];
      end
    end
    findSlave = res;
  end
  endfunction

endmodule


//////////////////////////////////////////////////////
module bus_manager(MSSigIn, MSSigOut, clock, reset, busSelector);
  parameter BITSIZE_in1=1, PORTSIZE_in1=2, BITSIZE_out1=1,
   BITSIZE_Sout_Rdata_ram=1, BITSIZE_Sout_DataRdy=1, BITSIZE_master
   =1, BITSIZE_slave=1;
  // IN
  input [(PORTSIZE_in1*BITSIZE_in1)+(-1):0] MSSigIn;
```

```
  input clock, reset;
  input busSelector;
  // OUT
  output [BITSIZE_out1-1:0] MSSigOut;

  reg [BITSIZE_out1-1:0] MSSigOut;

  wire read;
  wire [BITSIZE_master-1:0] bufferout;
  wire [BITSIZE_out1-1:0] out;
  wire empty;

always @(*)
  begin
    if(busSelector == 1) begin
      MSSigOut = out;
    end else begin
      MSSigOut = merge(MSSigIn);
    end
  end

//bus merger
  function [BITSIZE_out1-1:0] merge;
    input [BITSIZE_in1*PORTSIZE_in1-1:0] m;
    reg [BITSIZE_out1-1:0] res;
    integer i1;
  begin
    res={BITSIZE_in1{1'b0}};
    for(i1 = 0; i1 < PORTSIZE_in1; i1 = i1 + 1)
    begin
      res = res | m[i1*BITSIZE_in1 +:BITSIZE_in1];
    end
    merge = res;
  end
  endfunction

  controller_bus_manager
#(.BITSIZE_master(BITSIZE_master), .BITSIZE_slave(BITSIZE_slave), .
    BITSIZE_in1(BITSIZE_in1), .PORTSIZE_in1(PORTSIZE_in1), .
    BITSIZE_out1(BITSIZE_out1), .BITSIZE_Sout_Rdata_ram(
    BITSIZE_Sout_Rdata_ram), .BITSIZE_Sout_DataRdy(
    BITSIZE_Sout_DataRdy))
ControllerBM
(.busSelector(busSelector), .clock(clock), .reset(reset), .EMPTY(
    empty), .MSSigIn(MSSigIn), .RD(read), .MSSigOut(out), .
    bufferdata(bufferout));
  FIFObuffer
#(.BITSIZE_master(BITSIZE_master), .BITSIZE_in1(BITSIZE_in1), .
    PORTSIZE_in1(PORTSIZE_in1))
```

```
Fbuff
(.busSelector(busSelector), .EMPTY(empty), .clock(clock), .dataIn(
    MSSigIn), .RD(read), .dataOut(bufferout), .reset(reset));

endmodule
```

Listing B.6: Proxy manager

```
module proxy_manager(SigIn, SigOut, clock, reset, proxy_done);
  parameter BITSIZE_in1=1, PORTSIZE_in1=2, BITSIZE_out1=1;
  // IN
  input [(PORTSIZE_in1*BITSIZE_in1)+(-1):0] SigIn;
  input [31:0] proxy_done;
  input clock;
  input reset;
  // OUT
  output [BITSIZE_out1-1:0] SigOut;

  parameter [2:0] S_0 = 3'b001,
    S_1 = 3'b010,
    S_2 = 3'b100;
  reg [2:0] _present_state = S_0, _next_state;
  reg [BITSIZE_in1-1:0] result = 0;


  reg [BITSIZE_out1-1:0] SigOut;
  reg terminate = 1;

  always @ (posedge clock)
    if (reset == 1'b0)
    begin
      _present_state <= S_0;
      terminate <=1;
    end else begin
      _present_state <= _next_state;
    end

  always @(clock or proxy_done)
    begin
      _next_state = S_0;
      if(proxy_done != 0) begin
        terminate = 1;
        SigOut = 0;
      end else
      case (_present_state)
        S_0 :
```

