POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Advanced High-Level Synthesis strategies for a Logic-in-Memory exploration tool

Politecnico di Torino

Supervisors
Prof. Maurizio Zamboni
Prof.ssa Mariagrazia Graziano
Ph.D. Giovanna Turvani

Candidate
Alessio Nicola
ID: 264223

Academic year 2020 – 2021

"I've always believed that you should never, ever give up and you should always keep fighting even when there's only a slightest chance"
#KeepFightingMichael

[Michael Schumacher]

Summary

Nowadays, *von Neumann architectures* are reaching limitations in performance, due to the differences in terms of technology between the Central Processing Unit (CPU) and the memory: this critical condition is known as *von Neumann bottleneck*.

New computational paradigms are emerging to overcome the problem. Among them, the *Logic-in-Memory* (LiM) architectures that bring the computations inside the memory itself, minimizing data transfers between the memory and the CPU.

Octantis is a High-Level Synthesis tool, introduced in its first version in 2020 and developed within the VLSI Laboratory of Politecnico di Torino, in order to support designers in the exploration and development of LiM architectures. The program generates a Register Transfer Level (RTL) design of a LiM architecture, which implements an input algorithm described through a high-level programming language. The main advantage in its use consists in reducing the effort and time required to design and verify the whole architecture.

This thesis work has been focused on the *Optimization*, *Code generator*, and *Scheduling* steps of Octantis.

Regarding the *Optimization* step, several optimization techniques, suitable for HLS and, especially, those that can exploit the capabilities of Logic-in-Memory architectures, have been investigated and proposed.

The first version of Octantis when it was developed, followed the DExIMA Input Reference Language guidelines of that time, now obsolete. The software DExIMA has recently received a major update, in which it has obtained a different and new Input Language. Accordingly, a new *code generator* module in Octantis has been developed, to make it compatible with the new DExIMA Input Reference Language. Furthermore, parts of the source code of Octantis have been maintained, fixing bugs and introducing general optimizations to increase the capabilities and performances of the tool.

The scheduling algorithm is one of the most important modules of a High-Level Synthesis tool. It computes in which clock cycle the different operations belonging to an algorithm must be performed, paying attention to the data dependencies present and also, to the resource constraints of the target architecture. Originally, Octantis only implemented an unconstrained ASAP algorithm, so it has been decided to expand its capabilities, introducing a new scheduling algorithm. In particular, the objective was to enforce the abilities of the scheduling, providing the tool with the capability to explore a larger design space during the elaboration and to adopt advanced management of data dependencies. Furthermore, also the introduction of the possibility to set some meaningful constraints suitable for a LiM architecture has been considered. In this work, several scheduling algorithms have been studied in depth and some of them have been developed and introduced inside Octantis. Advanced strategies derived from graphs theory have been considered too, in order to introduce the support to a more flexible representation, analysis and elaboration of the input algorithms. The scheduling algorithms implemented are all based on the System of Difference Constraints (SDC) formulation, which allows to manage new and advanced constraints during the synthesis process.

Several tests have been conducted to prove the effectiveness of the proposed solutions. The considered algorithms revealed useful to verify the correct management of the data dependencies problem, and the improvements in terms of performance with respect to an ASAP scheduling algorithm. The derived results have shown that the synthesized architectures are well optimized according to the memory size constraint. In particular, doing a comparison with the ASAP scheduling, it is noteworthy the overall reduction of the memory size by at least half, at the worst case cost of

doubling the clock steps necessary to complete the execution. The possibility to constraint the memory size allows the designer to find the best trade off between the occupied area and the performance of the digital circuit.

From this Octantis update, some improvement could be brought. One idea could be an update on the input module where can be specified the scheduling preferences. Further, another idea could be to integrate in Octantis, some technological libraries containing timing, and area information of the logic gates. Allowing to expand the scheduling by including timing constraint.

In conclusion, the new scheduling algorithms have improved in a tangible way the output architecture generation, in addition constraining the memory size.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 In-Memory computation

Nowadays' architectures are mostly based on *von Neumann architecture* [1] — also known as *Princeton architecture* — where *data* and *instructions* share the memory section. The Central Processing Unit (CPU) fetches the instructions from the memory, where it also reads and writes data by using the shared bus, Figure 1.1. Hence, the CPU uses very frequently the memory in order to execute the desired computations.

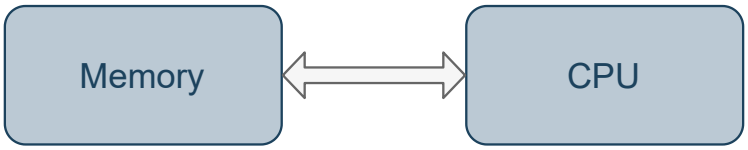

Figure 1.1. *von Neumann architecture.*

The main problem of the *von Neumann architecture* is the bandwidth limitation between the CPU and the memory, because the latter is not able to quickly provide the amount of data required by the CPU. This issue is called *von Neumann bottleneck*.

The *von Neumann bottleneck* is a limitation caused by performance

differences between the CPU and the memory. Since the CPUs can benefit of CMOS technology scaling [2] and at the same time they become faster; instead, the memories, being technologically different, do not benefit from this scaling, and moreover they are big and slow.

The *von Neumann bottleneck* is a limitation caused by performance differences between the CPU and the memory, for the reason that they are technologically different. Although the CPUs become faster as long as there will be benefits of CMOS technology scaling [2]; the memories, since they are large, are very slow compared to the CPUs.

To partially fill the gap of frequency rate and to achieve comparable speed between the CPU and the memory, the hierarchy of some levels of memory is introduced, as depicted in Figure 1.2; thusly near the CPU there is a small and fast memory, as much as the CPU speed. Gradually moving away from the CPU, there are ever larger and slower memories. In this way the slowness of the furthest and biggest memory become less relevant, and the latency is hidden. The limitation on memory hierarchy approach arises when it is needed to process a data intensive application, because the power dissipated for data exchanging between the CPU and the memory is not negligible, where the memory accesses power contribution has to be taken into account.

To overcome the *von Neumann bottleneck* and to stem the problems before described, some research groups are exploring the *in-memory computation* technology.

The in-memory computation technology is classified in different classes, where the main difference consists on the use and integration of processing elements' logic within the memory cells:

- Computation-near-Memory (CnM): logic and memory are separated, however implementing the 3D-SIC technology (3D Stacked Integrated Circuit[1]) they are very close between them, Figure 1.3(A);
- Computation-in-Memory (CiM): computation is performed by peripheral circuitry. The memory array is not modified, the difference consists on its analog peripheral circuitry (e.g. sense amplifiers) in order

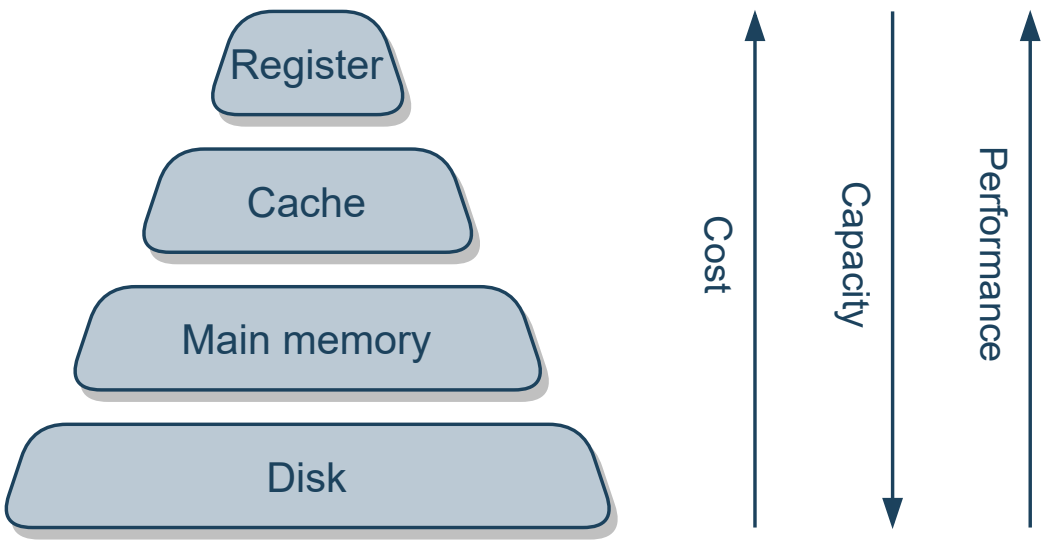

Figure 1.2. Memory hierarchy.

to compute information when the data is sensed inside the array, Figure 1.3(B);

- Computation-with-Memory (CwM): LUT-based memory for data precomputed, Figure 1.3(C);

- Logic-in-Memory (LiM): elementary logic gates are integrated inside the memory cell, in this case the data do not have to go outside the memory to perform basic operations since they can be computed by the logic between the memory cells, Figure 1.3(D).

At *VLSI Laboratory* of Politecnico di Torino a great research effort has been focused on LiM and CLiMA technology.

[1]3D-SIC is a 3D integration technology where the silicon wafers are stacked one above the other exploiting the vertical interconnection through-silicon vias (TSVs).

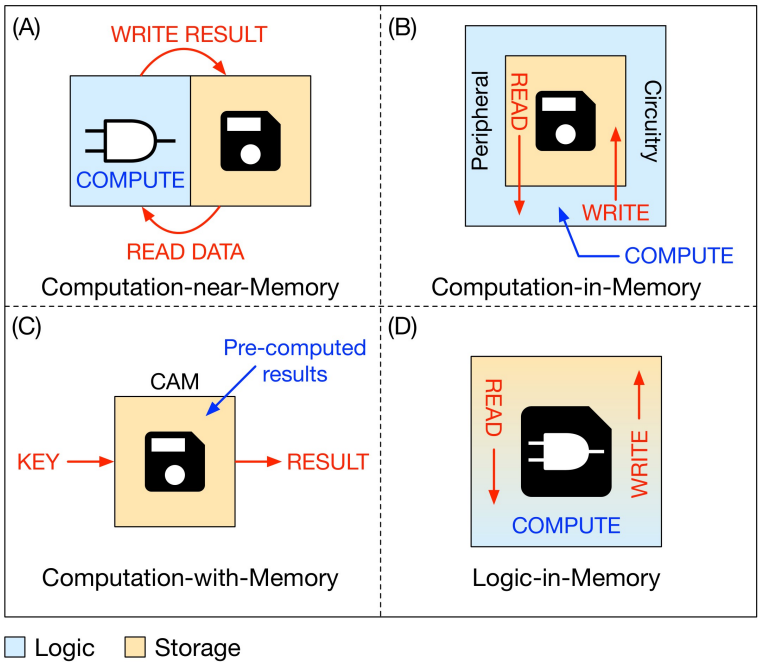

Figure 1.3. In-memory computation classification [3].

1.1.1 Logic-in-Memory

The *Logic-in-Memory* architecture allows to bring the computation inside the memory, and it consists of memory cells linked together by elementary logic circuits. Simple computations can be done within the memory itself, moving data from one memory cell to another one, without transferring data between memory and CPU, exploiting the full memory bandwidth, saving time and energy.

1.1.2 CLiMA architecture

An example of an Logic-in-Memory computation architecture is the Configurable Logic-in-Memory Architecture (CLiMA) [4], a flexible architecture overcoming as much possible the limitations of a traditional von Neumann system. The CLiMA architecture can process complex computation, even those that are not suitable with LiM approach, using peripheral or

complementary logic circuits, a concept schematic is depicted in Figure 1.4. In this way the flexibility is increased, and the architecture can be tailor made to the algorithm, exploiting different In-memory computation approaches if necessary. In CLiMA, between rows and columns, some data manipulation can be performed; in particular there are five possibilities:

- Local;
- Intra-row;
- Intra-column;
- Inter-row;
- Inter-column.

