POLITECNICO DI MILANO

Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

AN ARCHITECTURE FOR FUNCTION POINTERS AND NON-INLINED FUNCTION CALL IN HIGH LEVEL SYNTHESIS

Relatore: Prof. Fabrizio Ferrandi

Tesi di Laurea di:
Marco Minutoli, matr. 736023

ANNO ACCADEMICO 2012-2013

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

Modern embedded systems design increasingly couple hardware accelerators and processing units to obtain the right balance between energy efficiency and platform specialization. In this context High Level Synthesis (HLS) design flows can significantly reduce the effort needed to design these architectures. This thesis introduces a new HLS methodology that supports the synthesis of tightly coupled clusters of accelerators suitable to be used as building blocks for architecture design with the introduction of PorkSoC, a templated System on Chip (SoC) suitable for automatic generation. The proposed methodology introduces the concept of non-inlined function call and of function pointers to the traditional HLS methodology to synthesize cluster of accelerators able to call other accelerators to perform their computation without the mediation of a General Purpose Processor (GPP).

1 INTRODUCTION

This chapter is an introduction to the work done. It starts with a brief premise on the motivation of this thesis. Then it continue describing the problem addressed and giving a brief description of the proposed solutions. Finally it presents briefly the results obtained. Last section of the chapter overviews the structure of the thesis.

1.1 PREMISE

Modern embedded system design is increasingly exploiting architecture heterogeneity to build faster and more power efficient systems. High-end products like smartphones and tablets typically include in their architecture hardware accelerators. The design of this kind of systems is generally complex. The design process, testing and validation of these architectures require big efforts from designers.

The adoption of High Level Synthesis (HLS) design flows can help to lower the complexity and the cost of the design of such architectures. For this reason, the current generation of HLS tools is constantly gaining market shares supported by the appealing features of recent tools.

Recently third party companies, such as Berkeley Design Technology Inc. (BDTI) [BTD], have started offering tool certification programs for the evaluation of HLS tools in terms of quality of results and usability. For example, BDTI has evaluated the usability and the quality of results offered by AutoPilot in synthesizing accelerators for Digital Signal Processing (DSP) applications. The comparison has been made against mainstream DSP processors. Results have show that accelerators synthesized by the HLS tool have $40X$ better performance of the DSP implementation. Moreover, the Field Programmable Gate Array (FPGA) resources utilization

obtained with the HLS tool are comparable with hand-written Register Transfer Level (RTL) designs.

In this context, this thesis extend the current HLS methodologies proposing an architecture supporting the synthesis of accelerators using a new non-inlined call mechanism, the introduction of function pointers and a System on Chip (SoC) designed to be generated inside a hybrid HLS flow.

1.2 THE PROBLEM

The usual approach in HLS is to translate each function in a module implementing the corresponding Finite State Machine with datapath (FSMD) [GR94]. When the FSMD reach a function call, it transfer the control to an instance of the corresponding module and wait for the computed result. This approach implies that a distinct module instance is needed for each called function inside the data-path of its caller. In some sense this approach is similar to function inlining in compilers. The advantage of this approach is that each function call has no cost in terms of performance at the expense of duplication and therefore area consumption.

This implementation strategy has another non negligible drawback. In this model a function is seen as mere computational object that taking input values from input ports produces a results on the output port. This approach does not allow to associate addresses to the synthesized function and this makes impossible to implement language features like function pointers and everything that relies on it.

This work proposes a methodology that improves the usual implemented model defining addresses for function and implementing a new call mechanics that support this model and improves resource sharing trading some clock cycles.

1.3 PROPOSED SOLUTIONS

The proposed methodology takes inspiration from the design pattern of memory mapped peripherals widely adopted in the computer architecture world. This architectural solution consists of a set memory mapped registers used to configure the behavior of the unit, to pass inputs and retrieve outputs. This architectural pattern carries the advantages of clearly defining the interface between software and hardware layers without imposing complexities on the programming model.

The implemented solution translates function inputs and the return value as memory mapped register plus a control register used to start the computation. This strategy is not followed globally but only applied to marked function in the behavioral description leaving freedom to designers to easily explore different solutions. As a final step the generated accelerators are connected through a shared bus that can be easily connected to a more complex architecture.

This methodology has been implemented inside Bambu which is the HLS tool of the PandA framework [1]. All the generated accelerators and the final architecture are Wishbone B4 (WB4) [Ope10] compliant which is a widely adopted bus communication protocol inside the open hardware community.

Integration in a more complex architecture has been tested inside PorkSoC. PorkSoC is a minimalistic SoC, based on the OpenRisc Or1k processor, that has been developed during this work.

1.4 RESULTS

The evaluation of the performance obtained with the architecture defined by the proposed methodology has been performed using the study case of chapter 7. In the considered study case, results show that the proposed methodology allows to obtain area savings of around the 15% of the logic elements contained in the targeted FPGA over the design synthesized using the traditional methodology. The area savings obtained come at the cost of the overhead of the non-inlined function call mechanism introduced by the proposed methodology. The overhead of the non-inlined call mechanism can be divided in two components. The first component is due to the new parameter passing mechanism and it is proportional to the number of the function parameters to be transferred. The second constant component is due to the introduced notification mechanism. The final result is that the propose methodology trades off the introduction of the overhead of the non-inlined call mechanism to obtain a smaller architecture. This approach can make the difference synthesizing big accelerators with complex call graphs. In these cases the difference can be between faster designs that does not fit in the targeted FPGA and slower one that does fit in it.

A more detailed discussion of the results obtained can be found in chapter 7.

1.5 STRUCTURE OF THE THESIS

Following chapters describe in more details the proposed methodology and give a detailed overview of the implemented solutions.

chapter 2 contains a presentation of the current state of the art.

chapter 3 presents the proposed methodology and gives insights about the motivation of the step followed during the design of the presented solution.

chapter 4 describe the design of the bus interface built around accelerators synthesized by the PandA framework.

chapter 5 goes into the detail of the implementation of non-inlined function call and function pointers.

1 More information can be found at http://panda.dei.polimi.it

chapter 6 presents PorkSoC, the SoC that has has been designed for automatic generation. PorkSoc has also served as testing environment for the integration of the synthesized accelerators inside more complex architectures.

chapter 7 presents simulation and synthesis results.

chapter 8 concludes the thesis with final discussion on the obtained results and gives an overview of the possible future developments.

2

STATE OF THE ART

This chapter starts giving an introduction to HLS. Then it continues giving an overview of the tools implementing HLS flows. Finally, it presents the state of the art of the target architecture and pointer synthesis.

2.1 INTRODUCTION TO HIGH LEVEL SYNTHESIS

HLS investigates techniques and methodologies that translate algorithmic descriptions given in a high level language to a hardware micro-architecture description implementing the input algorithm. In a typical HLS flow, the source language is a restricted dialect of a high level programming language (e.g. C/C++ Mathlab) used to capture the *behavioral description* of the design. The output of the flow will be in a Hardware Description Language (HDL) (e.g. Verilog, VHDL) capturing the RTL description of the resulting micro-architecture.

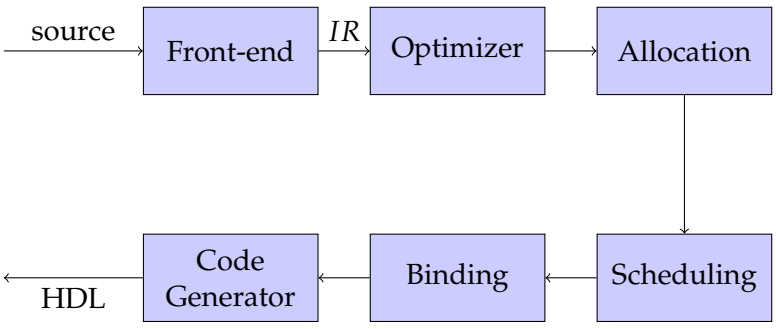


Figure 2.1: HLS flow

The architecture of a HLS tool follows the usual organization of a compiler tool. The first component of the chain is the *front-end*. Front-ends in compiler technology have the responsibility of performing lexical and syntactical analysis. The output produced by the front-end is an intermediate representation (IR) used as as input language by the compiler optimizer. The optimizer performs program analysis and transformation on the IR with the aim of producing a semantically equivalent but improved IR. In this context all code optimization algorithm known from compiler literature can be used. After the optimizer there is the HLS back-end.

The back-end contains several steps that transforms the optimized IR into an RTL representation. This representation is enriched with detailed implementation information needed for the generation of a FSMD in a hardware description language. The process of generating the FSMD includes several steps but of the whole process three can be considered critical: allocation, scheduling and binding.

Allocation selects functional units to be used to performs the operation contained into the IR. Usually synthesis tools have a library of components to choose from. The library usually contains multiple modules that can perform an operation of the IR with different area and timing. The allocation process is generally constrained by some performance goal of the design.

Scheduling maps each operation of the IR to the clock cycle when it will be executed. In HLS this is called *control step*. It is important to notice that more than one operation can be mapped to the same control step as far as data dependencies are met. This can be a challenging task especially when memory operation are involved. Another constraint that has to be taken into account is the resource availability determined during the allocation step.

Binding maps operation to a functional unit and values computed to storage resources. The mapping is computed with the objective of minimizing the number of needed hardware resources trying to share them whenever possible.

It is important to note that those three steps are dependent to each other. A consequence of this tight relation is that an optimal solution of one of those step can lead to a sub-optimal one of the others. This is known as the *phase ordering problem*.

The last block of the tool chain is the code generator. This will produce the Verilog or VHDL source to be synthesized. As said it is usually in a FSMD form that is obtained by the previous step. The datapath is obtained by the allocation and binding decisions. The controller is obtained by the scheduling and binding decisions.

This concludes the brief description of the HLS flow. A more deep and detailed discussion of problem and solution can be found in [WCC09].

2.2 PANDA AND HIGH LEVEL SYNTHESIS

PandA is a framework developed as research project at Politecnico di Milano. Its primary goal is to provide an environment to experiment with new ideas and methodologies in the hardware-software co-design field.

The PandA framework hosts a set of tools, named after wild beasts and vegetation, each covering a piece of the proposed work-flow. Following this convention the tool implementing HLS is named Bambu.

Bambu receives as input a behavioral description of an algorithm expressed in the C language and generates a Verilog[1] RTL description as output. The HDL description follows best-practices for FPGA design and is vendor independent. This design choice has permitted to easily integrate back-ends for RTL synthesis tools for Altera, Lattice and Xilinx FPGAs.

Bambu implements the HLS flow as described in the overview of section 2.1. The following subsections give a more detailed description of how the flow has been implemented inside Bambu.

2.2.1 *Frontend analysis*

Bambu relies on the GNU Compiler Collection (GCC) for the early stages in the compiler pipeline. The source language is parsed and translated in a Static Single Assignment (SSA) IR called GIMPLE. GIMPLE is the IR used by GCC to perform optimization in the compiler middle-end. The result of the middle-end optimization inside GCC is fed as input to the front-end of Bambu for additional analysis and transformation passes specific to the HLS flow. For example the tool computes in this stage Control Flow Graph (CFG), Data Flow Graph (DFG) and Program Dependency Graph (PDG) that will be used in following stages to extract parallelism from the sequential description.

2.2.2 *Resource allocation*

The resource allocation step maps operation contained in the IR to Functional Units (FUs) contained in the resource library. The mapping is done taking into account type and size of operands. The library can contain multiple FUs with different timing and area usage that can be allocated for a given operation. In this case the tool takes a decision based on the constraints and objectives given by the designer.

1 Bambu generates also functional units in VHDL when floating point operations are involved. The generation of floating point units is performed using the flopoco library.

2.2.3 *Scheduling*

The objective of this step is to build a scheduling of the operation that satisfy the given constraints (e.g. timing constraints and resource availability). The scheduling algorithm implemented inside the tool is a priority list-based one. In priority list-based algorithms, operations have priorities used to decide the scheduling. The algorithm proceeds iteratively mapping each operation to a control step of the final FSMD.

2.2.4 *Module binding*

The module binding step associates physical FUs to operations. The objective of this step is to maximize resource sharing among operations taking into account that concurrent operations need different FUs. Bambu solve this problem reducing it to a weighted graph coloring problem. In the implemented formulation weights represent the convenience of sharing the same FU for two given operations. Weights are computed taking into account timing and area characteristics of each FU.

2.2.5 *Register binding*

This step associates a physical register to each storage value contained in IR operations starting from the information given by the Liveness Analysis. Again the problem is reduced to a graph coloring problem on the conflict graph representing liveness overlapping between SSA temporaries. Must be noticed that the same problem is solved inside the back-end of a software compiler with the additional constraint that the number of available registers is bounded by the number of registers of the target architecture. HLS does not have this constraint because tools can add additional registers to the architecture when necessary.

2.2.6 *Interconnection*

This step starts from the work done by previous steps and builds the logic needed to interconnect resources. It introduces all the required logic to implement resource sharing and it binds control signals that will be set by the controller.

2.2.7 *Memory architecture*

The step of memory allocation takes care of where each variable is allocated. With respect to each function a variable can be allocated internally to the data-path or externally. Internally allocated memories are accessed directly through the

data-path while externally allocated memories are accessed through a memory interface.

The memory interface is composed of two chains, one for masters and one for slaves. The chain is closed by a global external interface that determines if the address accessed is internal or external to the core. A more detailed description of the memory architecture can be found in [PFS11].

2.3 HIGH LEVEL SYNTHESIS TOOLS

One of the most important feature of the current generation of HLS tools is the adoption of C-like (i.e. C, C++ and SystemC) specification as input of the synthesis flow. The introduction of support of C-like specification has contributed to increase the HLS user base. In fact, a C based specification language extends the range of potential user to designers more familiar with programming languages than with HDLs. It also facilitates complex tasks like HW/SW codesign, HW/SW partitioning and design space exploration. It allows to use compiler technologies to performs design optimization. For this reason several HLS flows have been implemented on top of compiler tools (i.e. Bambu, LegUp and AutoPilot).

Another important key aspect that has influenced positively the evolution and adoption of HLS flow is the rise of FPGA devices. In the last years, FPGAs have improved in capacitance, speed and power efficiency. For this reason many HLS tools have been designed specifically targeting FPGAs, i.e. LegUp [Can+11], GAUT [CM08], ROCCC [Guo+08], SPARK [Gup+04] and Trident [TGP07] [Tri+05].

The past generations of HLS tools have failed to gain significant market shares despite the interest of the industries and research communities. A good survey of the history and the reasons of the failures of HLS technology of the past can be found in [MS09] and [CCF03].

The current generation of HLS is instead increasingly gaining success in the industry world mostly because the current generation has:

- better support of the C based input language
- better quality of results
- motivated the adoption of the new design methodologies in the industry world with its better results and usability.

Most widespread HLS tools include AutoESL's AutoPilot (after its acquisition from Xilinx is known as Vivado HLS), Cadence's C-to-Silicon Compiler [Cad08], Forte's Cynthesizer [CM08], Mentor's Catapult C [Bol08], NEC's Cyber Workbench [CM08], and Synopsys Synphony C [Syn] (formerly, Synfora's PICO Express, originated from a long-range research effort in HP Labs [Kat+02]).