```
          begin
            if(terminate != 0) begin
              result = findSig(SigIn);
              if(result != 0) begin
                SigOut = result;
                _next_state = S_0;
                terminate = 0;
              end else begin
                SigOut = 0;
                _next_state = S_0;
              end
            end
          end
        default :
          begin
            SigOut = 0;
          end
      endcase
    end


  function [BITSIZE_in1-1:0] findSig;
    input [BITSIZE_in1*PORTSIZE_in1-1:0] m;
    reg [BITSIZE_in1-1:0] res;
    reg found;
    integer i1;
  begin
    res={BITSIZE_in1{1'b0}};
    found = 0;

    for(i1 = 0; i1 < PORTSIZE_in1; i1 = i1 + 1)
    begin
      if(found == 0) begin
        if(m[(i1*BITSIZE_in1) +: BITSIZE_in1] != 0) begin
          res = m[(i1*BITSIZE_in1) +: BITSIZE_in1];
          res[0:0] = 1;
          found = 1;
        end
      end
    end
    findSig = res;
  end
  endfunction

endmodule
```

Listing B.7: compacting_proxy_FU

```
module compacting_proxy_FU(this, val, start, out1);
  parameter BITSIZE_this=1, BITSIZE_val=1;
  // IN
  input [BITSIZE_this-1:0] this;
  input start;
  input [BITSIZE_val-1:0] val;
  // OUT
  output [(BITSIZE_val + BITSIZE_this + 1)+(-1):0] out1; // +1 Ã¨
   per start


  assign out1 = {this, val, start};

endmodule
```

Listing B.8: unzip_proxy_FU

```
module unzip_proxy_FU(start, val, this, in1);
  parameter BITSIZE_this=1, BITSIZE_val=1;
  // IN
  input [(BITSIZE_this + BITSIZE_val + 1)+(-1):0] in1; // +1 Ã¨ per
    start
  // OUT
  output [BITSIZE_this-1:0] this;
  output start;
  output [BITSIZE_val-1:0] val;


  assign this = in1[(BITSIZE_this + BITSIZE_val + 1) +(-1) : (
   BITSIZE_val + 1)];

  assign val = in1[(BITSIZE_val + 1) +(-1) : 1];

  assign start = in1[0:0];

endmodule
```

Bibliography

[1] P. Coussy, D. D. Gajski, M. Meredith, A. Takach, "An Introduction to High-Level Synthesis" in *IEEE Design & Test of Computers*, v. 26, pp. 8–17, August 2009.

[2] "C++11 Multithreading - Part 8: std::future , std::promise and Returning values from Thread" https://thispointer.com//c11-multithreading-part-8-stdfuture-stdpromise-and-returning-values-from-thread/.

[3] "cppreference, std::promise" https://en.cppreference.com/w/cpp/thread/promise.

[4] Wikipedia, "Logic synthesis" https://en.wikipedia.org/wiki/Logic_synthesis, Last access on 15 April 2020.

[5] D. Gajski, L. Ramachandran, "Introduction to high-level synthesis" in *IEEE Design & Test of Computers*, v. 11, pp. 44–54, 1994.

[6] S. Ravi, M. Joseph, "Open source HLS tools: A stepping stone for modern electronic CAD" in *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, 2016.

[7] W. Meeus, K. V. Beeck, T. Goedemé, J. Meel, D. Stroobandt, "An overview of today's high-level synthesis tools" in *Design Automation for Embedded Systems*, v. 16, pp. 31–51, September 2012.

[8] "MyHDL, from Python to Silicon!" http://www.myhdl.org/, Last access on 19 April 2020.

[9] "Kiwi Scientific Acceleration using FPGA" https://www.cl.cam.ac.uk/~djg11/kiwi/, Last access on 17 April 2020.

[10] Xilinx, "Introduction to FPGA Design with Vivado High-Level Synthesis" http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf.

[11] "Futures and Promises" http://dist-prog-book.com/chapter/2/futures.html\#brief-history.

[12] "Tutorialspoint, C++ Inheritance" https://www.tutorialspoint.com/cplusplus/cpp_inheritance.

[13] "Panda: a framework for hardware-software co-design of embedded systems" https://panda.dei.polimi.it/.

[14] M. Minutoli, V. G. Castellana, A. Tumeo, F. Ferrandi, "Inter-procedural resource sharing in High Level Synthesis through function proxies" in *25th International Conference on Field Programmable Logic and Applications*, pp. 1–8, 2015.

[15] "Autotools FAQ" https://www.gnu.org/software/automake/faq/autotools-faq.html.

[16] "PandA-bambu fir_filter" https://github.com/ferrandi/PandA-bambu/tree/master/examples/cpp_examples/fir_filter.

[17] "Vivado Design Suite - HLx Editions" https://www.xilinx.com/products/design-tools/vivado.html.