Thus, between memory cells can be achieved operations such as AND, OR, XOR, Ripple Carry Adder (RCA) and Array Multiplier (AM).

Definitely, in-memory computation is a promising technology especially for parallel computing and data intensive applications, as highlighted in [4, 5, 6].

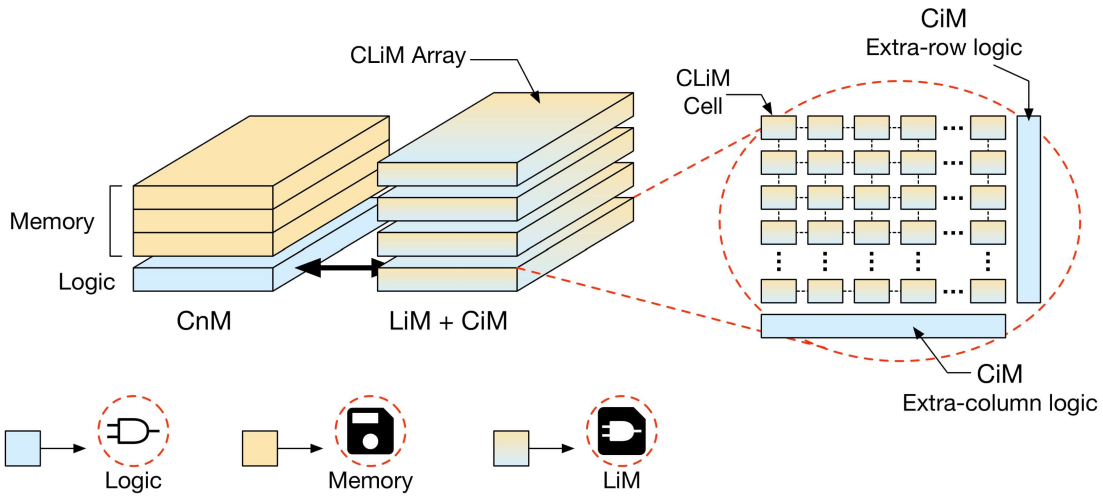

Figure 1.4. Configurable Logic-in-Memory Architecture [4].

1.2 LiM exploration tools

To aid the exploration of Logic-in-Memory architectures, at VLSI Lab of Politecnico di Torino, different tools have been developed. The tools of interest are DExIMA [7] and Octantis [8].

DExIMA — **D**esign **Ex**plorer for **I**n **M**emory **A**rchitectures — is a performance estimator that produces detailed analysis of a LiM architecture in terms of area occupation, static and dynamic power consumption, and timing analysis (e.g. critical paths computation). It gets in input a configuration file where the reference architecture and the finite state machine of a Logic-in-Memory are described.

Octantis is a High-Level Synthesis (HLS) tool, introduced in its first version in 2020 and developed within the VLSI Laboratory of Politecnico di Torino, to support designers in the exploration of LiM architectures. The synthesizer, starting from an input algorithm described in C programming language, generates a complete design of LiM architecture, composed by both a datapath and its control, and expressed through *DExIMA* description language. A deep study about Octantis follows in this dissertation.

Octantis and DExIMA are conceived to work in tandem. The performances of the LiM architecture designed by Octantis are analysed by means of DExIMA. Clearly, the process of understanding if an input algorithm gains the advantages of a LiM implementation, is speeded up by making the exploration process simpler for the user, reducing the effort and the time required to design and verify the whole architecture.

Chapter 2

Compiler

This chapter introduces some notions on Compilers, deepening their structure and functionality, and at end is provided a classification. This study is provided in order to understand the origins and the basic structure of a High-Level Synthesis and, in particular, the one of Octantis.

2.1 Introduction

Many years ago, with the advent of the High-Level Language (HLL), the Compilers have begun to be used in an overbearing way, due to the low-level code — Assembly language — that requires in-depth knowledge of the target hardware. Avoiding others issues arising from it, such as a more difficult verification and the knowledge of different hardware machines. The advantages to writing an algorithm in HLL are many, it is much simpler and faster than any low-level languages.

The compiler allows to translate an HLL code into a low-level target program (Figure 2.1), applying some optimizations and transformations for a target hardware machine, not forgetting the capability to check and find errors on the input source code.

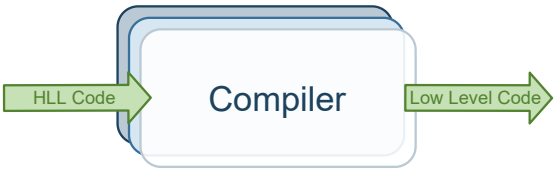

Figure 2.1. Compiler

2.2 Compiler's structure

The structure of a compiler is modular and follows some phases in order to translate the source program [9], having as landmark the intermediate representation of the input code; that divides logically the structure in two parts: *front end* and *back end*. The *front end* takes care on translating into intermediate representation, moreover checks that there are not syntactic and semantic errors, and in case of inaccuracy on the input code an error message is provided to the user. On the other side, the *back end* transforms the intermediate representation into a target machine code, applying several optimization on the code. In Figure 2.2 the steps that make up a compiler are depicted.

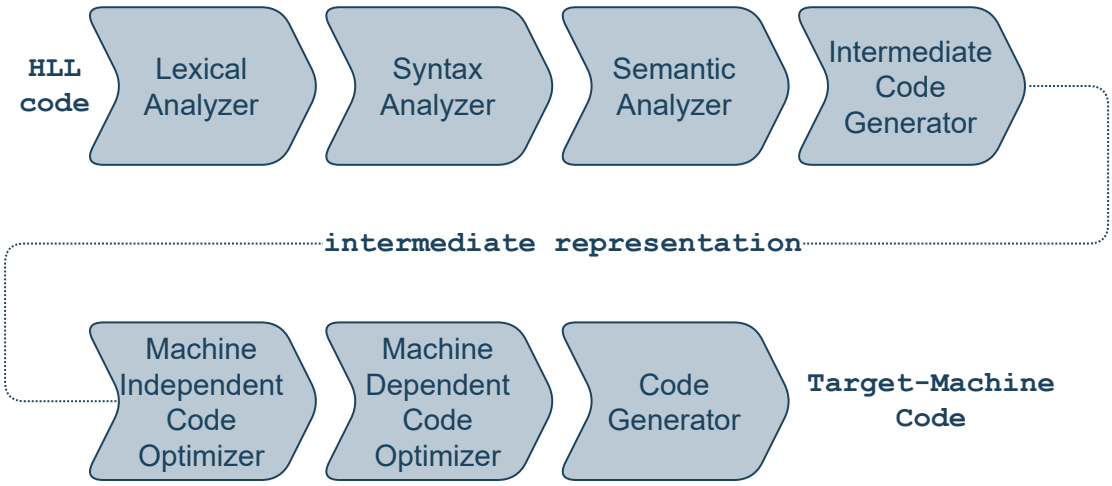

Figure 2.2. Compiler's Phases

The following sections describe each phase depicted in 2.2

Lexical analysis

The *lexical analysis*, or *scanning*, elaborates the data into tokens. The analysis is performed reading and grouping the input stream into sequences named *lexemes*. After that, from the lexemes, the tokens are produced. For example, the simple addition of the equation 2.1 becomes into tokens as described in 2.2. A token is composed by two elements, the first is the *token name*, the second one is the *token entry*.

$$c = a + b \tag{2.1}$$

$$< \textbf{id},1 > \quad <=> \quad < \textbf{id},2 > \quad < + > \quad < \textbf{id},3 > \tag{2.2}$$

Syntax analysis

The phase after the lexical analysis is the *syntax analysis* or *parser*. In this phase, it is created a parse tree from the first tokens' element determining the order in which to perform the operations, moreover checking the grammar correctness. In 2.3 is shown the parse tree of the previous example.

$$\begin{array}{ccc} & = & \\ / & & \backslash \\ < \textbf{id},1 > & & + \\ & / \quad \backslash & \\ & < \textbf{id},2 > \quad < \textbf{id},3 > & \end{array} \tag{2.3}$$

Semantic analyzer

The *semantic analyzer*, from the parse tree, checks the semantic coherence of the instructions. Moreover, in this phase the uniformity of the data's type is checked; for simple instructions, the compiler is able to apply a conversion of type (e.g. from integer to floating point) in order to obtain the correct compatibility.

Intermediate code generation

During the *intermediate code generation* phase a low level intermediate representation of the input HLL code is generated; it must be easy to generate and to use in order to produce a better target machine code. There are some varieties of intermediate code representation, the most common is the *three-address code* with three operands per instruction, as for reduced instruction set computer (RISC) instruction set architecture (ISA).

Machine-Independent code optimizer

The *machine-independent code optimizer* phase transforms the intermediate representation in order to improve and optimize the sequence of instructions. These optimizations are good for all types of machines, being a machine-independent optimizer. Thus, examples of optimizations are a faster code to execute, shorter lines code or less power code to execute.

Machine-Dependent code optimizer

The *machine-dependent code optimizer* applies optimization that depends on the type of the hardware architecture, such as the CPU, memory size, etc...

Code generator

From the optimized intermediate representation, the *code generator* generates the target machine code. In this phase the allocation of registers and memories is done, and also the instructions scheduling.

2.3 Conclusion

All these phases seen so far, can be grouped in *Passes*. For example, the front-end pass can be made up from lexical analyzer to intermediate code generator phases. The back-end pass with machine-dependent code optimizer and code generator. The machine-independent code optimizer sometimes is needless, in fact it is not confined in any front or back end. The modular structure of the compiler allows an easier compilation using the same target architecture with different programming languages using a different front-end, or using the same algorithm for different hardware architecture just changing the back-end.

2.3.1 Compiler classification

The compilers are classified in different ways [10], depending on the combination of input code and output format code or better called *target platform*.

- **Native compiler**: it is also called as *hosted compiler*, compiles from HLL to the machine code suitable for the hosted computer and its operating system (OS), where the the compiler itself is executed.
- **Cross compiler**: it compiles from HLL to a different output target code of the hosted architecture and/or OS. It is mostly used to compile for systems without an operating system, as for the bare metal embedded systems.

- **Source-to-Source compiler**: it is also known as *language translator*, compiles from HLL to HLL. It is used to obtain the same algorithm on different programming languages, or to apply some transformations and optimizations; such as an automatic parallelization of the algorithm, in order to exploit multi-core execution with the application programming interface (API) OpenMP (Open Multi-Processing).

- **Just-in-time compiler**: the *JIT compiler* or *run-time compiler* is regularly inside the interpreters of scripting languages (e.g. JavaScript, Matlab etc...), thus this compiler does not used to compile before the execution, but the compilation is carried out in run-time.

- **Assembler/Disassembler**: it compiles from a low-level programming language (e.g. Assembly language) to a target machine code. The opposite compilation is obtained with a disassembler. This kind of compilers overlaps the Source-to-Source typology.

- **Decompiler**: it compiles from a low-level programming language, or even from a target machine code if supported, to a HLL code. Basically, it is the opposite process of a simple compiler.

- **Hardware compiler**: it is better known as ***High-Level Synthesizer***, and it is used to translate a high level language into a hardware description language or gate level. Hence, from some phases of a generic compiler as studied in Chapter 2.2, it can be built a High-Level Synthesizer.

Chapter 3

High-Level Synthesis

In recent years, to the develop increasingly complex digital design the industries moved on *Electronic System-Level* (ESL) methodology [11]. An innovative technology among ESL methodology is the *High-Level Synthesis* (HLS) one. A High-Level Synthesizer is an automatic compiler, that is able to generate an RTL design from a high-level input code.

HLS aids in the development of a digital circuit, reducing the effort and the time required to design from the scratch the whole digital architecture. Furthermore, the design generated is correct by construction, thus, the verification phase is also less expensive in terms of time and effort.