2.3.1 *Bambu*

The current generation HLS tools accept as input a dialect of the C/C++ language that include only a subset of the language features. For example common restrictions are on the usage of pointers and some commercial tool requires that every pointer must be statically resolved at compile time. Bambu tries to not impose such restriction on the designer by generating architectures capable of dealing with language construct that other tools simply consider illegal. In this spirit this work add the last missing feature to cover the whole C language: function pointers.

Another language feature relevant to this work is the concept of function and its translation to hardware modules. The HLS flow, as implemented inside Bambu, translates each function in a module that takes function parameters as input and eventually produces one output at the end of the computation. The computation inside the function module is implemented by the means of a FSMD. Function calls are translated instantiating the called function module inside the data-path of the caller. When a function is called multiple times inside the body of a function it is possible to share the instantiated function module in order to save area. If the same function is called inside the body of another function an additional instance of the callee is needed and it will be included into the data-path of the caller. This mechanism is similar to function inlining in software compilers with the difference that the inlined function can be shared inside a hardware module, if called multiple times. This is what can be considered as the state-of-the-art approach to function calls translations and it is the strategy used by every tool.

2.3.2 *Legup*

LegUp [Can+11] is an open source HLS tool being developed at University of Toronto. As stated on the homepage of the project[2] its long-term goal is to make FPGA programming easier for software developer. Moreover this project is traditionally used by PandA as a term of comparison.

LegUp is similar to Bambu in terms of functionality and proposed methodology. The first notable difference between the two tools is that LegUp uses the LLVM infrastructure to build the HLS flow. But this is not the only difference.

At the architectural level the distinguishing difference is the memory architecture. While both tools have the concept of local memory and external memory, they differ radically in the architecture and decoding mechanism. Particularly the key difference is that LegUp reserves the higher bit of the address to store a tag that is uniquely bound to each memory allocated into the architecture.

Another notable difference between the two tools is that LegUp offers what they call *hybrid flow*. The *hybrid flow* synthesize an architecture that contains a TigerMIPS

2 http://legup.eecg.utoronto.ca

processor with a set of accelerators built from marked functions and offers to designers an easy way to test HW/SW partitioning. The PandA framework does not implements currently an equivalent of the Legup hybrid flow.

A noticeable missing feature of the tool is the lack of support for floating point operations.

2.3.3 *Vivado HLS*

One of the tools that can be considered state-of-the-art in the industry world is Vivado HLS by Xilinx. This section presents some information of the HLS methodology extracted from the documentation of the product.

Xilinx has released a short manual [Xil13] with the goal of introducing FPGA design with their HLS synthesis tool, Vivado HLS, to software engineers. The document explains general concepts of hardware design and covers feature and limitation of the implemented HLS flow. Unfortunately, it does not give any insight of the implementation details.

Two of the concepts presented inside the document are of interest to our discussion: functions and pointers.

The description given about function is quite vague but it is clear that inside the framework their treatment is similar to the one implemented for loops. It is clearly stated that for each function it is created an independent hardware module. Those modules can be instantiated multiple times and can execute in parallel.

It also presents the memory allocation policy of the tool warning the reader that any dynamic allocation mechanism is not supported. Moreover, it explains that pointers can be used for two types of memory access: internal and external.

Pointers declared inside the body of a function are permitted when pointing internally allocated objects. Constraints are sightly different when the pointer is declared in a function parameter. In fact, any pointer access on function parameters must imply an internal allocated object or an external memory that will be accessed through an external bus protocol such as the Advanced eXtensible Interface (AXI). Moreover, the tool imposes that any pointer accessing internal objects must be statically resolvable at compile time. The following quote from [Xil13] clarifies the previous statement of what is considered to be legal by Vivado HLS in terms of pointers to internally allocated objects.

> The HLS compiler supports pointer usage that can be completely analyzed at compile time. An analyzable pointer usage is a usage that can be fully expressed and computed in a pen and paper computation without the need for runtime information.

2.4 TARGET ARCHITECTURE

Modern systems design is increasingly moving to heterogeneous architecture to exploit the performance and the energy efficiency of such systems. The design of such large systems has exposed the limitations of traditional system architectures favoring the emergence of new architectural solutions. For example, architecture composed with hundreds of simple cores connected together have exposed scalability issue in the interconnection of such a number of cores. In order to address this problem, many architecture have defined tightly coupled clusters of accelerators and used them as building blocks [Mel+12] [Bur+13] [Plu] [Cor].

There are essentially two different approaches in defining the interaction between accelerators included in this kind of architecture. The first makes use of a shared memory communication mechanism. The second approach makes use of specialized instructions specifically defined to communicate with hardware accelerators.

An example of the first solution is presented in [Bur+13] and [Bur+12]. In these two works the authors defines the architecture of an heterogeneous shared memory clusters and its OpenMP based programming model. The defined clusters includes General Purpose Processors (GPPs) and hardware accelerators and a shared memory. Inside the cluster, the connection between accelerators and the cluster interconnect is mediated by the *Data Pump* module. This modules multiplexes data accesses to/from the accelerators. It has been introduced to allow to include a great number of accelerators without increasing the complexity of the interconnection. The communication between GPPs and hardware accelerators is performed by means of memory mapped registers. Other works have approached the scalability issue with the introduction of hardware modules hiding the increasing latency between the shared memories and the processing units. The work of Burgio et al. defines an architectural solution that is similar to the one proposed in this thesis. The two key difference between the proposed architecture and the one defined in [Bur+13] are the absence an equivalent of the *Data Pump* and the possibility to synthesize clusters that do not include GPPs.

An example of the second approach is the Molen architecture [Vas+04] [PBV07]. In this two works the authors define the architecture of the Molen machine and its programming paradigm. The works of Vassiliadis et al. and Panainte, Bertels, and Vassiliadis are in the context of Reconfigurable Computing (RC). Their works start analyzing the common issue of the RC programming models. Then they define a set of specialized instruction to reconfigure the programmable logic included into the design and to communicate with loaded accelerators. In particular parameter passing and result retrieval is performed by means of two specialized instructions that moves data between the GPP register file and the XREGs File. The XREGs File is a set dedicated register for input and output of hardware accelerators. Even in this case the GPP and the reconfigurable unit have access to a shared memory. The

design proposed with the Molen architecture overcome some of the limitation of the previous works using the same approach. For example, their works does not suffers of the op-code space explosion because they do not use a different specialized instruction for each accelerator that can be configured in the programmable logic included inside the design. Another important improvement is that their proposed solution does not have any limitation on the number of input and output that accelerators can have.

Both the Molen architecture and the template architecture defined in [Bur+13] include mechanism to perform accelerator call with the cooperation of a GPP. As it will be exposed in chapter 3, this thesis overcome this limitation building a mechanism that allows accelerators to make use of other accelerators to perform their computation without the need of a GPP.

2.5 SYNTHESIS OF POINTERS

The semantic of pointers, as defined by the C language, represents the address of data in memory. This definition starts from the assumption that the target architecture consists of a single contiguous memory space containing all the application data. Unfortunately this assumption is not valid in HLS and the treatment of pointers have required special attentions from designers. In fact, in hardware data may be stored in registers, memories or even wires.

Moreover, pointers can be used to read and write the data they point to but also to allocate and de-allocate memory dynamically.

The synthesis of pointers and their optimization has been analyzed in [SSDM01] and [SDM01]. These two works defines the foundations of the synthesis and optimization of pointers in the current HLS methodologies.

[SSDM01] analyzes the operations that can be performed through pointers in the C programming language and propose strategies for their synthesis. The three operation taken into account by this work are loads, stores and dynamic memory allocation and de-allocation.

For the treatment of loads and stores the authors distinguish between two situations: pointers to a single location and pointers to multiple locations. Pointers to a single location can be removed replacing loads and stores operation with equivalent assignment operation to the pointed object. In the case of pointers to multiple locations, load and store operations are replaced by a set of assignments in which the location is dynamically accessed according to the pointer value. Addresses (pointer values) are encoded using two fields: a tag and an index. The tag is used to recognize the memory storing the accessed data. The index is used as offset to retrieve the data from the memory. With this strategy, loads and stores can be removed using temporary variables and branching instructions. [SDM01]

defines analysis and transformations that can be performed in order to optimize the synthesis of loads and stores operation through pointers.

The dynamically memory allocation and de-allocation is addressed using both hardware and software memory allocators. [SSDM01] concentrates on the hardware solution giving hints on a possible software strategy. The proposed solution makes use of a hardware module implementing the malloc and free operation. The proposed module is able to allocate storage on multiple memories in parallel. As a consequence of this design choice the dynamically allocated space is partitioned into memory segments.

[SSDM01] and [SDM01] does not take into account the synthesis of function pointers. The reason behind this missing is that traditionally HLS considers functions as computational units taking inputs and producing outputs that do not have a location in memory.

This thesis extends on these works proposing a methodology that includes accelerators in a memory mapped architecture and defines mechanisms to use function pointers for accelerator invocation.

2.6 FUNCTION CALLS

The are two traditional synthesis methods to performs function calls: inlining as performed by a software compiler and the inlining as performed HLS tools.

The inlining method used by software compilers replaces function call with the body of the called functions. In HLS, it has the advantage of keeping the datapath small because it exposes more resource sharing opportunities during the synthesis. In fact, performing the inlining of all the functions of a design gives the possibility to share resources between functions boundaries. The inlining has also the advantage of removing any kind of performance degradation caused by module communication. Unfortunately, the inlining approach increases the area and the delay of the controller due to the explosion of its number of states.

The inlining method used by HLS tools instead produces a different module for each function contained in the design. The synthesized modules are then included inside the datapath of its callers. This approach as the advantage of synthesizing smaller function modules. This is reflected in the size the function module controller that is smaller and faster than in the previous case. In the rest of this thesis this is the approach referenced as inlined call mechanism.

[Har+06] starts from this premises and defines a mechanism of function merging to be associated to the inlining mechanism of HLS tools. Function merging allows the synthesized module to increase the resource sharing inside the synthesized module without incurring in the drawback of the inlining mechanism. The work of Hara et al. proposes an Integer Linear Programming (ILP) formulation to decide the merging strategy.

The work of Hara et al. proposes a solution to increase resource sharing at fine grain level. The methodology proposed in this thesis instead proposes a coarse grain approach that can be supported by the methodology proposed in [Har+06].

3 METHODOLOGIES

This chapter presents the methodology followed during the design and the implementation of the proposed solutions. It starts giving an overview of the overall methodology. Then it defines all the concepts introduced by methodology.

3.1 METHODOLOGY INTRODUCTION

The purpose of the methodology proposed by this thesis is to introduce the concept of non-inlined function calls and function pointers to the framework of HLS. As introduced in section 1.2, HLS tools translate functions to FSMD. With the current methodologies, function calls are translated instantiating the module implementing the called function into the data-path of the caller. The implication of this methodology is that the module of a function is instantiated into the data-path of its callers. This approach limits the resource sharing of function modules to the possibility of multiple calls from the same caller. This thesis proposes a new methodology that extends the current approach with the possibility of inter-procedural sharing of function modules. The final result of the new methodology will be an architecture interconnected through a bus. The accelerators composing the architecture will be able to call other accelerators using the mechanism defined by the methodology. Figure 3.1 shows a block diagram of the final architecture containing two accelerators.

In order to obtain the desired result, this thesis defines a suitable interface for the accelerators, a call mechanism and a notification mechanism. The definition of this concepts suggests the introduction of function pointers and their architectural meaning in the context of HLS. The following sections introduce the newly defined entities and mechanisms.

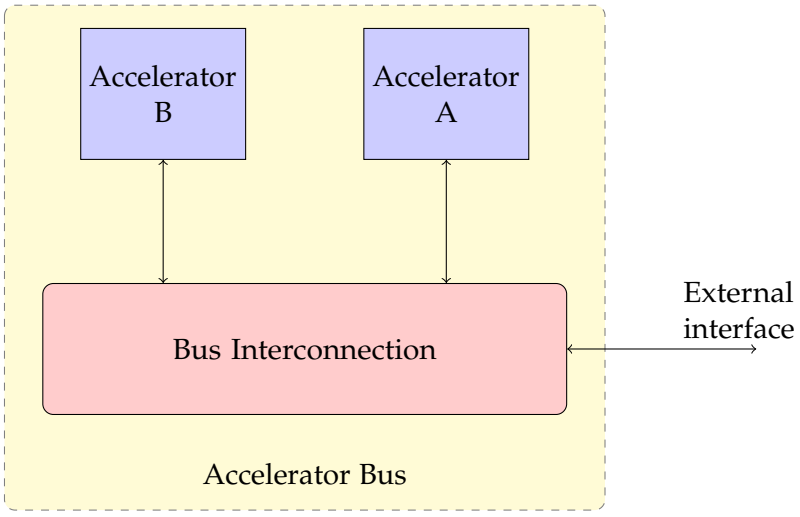


Figure 3.1: Block diagram of the proposed architecture

3.2 ACCELERATOR INTERFACE

The first entity that must be defined is the accelerator interface. It is the fundamental building block underpinning other concepts defined by the introduced methodology. Its definition must fulfill two requirements. First, the defined interface must enable to easily plug synthesized accelerator in bigger modular design. Second, it must ensures ease of use from hardware and software components. These characteristics will enable to integrate the proposed methodology in more complex hardware/software co-design flows.

The architectural solution proposed for the accelerator interface definition is the usage of memory mapped registers. This is a well established architectural design pattern in the industry world. It is used for example in computers to access the video card and other peripherals. The pattern is extremely simple. The interface exposes a pool of registers that are mapped into the address space of the architecture. The exposed registers can be of three kind: input, output and control registers. Input and output register are used to pass and retrieve data through the interface. Control registers are used to set operating modes.

Following the pattern just described, the proposed interfaces includes a set of input registers, an output register and a control register. The set of input registers stores input values necessary to the accelerator. The output register stores the return value of the computation performed by the accelerator. The control register allows to start the computation and to monitor its state.

To better explain the proposed solution lets look at how the behavioral description of Listing 3.1 translates to the memory mapped interface of an accelerator.

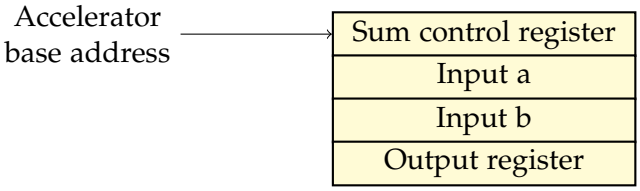


Figure 3.2: Memory mapped interface of function sum

```
int sum(int a, int b)
{
  return a + b;
}
```

Listing 3.1: Example behavioral description

The behavioral description of the accelerator `sum` in Listing 3.1 gives all the needed information to define its interface. The function `sum` takes as input two integer values and produces as output another integer value. This information translates in a memory mapped interface that contains four registers. The first register is the control register. Its address will be associated to the function. It will be the first thing allocated inside the address space of the accelerator. The second and third registers are used to store the two inputs of the function. The last one is return value. It will store the result at the end of the computation. Input and output registers are optional because functions may not need inputs and may not have return values.

Traditionally HLS tools translates function input and output as module ports. In this case the interface may be implemented as a wrapper around the synthesized accelerators. The same strategy has been adopted to integrate the methodology into the PandA framework.

3.3 WRAPPING ACCELERATORS

One of the most important design choice of the PandA framework is to be vendor agnostic in all its part. This choice was made in order to avoid problems that can arise by product discontinuity. For this reason, and in the spirit of supporting the open-hardware community, the implemented solution makes use of the WB4 bus protocol instead of other available proprietary alternatives.

The WB4 protocol can be considered the de facto standard for what concern communication protocols inside the open-hardware community. The WB4 specification has been developed and is currently maintained at OpenCores [Ope10].

The standard flow implemented in Bambu translates function to accelerators with a *minimal interface*. This interface contains a set of ports for inputs and for return

value, a set of control signal (i.e. clock, reset and start) and, when needed, a set of ports used for memory operations.

The current memory architecture is based on the work described in [PFS11] as detailed in section 4.1.

The bus interface and the communication protocol defined by the *minimal interface* for memory operations and by the WB4 specification are very similar. For this reason the accelerator WB4 interface has been implemented as a wrapper layer around the minimal interface accelerator. The wrapper logic can be decomposed in three elements: a set of registers, a controller and the interconnection logic. The set of register is used to implement the memory mapped interface described in section 3.2. The controller is a state machine activated by the control register used to start and notify the end of the computation. Wrapped accelerators may need to access the external bus or may need to connect with the internal slave chain during the computation. The interconnection logic takes care of discriminating bus access types and dynamically build the needed connection to complete the communication.

3.4 FUNCTION CALL MECHANISM

Once defined the accelerator interface, it can be defined the call mechanism using the memory mapped interface of the accelerator. The first operation that the caller must perform is passing parameters to the accelerators. In a memory mapped architecture, this step is performed by a series of write operation to the addresses of the accelerator input registers. After that, the caller can start the computation. The command that start the computation is a write operation to the control register. The caller has two options at this stage. It can write zero or a notification address inside the control register. The meaning of the notification address will be explained in subsection 3.4.1 while introducing the notification mechanism.

Following the analogy of the software function call, the described mechanism is equivalent to fill the activation record of a function and then jump to the address of its first instruction.

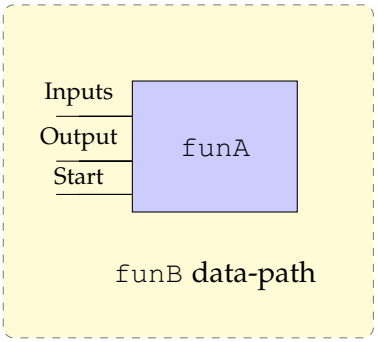
When an accelerator completes the computation, the caller get notified by a change in the state of the control register and, if activated, by the notification mechanism. The notification mechanism can be deactivated in order to implement different hardware-software interaction models (i.e. interrupt driven). The default hardware behavior always activate the notification mechanism.

At this point the caller can read the return value from the callee interface and continue its computation.

Figure 3.3 shows the inlined function call mechanism and its architectural equivalent. HLS flows tools traditionally synthesize functions as modules having input ports for function parameters and an output port for the return value. The call is

```
int funA(int a, int b)
{
  // ...
}

int funB{int a, int b)
{
  // ...
  funA(a, b);
  // ...
}
```

(a) code



(b) architecture

Figure 3.3: Synthesis of inlined call mechanism

performed asserting the start signal of the called function FSMD and keeping its input stable until the computation is complete. Using this methodology, the module of the caller function contains an instance of the module of the called accelerator.

In order to distinguish between the inlined and the non-inlined function call mechanism, the methodology defines the `hwcall` attribute. The `hwcall` attribute is used to mark the functions that are synthesized as top accelerators and that use the non-inlined function call mechanism. Calls to marked function are translated using a builtin function hiding the implementation details of the call mechanism. Listing 3.2 and Listing 3.3 show the transformation performed on the call to function sum.

```
__attribute__((hwcall))
int sum(int a, int b)
{
  return a + b;
}

int funA(int a, int b, int c)
{
  int e;

  e = sum(a, b);
  return c * e;
}
```

Listing 3.2: Before transformation

```
__attribute__((hwcall))
int sum(int a, int b) { ... }

void
__builtin_wait_call(void *, ...);

int funA(int a, int b, int c)
{
  int e;
  __builtin_wait_call(
      sum, 1, a, b, &e);
  return c * e;
}
```

Listing 3.3: After transformation

The `__builtin_wait_call` function is a var-arg function. The first argument is the address of the accelerator to be invoked. The second argument is a Boolean flag that used to distinguish if the return value of the called function must be stored or not. Subsequent arguments are the input to pass to the called function. When

```
void __builtin_wait_call(void * fun, ...)
{
  sendParameters(fun);
  startComputation(fun, CALL_SITE_ADDRESS);
  waitNotification();
  if (was_rhs_of_assignment())
  {
    readReturnValue();
    writeReturnValue();
  }
  return;
}
```

Listing 3.4: Pseudo code of the builtin function

the second parameter is one, the last argument passed to the builtin function is the address of the variable where the return value must be stored at the end of the computation.

Listing 3.4 shows in a C-like pseudo code macro operation that an hardware implementation of the `__builtin_wait_call` performs. The instantiated Intellectual Property (IP) performing the hardware call will start passing all the input parameters to the called accelerator using the architecture bus (`sendParameters(fun)`). Then it starts the computation writing in the control register of the called accelerator the notification address associated with the call site of the replaced function call (`startComputation(fun, CALL_SITE_ADDRESS)`). The call site address is an address allocated for the instruction calling the `__builtin_wait_call`. Its value is defined during the memory allocation step. After that, the IP waits the notification message declaring the end of the computation (`waitNotification()`). If the original function was in the right-hand side of an assignment (`was_rhs_of_assignment()`), the `__builtin_wait_call` will read the return value (`readReturnValue()`) from the called accelerator and it stores its value to the left-hand side (`writeResultValue`).

3.4.1 *Notification mechanism*

The introduction of non-inlined function calls carries with it the definition of a notification mechanism for computation completion.

The interface definition of section 3.2 specifies that the information of the status of the accelerator is exposed through the control register. Using this information, a simple notification mechanism could be to let the caller periodically check the control register of the callee. Unfortunately this strategy does not scale with the number of hardware callable accelerators. In fact, the periodical check of callers can congest the bus. Moreover due to bus congestion, called accelerators may be unable

to perform memory operations needed to complete the computation. In the end, the polling strategy can lead to deadlocks.

The proposed methodology solves this problem with the introduction of notification addresses. Performing the hardware call, the caller can ask to be notified by the callee when the computation is done. At the end of its computation, the callee will notify the caller performing a write operation to the notification address provided during the non-inlined function call. The notification address is associated during the memory allocation step with the call site of the function. Giving different address to each call site helps to distinguish between calls to the same function that are performed in the same caller. The allocation of the notification address does not impose the allocation of a memory or a memory mapped register.

3.4.2 *Simulation of the call mechanism*

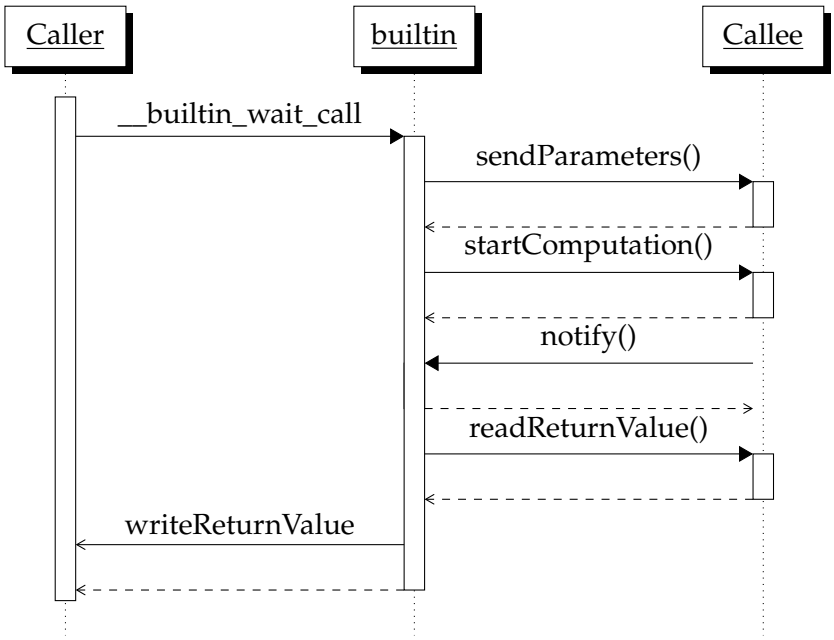


Figure 3.4: Sequence diagram of the notification mechanism

The diagram in Figure 3.4 shows the complete sequence of events happening during a non-inlined function call. When the control flow of the caller reach the call site of the original function call, it starts the execution of the IP implementing the `__builtin_wait_call`. The IP start writing parameters to the registers of the called accelerator. Then, it starts the computation of the callee writing the notification address into the callee control register. The notification address used is the address associated with the call site of builtin call. When the callee completes

```
void sort(char * vector, size_t n,
          int (*compare)(int a, int b))
{
  // ...
  for (...)
  {
    if (compare(vector[i], vector[i-1]))
    {
      // ...
    }
  }
  // ...
}

int less(int a, int b){ return a < b; }
int greater(int a, int b){ return a > b; }

int f(int a)
{
  char vec[] = { 'b', 'c', 'a' };
  if (a)
    sort(vec, 3, less);
  else
    sort(vec, 3, greater);
}
```

Listing 3.5: Function call using function pointer

its work, it send the notification message performing a write operation at the notification address stored in the control register. The builtin module then retrieve the return value and write it to the variable used by the caller to store the return value of the original call. This last step is performed only when the original call was a right-hand-side of an assignment. After that, the builtin module return the control to the caller.

3.5 FUNCTION POINTERS

The presented methodology supports function pointers. In the context of the proposed architecture, a function pointer is the first address of the memory mapped interface of the accelerator. As described during interface definition, the first address of the memory mapped interface corresponds to the address of the control register.

According to the proposed methodology, function calls through function pointers can be performed using the defined call mechanism.

Listing 3.5 shows an example using function pointers. The function `sort` use a function pointer to call a compare function passed as parameter. This call is translated injecting the `__builtin_wait_call` described in section 3.4. In this

```
void sort(char * vector, size_t n,
          int (*compare)(int a, int b))
{
  // ...
  for (...)
  {
    __builtin_wait_call(
        compare, 1, vector[i], vector[i-1], &tmp);
    if (tmp)
    {
      // ...
    }
  }
  // ...
}

int less(int a, int b){ return a < b; }
int greater(int a, int b){ return a > b; }

int f(int a)
{
  char vec[] = { 'b', 'c', 'a' };
  if (a)
    sort(vec, 3, less);
  else
    sort(vec, 3, greater);
}
```

Listing 3.6: Transformed function call using function pointer

case the, its first parameter will be the `compare` function pointer. Listing 3.6 show the result of the transformation of the call through `compare`. The difference between this case and the previous one is that the value of the compare function pointer is passed at run-time. Its value will determine the accelerator invoked inside the architecture.

3.6 TESTING ENVIRONMENT

Mechanism defined in the proposed methodology have been tested using PorkSoC as a playground. PorkSoC is a System on Chip designed to be automatically generated. The long term goal is to include it in a hybrid HLS flow. A detailed discussion of its design can be found in chapter 6.

PorkSoC has been used also to test the accelerator integration inside a more complex designs. As part of the testing, we had the opportunity to verify the interaction between accelerators and software executed by the processor included in PorkSoC.

4 WISHBONE WRAPPER

This chapter gives a detailed description of the design and implementation of the WB4 wrapper that the proposed methodology builds around accelerators synthesized by Bambu. The chapter starts giving a description of the minimal interface and the protocol used by its memory interface. Then it continues presenting the WB4 interface and the communication protocol. The chapter ends describing the details of the implementation of the wrapper circuit.

4.1 MINIMAL INTERFACE

In the PandA framework the *minimal interface* is the interface obtained following the standard synthesis flow of hardware accelerators. It has been designed to be extremely simple in order to obtain maximum performance from the synthesis flow. The architecture of the accelerators is organized in two daisy chains. One for masters components and one for slaves components. The two chains are used to perform internal and external memory operation. During external memory operation, master and slave chains of an accelerator have direct access to the minimal interface and, through it, to the outside world. During internal memory operation the two chains are closed on each others to build the communication channel.

Figure 4.1 shows a block diagram of the architecture. The accelerator `funC` contains two functional units used to perform memory operation (`LD/ST`) and two functional units to compute function `funA` and `funB`. The bus merger has the task to forward requests from internal module through the exposed interface.

The minimal interface of synthesized accelerators is dynamically generated. The generation starts from the signature of the synthesized function and from the information of the memory allocation. Its interface signals can be organized in four

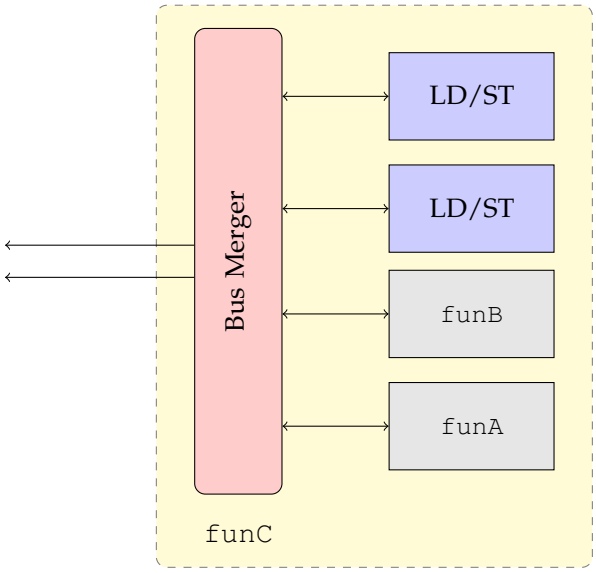


Figure 4.1: Block diagram of the minimal interface

```
int funA(int a, int b) {}

void funB(int a, int b) {}

void funC(){}
```

Listing 4.1: Example of function with different input output interface

sets: *control signals, accelerator inputs and output, master chain signals and slave chain signals*.

Of the four sets, only the set of control signals is always included in the interface. The set of control signals contains four elements: *clock, reset, start_port, done_port*. The *clock* and *reset* have the usual meaning. The other two control signals allows to start (*start_port*) and get notified (*done_port*) when the computation is done.

The other three sets are optional and they are included when needed by the synthesized accelerator. Their inclusion and their content depends from the signature of the function being synthesized and from the decision taken during the memory allocation step of the synthesis flow. Their meaning and their inclusion policy will be explained in the following subsections.

4.1.1 *Inputs and Output of the accelerator*

The first optional part of the minimal interface is the set of module ports that represents inputs and the output of the synthesized function. Listing 4.1 shows all the possible cases that can happen.