3.1 High-Level Synthesis' structure

The development of complex digital circuits made these tools necessary to raise the abstraction level, at the design moving from the *Logic Synthesis* to the *High-Level Synthesis* (HLS) [12]. the transition between a High Level Language to a Hardware Description Language (HDL) is called *High-Level Synthesis*: for example from C/C++ code to VHDL/Verilog design. Instead, the transition between the Register Transfer Level (RTL), using a Hardware Description Language such as VHDL or Verilog, to the gate level is known as *Logic Synthesis*.

The HLS generates a complete RTL design, composed by both the datapath and its finite-state machine (FSM). Hence, the datapath is made up of registers, functional units, ALUs, and etc... All these elements of the datapath are timed and driven by the signals coming from FSM, where also it reads the signals generated by the datapath.

The main motivation of their adoption is that the digital design become easier and faster, exploring better the different solutions from the design space and avoiding designing by hand from scratch. Furthermore, the verification phase, one of the most important phases together to the design, gain all the advantages of using an HLS, decreasing the time required to test the behaviour of the design, escaping from common mistakes.

HLS is an NP-hard combinatorial problem, and it takes in input an algorithm written in a high-level language that contains only the algorithm itself without any timing information and/or parallel instructions as can be done in an RTL description.

To solve the HLS problem, it is divided in smaller individual blocks and consequently in smaller problems, 3.1. The front-end takes in input the C code and generate the intermediate representation. During the front-end compilation, are done the lexical, syntax, and semantic analysis as seen in chapter 2.

The optimization phase, after the analysis of the IR code, is relegated to apply some techniques in order to have a more efficient code, for example the dead code elimination, algebraic optimization, loop unrolling and other techniques can be applied.

The core of an HLS is the back-end, made from the allocation, scheduling and binding. These phases are executed one after the other. The back-end together to the Code generator is able to produce the RTL architecture, getting in input the intermediate representation optimized.

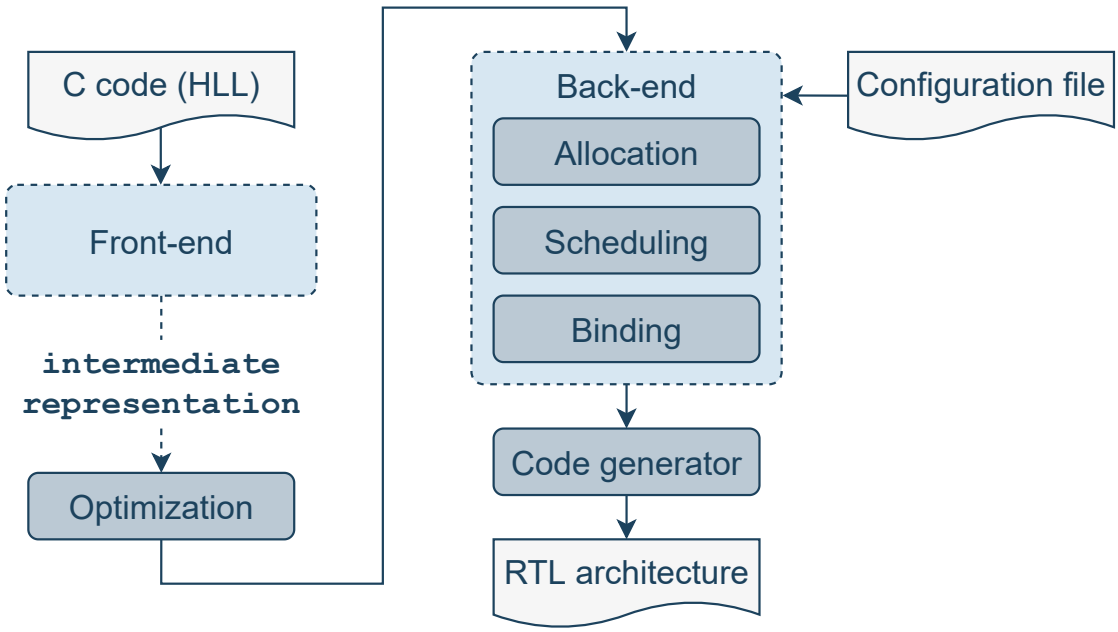

Figure 3.1. High-Level synthesis basic blocks

Allocation

The Allocation step takes in input the optimized intermediate representation, and from it, the required hardware resources will be instantiated, such as logic and registers. The configuration file is used to define several compilation directives and constraints that the HLS must considers; for example the optimization to apply, the memory size, the number of functional units, or constraints in terms of speed and timing.

Scheduling

Once the allocation is completed, there is one of the most important steps of a HLS, that is the scheduling. The scheduling computes in which control step (clock cycle) an operator must be enabled, generating the FSM of the architecture. When there are operators without any data dependency between them, the sequential input code can be scheduled increasing the actual parallelism and optimizing the overall performances. Otherwise, if

operators have data dependencies, the inputs are correlated to the prior outputs such that the code cannot be scheduled in parallel.

Binding

The binding is the step where the variables are assigned to the registers, and the operations to the functional units. In order to optimize the output architecture, the sharing of the hardware is implemented.

Code generator

The last stage of a HLS pipeline is the code generator, which uses the allocation, scheduling, and binding information, to describe the datapath and the FSM of the architecture in a hardware description language as VHDL or Verilog.

3.2 Octantis

The last born tool is a *high-level synthesis* (HLS) tool called Octantis . It takes in input a C code and generates in output a hardware design description DExIMA compliant. In this way, the exploration and the design of Logic-in-Memory architectures become easier and faster and furthermore, the verification phase is simplified. Octantis is written in modular C++, in order to allow new improvements and experimentation on different HLS algorithms easily. Octantis is developed as a Pass on the LLVM[1] Compiler Infrastructure framework [13].

[1]LLVM is not an acronym.

3.2.1 LLVM compiler infrastructure

The LLVM compiler infrastructure is a framework which provides a complete and configurable compiler system. It is made in a way that the input code goes through many layers, where several improvement and optimization are executed. The advantages to use a compiler like this, is to have a good infrastructure and some optimization algorithm ready to use, as those machine-independent. Thus, can be developed compilers, or parts of them, in a simple and modular way.

The most important aspect of the whole LLVM framework is the *LLVM's Intermediate Representation* (IR) [14], because it is the intersect point between the input code and the optimized output code. LLVM's Intermediate Representation is a machine-independent high-level assembly language. It has some different forms of representation, however the *"human readable assembly language"* is one of the most convenient because it is easier to understand and for this reason the debug of the generated code is facilitated. The LLVM IR instructions are expressed in *Static Single Assignment* (SSA), overcoming problems on dependency's analysis such as *write after write* (WAW) and *write after read* (WAR), simply because each variable is assigned only one time. For example if there is a multiple use of a variable, it will be assigned as many times as the use with different names and different memory locations.

The Octantis project is designed around the LLVM's Intermediate Representation, in a way that the input code is compiled into the LLVM IR and then it is analysed by several optimization phases until the output code is generated. Now, for simplicity are defined: *Front-end* – the parsing process from the input code to LLVM IR, and *Back-end* – the optimization and code generation process from LLVM IR to the output description code; an in-depth study follows in next chapter. This kind of conception allows to write the Front-end for any desired high level language, and the Back-end for any target architecture.

In Figure 3.2 the general structure of Octantis is depicted: it takes in input the C code, then the Intermediate Representation is generated, and finally the DExIMA file configuration is produced. In particular, the

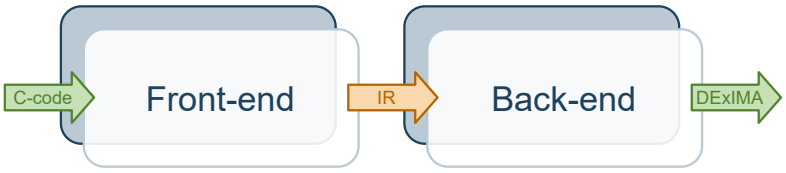

Figure 3.2. Octantis' general structure

Front-end phase is implemented through the Clang[2]compiler; the Back-end phase is addressed optimizing the input IR representation through the LLVM Passes; and at the end the output DExIMA description is generated.

The Back-end can be composed by several LLVM Passes [16], which transform and optimize the Intermediate Representation, each of which takes in input and puts in output always an IR file description. Each pass can be used for a peculiar optimization algorithm. These Passes are used one after the other in sequence, applying all the desired optimizations.

LLVM Compiler Infrastructure, as for the Pass, provides a suite of libraries [17] that can be customized in order to translate the Intermediate Representation in a specific target code; in Octantis the output code is the DExIMA file description.

In conclusion, the Octantis' Back-end is composed by some appropriate Passes to increase the parallelism of the execution, being one of the main features of LiM architectures. Furthermore, there are some phases — as described in High-Level Synthesis chapter — that deal with doing the allocation, the scheduling, the binding and the generation, in order to get the LiM architecture and the finite state machine (FSM).

[2]Clang is a Front-end compiler able to compile C, C++, Objective C/C++. It is compatible with OpenMP, OpenCL and CUDA frameworks [15].

3.2.2 Octantis' main classes

Octantis is a LLVM Pass made up by different classes in order to understand the input Intermediate Representation (it doesn't matter if it is optimized or not), so that in output the LiM architecture and the FSM described inside the DExIMA file configuration will be generated. In Figure 3.3 the organization of Octantis' Classes is shown, where *Octantis Pass* is at the top, which manages the correct execution of each sub classes and methods, with the order:

1. *As Soon As Possible (ASAP) Scheduling*;
2. *LiM compiler*;
3. *Print DExIMA File.*

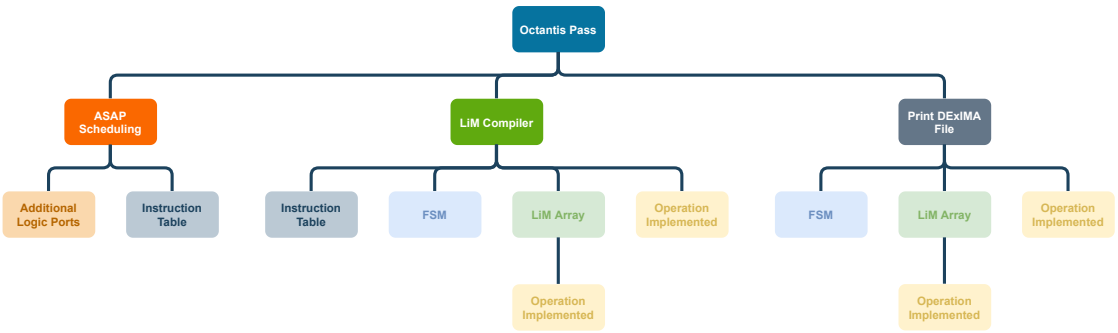

Figure 3.3. Octantis' Classes

Each of these classes in turn use other subclasses and methods, that will be explained below.

Additional logic ports

The *Additional Logic Ports* class extends the LLVM Intermediate Representation language introducing additional logic gates used in RTL design process which are not present, for example the negative logic gates: NAND, NOR and XNOR.

Instruction table

The *Instruction Table* class implements methods to store the scheduled instructions in a data structure. Subsequently, these instructions are used by *LiM Compiler* class.

Finite State Machine

The *Finite State Machine* (FSM) class includes a data structure where to store the information about the scheduling of the architecture according to the ASAP scheduling.

Logic-in-Memory array

The *LiM Array* class keeps information about the Logic-in-Memory structure to implement in hardware, with methods to add new memory rows, to add new Logic-in-Memory rows, and to modify the logic between memory rows.

Operation implemented

The *Operation Implemented* class checks that there are not any irregular operations that could be unrecognized by Octantis and/or DExIMA tool.

As Soon As Possible scheduling

The *As Soon As Possible Scheduling* class identifies the input instructions and it schedules them according to the ASAP algorithm. The LLVM's recognized instructions are listed in Table 3.1. The *load* instruction is considered as definitions of new variables inside the LiM array, while the *store* ones are needed when an elaborated data is available, and useful to define the data dependency.

Instruction	*Description*
alloca	Allocate memory
load	Read from memory
store	Write to memory
binary	Shift, AND, OR, XOR
ptr	Get address
switch	Multiplexer
ret	Return control flow
br	Branch
sext	Sign extension
icmp	Compare

Table 3.1. LLVM's recognized instructions

Logic-in-Memory compiler

The *LiM Compiler* class extracts the information just saved by *ASAP Scheduling* inside the data structure of *Instruction Table* and it generates the *Finite State Machine* and the LiM architecture (*LiM Array*).

Print DExIMA file

The *Print DExIMA File* class, in turn reading the data just processed by *LiM Compiler*, produces the DExIMA file configuration.

Octantis pass

Summarizing, the Octantis Pass starts with the *ASAP Scheduling*, where it reads instruction by instruction and produces the Instruction Table. This last structure is read by *LiM Compiler* in order to generate the FSM and

the LiM architecture storing information respectively in data structures on *FSM* class and on *LiM Array* class. At the end, *Print DExIMA File*, from the last two, generates the output file.

Chapter 4

High-Level Synthesis optimizations

The High-Level Synthesis make extensively use of optimization techniques in order to improve the performance and the quality of the output architecture. Some of these optimizations are often used during the design of a digital architecture, and they are not exclusively used by HLS tools.

In this chapter some optimization techniques that are suitable for a HLS are explored and, especially, that can exploit the capabilities of the Logic-in-Memory.

4.1 Introduction

Once the input C code is acquired by the HLS, it is transformed into a Intermediate Representation form by the front-end. As can be seen in Figure 3.1, there is then a phase called *Optimization*. The Optimization step is not mandatory, but a better output architectures could be obtained through the subsequent processing applied by the HLS tool. Especially, when these techniques are able to exploit some peculiar specification of the target architecture. Several techniques are available as LLVM passes

(e.g. dead code elimination, loop optimization, loop unrolling) that were just studied in [8]. The techniques discussed in the following are oriented to be used on a HLS for LiM technology.

4.2 Loop unrolling with cross iteration dependency

Knowing the pros of a LiM architecture, the primary objective of a HLS is to parallelize as much as possible the input code. When there is a cross iteration dependency (e.g. loop accumulation) the *Loop unrolling* can be applied, following the well-know technique of *look-ahead* expansion. An example is provided where the Algorithm 1 can be optimized as in Algorithm 2:

Algorithm 1: Cross iteration dependency example

```
1 for i ← 0 to 4 do
2 |   sum += V[i]
3 end
```

Algorithm 2: Look-ahead applied on a cross iteration dependency

```
1 sum = V[0] + V[1] + V[2] + V[3] + V[4]
```

4.3 Tree height reduction

Computations and algorithms can be implemented in different ways according to the design constrains: optimizing the performances or minimizing the number of resources at disposal are two typical examples which move the designer from an implementation to another. Starting from the algorithm example in Figure 4.1 the *tree height reduction* technique is used to minimize the number of computational cycles, and the derived results are shown in Figure 4.2. A chain of identical operators can be reordered as

a tree of the same operators, and executing their computations in parallel to save time. In the discussed example this technique allows to compute the SUM in 2 cycles instead of 3.

The tree height reduction technique mathematically is nothing more the union of simple commutative and associative properties. However, its software implementation can be challenging due to the data dependencies that should be upset.

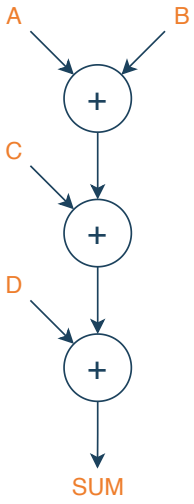

Figure 4.1. Accumulation example

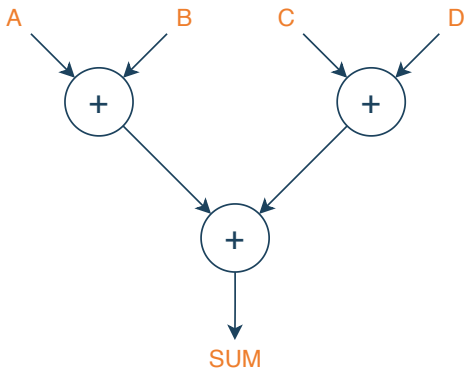

Figure 4.2. Tree height reduction example

4.4 Folding – Time multiplexing

When the area of a circuit has to be reduced or the number of available resources are limited, the *folding method* can be applied. In Figure 4.4 a single adder is used to implement multiple sums. Applying the time multiplexing, a feedback and an accumulator allows to perform the same operation saving 1 adder, but at the cost of a small overhead and additional control signals (e.g. multiplexers selectors). Therefore, while the original system, depicted in Figure 4.3, needed 2 adders and 3 input registers, the folded circuit shows 1 adder, 3 input registers, 1 accumulator and 2 multiplexers.

In terms of performances, the negative consequence of the time multiplexing technique is the decrease of the throughput due to the need of more clock cycles (two in this example) to complete the same operation.

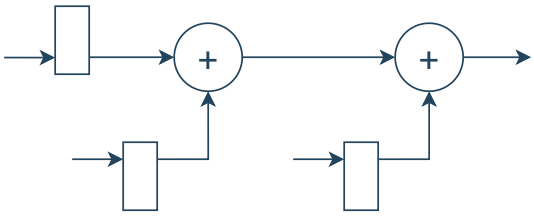

Figure 4.3. Unfolded accumulation example

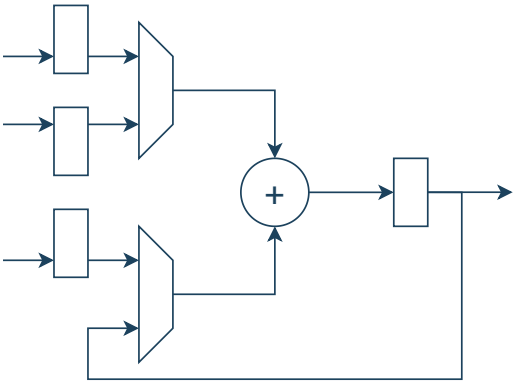

Figure 4.4. Folding example

4.5 If-Conversion

To speed up the execution when the algorithm encounters an If-Else construct, the *If-conversion* technique can be applied [18]. This technique enables the execution of the predication, where the two execution paths (both the ones following the If and Else statements) are scheduled in parallel and the choice of the correct final results is postponed at the end of the two paths. Attention must be paid in the implementation of this technique, as long as there is not any advantage if the two paths are not balanced, because the choice is made once two paths have been completed.

4.6 Multi-threading

The *multi-threading* technique allows to execute multiple threads in parallel, increasing the execution parallelism. There are different standards in the panorama of high level programming languages: OpenMP, OpenCL, CUDA. Each of these have pros and cons in their adoption, and the choice on which one to consider mainly depends on the target architecture and its features. OpenMP is ideal for CPUs, instead OpenCL and CUDA are fitted to be used on GPUs. The advantages of using the multi-threading are to increase the performance thanks to the parallel execution, and an additional way to force additional design specifications inside the compilation process.

Chapter 5

Scheduling algorithms

The scheduling algorithm is a very important component of a High-Level Synthesis tool and it contributes to determine the goodness of the program itself. It computes in which clock cycle an operation has to be performed, paying attention to the data dependencies present in the code and to the resource constraints of the target architecture. In this chapter, different scheduling algorithms are presented, from the simplest (e.g. ASAP, ALAP) to the optimal one based on the System of Difference Constraints (SDC) formulation, which allows to manage advanced constraints during the synthesis process.

5.1 Introduction

The goal of a scheduling algorithm is to reduce as much as possible the control steps required to execute an input algorithm, trying to schedule more instructions in the same time unit, thereby parallelizing at most the instructions to execute. In this way, the performance of the output architecture will be maximized. Obviously, this will not be always possible, there can be some *data dependencies* that must be satisfied, otherwise we loose the semantic of the input algorithm. In addition to timing constraints, when an architecture is developed, also some design specifications

have to be taken into consideration, e.g. resource constraint, timing constraint and interfacing of the architecture.

A Data-Flow Graph (DFG) is generated by parsing the IR code. It is used to analyze the dependencies among the instructions, and the scheduling algorithms organize its structure considering the various constraints. In Figure 5.1 an example of a DFG is depicted.

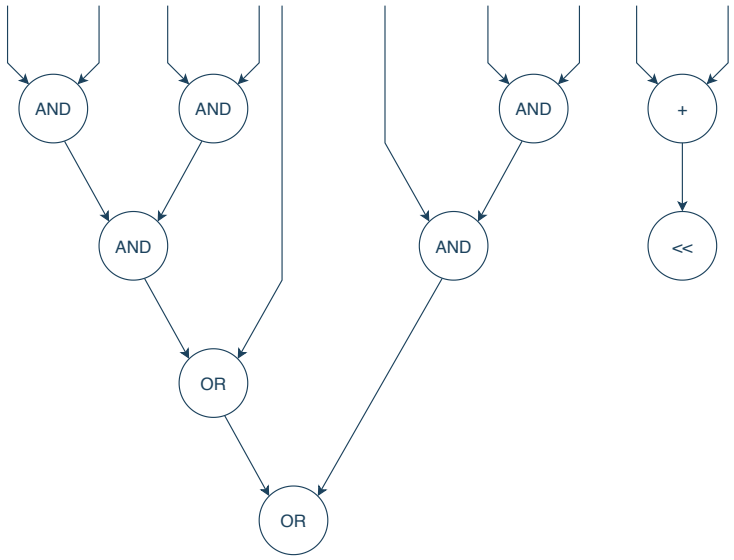

Figure 5.1. DFG example

The scheduling algorithms are classified in:

- **Basic**: As Soon As Possible (ASAP), As Late As Possible (ALAP);
- **Resource Constraint (RC)**: List Scheduling (LS);
- **Time Constraint (TC)**: Force Directed Scheduling (FDS), Integer Linear Programming, Iterative Refinement;
- **Miscellaneous**: Simulated Annealing, Path-Based, Force Directed List Scheduling (FDLS), FDS with FDLS, System of Difference Constraint (SDC) Based

The most significant scheduling algorithms are deepened in the sections below, while further details of all mentioned scheduling algorithms can be

found in [19, 20, 21, 22, 23, 24]. Instead, additional readings about other scheduling algorithms that are not discussed in this thesis, can be found in [25, 26, 27, 28, 29].

5.2 Basic scheduling algorithms

The basic scheduling algorithms, *As Soon As Possible* and the *As Late As Possible* scheduling, are very useful to find two bounds for a feasible scheduling. These algorithms consider infinite resources, as any resource constraint can be applied, they must consider the data dependencies.

5.2.1 As Soon As Possible scheduling algorithm

The ASAP is one of the simplest scheduling algorithm, and it is used to obtain the fastest possible execution: every instruction is allocated as soon as it satisfies any dependency. The scheduling obtained uses the minimum number of control steps, consequently the minimum execution time. Algorithmically, it is obtained analyzing the graph from top to bottom. In Figure 5.2 an example of an ASAP scheduling related to the DFG discussed in the previous section in Figure 5.1.

5.2.2 As Late As Possible scheduling algorithm

As can be imaged, the ALAP scheduling algorithm is the opposite of the ASAP one. In this scheduling, the instructions are scheduled during the last control step available. The obtained scheduling takes the maximum control steps possible. In this case, the graph is analyzed starting from the bottom to the top. The example of ALAP scheduling applyied to the DFG in Figure 5.1 is depicted in Figure 5.3.