The first case is represented by the function `funA`. This function represents all the functions that take any given number of arguments and produce a return value. In this case the accelerator with the minimal interface will have one input port for each parameter and one output port, named *return_port*, that will contain the return value.

The opposite case is represented by the function `funC`. In this case the function does not need any inputs and does not produce any output. The minimal interface of the synthesized accelerator will reflect this characteristic. As a consequence, the generated minimal interface for the accelerator will not have any ports for inputs and output.

The last case represented by `funB` lays in between the two preceding. In this case the function has a list of inputs but does not have a return value. This will be reflected into the minimal interface including an input port for each function parameter but omitting the *return_port*.

4.1.2 *Master chain signals*

Master chain signals are the portion of the interface that is used to start memory operations. Their inclusion in the interface is conditioned by the kind of operation included in the synthesized function and by the decision taken during memory allocation. The need or not of these signals depends on where the memory allocation step has allocated memories that the synthesized accelerator accesses during the computation. Essentially there are two cases that make this interface needed. The first case is that the synthesized function need to access memory located higher in the hierarchy. The other case is that the accelerator may access memories outside the boundaries of the accelerator.

Master chain signals can be organized in two distinct sets. The first is composed of the following signals:

MOUT_DATA_RAM_SIZE Size of the data to be read or written by the master initiating the bus cycle. The size is transmitted as the number of bit to be read or written.

MOUT_WDATA_RAM The data to be written. This is meaningful only during a write cycle.

MOUT_ADDR_RAM The address of the data to be read or written.

MOUT_WE_RAM Asserted during write cycles otherwise set to zero.

MOUT_OE_RAM Asserted during read cycles otherwise set to zero.

For each of the previous signals exist an equivalent input signal used to build the master chain. A master making a request will use these output signals to start a bus cycle.

The other set is the return channel of the communication. It contains the information coming back from the addressed slave. It contains the following two signals:

M_DATA_RDY Acknowledge signal for the master waiting the completion of the requested read or write operation.

M_RDATA_RAM Data read from the selected address. Contains a meaningful value only at the end of a reading cycle.

4.1.3 *Slave chain signals*

Like the master chain signals, even the slave chain signals are included by the minimal interface generation step on a need basis. Again the inclusion or not depends on the decisions taken during the memory allocation step of the synthesis. They will be included when the synthesized accelerator contains memories that must be accessible through the architecture bus.

Slave chain signals can be organized in distinct sets. The first set is used to propagate the request coming from the master chain inside the slave chain. It includes the following signals:

S_DATA_RAM_SIZE Size of the data involved in the operation. The size is transmitted as the number of bit to be read or written.

S_WDATA_RAM The input data to be written in the destination address.

S_ADDR_RAM The destination address for the current operation.

S_WE_RAM Input to be asserted to perform write operation otherwise set to zero.

S_OE_RAM Input to be asserted to perform read operation otherwise set to zero.

The other set is used to send back the response to the master chain. It includes the following two signals:

SOUT_DATA_RDY Acknowledge signal for the master waiting the completion of the requested read or write operation.

SOUT_RDATA_RAM Data read from the selected address. Contains a meaningful value only at the end of a reading cycle.

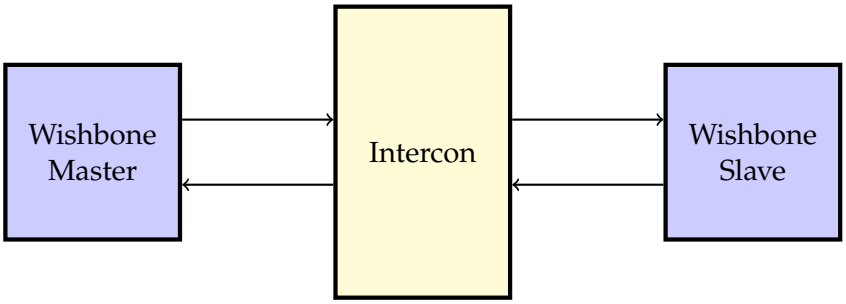


Figure 4.2: Wishbone entities interconnected

4.2 WISHBONE 4 PROTOCOL BASICS

This section presents the features of the WB4 protocol used by the implementation of the wishbone wrapper. The interested reader can find the complete specification of the interface and its bus cycles inside the wishbone specification [Ope10].

The wishbone specification is very general. Its generality lays on the absence of architectural constraints imposed by the specification other than a precise definition of the interfaces and the communication protocol.

The specification defines the three entities that are involved during the communication: the wishbone master, the intercon and the wishbone slave. Wishbone masters and slaves are object implementing the wishbone master and slave interface respectively. The intercon is a wishbone module that interconnects master and slave interfaces. Figure 4.2 gives a schematic view of the interconnection in a wishbone bus.

The specification does not prescribe any particular topology for the interconnection or any specific interconnection mechanism. The choice is left to the architecture designer but any possible interconnection mechanism is compliant with the specification. Inside the specification document there are outlined a few possible architectural solution for the implementation of the intercon including: point-to-point connection between wishbone masters and slaves, crossbar and a shared bus.

The specification precisely defines the communication protocols between wishbone masters and slaves defining *bus cycles*. According to the definition given by the specification, a *bus cycles* is the process whereby digital signals affect the transfer of data across a bus by means of an interlocked sequence of control signals. The specification defines two bus cycles: the wishbone classic bus cycle and the wishbone registered feedback bus cycle. Moreover, the wishbone classic bus cycle has two possible variants: the standard bus cycle and the pipelined bus cycle.

The wishbone specification does not impose to support every variant of the communication protocol. In order to be compliant with the specification, it is enough to support only the wishbone bus cycle chosen by the designer.

The following subsections describe the wishbone interface as implemented by the wrapper logic built around Bambu synthesized accelerators and the classic bus cycle in the standard version.

4.2.1 *The interface*

The wishbone specification defines a rich interface in order to support all the defined bus cycles. An interface is not required to implement all of them to be compliant with the specification. But at the same time the specification defines the minimal set of signals that master and slave interface must have to implement the communication protocol. The wrapper wishbone has been designed to use the wishbone classic bus cycle. Its interface exposes only the minimal set of signals necessary to implement the communication protocol.

The following subsection give the definition of the used signals following the organization of the WB4 specification. The specification organizes the signal description in three groups: signal common to master and slave, master signal and slave signal. The description will follow the same organization and the same name convention of the wishbone specification.

Signals common to master and slave

CLK_I The clock input coordinates the activity of internal logic with the interconnect. According to the specification all output signals must be registered with the clock and all the input must be stable before the rising edge.

RST_I The reset input force the interface to restart. At the same time all the internal state machines must be set to their initial state.

DAT_I The data input is used to transfer binary data.

DAT_O The data output is used to transfer binary data.

Master signals

ACK_I The acknowledge input. Its assertion indicates the normal termination of a bus cycle.

ADR_O The address output is used to pass addresses.

CYC_O The cycle output. Its assertion gives the start to the communication. The cycle output must be asserted during whole bus cycle.

SEL_O The select output indicates where valid data is expected on `dat_i` during a read operation or where valid data is put on `dat_o` during a write operation. Its size is determined by the granularity of the port.

STB_O The strobe output identifies a valid data transfer. This signal has an important role in more complex bus cycles. In our discussion it will be asserted always when `cyc_o` is.

WE_O The write enable output identifies read and write operation. This signal is asserted during write operations and negated on read operations.

Slave signals

ACK_O The acknowledge output. The slave interface assert this signal to indicate the normal termination of a bus cycle.

ADR_I The address input is used to pass addresses.

CYC_I The cycle input indicates that a valid bus cycle is in progress.

SEL_I The select input indicates where valid data is expected on `dat_o` during a read operation or where valid data is put on `dat_i` during a write operation. Its size is determined by the granularity of the port.

STB_I The strobe input when asserted indicates that the slave is selected. Slaves must respond to other wishbone signal only when this is asserted.

WE_I The write enable input identifies read and write operation. This signal is asserted during write operations and negated read operations.

4.2.2 *The wishbone classic bus cycle*

The wishbone classic bus cycle is the simplest communication method described inside the specification document. According to the specification, every wishbone bus cycle is initiated by the master interface asserting the `cyc_o` signal. When the `cyc_o` signal is not asserted all other master signal must be considered invalid. On the other side, slave interfaces respond to slave signals only when the `cyc_i` is asserted.

All the bus cycles define a handshaking protocol between master and slave interfaces. The timing diagram in Figure 4.3 shows the handshaking protocol for the standard version of the wishbone classic bus cycle with a synchronous slave from the master perspective. As shown in Figure 4.3 the master interface assert the `stb_o` signal when it is ready to transfer data to the slave. The `stb_o` signal will remain asserted until the slave interface assert the `ack_i` signal to terminate

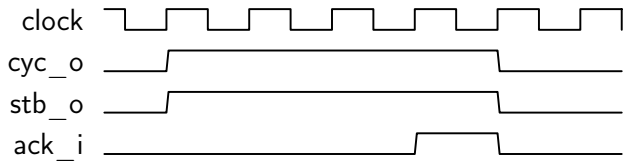


Figure 4.3: Wishbone handshake protocol

the bus cycle. On the rising edge of the `ack_i` signal, the master interface will de-assert the `stb_o` signal. This gives control over the data transfer rate to both master and slave interfaces.

The following two subsection presents the standard single write cycle and the standard single read cycle.

4.2.3 *The standard single write cycle*

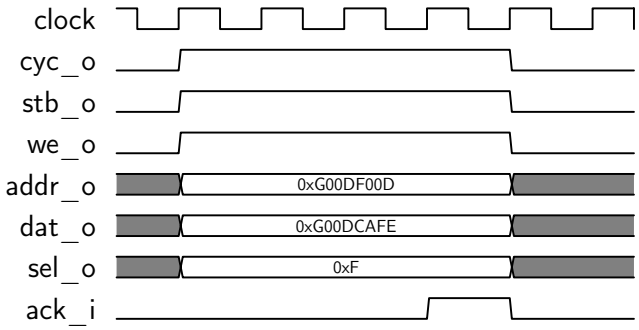


Figure 4.4: Wishbone standard write cycle

The timing diagram in Figure 4.4 shows a standard single write cycle from the master interface perspective. The bus cycle works as follow.

When a master is ready to perform a write operation it set to a valid address the `adr_o` signal and to the appropriate value the `dat_o` and `sel_o` signals. Concurrently it assert the `we_o`, `cyc_o` and `stb_o` signals. After that the master start monitoring the `ack_i` signal waiting for the slave response.

The selected slave will decode the inputs and register value of the `dat_o` signal. As soon as the operation is completed the slave asserts the `ack_i` signal to notify the master.

When the `ack_i` signal is asserted, the master de-asserts `stb_o` and `cyc_o` signals concluding the bus cycle.

4.2.4 *The standard single read cycle*

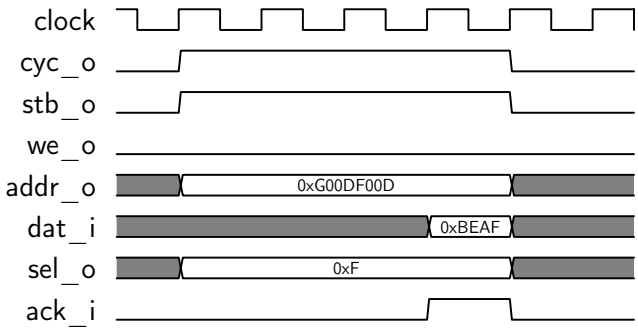


Figure 4.5: Wishbone standard read cycle

The timing diagram in Figure 4.5 shows a standard single read cycle from the master interface perspective. The bus cycle works as follow.

When a master is ready to perform a read operation it sets to a valid address the `adr_o` signal and to the appropriate value the `sel_o` signal. Concurrently it negates the `we_o` signal and asserts `cyc_o` and `stb_o` signals. After that the master start monitoring the `ack_i` signal waiting for the slave response.

The selected slave will decode the inputs coming from the master. As soon as the requested data is ready, it will output the value to the `dat_i` signal. At the same time it asserts the `ack_i` signal to notify the master it has done.

When the `ack_i` signal is asserted the master register the value of the `dat_i` signal and de-asserts `str_o` and `cyc_o` signals concluding the bus cycle.

4.3 WB4 WRAPPER CIRCUIT

The minimal interface and the WB4 interfaces have multiple common characteristics. Both protocols have dedicated lines to transmit data, data size and addresses. Both make use of an acknowledge line. The main design difference between the two communication protocol is mechanism that start bus cycles. The minimal interface bus cycle starts when the initiator assert one the line `Min_we_ram` and `Min_oe_ram`. The wishbone communication protocol makes use of two control lines `cyc_o` and `stb_o`. The wrapper circuit makes use of the similarities between the two communication protocols to build the wishbone interface on top of the minimal interface of synthesized accelerators. At the same time, it introduces connection logic and IPs designed to handle the difference between the two communication protocols.

The wrapper circuitry can be organized in three entities: a pool of memory mapped registers, the interface controller and the interface interconnection logic. In the previous chapter, section 3.2 explains how the pool of memory mapped registers

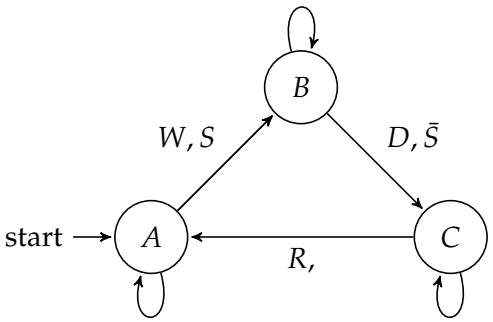


Figure 4.6: Wishbone interface controller state machine

is allocated. The following subsections explain the implementation details of the wishbone interface controller and the interconnection logic.

4.3.1 *The wishbone interface controller*

The wishbone interface controller is directly tied to the control register exposed by the memory mapped interface of the accelerator. The control register exposes to users the current operating state of the accelerator and, at the same time, acts as a locking mechanism.

Figure 4.6 shows the state machine of the controller. chapter 5 will extend the controller to build non-inlined function calls.

The controller in Figure 4.6 has three states. The starting state (A) represents that the accelerator is doing nothing an can start new a computation. An hypothetical caller can fill the memory mapped register of the input parameters with the appropriate value and then let the computation start writing in the control register. The write operation in the control register (W) makes the controller move to the second state (B) starting the computation raising the `start_port` (S) signal of the minimal interface. The controller stays in B until the accelerator asserts the `done_port` signal (D) then it moves to the third state (C) de-asserting the `start_port` signal. The controller moves to the initial state after the control register gets read. After that the accelerator is ready to start a new computation again.