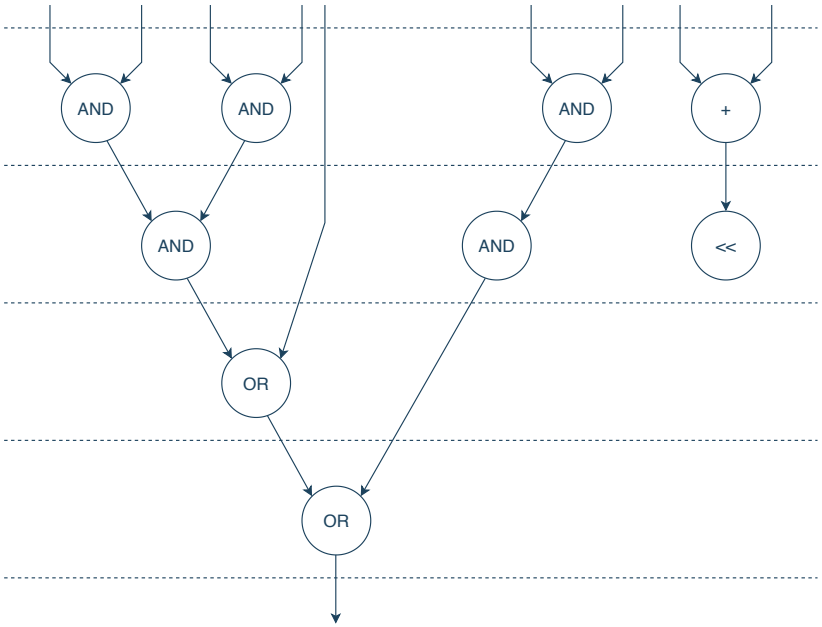

Figure 5.2. ASAP scheduling of DFG in Figure 5.1

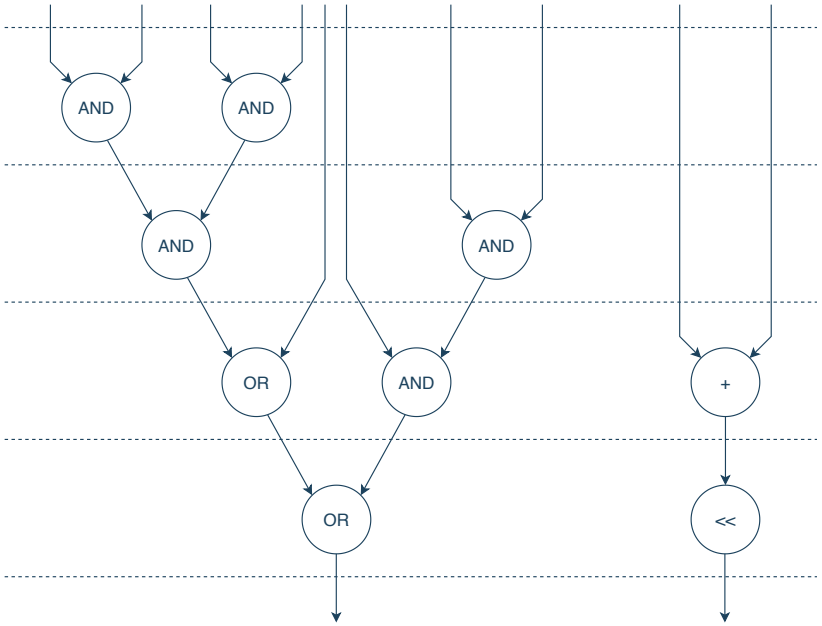

Figure 5.3. ALAP scheduling of DFG in Figure 5.1

5.3 Resource Constrained scheduling slgorithms

When we design an architecture, there are often some limitations in terms of available hardware resources, and in these conditions the Resource Constraint (RC) Scheduling algorithms are necessary, taking into account both the resource constraints and the data dependencies. An algorithm belonging to the RC scheduling is the *List Scheduling*.

5.3.1 List Scheduling algorithm

The List Scheduling algorithm is based on a sorted/priority ready list of the instructions to schedule.

The List Scheduling makes use of the results obtained from the execution of both the ASAP and ALAP algorithms, in order to obtain the two scheduling bounds among which an operator is included. After, the scheduling difference between the ALAP and ASAP scheduling is computed. This quantity gives the mobility that characterizes each operator. The mobility values are collected inside the ready list, that will be sorted with the lowest mobility to the top. Once the ready list has been completely filled, the operators that are at the top of this list, have a higher priority when the scheduling is performed.

Algorithm 3 gives the pseudo code of the list scheduling.

Algorithm 3: List Scheduling algorithm

```
1 ASAP Computation;
2 ALAP Computation;
3 Filling the list of ready operation computing the mobility of each
    operator;
4 Sort the list with the lowest mobility of operators on the top;
5 Scheduling using the priority list;
```

5.4 Time Constrained scheduling algorithms

During the design of an architecture, if the objective is to complete the execution of an algorithm in a fixed time, as happens for Digital Signal Processors, the most effective scheduling algorithms are those known as Time Constrained. In this case, the TC scheduling algorithms are able to optimize the cost of the hardware for a given execution time target.

There are different approaches to develop a TC scheduling. The problem can be solved by Mathematical Programming (e.g. Integer Linear Programming), Constructive heuristic[1](e.g. Force Directed Scheduling), or Iterative Refinement.

5.4.1 Force Directed Scheduling

The Force Directed Scheduling (FDS) is a heuristic algorithm. It is able to minimize the number of operators necessary to execute an algorithm, given a fixed execution time, as target. Furthermore, this scheduling manages the concurrency, balancing in an uniform way, the number of the operators assigned to each control step.

First of all, the ALAP and ASAP scheduling are computed, in order to get the degree of freedom for each operator. The mobility information is useful to find out the operators probability as $Prob = 1/mobility$. Then, the FDS scheduling generates a distribution graph as in Equation 5.1, and it computes the forces applied by each operator as in Equation 5.2.

$$DG(i) = \sum_{OpnType} Prob(Opn, i) \tag{5.1}$$

[1]A heuristic technique is an approach to solve the problem applying some simplifications in order to produce a solution that may not be optimal, but a reliable approximation. Used mainly when the classic methods are too slow, or can't be just solved.

$$Force(i) = DG(i) * p(i) \tag{5.2}$$

In Equation 5.1, $Prob(Opn, i)$ is the probability to use an operator in a given time step i. Instead in Equation 5.2, $p(i)$ is the probability of an operator in control step i, and its sign is negative if it is being removed, or it is positive if it is just added. At the end, when all the forces are computed, starting from the least force operator the scheduling can be done.

Using as example the DFG in 5.1 and helping using the FDS time frames in Figure 5.4, the force of the last AND gate on the right is calculated in this way:

$mobility = 2$
$Probability = 1/mobility = 1/2 = 0.5$
$DG(1) = Prob(AND',1) + Prob(AND'',1) + Prob(AND''',1) = 1 + 1 + 0.5 = 2.5$
$DG(2) = 1 + 0.5 + 0.5 = 2$
$Force(1) = (DG(1) * p(1)) + (DG(2) * p(2)) = (2.5 * (+0.5)) + (2 * (-0.5)) = +0.25$
$Force(2) = (DG(2) * p(2)) + (DG(1) * p(1)) = (2 * (+0.5)) + (2.5 * (-0.5)) = -0.25$

Algorithm 4 gives the pseudo code of the Force Directed Scheduling.

Algorithm 4: Force Directed Scheduling algorithm

1 ASAP Computation;
2 ALAP Computation;
3 Mobility computation of each operator to evaluate the possible time frames. For each time frame is associate the probability (Uniform probability);
4 Distributed graph and Force computation for each operator;
5 Scheduling of the operators that all together give the lowest force;

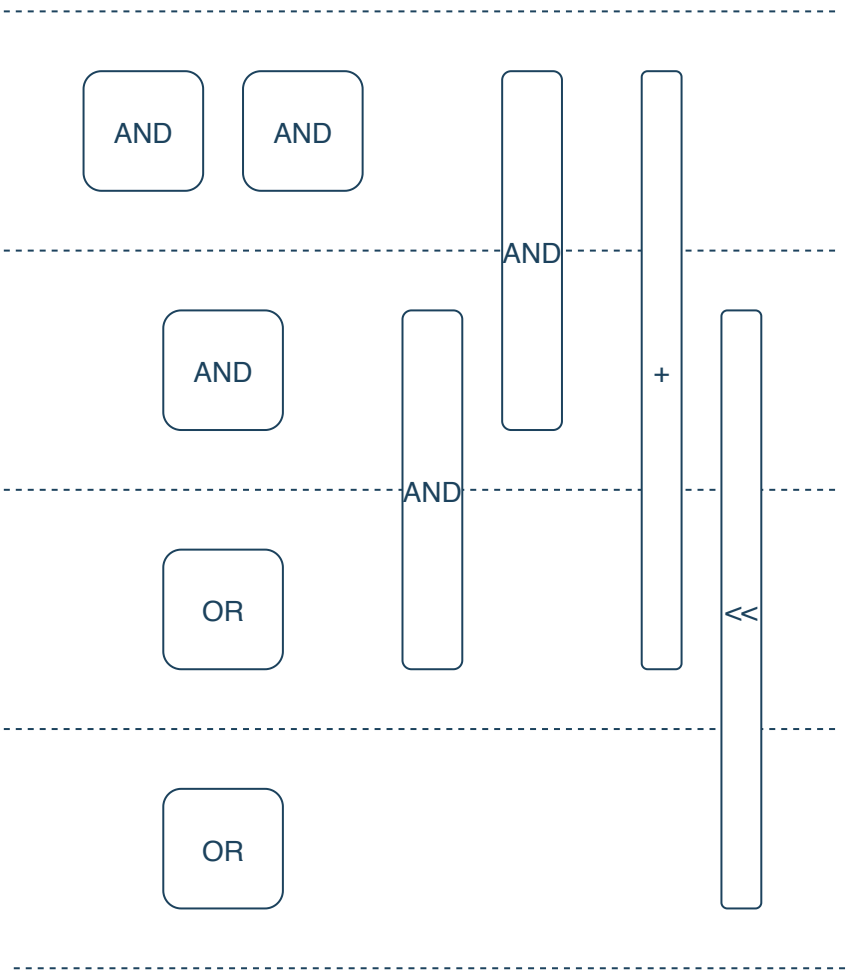

Figure 5.4. FDS time frames of DFG in Figure 5.1

5.5 Miscellaneous scheduling algorithms

In this section, the miscellaneous scheduling algorithms are able to optimize a larger space of constraints, for example both resource and timing constraint. Obviously, the performances are not always as good as the ones that can be obtained using a more specific algorithms (TC and RC scheduling algorithm). However the solutions scheduled are often very close to them. The algorithm able to consider more than one constraint is preferred to find out the optimal solution. Reaching a good trade-off implementation. The Scheduling Algorithms among this family are the

Force Directed List Scheduling, and the *System of Difference Constraint* Based, discussed in the following.

5.5.1 Force Directed List Scheduling algorithm

The *Force Directed List Scheduling* Algorithm is able to solve the scheduling problem taking into account both timing and resource constraints. In details, it takes the best aspects from the *List Scheduling* and *Force Directed Scheduling* algorithms. The priority ready list is no more determined by the operator's mobility, but from the force of each operator as done in Force Directed Scheduling. The difference of FDLS from FDS, is that there isn't a target execution time, and the algorithm tries to minimize the control steps, given resource constraints.

The Algorithm 5 follows the main phases like the list scheduling, modifying the way of how the priority list is computed and filled.

Algorithm 5: Force Directed List Scheduling Algorithm

1 ASAP Computation;
2 ALAP Computation;
3 Mobility computation of each operator to evaluate the possible time frames. For each time frame is associate the probability (Uniform probability);
4 Distributed graph and Force computation for each operator;
5 Filling the list of ready operation whit all forces.;
6 Sort the list whit the lowest force of operators on the top;
7 Scheduling following the priority ready list taking into account the resource constraints;

A more powerful scheduling algorithm [21] can be obtained chaining the FDS and FDLS algorithms. Thus, at the beginning, the FDS is applied, setting a target execution time, and obtaining an architecture that is optimized from the timing point of view, with a balance between the load of the operators and, consequently, the hardware costs. Then, working in a smaller design space, the FDLS algorithm is performed, trying to decrease the execution time while the resource constraints are also optimized

in order to reduce the area.

5.5.2 System of Difference Constraint based scheduling algorithm

The heuristic scheduling algorithms seen so far, are not able to find the optimal scheduling, because they are prone to approximations. Instead, for large problems the optimal scheduling algorithms are often too slow.

The *System of Difference Constraint Based* scheduling algorithm can solve most of the previous problems. A different approach to the problem is introduced, solving a mathematical system where all the dependencies and constraints are described, relying to a mathematical framework called System of Difference [22].

Since we used a mathematical environment, we can used several constraints, spanning from the data dependencies, resource constraints, timing constraints, I/O Buses, to the clock frequency specification.

Given a connected graph as depicted in Figure 5.5, the formulation is as in Equation 5.3. In particular, t_u and t_v are respectively the scheduling time of the source node and the destination node; C is the transition time between t_u and t_v.

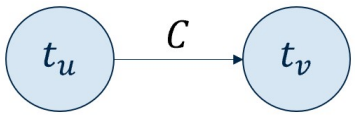

Figure 5.5. Set of difference constraints

$$t_v - t_u \geq C \tag{5.3}$$

An example of the SDC system is shown: given the algorithm 6, we produced the graph as in Figure 5.6. Furthermore, from the graph all the

data dependencies are easily visible. The system 5.4 reproduces the graph 5.6 inside the framework SDC.

Algorithm 6: Example code

1 $C = A \cdot B$
2 $D = A + C$
3 $E = C \cdot D$

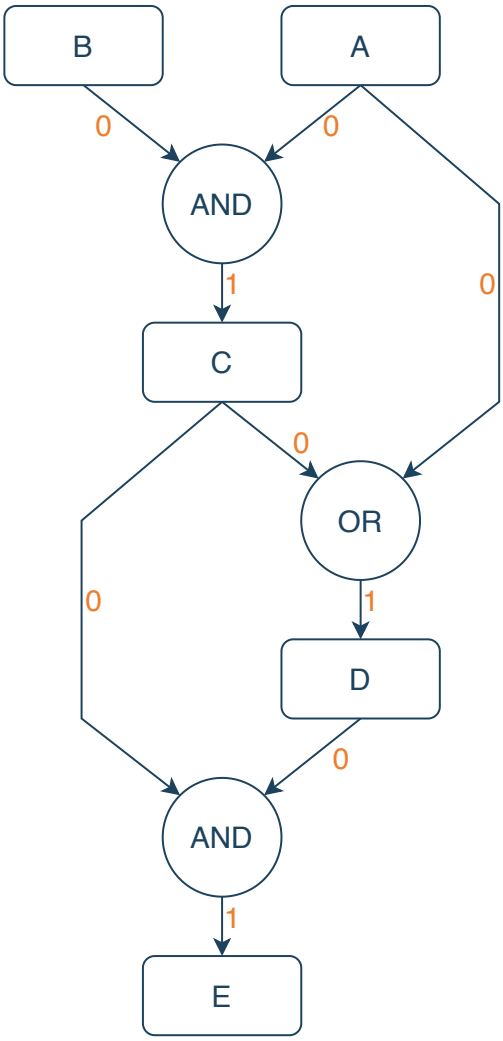

Figure 5.6. Graph SDC example

$$\begin{cases} t_{AND1} - t_B \geq 0 \\ t_{AND1} - t_A \geq 0 \\ t_{OR} - t_A \geq 0 \\ t_C - t_{AND1} \geq 1 \\ t_{OR} - t_C \geq 0 \\ t_D - t_{OR} \geq 1 \\ t_{AND2} - t_C \geq 0 \\ t_{AND2} - t_D \geq 0 \\ t_E - t_{AND2} \geq 1 \end{cases} \tag{5.4}$$

Since the Constants are integers, the matrix of the system is *totally unimodular*[2]. Taking into account also that a totally unimodular matrix has the property that every square submatrix has a determinant 0, -1 or 1, the system can be solved in *linear programming* (LP) [30, 31, 32] and in polynomial time, obtaining the scheduling result as timing integer values.

The results of the SDC system provides the scheduling time for each operator. Moreover, the SDC scheduling is very flexible because it allows to obtain the ASAP, as well as the ALAP scheduling. Setting as objective the minimum as in Equation 5.5, the ASAP scheduling time will be generated; vice versa, setting as objective the maximun as in Equation 5.6, the ALAP scheduling time will be generated.

Other constraints can be set, consider [22] for further options.

$$min \sum_i op_i \tag{5.5}$$

$$max \sum_i op_i \tag{5.6}$$

[2]A unimodular matrix is a square matrix with integer values having determinant +1 or -1.

5.5.3 Modulo SDC based scheduling algorithm

The input code given into the HLS, written in a high level coding language, often makes extensive use of *for* loops. To enable optimizations on these iterative constructs, we introduced the *Modulo SDC Based Scheduling Algorithm*, [33, 34]. The modulo scheduling is a kind of scheduling that takes care to apply the loop pipelining together with other optimizations. This scheduling is able to execute different loop iterations in parallel, increasing the parallelism and performance.

In a modulo scheduling there are several difficulties to overcome. In addition to the data dependencies, we introduced a new data dependency called *cross-iteration* dependency, and it occurs when there is a dependency between two different loop iteration. This kind of dependenciy can affect heavily the performances. The cross-iteration dependencies do not allow to start a new loop iteration in parallel, until the data required by the dependency is ready. Moreover, the modulo scheduling has to meet the resource constraints, because it could allocate a lot of resources trying to execute the loop iterations in parallel.

Modulo scheduling arranges the different operations in a way that can be repeated at a fixed iteration time (called *Initiation Interval* (II)) and avoiding all the problems deriving from the data dependencies, cross-iteration dependencies, and resources constraints. For example, if the II is equal to one, the architecture is able to start a new loop iteration at each control step. Instead, if the II is equal to three, the architecture begin a new loop iteration at every three control step. It's easy to understand that, ideally at every control step a new loop iteration should start to obtain the maximum performances.

In order to reach the maximum performances, the modulo scheduling algorithm must be able to schedule the instructions with the lowest II possible, considering that the II is bounded by constraints and dependencies.

To find the Minimum Initiation Interval (MII) [35], the *Minimum Recurrence Constrained Initiation Interval* (recMII) and the *Resource Constrained Minimum Initiation Interval* (resMII) must be calculated. RecMII

is a limitation caused by the cross-iteration and data dependencies. Instead, ResMII is a limitation resulted by the resource constraints.

The recMII can be found following the Equation 5.7: for every loop recurrence i, two quantities have to be evaluated, i.e. the $delay_i$ to cover the loop itself, and the value that indicates how many iterations of the loop separate the cross-iteration dependency, called just $distance_i$ of the recurrence.

$$recMII = max \left\lceil \frac{delay_i}{distance_i} \right\rceil \tag{5.7}$$

The resMII equation reported in 5.8, verifies the limitation due to the resource constraint, where $\#ops_i$ is the number of times that an operator is required inside the loop iteration, while $\#FU_i$ is the number of available resources of that operator. This is performed for every resource type i.

$$resMII = max \left\lceil \frac{\#ops_i}{\#FU_i} \right\rceil \tag{5.8}$$

The scheduling is feasible if the II is greater than or equal to the MII. MII is the minimum allowed Iteration Interval that guarantees the scheduling to obtain correct results, so as in Equation 5.9.

$$MII = max(resMII, recMII) \tag{5.9}$$

To implement the modulo SDC scheduling, the SDC dependency equations must be modified, introducing the information about the loop recurrence, as in Equation 5.10. After that the scheduling can start to work, and in particular it tries to schedule the loop using the MII precomputed. If it fails to find a feasible scheduling, then the MII will be incremented and the scheduler repeat the scheduling process.

$$(t_v + MII) - t_u \geq C \tag{5.10}$$

The modulo SDC scheduling uses the backtracking technique to consider also the resource constraints. The resource constraint introduces a non linearity to the linear SDC system, and the backtracking represents a workaround to ensure this linearity. Thus, during the execution of the algorithm when the modulo SDC scheduling can not schedule the instructions due to unavailable resources, it un-schedules them by means of the backtracking, and then attemps a new scheduling, modifying the SDC system. In [34], the algorithm related to the modulo SDC scheduling can be found.

An example of a modulo SDC scheduling is provided, not related to the LiM principle but useful to understand how the modulo SDC scheduling works.

In Figure 5.7 a graph that contains a loop dependency is depicted. Here, it is considered that the latency of the memory to perform a load is equal to 2, and to perform a store is equal to 1. The first example shows the scheduling of the loop graph in Figure 5.7 when the memory has three ports and we can read and write in parallel from it, so for this example 1 there isn't any resource constraint.

The MII is computed in order to set the loop constraint to the SDC system 5.11.

$$recMII = max\left\lceil \frac{delay}{distance} \right\rceil = max(\tfrac{3}{1}) = 3$$

$$resMII = max\left\lceil \frac{\#ops}{\#FU} \right\rceil = max(\tfrac{3}{3}) = 1$$

$$MII = max(resMII, recMII) = 3$$

$$\begin{cases} t_C - t_A \geq 2 \\ t_C - t_B \geq 2 \\ t_D - t_C \geq 0 \\ (t_A + MII) - t_D \geq 1 \end{cases} => \begin{cases} t_A = 0 \\ t_B = 0 \\ t_C = 2 \\ t_D = 2 \end{cases} \tag{5.11}$$

The solution of the system 5.11 is computed by using a LPsolve[3]and

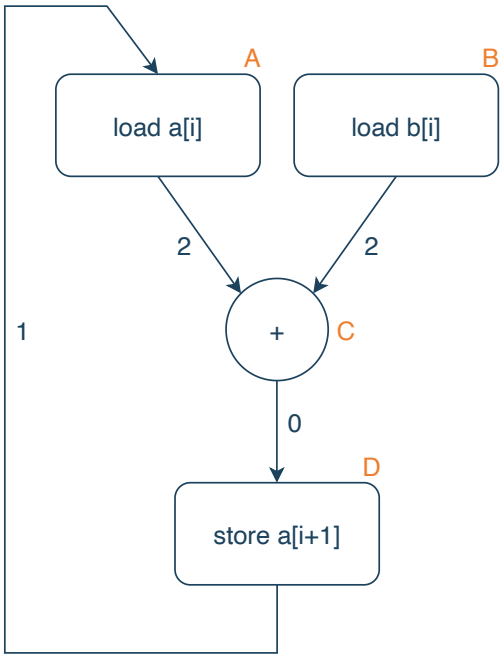

Figure 5.7. Loop Graph — modulo SDC example

it is represented graphically in Figure 5.8, where we can easily see that every three cycles a new loop iteration is executed.