4.3.2 *The interconnection logic*

As described in section 4.1, the minimal interface is generated dynamically according the needs of the synthesized function. For this reason the wishbone interface and the glue logic between the two interfaces is also generated on the need. The following paragraphs describe only the complete interconnection logic represented

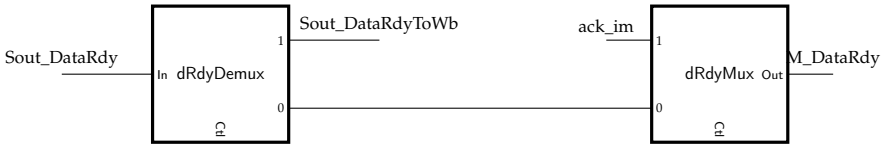


Figure 4.7: Wrapper logic for the DataRdy signal

in the circuit of Figure 4.9. The uncovered cases can be obtained removing not needed parts from the described circuit.

The circuit built around the minimal interface has two tasks. The first is to build the mapping between minimal interface signals and wishbone signals. The second is to connect master and slave chains of the minimal interface when necessary.

The `RangeChecker` is a module that constantly check if the address coming from the `Mout_addr_ram` resides in the memory space of the accelerator or outside. The `RangeChecker` has three inputs: an address and the starting and ending addresses of the internal range. The result of the check is used as control signal into the multiplexers that build the interconnections.

`DRSToS` and `SToDRS` are two modules used to convert respectively from the `Mout_data_ram_size` to the `sel_o` format and from the `sel_i` back to the `S_data_ram_size` format.

The multiplexer logic is similar for each of the interconnected signals. The thesis describes in detail only the `weMux` functionality. The behavior of the other multiplexers can be deduced from this.

The `weMux` connects the `Mout_oe_ram` signal to the corresponding `S_we_ram` signal from the master chain or the result of the or connected to port 1. As said, the selection is controlled by the `RangeChecker` module. If the destination address is internal the multiplexer will close the connection on the master chain otherwise it will select the signal coming from the wishbone slave interface. The output of the or port at input 1 of the multiplexer is asserted only during wishbone write cycles.

Figure 4.7 and Figure 4.8 present two other important aspects of the wrapper circuit.

The circuit in Figure 4.7 shows how the wrapper performs the interconnection of the acknowledge signals. The circuit is made by a cascade of a demultiplexer and a multiplexer controlled by the output of the `RangeChecker` module. The multiplexer–demultiplexer chain will close the connection between the `Sout_DataRdy` and the `M_DataRdy` signals when the destination address is internal to the accelerator. Otherwise the `Sout_DataRdy` signal will be used to respond to slave cycles from the wishbone interface while the `M_DataRdy` signal will be connected to the acknowledge signal coming from the wishbone master interface (`ack_i`).

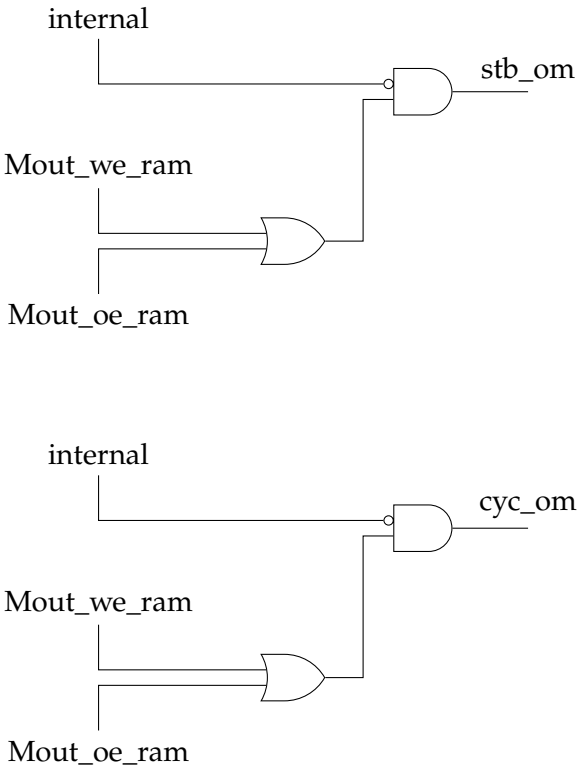


Figure 4.8: Wrapper logic for stb_om and cyc_om signals

The circuit in Figure 4.8 shows the logic that controls the `stb_o` and `cyc_o` of the wishbone master interface. The circuit reflects the fact that a wishbone bus cycle starts when the underlying master of the minimal interface starts a read or write operation on an externally allocated object.

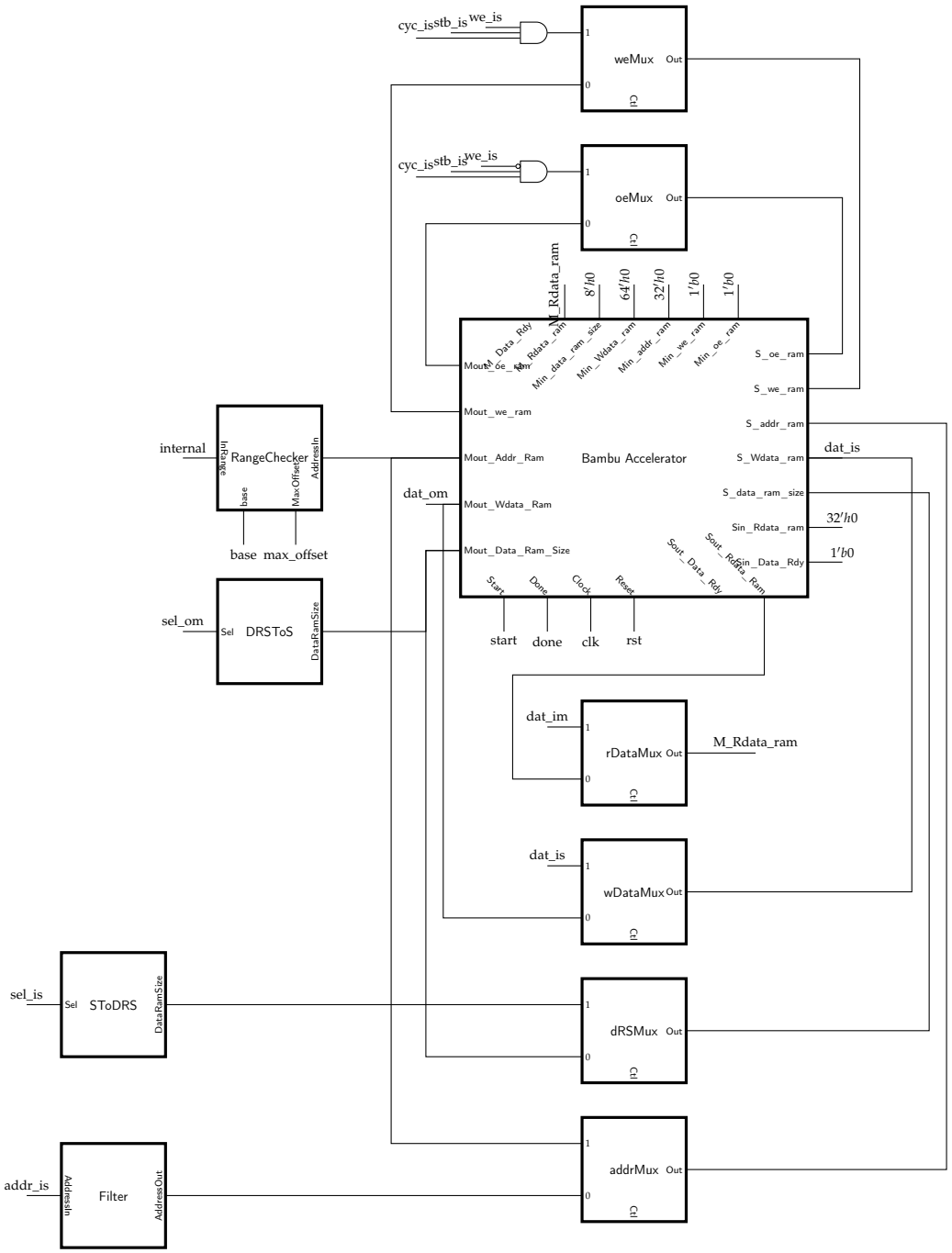


Figure 4.9: Wishbone 4 wrapper logic

5 FUNCTION CALLS AND FUNCTION POINTERS

This chapters presents the architectural solutions implemented inside the PandA framework to build non-inlined function call mechanism as described in chapter 3. The discussion starts giving an overview of the implementation of the architecture proposed by the methodology. Then it continues explaining transformation and analysis added to the HLS flow to support the methodology. The chapter ends presenting the details of the architecture generation.

5.1 ARCHITECTURAL OVERVIEW

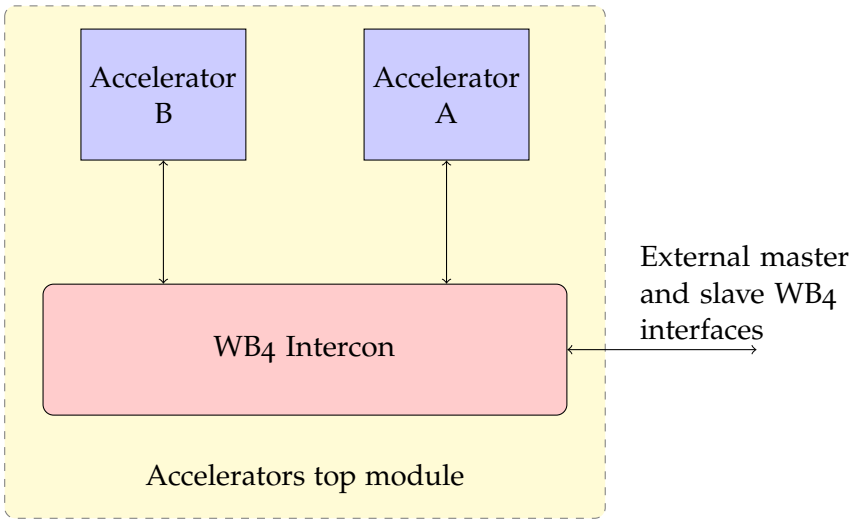


Figure 5.1: Block diagram of the implemented architecture

The architecture built of interconnected accelerators exposing memory mapped interfaces presented in section 3.1 is very general. Lots of the architectural details are unspecified leaving to the designer various degrees of freedom during its implementation. In fact, a designer approaching its implementation must define:

- the architecture interconnection
- the internal communication protocol
- the external interface
- the external communication protocol

Figure 5.1 shows the block diagram of the architecture as implemented inside the PandA framework.

Accelerators composing the architecture expose interface defined by the methodology and implemented as described in chapter 4. Accelerators are built with a WB4 interface and for this reason the internal communication protocol chosen is Wishbone B4. The whole architecture is synthesize inside a top module that can be used as a building block of more complex architecture.

5.1.1 *The wishbone interconnection*

The internal interconnection is implemented using a shared bus. The module implementing the WB4 intercon is composed by two cooperating entities: the connection logic and an arbiter. Figure 5.2 shows a block diagram of the internal structure of the wishbone interconnection.

Figure 5.2: Block diagram of the wishbone interconnection

The arbiter has the task of deciding which of the master initiating a connection takes the grant to use the shared channel. The decision of the arbiter is influenced by:

- the initiation requests coming by masters interfaces (`cyc_o` signal);
- the last granted master.

The arbitration logic always decide in favor of the last granted master when it is claiming the bus in subsequent clock cycles. This policy is implemented in order to support the pipelined version of the WB4 classic bus cycle. In case the previously granted master has released the bus, the arbiter decides in favor of first found master reclaiming the bus. The order of scanning is the order of the master interface

```
void funA(int a, int b)
{
  // ...
}

int funB(int a, int b)
{
  // ...
  funA(a, b);
  // ...
}

void funC(int a, int b)
{
  // ...
  funA(a, b);
  // ...
}
```

Listing 5.1: Example of non-inlined calls

ports. This means that master connected to interface with lower indexes have higher priority.

The decision process lasts one clock cycle. Once the decision is taken, the interconnection logic connects the granted master with its requested slave. Having to wait for one clock cycle for the arbiter, the interconnection logic has been enriched with a set of registers between master and slaves interfaces. This registers break critical paths allowing to obtain higher clock frequencies.

During architecture generation, the intercon is configured to expose two additional interfaces: a master and a slave interface. These two interfaces are used to give access to the architecture to and from the outside world. For example, the testing infrastructure uses the master interface to perform the call that start the computation. At the same time, the slave interface is used to connect a memory controller implementing a RAM to the design under test. The two additional ports can be used to connect the generated architecture to an higher level intercon. This allows to build a hierarchical architecture made of tightly-coupled shared-memory clusters as described in [Bur+13].

5.2 NON-INLINED FUNCTION CALLS

The function call mechanism described in section 3.4 introduces the non-inlined function calls to the HLS methodology. Non-inlined function calls can be used to reduce the number of instantiated modules inside the synthesized architecture with the advantage of reducing the area occupation inside the FPGA.

Listing 5.1 shows a situation where this strategy can be applied. Lets suppose that the final architecture must contain accelerators computing `funA`, `funB` and `funC`. A design synthesized using the current HLS methodology will contain three instances of the module implementing `funA`. The first instance is needed to directly compute `funA`. The other two instances are included inside the data-path of the modules that will compute `funB` and `funC`.

The proposed methodology offers the possibility to synthesize a design containing only one instance of the module implementing `funA`. Using this methodology, `funB` and `funC` will perform the function calls to `funA` using the mechanism defined in section 3.4.

In the context of PandA, the implementation of non-inlined function calls is built starting from three passes.

The first pass is implemented inside a GCC plugin. The plugin containing an attribute definition and a transformation pass. The attribute defines the concept of hardware call. The transformation pass inject a builtin function that implements the non-inlined function call mechanism.

The second component is an analysis pass implemented inside the front-end of Bambu. The transformation pass performed by the plugin hides function calls behind a builtin function. The front-end analysis performed augment the call graph adding the missing edges disappeared after the transformation.

The third component is the architecture generation step in the HLS back-end. The implementation generate a WB4 interconnection suitable to connect all the accelerators and to give access to/from the outside world.

The following sections present the details of their design and implementation.

5.3 SOURCE CODE TRANSFORMATION

PandA implementation of the non-inlined function calls mechanism makes use of a GCC plugin that defines an attribute and include a transformation pass. The attribute defined is called `hwcall`. This attribute is used to mark explicitly function that are synthesized as hardware accelerators. Marking a function definition or a function declaration with the attribute triggers the transformation pass.

Listing 5.2 and Listing 5.3 show the effect of the transformation. To better understand the transformation performed by the pass, lets see how Listing 5.2 is translated into Listing 5.3. The transformation search for function calls to function marked with the `hwcall` attribute. When it finds one, it replaces the original call with a call to the `__builtin_wait_call` builtin function. The injected builtin function is a var-arg function hiding the implementation of the non-inlined function call mechanism. The first argument of the `__builtin_wait_call` is a function pointer pointing to the function to be called. The second argument is a Boolean flag. It is used to discriminate the meaning of the last parameter passed to the builtin

```
__attribute__((hwcall))
int sum(int a, int b)
{
  return a + b;
}

int funA(int a, int b, int c)
{
  int e;

  e = sum(a, b);
  return c * e;
}
```

Listing 5.2: Before transformation

```
__attribute__((hwcall))
int sum(int a, int b) { ... }

void
__builtin_wait_call(void *, ...);

int funA(int a, int b, int c)
{
  int e;
  __builtin_wait_call(
      sum, 1, a, b, &e);
  return c * e;
}
```

Listing 5.3: After transformation

function. There are two cases: the original function call is in the right-hand-side of an assignment or not. When the original call is in the right-hand-side of an assignment, the last parameter of the builtin function is the address of the variable storing the return value of the called function. As is shown in Listing 5.3 the second parameter of the `__builtin_wait_call` wait call is one and the last parameter is the address of variable `e`. Between the second and the last parameter, the builtin function takes as input all the variable that were passed to the original function call. When the original function call is not in the right-hand-side of an assignment, the last parameter of the builtin function will be the last argument of the original call.