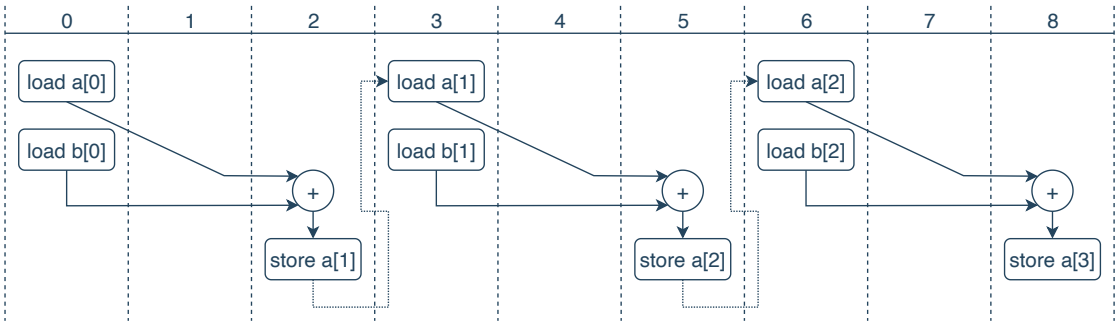

Figure 5.8. Scheduling — modulo SDC example 1

Instead, if it is decided that the memory can have only one input/output port as resource constraint, the $\#FU$ of the equation useful to compute

[3]LPsolve is a Mixed Integer Linear Programming (MILP) solver.

the $resMII$ become equal to 1, as follow:

$$recMII = max\left\lceil \frac{delay}{distance} \right\rceil = max(\frac{3}{1}) = 3$$

$$resMII = max\left\lceil \frac{\#ops}{\#FU} \right\rceil = max(\frac{3}{1}) = 3$$

$$MII = max(resMII, recMII) = 3$$

$$\begin{cases} t_C - t_A \geq 2 \\ t_C - t_B \geq 2 \\ t_D - t_C \geq 0 \\ (t_A + MII) - t_D \geq 1 \end{cases} => ...backtracking... => \begin{cases} t_A = 2 \\ t_B = 0 \\ t_C = 4 \\ t_D = 4 \end{cases} \tag{5.12}$$

Before finding a feasible solution, the algorithm tries several times to apply the backtracking technique to un-schedule the unworkable scheduling of the graph. The solution is shown in Equation 5.12 and depicted in Figure 5.9.

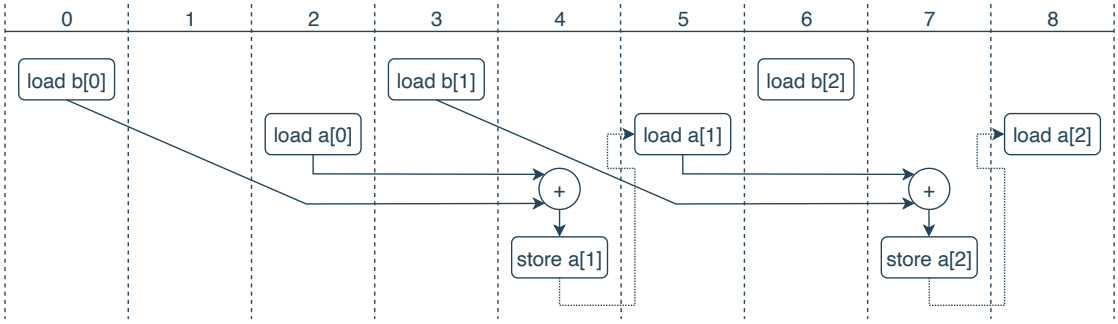

Figure 5.9. Scheduling — modulo SDC example 2

Looking at Figure 5.9, a new loop iteration begin every three cycles, even if the latency of this scheduling corresponds to two more cycles, however, as a matter of performances, this is not important as it would be for the throughput.

Chapter 6

The optimizations introduced on Octantis

Octantis is a software recently born, and it presents several areas that can be improved and optimized. The areas of focus for the present thesis work are the ones corresponding to the *Code Generator* and to *Scheduling* phase, as outlined in 3.1. Together with these specific improvements, also other parts of the source code of Octantis have been maintained, fixing bugs and introducing general optimizations to increase the capabilities and performances of the tool.

6.1 Octantis code generator

The first version of Octantis was compliant with an older version of the DExIMA Input Reference Language, now obsolete. In fact, the software DExIMA has recently received a major update [7], where a new imput interface has been designed.

Accordingly, the code generator of Octantis has been rewritten following the new guidelines of DExIMA Input Reference Language.

Recalling, the code generation stage, depicted in Figure 3.1, is responsible to generate the RTL architecture of the final description from the elaborated information in the previous steps.

6.2 Octantis scheduling algorithm

As seen in Chapter 3, one of the most important phases of an HLS is the scheduling algorithm. Since Octantis implemented only an unconstrained ASAP algorithm, it has been decided to expand its capabilities, introducing a new scheduling algorithm. In particular, the objective was to enforce the abilities of the scheduling, providing a larger design space during the elaboration and advanced management of data dependencies. Furthermore, also the introduction of the possibility to set some meaningful constraints suitable for a LiM architecture has been considered.

The first version of Octantis was able to schedule according to the ASAP scheduling, but was not conceived to work with graph data structures. Thus, the first thing to do was to redesign the ASAP scheduling. The Intermediate Representation once acquired, is saved in a directed graph as data structure, as in this way the elaboration of the data dependencies becomes easier and it allows to apply several graph strategies/analysis.

Thanks to the ASAP scheduling, can be obtained the fastest possible unconstrained architecture. To obtain the slowest possible architecture, the ALAP scheduling was developed, mainly used by more elaborated scheduling algorithms.

Before choosing the best scheduling algorithms that are able to exploit the main characteristic of a LiM architecture, among those described in Chapter 5, it is better to describe the capabilities that the scheduler should have.

During the design, in a LiM architecture the possibility to define the memory size is important. So, the first resource constraint that can be chosen in Octantis is the memory size. In addition the possibilities to manage loop recurrences should be taken into account in order to exploit

the *for* loops written in C language.

The most promising scheduling algorithms for a LiM architecture purpose were revealed:

- FDLS scheduling algorithm;
- FDLS used in combination with FDS scheduling algorithm;
- SDC based scheduling algorithm

They allow to constrain resource and timing. At the end, the choice fell on the SDC based scheduling algorithm with some adjustments, as for its extreme flexibility in the introduction of further constraints consistent with continuous evolution of the LiM technology. Furthermore, it allows to manage cross dependencies between different loop iterations reaching very good quality of scheduling performances.

To summarize, the scheduling algorithms implemented in Octantis, and that can be chosen by the user, are:

- ASAP;
- ALAP;
- SDC based with memory constraint: Fixed size;
- SDC based with memory constraint: Upper limit size;
- SDC based: Optimal -> performance + size;

SDC based scheduling algorithms allow to give in input as constraint the memory size. The constraint memory size indicates the desired dimension of the LiM. The three algorithms use this constraint in a different way from each other, in order to obtain the best scheduling according to the designer objectives. These three share the first part of the algorithm, where first of all they compute the ASAP and ALAP by means the SDC framework. Once they are obtained, the scheduling calculates the mobility

for each operator, filling a list of ready operations. When all mobilities are computed, they will be sorted with the lowest mobility on the top. From this point, the three algorithms follow a different algorithm. In the subsequent sections the three SDC based algorithms are described.

SDC based with memory constraint: fixed size

The scheduling algorithm "*SDC based with memory constraint: Fixed Size*" allows to obtain a scheduling that uses as much as possible the whole memory space, even if there are operators that are still available. The scheduling start to reuse the operators just allocated when the memory is totally allocated. If the scheduling become infeasible, the scheduler tries to extend the execution by adding a new constraint, and then it executes the scheduling process again from the beginning (ASAP, ALAP, mobility list and etc...). The algorithm is depicted in Figure 6.1.

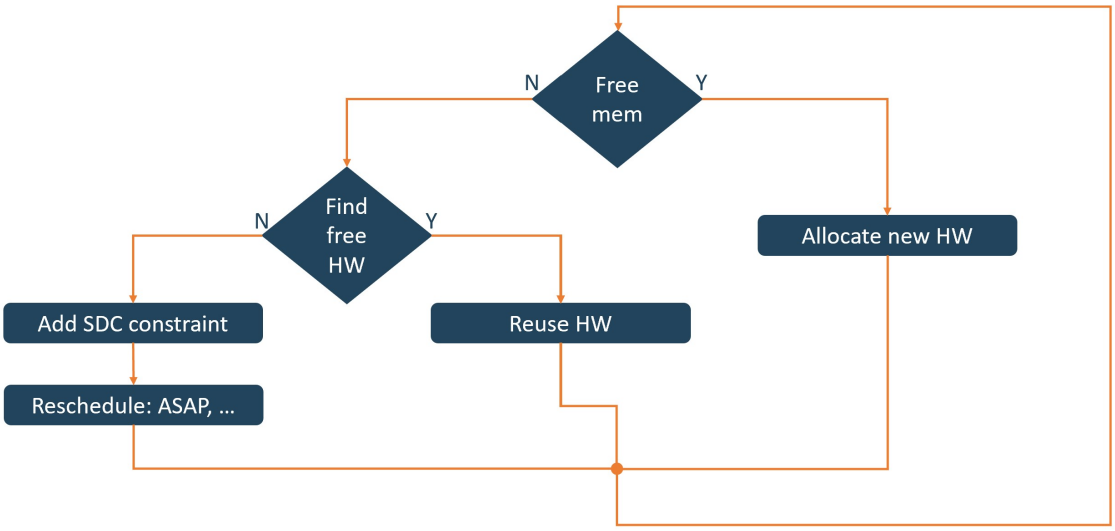

Figure 6.1. SDC based with memory constraint: Fixed size - Scheduling algorithm

SDC based with memory constraint: upper limit size

The scheduling algorithm "*SDC based with memory constraint: Upper Limit Size*" has a different approach, compared to the previous one. In fact, before allocating a new hardware resource, it verifies if there are some allocated free resources. In this way, this algorithm tries to use the memory as little as possible. If there are not free resources, the scheduler allocates a new memory row, until the memory is full. If it happens that the scheduling is infeasible, the scheduler adds a new SDC constraint in order to extend the execution time, and avoids to use more times an operator in the same time step. The algorithm just described is shown in Figure 6.2.

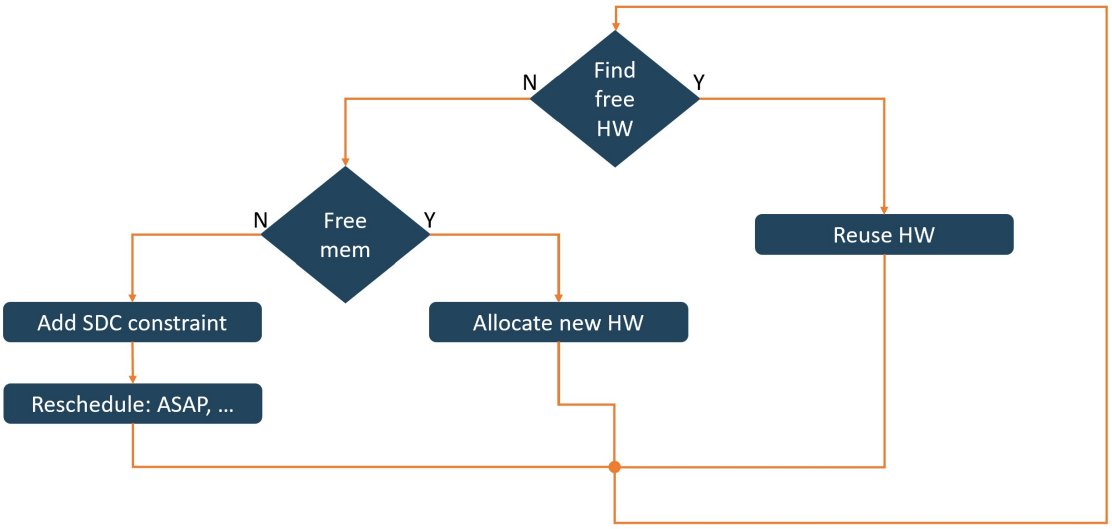

Figure 6.2. SDC based with memory constraint: Upper Limit size - Scheduling algorithm

SDC based: Optimal -> performance plus size

The "*SDC based: Optimal -> performance + size*" generates a scheduling where there is not the memory size constraint, but the objective consists in obtaining the best possible scheduler in terms of area and performances. The algorithm constantly check if there are free hardware operators to be reused. The algorithm is depicted in Figure 6.3.