This implementation strategy allows to use the same transformation to implement hardware calls from hardware and software components. In fact, the transformation pass will only declare the builtin call leaving to the designer the responsibility of its implementation. For what concern the hardware implementation, the IP library of the PandA framework already contains an IP implementing the builtin. During the synthesis flow all function calls to the builtin will be translated using the implemented IP. Moreover, the designer has the freedom to change its implementation using a different IP library.

As said, the builtin function takes a variable number of arguments that are unknown until synthesis time. This implies that the IP implementing the builtin non-inlined function mechanism must be dynamically generated during the synthesis. The builtin IP generation makes use of the IP specialization process offered by the PandA framework. When the synthesis flow encounters a `__builtin_wait_call`, it invokes the specialization process. This generates a new functional units taking as input the exact number and types of arguments of the synthesized builtin call. The generated functional unit implements a state machine that performs the non-inlined function call. Figure 5.3 shows the generated state machine for a function taking an argument and returning a value.

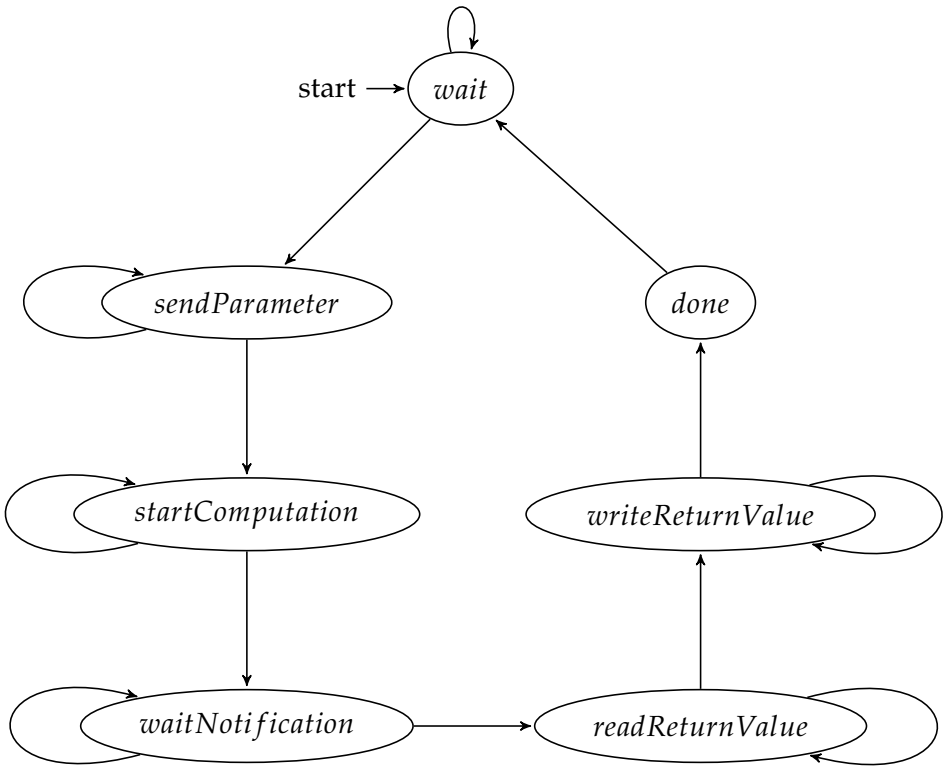


Figure 5.3: Builtin state machine of a function that takes a parameter and has a return value

The specialized `__builtin_wait_call` is also able to perform non-inlined function calls using function pointers. This functionality is implemented using relative addressing to access the accelerator memory mapped interface. This requires that the memory mapped interface of accelerators having the same types as input and outputs have the same relative addressing between registers. This property is guaranteed by the implemented memory allocation algorithm.

Listing 5.4 shows the same example of chapter 3. The first argument passed to the builtin function is the address contained in the `compare` function pointer. As introduced in section 3.5, the address associated to the function is the address of the accelerator control register that is the first element of the memory mapped interface.

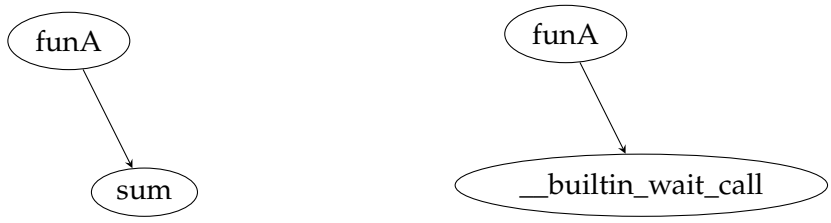
The framework is also able to generate a software implementation of the builtin function during the testbench generation. In that context, the software implementation of the builtin is used in the process of automatic generation of testbench stimuli.

```
void sort(char * vector, size_t n,
          int (*compare)(int a, int b))
{
  // ...
  for (...)
  {
    __builtin_wait_call(
        compare, 1, vector[i], vector[i-1], &tmp);
    if (tmp)
    {
      // ...
    }
  }
  // ...
}

int less(int a, int b) { return a < b; }

int f()
{
  char vec[] = { 'b', 'c', 'a' };
  sort(vec, 3, less);
}
```

Listing 5.4: Function call using function pointer

5.4 FRONT-END ANALYSIS

The transformation described in section 5.3 has the effect of changing the call graph of the original code. Due to the fact that the builtin function injected is declared but not defined, the information of the original function call is lost.

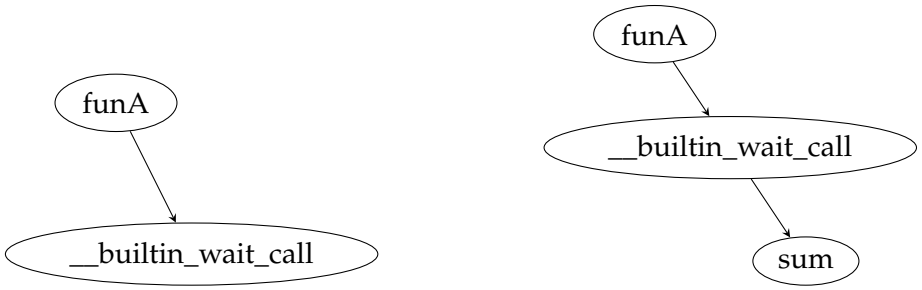


(a) before (b) after

Figure 5.4: Call graph before and after transformation

Figure 5.4 shows how the transformation modifies the call graph of the code in Listing 5.2. The missing information of `__builtin_wait_call` calling `sum` has the consequence that the HLS flow will not include `sum` during the synthesis despite its presence in the source code. The reason of this exclusion is that `sum` is not reachable navigating the call graph from the top function (`funA`).

To restore the information lost during the transformation pass, the HLS flow was enriched with an analysis pass adding the missing information to the call graph.



(a) before (b) after

Figure 5.5: Call graph before and after front-end analysis

Figure 5.5 shows how the analysis pass effects the call graph of the transformed code of Listing 5.3. The analysis search calls to the `__builtin_wait_call`. When it finds one, it adds an edge between the `__builtin_wait_call` function and the function passed as its first argument.

At the end of the analysis, all the function called through the non-inlined function call mechanism will be included again in the call graph and the HLS flow will perform their synthesis as top modules having WB4 interfaces.

5.5 ARCHITECTURE GENERATION

Once that the analysis has completed its work, the HLS tool starts the synthesis flow in order to build the architecture of Figure 5.1.

The first step of the architecture generation is the synthesis of the accelerators to be included in the final design. The architecture will contain accelerators for the top function and for functions marked with the `hwcall` attribute. All the synthesized accelerators expose a WB4 interface generated following the steps described in chapter 4. In order to support the notification mechanism, the synthesis flow of accelerator with the wishbone interface wrapper has been extended with new features.

First, accelerator wrapper has been extended to include the `notify_caller` IP. This IP implements the notification mechanism of the proposed methodology described in subsection 3.4.1. The notification mechanism is implemented by means of a little state machine performing a write operation of the wishbone classic bus cycle. The destination address used is the notification address stored in the high bits of the control register of the accelerator interface. Figure 5.6 shows the implemented state machine.

The second feature is the introduction of a new memory allocation policy.

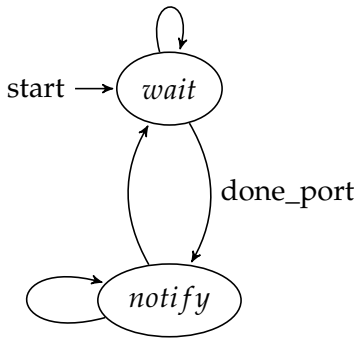


Figure 5.6: `notify_caller` state machine.

Once that the synthesis of the accelerators composing the final architecture is completed, it is performed the generation of the architecture of Figure 5.1.

The generation step starts instantiating the top module and modules of function called using the non-inlined call mechanism. Then it instantiate the WB4 intercon. The instantiated WB4 intercon contains all the needed master and slave interface to connect the accelerator composing the architecture plus a master and a slave WB4 interface. The two additional master and slave interface are used to give access to the generated architecture from/to the outside world.

The whole architecture is synthesized as a module exposing a master and a slave interface directly connected to the additional interface of the WB4 intercon. All the core included into the generated architecture are relocatable in memory using module parameters. This allows to specify the new base address of each module included. This characteristics makes the generate modules easy to use and integrate in more complex modular designs like SoC.

5.5.1 *The new memory allocation policy*

The original memory allocation algorithm implemented inside the PandA framework starts deciding the accelerator where each object must be allocated inside the architecture. The policy implemented allocates memory objects in the common ancestor of the function modules using them. After that, the memory allocation policy starts assigning addresses to each object one function at time but without following any particular order.

Unfortunately, the original implemented algorithm does not guaranties that address space contiguity when generating the architecture for non-inlined function calls. The address space contiguity is a fundamental property needed by the WB4 intercon present in the PandA IP library. For this reason, the allocation policy has been changed with a new one giving this guarantee.

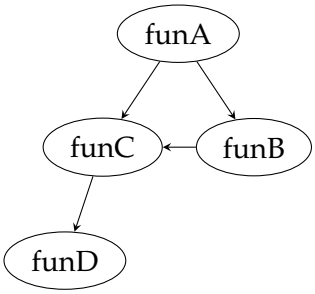


Figure 5.7: Call graph example for the allocation policy

The newly implemented policy performs a partition of the object to be allocated taking into account the accelerator that will contain them. After that it start allocating the WB4 interface registers and continues with all the object allocated inside the accelerator and its called function. This will guarantee that address spaces of the accelerators included in the final architecture will not be overlapping.

Figure 5.7 shows an example of call graph that allows to better explain the allocation policy. Lets suppose that `funA` must be synthesized as an hardware accelerator and that `funC` has been marked to use the non inlined function call mechanism. At the end of the synthesis, the generated architecture will contains two top accelerators one for `funA` and one for `funC`. At this point we can ideally partition the call graph cutting all the edges in the backward star[1] of the `funC` node. The two sub-graphs obtained represent the partitioning of function modules in top accelerators. The requirement of the interconnection logic is that all the memories that are reachable through the interface of an accelerator must be in a compact range. This implies that memories allocated inside function modules contained in a partition cannot be allocated in between memories contained inside function modules of other partitions.

The new memory allocation policy computes the call graph partitioning and subsequently perform the memory allocation one partition at time. Partitions are processed performing the allocation of memories considering on function at time starting from the top of the sub-graph. This guaranties address space contiguity.

1 Given a graph $G = (N, E)$ and a node i of the graph, the backward star of i is the set $\delta^-(i) = \{(x, y) \in E : y = i\}$.

6 PORKSOC

This chapter introduce PorkSoC, the System on Chip built to test the integration of synthesized accelerators. The chapter starts explaining why it has been implemented another SoC instead of using one already available. It continues stating the architecture requirements and then giving an overview of the built architecture. Finally, it presents a study case with the obtained results.

6.1 MOTIVATION

The long term goal in having a SoC as part of the PandA framework is the introduction of a hybrid HLS flow. The hybrid HLS methodology will enable the generation of an application specific heterogeneous multi-core architecture. The generated heterogeneous multi-core architecture will contain a GPP and a set of hardware accelerator connected through a WB4 intercon. The hardware accelerators will be synthesized using the methodology proposed in this work. This long term goals motivates the introduction of PorkSoC into the framework. In the short term the SoC will be used to test the integration and the interaction between accelerators and the GPP.

During the design of the SoC, the choice of which GPP to include has led to a critical analysis of the available alternatives. The final decision has been in favor of the Or1200 processor.

The industry already supports methodologies allowing to build heterogeneous multi-core architectures. In fact all the three leading companies in the FPGA market offer a soft-core IP ready to be included into designs.

Xilinx offers MicroBlaze [Xil]. Microblaze is a 32-bit Reduced Instruction Set Computing (RISC) soft-core that is included free with some of the Xilinx tools. It has an Harvard architecture and it is highly configurable. It can be configured to

include a Memory Management Unit (MMU), instruction and data caches or to have different pipeline depths.

Altera offers Nios II [Alt]. It is a family of three configurable 32-bit RISC Harvard architecture cores. The first architecture is a six stage pipeline optimized for performance with an optional MMU or Memory Protection Unit (MPU). The second architecture is optimized for smallest size. The third option lays in between balancing performance and size of the core.

Lattice offers LatticeMico32 [Lat]. It is a 32-bit RISC Harvard architecture with a six stage pipeline. It optionally can include caches, with various sizes and arrangements, and pipelined memories. LatticeMico32 has a dual wishbone interface for data and instruction bus.

Besides industry products there are also alternatives coming from academia and the open-hardware community. Of all the available alternatives two have been taken into account for the inclusion inside the SoC.

The first one is SecretBlaze [Bar+11]. SecretBlaze is an open-source processor developed at LIRMM. It is a 32-bit RISC Harvard processor with a five stages pipeline. Its Instruction Set Architecture (ISA) is compatible with Xilinx's MicroBlaze ISA. The second one is Or1200. It is a 32-bit RISC Harvard processor with a five stage pipeline. It can be configured to include instruction and data caches and MMUs, a ticktimer, floating point units and power management unit. According to the project page it has been implemented in various commercial Application Specific Integrated Circuits (ASICs) and FPGAs.

6.2 ARCHITECTURE REQUIREMENTS

To better understand the architecture requirements it is important to understand some facts about the PandA user base. PandA is a research project developed at Politecnico di Milano quite exclusively by researchers and students. While the main core of the user base is still from academia it is gaining the attention of industry people and of the open hardware community. For this reason the design of the architecture must take into account that some of the intended audience of the tool may not have access to expensive commercial tools and devices. Requirements will reflect this background consideration.

The first requirement is that all of the IPs used to build the architecture must be open-source. For this reason it has been chosen to use IPs freely available at OpenCores. This will avoid problems with products discontinuity and to respect the vendor agnostic philosophy of the PandA framework.

The second requirement is that the designed architecture must be modular and easily extensible with accelerators. In the spirit of design for the future, this will help to purse the long time goal of implementing a hybrid HLS flow inside the PandA framework. Moreover, this characteristic will help to achieve the short

term goal of building a good testing framework usable to refine the methodology proposed in this thesis.

The last important requirement is that the complete design should fit inside small FPGAs that are usually affordable for students.

A few final words must be spent on the choice of the GPP at the light of the just stated requirements. The two main alternatives were SecretBlaze and Or1200. While both architecture are well supported by toolchains and debugging software, SecrectBlaze has a not negligible drawback in this context. It is modeled using VHDL while accelerators produced by the PandA framework are modeled using Verilog. This implies that the simulation of an architecture containing SecretBlaze and accelerators produced by PandA needs the support of a simulation tool capable of mixed HDL simulation. All the leading commercial tools are capable of such a kind of simulation but none of the freely available versions and open-source alternatives are. This mostly motivates the choice of Or1200 over SecretBlaze but there other advantages.

6.3 PORKSOC BASE ARCHITECTURE

Before going into the details of the PorkSoC architecture, it is important to motivate the choice of designing a SoC from scratch instead of using some of the already available SoC based on the same architecture.

After a careful analysis of the available options two alternatives have emerged: OrpSoC and MinSoC. OrpSoC is a SoC designed to be a feature rich testing environment for the OperRisc architecture. MinSoC instead is the Minimalistic OperRisc SoC. It is designed to fit in small FPGAs. Of these two alternatives, MinSoC is the nearest to fulfill the requirements stated in section 6.2. In fact, it has positively influenced the design and implementation of PorkSoC. The main problem with MinSoC design is that it is not intended to be automatically generated. The availability of unused wishbone interface in the intercon included in MinSoC makes it a potential candidate to solve the short term goal of accelerator testing. On the contrary it is impossible to use this design to solve the long term goal of SoC generation. The main reason is that the size of the intercon is fixed and can not scale with the number of accelerators to be included into the design.

This consideration and the requirements stated in section 6.2 have led to the current implementation of PorkSoC. PorkSoC is a highly configurable System on Chip based on the OperRisc architecture.

Figure 6.1 shows the base architecture of PorkSoC. Blue components are mandatory to implement a SoC based on the Or1200 processor. The two gray components are not strictly necessary but can be useful in interfacing with the SoC and for debugging purposes.

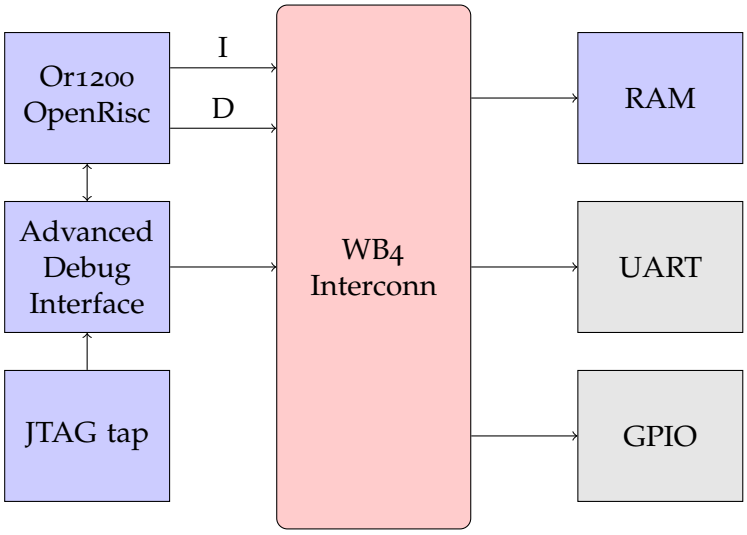


Figure 6.1: PorkSoC base architecture

The block diagram of Figure 6.1 shows that Or1200 has three interfaces. Two wishbone interfaces are connected to the wishbone intercon and are used to access data and instruction memory. The third interface is for debugging support and is connected to the *Advanced Debug Interface* module. The Advanced Debug Interface is a component that control the execution of the processor during debugging. This enable to control program execution using software debugger like GNU Project Debugger (GDB). The Advanced Debug Interface is also connected to the wishbone intercon in order to have access to memories connected to the architecture. The Advanced Debug Interface module is controlled by the *JTAG tap* module that receives debugging commands from the software debugger.

Instruction memory and data memory is implemented with as a Random Access Memory (RAM). The whole memory is allocated inside the FPGA instead of using external resources. This design choice makes unnecessary the use of instruction and data caches. The reason is that synthesis tools translate both with the same FPGA resources so they have the same access times voiding the advantages of caches usage. The size of the memory instantiated can be configured during component instantiation.

The two optional component are a Universal Asynchronous Receiver/Transmitter (UART) controller and a General Purpose Input/Output (GPIO) controller. The UART controller is a hardware module that translates data between parallel and serial form. It is usually used in conjunction with other communication standards allowing data transfer through serial ports. The GPIO controller is a hardware module that allows the user to interact with a set of external pins. The user by means of the GPIO controller can read or write logical values on the pin. In order

to perform read or write operation on pins, the user must configure pins working direction (input or output) using a memory mapped control register of the GPIO controller. It has been included because during the debugging of such systems can be very useful to have visual feedback turning on and off LEDs.

6.4 EXTENDING THE ARCHITECTURE WITH ACCELERATORS

In order to easily extend the base architecture of PorkSoC the intercon is not statically defined. It is generated to host the connection of the needed number of master and slave interfaces. In its base configuration of Figure 6.1, the PorkSoC interconnect need to expose three master interfaces and two slave interfaces. The three master interfaces are needed for modules that are capable of initiating wishbone bus cycles. There are two of them, the Or1200 and Advanced Debug Interface, but three interfaces are needed because Or1200 uses different interfaces to interact with data memory and instruction memory.

Extending PorkSoC base architecture with additional accelerators is a three step process. The first step is the implementation of the accelerators to be included. They can be automatically generated with an HLS tool or not. In any case the accelerator must expose a wishbone interface. The second step is the instantiation of the accelerator inside the top module of PorkSoC and its configuration. The third and final step is performing the connection between the accelerator and the intercon.

At the current stage, the inclusion of an accelerator into the PorkSoC architecture must be performed by the designer. The long term goal is to automate the process with the introduction of an hybrid HLS flow inside the PandA framework.

6.5 EXPERIMENTS

PorkSoC has been used to test accelerators integration during the development of the methodology proposed in this thesis. The following paragraphs will present an experiment conducted to test the call mechanism of accelerators from software executed by the Or1200 processor. The accelerator used to conduct the test was synthesized by a function computing the crc32 of an input string. The crc32 computation is performed by the accelerator by means of a table allocated in RAM.

Listing 6.1 shows the source code used in our experiment. The program entry point is the function `main`. The first three function calls inside `main` are service routines used to initialize the environment. The `gpio_init` function initializes the GPIO controller. The next function calls to `int_init` and `int_add` initialize the interrupt controller and define an interrupt handler for the GPIO.

The following function call to `crc32` is the core of the test application. At the beginning of Listing 6.1, the declaration of the function `crc32` is marked with the

```
__attribute__((synthesize))
ulong crc32(ulong crc, unsigned char* buf,
            ulong offset, ulong len,
            ulong crc32tab[]);

int main() {
  gpio_init();

  int_init();
  int_add(GPIO_IRQ, &gpio_interrupt, NULL);

  int crc =
      crc32(10, "0,1,2,3,4", 0, 5, crc_32_tab);

  REG32(GPIO_BASE + RGPIO_OE) = 0x000000ff;
  REG32(GPIO_BASE + RGPIO_OUT) = crc;
  or32_exit(0);
}
```

Listing 6.1: Test program for crc32

attribute *hwcall*. The attribute defines that the marked function is implemented as an hardware accelerator. The presence of the attribute is used to trigger a transformation pass implemented as a GCC plugin, as discussed in section 5.3. The transformation pass translates calls to marked functions with the instruction needed to call the underlying accelerator.

Listing 6.2 shows the pseudo-code of the transformation made on the original function call by the pass. The first operation performed by the transformed code is parameter passing. Each of the function parameters is written inside the corresponding memory mapped register of the accelerator. Then the computation is started writing into the accelerator control register. After that the program enters a busy waiting loop. The software stays into the loop until the accelerator notifies the end of the computation. The program get notified of the computation end by reading the accelerator control register. At the exit of the waiting loop, the software retrieves the return value of the computation reading the memory mapped register of the accelerator that stores it.

As shown in Listing 6.2, the current implementation of the accelerator call mechanism in software is performed with the injection of a busy waiting loop during the transformation pass. Simulation and testing on FPGA have demonstrated that the GPP does not send request faster enough to cause bus congestion. Polling is not the only mechanism that can be used to implement the hardware-software interaction mechanism with accelerators. All the synthesized accelerators expose interrupt lines that can be connected to the interrupt controller of the GPP. This means that, with the support of an operating system, the GPP can schedule other

```
int main() {
  // ...
  // ulong crc = crc32(
  //     10, "0,1,2,3,4", 0, 5, crc_32_tab);
  *REG32(CRC32_PARAM1_ADDRESS) = 10;
  *REG32(CRC32_PARAM2_ADDRESS) = "0,1,2,3,4";
  *REG32(CRC32_PARAM3_ADDRESS) = 0;
  *REG32(CRC32_PARAM4_ADDRESS) = 5;
  *REG32(CRC32_PARAM5_ADDRESS) = crc_32_tab;
  *REG32(CRC32_CTRLREG_ADDRESS) = START;
  while (*REG32(CRC32_CTRLREG_ADDRESS) != END);
  ulong crc = *REG32(CRC32_RETVAL_ADDRESS);
  // ...
}
```

Listing 6.2: Call mechanism code

Total logic elements	7,940 / 18,752 (42 %)
Total combinational functions	7,183 / 18,752 (38 %)
Dedicated logic registers	3,405 / 18,752 (18 %)
Total registers	3405
Total pins	36 / 315 (11 %)
Total virtual pins	0
Total memory bits	133,376 / 239,616 (56 %)
Embedded Multiplier 9-bit elements	8 / 52 (15 %)
Total PLLs	1 / 4 (25 %)

Table 6.1: Synthesis report for PorkSoC + crc32 on the DE1

tasks in parallel with the accelerator computation and get notified of its end by means of an interrupt.

The program in Listing 6.1 ends by writing one byte of the result of the `crc32` through the GPIO. This operation lights up some LEDs of the testing board as a visual feedback.

6.6 SYNTHESIS RESULTS

The synthesis of the PorkSoC architecture with the crc32 accelerator has proved some of the strength of the design and has pointed out some of its actual limitations.

The report in Table 6.1 shows the area occupation of the architecture inside the used FPGA. The report indicates that the synthesized architecture uses roughly half the resources available on the FPGA included in the Altera DE1 board. The Altera DE1 board contains an Altera Cyclone II FPGA that is one of the smallest and cheapest FPGA on the market. The synthesis proves that the design satisfies the

small footprint requirement stated in section 6.2. Also it shows that the remaining resources on the FPGA are enough to include other accelerators to the architecture.

The synthesis has also exposed some modeling issue inside Or1200. At the first synthesis the tool was not able to synthesize a design that was working at an acceptable clock frequency. The result was around 25MHz. By inspecting the synthesis reports obtained by the tool and the Or1200 HDL description, some of the problems have been pointed out. Most of them where due to the fact that Or1200 is targeted both at ASIC and FPGA. For this reason some of the component inside the processor were modeled using patterns from the ASIC world. The two problems found inside the description of Or1200 are in the register file and in the multiplier. The register file was modeled to be synchronous on both raising and falling edge of the clock signal. This was requesting the register file to work twice faster than the rest of the design. The multiplier was instead described using a patter from the ASIC world that is not compliant with the FPGA best practice. The result of its usage was that it was blocking the retiming algorithm of the synthesis tool.

Once these modeling issue have been fixed, the synthesis of the architecture has started to obtain acceptable working frequencies. The final result obtained was around 80MHz.

7 DISCUSSION

The chapters starts presenting a study case used to test the methodology proposed in chapter 3. Then it continues presenting the testing environment. The final section presents the obtained results with architecture synthesis and simulation.

7.1 STUDY CASE

The study case chosen is an application of the LU factorization to solve a linear system and compute the inverse of a matrix.

Let A be a square non singular matrix. The LU factorization decompose a given matrix A as the product of two factors L and U.

$$A = L \cdot U \tag{7.1}$$

Where L is a lower triangular matrix and U is an upper triangular matrix.

The LU decomposition can be used to solve linear systems of the form:

$$A \cdot \underline{x} = \underline{b} \tag{7.2}$$

transforming the system to the equivalent:

$$U \cdot \underline{x} = \underline{y} \tag{7.3}$$
$$L \cdot \underline{y} = \underline{b} \tag{7.4}$$

This transformation allows to split the resolution of original system of Equation 7.2 in the resolution of two triangular systems. At first it may seem that the problem has been complicated but triangular systems can be efficiently resolved using forward and backward substitution algorithms. These two algorithms have

a computational complexity of $O(n^2)$ while the LU decomposition algorithm has a computational complexity of $O(n^3)$. The cost of this resolution method is dominated by the cost of the computation of the LU factorization. But this is a one time cost. In fact, the LU decomposition of a matrix can be reused multiple times to solve the systems having the same matrix but changing the $\underline{b}$ vector. For example, this method can be used to compute the inverse of the A matrix solving n linear systems using as $\underline{b}$ vector the columns of the identity matrix.

7.2 SYNTHESIS OF THE STUDY CASE

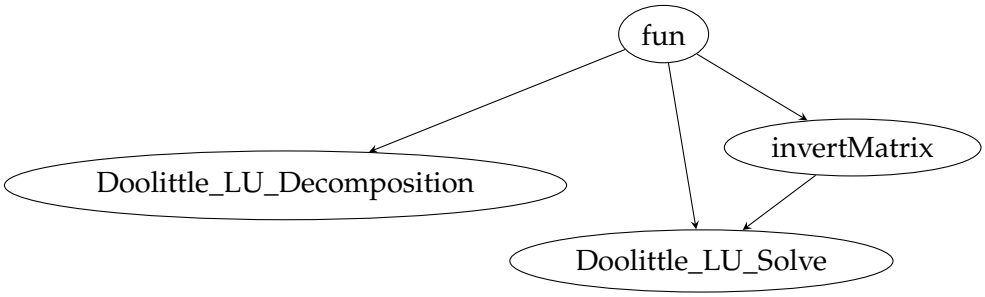


Figure 7.1: Call graph of the study case

Listing 7.1 shows the core of the study case used to evaluate the performance of the proposed methodology. The program entry point is the `fun` function. The program starts computing the LU decomposition of the matrix `A`. The result of the computation is stored packing the L and U matrix inside `A`. Then the program continues solving a linear system calling `Doolittle_LU_Solve`. After that, the program computes the inverse of A using its LU decomposition as outlined in section 7.1.

Figure 7.1 shows the segment of interest of the call graph of the study case.

To evaluate the impact of the proposed methodology, two different synthesis and simulation of the application in Listing 7.1 have been performed. In the first experiment, the application was synthesized as an accelerator with the WB4 interface using only the traditional function call mechanism. In the second experiment, the application was synthesized using the non-inlined function mechanism for calls to `Doolittle_LU_Solve`. As the call graph in Figure 7.1, this function is shared between the `fun` and `invertMatrix` functions. Moreover, the considered experiment setup does not include PorkSoC to better isolate results obtained following the proposed methodology.

```
int invertMatrix(float *LU, float *invA)
{
  int i, j;
  float I[4][4] = {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0,
      1}};
  float resultColumn[4];

  for (i = 0; i < 4; ++i)
  {
    int res = Doolittle_LU_Solve(LU, I[i], resultColumn, 4);

    if (res != 0) return res;
    for (j = 0; j < 4; ++j)
      *(invA + i  + j * 4) = resultColumn[j];
  }

  return 0;
}

//float A[4][4] = {{1, 1, 1, 1}, {1, 4, 2, 3}, {1, 2, 1, 2}, {1, 1, 1,
    0}};
//float invA[4][4]= {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0,
    0}};

int fun(float *A, float *invA, float *b, float *x)
{
  int res = Doolittle_LU_Decomposition((float *)A, 4);

  if (res != 0) return res;

  //  float b[4] = {63, 105, 48, 186};
  // float x[4];

  res = Doolittle_LU_Solve((float *)A, b, x, 4);

  if (res != 0) return res;

  res = invertMatrix((float *)A, (float *)invA);

  return res;
}
```

Listing 7.1: Study case with LU decomposition and LU solve

7.3 RESULTS

The synthesis of the two experiments have shown that the non-inlined function call mechanism produces a significant reduction of the area usage. Comparing the

Total logic elements	37,700 / 68,416 (55 %)
Total combinational functions	32,096 / 68,416 (47 %)
Dedicated logic registers	23,389 / 68,416 (34 %)
Total registers	23389
Total pins	211 / 622 (34 %)
Total virtual pins	0
Total memory bits	20,211 / 1,152,000 (2 %)
Embedded Multiplier 9-bit elements	72 / 300 (24 %)
Total PLLs	0 / 4 (0 %)

Table 7.1: Synthesis report of architecture with wishbone interface on *Altera Cyclone II EP2C70F896C6*

Total logic elements	25,724 / 68,416 (38 %)
Total combinational functions	21,834 / 68,416 (32 %)
Dedicated logic registers	15,914 / 68,416 (23 %)
Total registers	15914
Total pins	212 / 622 (34 %)
Total virtual pins	0
Total memory bits	12,962 / 1,152,000 (1 %)
Embedded Multiplier 9-bit elements	46 / 300 (15 %)
Total PLLs	0 / 4 (0 %)

Table 7.2: Synthesis report of architecture with non-inlined call on *Altera Cyclone II EP2C70F896C6*

results of Table 7.1 and Table 7.2, it can be seen that the introduction of the non-inlined function call mechanism has reduced the area occupation of the synthesized design of about 12000 logic elements. The synthesis has been performed targeting an FPGA of the *Cyclone II* family. The obtained saving on the used model correspond to the 17% the logic elements available.

The results show that the area saving obtained with the introduction of the non-inlined call of `Doolittle_LU_Solve` saves enough space to include into the design additional components for the size a little SoC like PorkSoc.

The choice of the functions to be marked to use the non-inlined function call mechanism becomes critical. Good candidates to use the introduced methodology are function that performs non trivial computations and that are called by multiple functions inside the synthesized application.

Both version of study test case have been simulated using ModelSim[Gra] by Mentor Graphics. The measurements performed on the simulation results are summarized in Table 7.3.

The simulation has shown that the architecture using the non-inlined function mechanism takes 4902 clock cycles while the architecture using the inlined mech-

	inlined	non-inlined	speepup
fun	4552	4902	0.93
Doolittle_LU_Decomposition	899	899	1
__builtin_wait_call	-	862	-
Doolittle_LU_Solve	834	834	1
invertMatrix	2814	3134	0.89

Table 7.3: Number of clock cycle of function execution measured from simulation. Numbers for __builtin_wait_call and Doolittle_LU_Solve are measures from the first call. Case of internal I.

anism takes 4552 clock cycles. The difference in performance between the two architectures is caused by two factors: the non-inlined function calls overhead and by the memory allocation strategy.

As described in section 3.4, the non-inlined function call mechanism needs a number of clock cycles proportional to the number of the function parameters to complete. Measures from the simulation results have shown that the function call mechanism takes 14 clock cycles to start the computation of `Doolittle_LU_Solve`. On the other hand, inlined mechanism can start the function computation in one cycle.

The new methodology imposes an overhead also on the function return with the addition of the notification mechanism described in subsection 3.4.1. In this case the overhead can change only between function with or without return values. Measures on the simulation have show that the notification mechanism for `Doolittle_LU_Solve` lasts in 14 clock cycles. The cumulative overhead introduced by the methodology for the `Doolittle_LU_Solve` function is 28 clock cycle. The datum can also be extracted by Table 7.3. The lines of the `__builtin_wait_call` shows the number of clock cycles needed to perform the first call to `Doolittle_LU_Solve` from caller perspective. The subsequent line contains the effective number of clock cycles need to complete the computation of `Doolittle_LU_Solve`. The difference between the two numbers is exactly 28. To this overhead must be added 2 additional clock cycle to read the return value from the memory where the builtin call stores it. This brings the total over head to 30 clock cycle per call to the `Doolittle_LU_Solve` function.

This justifies part of the performance lost. In fact, some simple math shows that the call to `invertMatrix` is still 200 clock cycles slower than its counterpart in the inlined test considering the 120 clock cycle of overhead of the 4 calls to `Doolittle_LU_Solve`. The difference in the computation times is due to the memory allocation strategy followed by the tool during the architecture generation. In fact, the difference in performance between the two architecture is due to how they have access to the I matrix. In both architecture the identity matrix is allocated inside the `invertMatrix` accelerator. The difference in performance between

the two architectures lays in the way they access it. In fact, the version using the non-inlined function call mechanism has to traverse the bus to read data from it. This shows that the designer must take particular care of where memories are allocated inside the architecture using the proposed methodology.

8 CONCLUSION

8.1 FINAL CONSIDERATIONS

As introduced in section 1.1, the contribution of this thesis to HLS is the introduction of a new methodology supporting the synthesis of function pointers and the definition of the non-inlined function call mechanism.

As explained in chapter 3, the new methodology defines the following entities and mechanisms:

- the memory mapped interface for the synthesized accelerators;
- the architecture connecting together cooperating accelerators;
- the non-inlined call mechanism used by accelerators to invoke other accelerators;
- the notification mechanism to notify caller accelerators of the end of the computation in the callee;
- a SoC suitable to be generated with the introduced architecture.

The proposed methodology has been implemented inside Bambu, the HLS tool of the PandA framework. This implementation has been used to evaluate the overhead of the non-inlined function call mechanism and the area saving obtained with the new methodology. A detailed discussion of the results can be found in chapter 7. The analyzed study case has exposed the strengths and weaknesses of the implementation of the proposed methodology. It has pointed out the areas where the implementation can be improved in order to reduce the overhead imposed by the non-inlined function call mechanism. For example, the WB4 pipelined classical

bus cycle can be used to reduce the overhead of parameters passing and of the notification mechanism.

8.2 FUTURE DEVELOPMENTS

The methodology proposed in this work lays the foundations of two main future developments. The first is its integration in a hybrid HLS flow. The second possible future development lay its foundation on the architectural meaning of function pointers given by the methodology proposed in this thesis. Its definition and the mechanism defined in this work can be the starting point to implement functionality and libraries that needed function pointers support in HLS synthesis flows. An important example in this context are the `pthread` library and OpenMP language.

BIBLIOGRAPHY

[Alt] Altera. *Nios II Processor: The World's Most Versatile Embedded Processor* (cit. on p. 52).

[Bar+11] Lyonel Barthe, Luis Vitório Cargnini, Pascal Benoit, and Lionel Torres. "The secretblaze: A configurable and cost-effective open-source soft-core processor". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 310–313 (cit. on p. 52).

[Bolo8] Thomas Bollaert. "Catapult synthesis: a practical introduction to interactive C synthesis". In: *High-Level Synthesis*. Springer, 2008, pp. 29–52 (cit. on p. 9).

[BTD] BTDI. *High level synthesis tool certification program resuts*. URL: http://www.btdi.com/resources/benchmarkresults/hlstcp (cit. on p. 1).

[Bur+12] Paolo Burgio, Andrea Marongiu, Dominique Heller, Cyrille Chavet, Philippe Coussy, and Luca Benini. "OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters". In: *Digital System Design (DSD), 2012 15th Euromicro Conference on*. IEEE. 2012, pp. 751–758 (cit. on p. 12).

[Bur+13] Paolo Burgio, Andrea Marongiu, Robin Danilo, Philippe Coussy, and Luca Benini. "Architecture and programming model support for efficient heterogeneous computing on tigthly-coupled shared-memory clusters". In: *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*. IEEE. 2013, pp. 22–29 (cit. on pp. 12, 13, 43).

[Cad08] Cadence. *Cadence c-to-silicon white paper*. Tech. rep. 2008. URL: http://www.cadence.com/rl/resources/technical_papers/c_tosilicon_tp.pdf (cit. on p. 9).

[Can+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 33–36 (cit. on pp. 9, 10).

[CCF03] Deming Chen, Jason Cong, and Yiping Fan. "Low-power high-level synthesis for FPGA architectures". In: *Proceedings of the 2003 international symposium on Low power electronics and design*. ACM. 2003, pp. 134–139 (cit. on p. 9).

[CM08] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008 (cit. on p. 9).

[Cor] Kalray Corporation. *Many-core Kalray MPPA*. URL: http://www.kalray.eu (cit. on p. 12).

[GR94] Daniel D Gajski and Loganath Ramachandran. "Introduction to high-level synthesis". In: *Design & Test of Computers, IEEE* 11.4 (1994), pp. 44–54 (cit. on p. 2).

[Gra] Menthor Graphics. *ModelSim*. URL: http://www.mentor.com/products/fv/modelsim/ (cit. on p. 62).

[Guo+08] Zhi Guo, Betul Buyukkurt, John Cortes, Abhishek Mitra, and Walild Najjar. "A compiler intermediate representation for reconfigurable fabrics". In: *International Journal of Parallel Programming* 36.5 (2008), pp. 493–520 (cit. on p. 9).

[Gup+04] Sumit Gupta, Rajesh Kumar Gupta, Nikil D Dutt, and Alexandru Nicolau. "Coordinated parallelizing compiler optimizations and high-level synthesis". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 9.4 (2004), pp. 441–470 (cit. on p. 9).

[Har+06] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. "Function call optimization in behavioral synthesis". In: *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*. IEEE. 2006, pp. 522–529 (cit. on pp. 14, 15).

[Kat+02] Vinod Kathail, Shail Aditya, Robert Schreiber, B Ramakrishna Rau, Darren C Cronquist, and Mukund Sivaraman. "PICO: automatically designing custom computers". In: *Computer* 35.9 (2002), pp. 39–47 (cit. on p. 9).

[Lat] Lattice. *LatticeMico32 Open, Free 32-Bit Soft Processor*. URL: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx (cit. on p. 52).

[Mel+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. "Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1137–1142 (cit. on p. 12).

[MS09] Grant Martin and Gary Smith. "High-level synthesis: Past, present, and future". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 18–25 (cit. on p. 9).

[Ope10] OpenCores. *Wishbone B4*. Tech. rep. http://cdn.opencores.org/downloads/wbspec_b4.pdf. OpenCores, 2010 (cit. on pp. 3, 19, 31).

[PBV07] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. "The Molen compiler for reconfigurable processors". In: *ACM Transactions on Embedded Computing Systems (TECS)* 6.1 (2007), p. 6 (cit. on p. 12).

[PFS11] Christian Pilato, Fabrizio Ferrandi, and Donatella Sciuto. "A design methodology to implement memory accesses in high-level synthesis". In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*. IEEE. 2011, pp. 49–58 (cit. on pp. 9, 20).

[Plu] Plural. *The Hypercore Processor*. URL: http://www.plurality.com/hypercore.html (cit. on p. 12).

[SDM01] Luc Séméria and Giovanni De Micheli. "Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 20.2 (2001), pp. 213–233 (cit. on pp. 13, 14).

[SSDM01] Luc Séméria, Koichi Sato, and Giovanni De Micheli. "Synthesis of hardware models in C with pointers and complex data structures". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 9.6 (2001), pp. 743–756 (cit. on pp. 13, 14).

[Syn] Synopsys. *Synopsis Synphony*. URL: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx (cit. on p. 9).

[TGP07] Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. "Trident: From high-level language to hardware circuitry". In: *IEEE Computer* 40.3 (2007), pp. 28–37 (cit. on p. 9).

[Tri+05] Justin L Tripp, Kristopher D Peterson, Christine Ahrens, Jeffrey D Poznanovic, and Maya Gokhale. "Trident: An FPGA Compiler Framework for Floating-Point Algorithms." In: *FPL*. 2005, pp. 317–322 (cit. on p. 9).

[Vas+04] Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, and Elena Moscu Panainte. "The molen programming paradigm". In: *Computer Systems: Architectures, Modeling, and Simulation*. Springer, 2004, pp. 1–10 (cit. on p. 12).

[WCC09] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009 (cit. on p. 6).

[Xil] Xilinx. *MicroBlaze Soft Processor Core*. URL: http://www.xilinx.com/tools/microblaze.htm (cit. on p. 51).

[Xil13] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*. Tech. rep. http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf. Xilinx, 2013 (cit. on p. 11).

ACRONYMS

ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BDTI	Berkeley Design Technology Inc.
CAD	Computer Aided Design
CFG	Control Flow Graph
DFG	Data Flow Graph
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FSMD	Finite State Machine with datapath
FU	Functional Unit
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GPIO	General Purpose Input/Output
GPP	General Purpose Processor
HDL	Hardware Description Language
HLS	High Level Synthesis
ILP	Integer Linear Programming

IP	Intellectual Property
IR	intermediate representation
ISA	Instruction Set Architecture
MMU	Memory Management Unit
MPU	Memory Protection Unit
PDG	Program Dependency Graph
RAM	Random Access Memory
RC	Reconfigurable Computing
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SSA	Static Single Assignment
SoC	System on Chip
UART	Universal Asynchronous Receiver/Transmitter
WB4	Wishbone B4