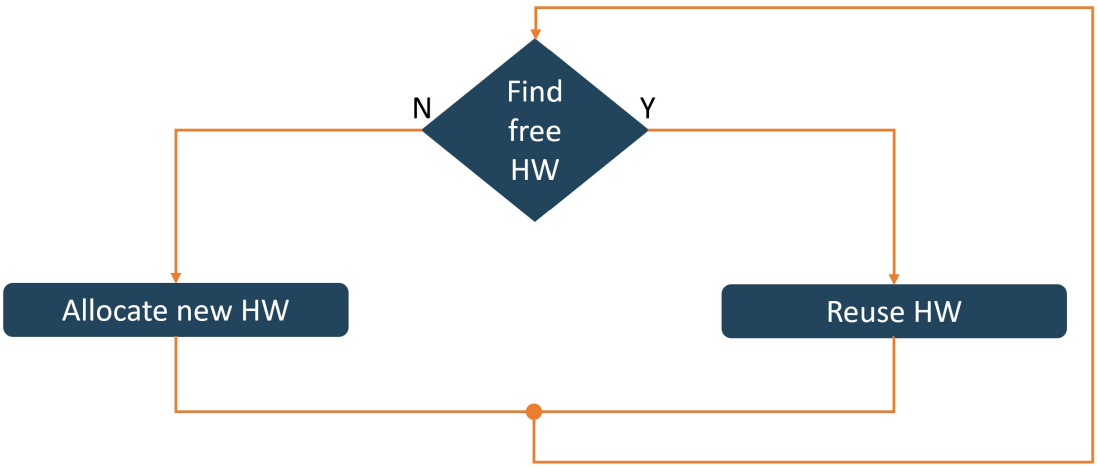

Figure 6.3. SDC based: Optimal -> Performance + Size - Scheduling algorithm

Chapter 7

Tests on the Scheduling Algorithms

To prove the effectiveness of the proposed scheduling algorithms, several tests have been conducted. In particular, it has been checked if the output architectures correctly implemented the input algorithms. Some algorithms have been developed for debug purpose in order to check the correctness of data dependencies analysis, instead others have revealed useful to have a benchmark with the previous version of Octantis, where the capabilities of the new scheduling algorithms can be put into evidence. The correctness of the obtained results has been verified by *inspection*.

7.1 Test algorithm 1 with data dependencies

The first test algorithm with data dependencies (Listing 7.1) implements an algorithm for debug purpose, where simple logic operators, with different data dependencies are present. The related graph is pictured in Figure 7.1.

```
1 void algorithm1()
2 {
```

```
3   //Allocation of the data
4   unsigned A, B, C, D, E;
5
6   C = A & B;
7   D = A | C;
8   E = C & D;
9 }
```

Listing 7.1. Test algorithm 1 with data dependencies

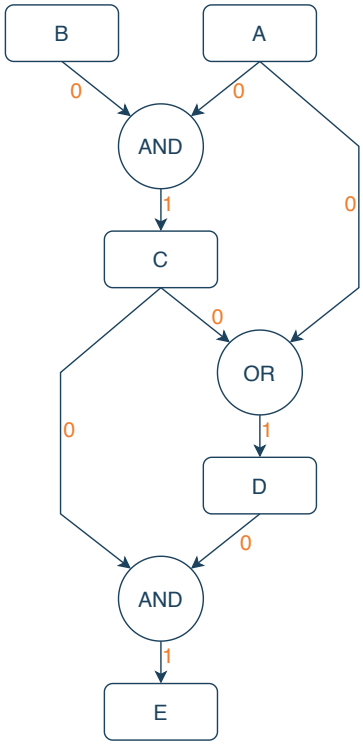

Figure 7.1. Graph test algorithm 1

In Table 7.1 the results of all the newly implemented scheduling algorithms are compared. The three SDC scheduling options are able to reuse (reducing the size memory of 40% with respect to the results derived from the ASAP) the memory rows allocated without any performance loss. In particular, for this algorithm two LiM rows are enough: one that is enriched with AND gates and the other with OR gates. An additional simple memory row is required to save the result.

	ASAP (ref)	ALAP	SDC – Fix:4	SDC – Fix:3	SDC – UP:5	SDC – Opt
#MemRows	5	5	4	3 (-40%)	3 (-40%)	3 (-40%)
#ClkSteps	4	4	4	4 (+0%)	4 (+0%)	4 (+0%)

Table 7.1. Test algorithm 1 results

7.2 Test algorithm 2 with data dependencies

Also the second test algorithm Listed in 7.2, with different data dependencies, has been considered for debug purposes, where the simple logic operators are not fully constrained with null mobility. The related graph is pictured in Figure 7.2.

```
1  void algorithm2()
2  {
3    //Allocation of the data
4    unsigned IN1, IN2, IN3, IN4;
5    unsigned P1, P2, P3, P4, P5;
6
7    P1 = IN1 & IN2;
8    P2 = P1 & IN3;
9    P3 = P1 | IN4;
10   P4 = P2 ^ P3
11   P5 = ~ P2
12 }
```

Listing 7.2. Test algorithm 2 with data dependencies

	ASAP (ref)	ALAP	SDC – Fix:8	SDC – Fix:6	SDC – UP:5	SDC – Opt
#MemRows	9	9	8	6 (-33%)	5 (-44%)	6 (-33%)
#ClkSteps	4	4	5	5 (+25%)	5 (+25%)	4 (+0%)

Table 7.2. Test algorithm 2 results

In this case, the three SDC scheduling options, compared to the ASAP scheduling, are able to considerably reduce the memory size, paying a little penalty as drawback in terms of the time required to execute the whole algorithm. However, if the user want to obtain an architecture characterized by the maximum performance, the memory reduction is still high,

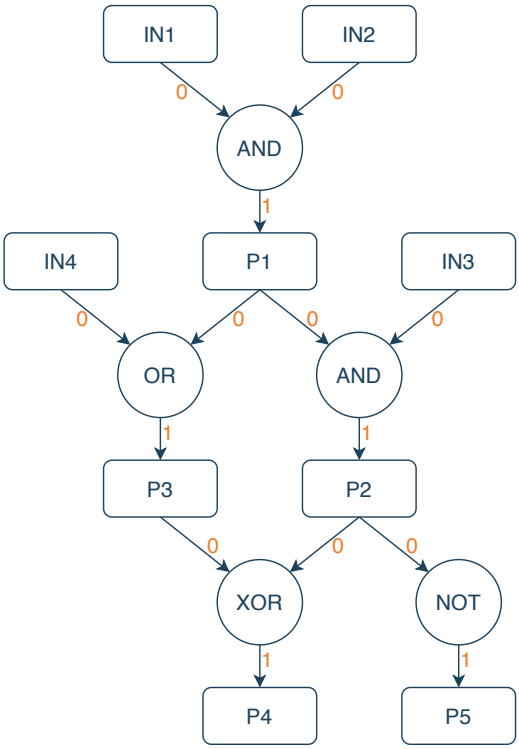

Figure 7.2. Graph test algorithm 2

approximately of -33%, compared to the results derived from the ASAP scheduling. The output LiM architecture is made of both LiM memory cells including all the necessary operators (AND, OR, XOR, NOT), together with a simple memory row.

7.3 XNOR Net for an approximated CNN

To compare the new scheduling algorithms with a benchmark, the tests used to verify the first version of Octantis have been used again. We recall that the first version of the tool allowed to compile the input algorithm only according to the ASAP scheduling.

This test uses, as input algorithm the Listing 7.3, a circuit in *XNOR Net for an approximated CNN* [5] as explained in [8].

```
//Code for the implementation of a XNor Net
void XNOR_Net(){

    //Allocation of the weight
    unsigned weight;
    //Allocation of the matrix for the input data
    unsigned dataMatrix[5];
    //Allocation of the rows for the output results
    unsigned outData[5];

    //Execution of the Xor operations on the data
    for(int i=0;i<5;++i)
        outData[i] = ~ (weight ^ dataMatrix[i]);

}
```

Listing 7.3. XNOR Net input algorithm

	ASAP (ref)	ALAP	SDC – Fix:6	SDC – UP:6	SDC – UP:5	SDC – Opt
#MemRows	11	11	6	6 (-45%)	5 (-55%)	10 (-9%)
#ClkSteps	2	2	3	3 (+50%)	4 (+100%)	2 (+0%)

Table 7.3. XNOR Net algorithm results

The results of the XNOR Net algorithm, reported in Table 7.3, shows the flexibility of the different SDC scheduling applied. Comparing the results to the ones obtained through the ASAP scheduling, a reduction of 55% of the memory size can be achieved, at the cost of doubling the required clock cycles to execute the algorithm. This means that the SDC scheduling allows the designer to minimize the memory size if the timing constraint is not the main issue; otherwise the designer can find a good trade-off between the memory size and the execution time. In table 7.3, as well as in the previous results, only the most relevant memory size cases are shown.

7.4 CLiMA CNN

The benchmark Listed in 7.4 is a CNN algorithm, which implements a *Quantized Convolutional Neural Network* exploiting the capabilities of a

CLiMA architecture [4].

```
//Code for the implementation of the CNN algorithm
void CNN(){

    //Allocation of the LiM rows for the pixels
    int pixels[9];
    //Allocation of the map for the input weights
    int weights[9];
    //Allocation of the vector for the partial results
    int partial[9];
    //Allocation of the row for the result of accumulation
    int result=0;

    //Definition of the shift operations and computation
    //of the partial results
    for(int i=0; i<9; ++i)
        partial[i]=pixels[i] >> weights[i];

    //Accumulation of the generated partial results
    for(int j=0; j<9; ++j)
        result+=partial[i];

}
```

Listing 7.4. CLiMA CNN algorithm

	ASAP (ref)	ALAP	SDC – Fix:34	SDC – Fix:13	SDC – UP:10	SDC – Opt
#MemRows	35	35	34	13 (-63%)	10 (-71%)	10 (-71%)
#ClkSteps	10	10	10	10 (+0%)	10 (+0%)	10 (+0%)

Table 7.4. CLiMA CNN algorithm results

The properties of the architectures generated by the different scheduling algorithms are shown in Table 7.4. The results are very interesting because the SDC based scheduling is able to save a huge part of the memory, even up to 71% compared to the ASAP scheduling. For example, the SDC – Optimal is able to use just 3 Shift LiM, 2 Adder LiM, and 5 simple memory rows; while the ASAP scheduling based implementation of the [4] algorithm needed 9 Shift LiM and 6 Adder LiM.

Chapter 8

Conclusion and future works

Octantis is a tool with a lot of potential. It aids to design a Logic-in-Memory architecture, avoiding design errors, speeding up the overall process, and in addition reducing the effort to verify the generated architecture.

Octantis is an HLS tool able to translate an input algorithm implemented in C language into a digital circuit architecture. The focus of this thesis was to improve the quality of the output architecture, through scheduling algorithms with higher performances.

In particular, while the previous version of Octantis implemented only an unconstrained ASAP scheduling algorithm, the new scheduling algorithms studied and implemented in this work are based on the SDC mathematical framework, introducing the size memory as resource constraint. In this way the LiM architecture generated, noticeably improved its performance in terms of performance and area, and its memory size has been reduced.

The overall reduction of the memory size up to 50% was achieved at the worst case cost of doubling the clock steps necessary to complete the

execution. The possibility to reduce the memory size allows the designer to find the best trade-off between the occupied area and the performance of the digital circuit.

In future, the areas of improvement where Octantis could be benefit are the following:

- introducing a wider customization from the configuration file, allowing the designer to choose the scheduling algorithm;
- development of an alternative *code generation* module in order to have the possibility to generate the VHDL/Verilog, useful on two fronts. The first one is to have the description architecture in a standard HDL, the second one is to consent using the commercial tools to test in depth the output architecture.

Instead, improvements related to the scheduling phase could be:

- integrating in Octantis some technological libraries of the logic gates containing information like the timing, the area, and the power. Allowing to expand the scheduling by including timing constraints (i.e. critical paths, frequency, execution time);
- improving the binding module in order to fully benefit the new scheduling potential.

Nomenclature

ALAP As Late As Possible

AM Array Multiplier

API Application Programming Interface

ASAP As Soon As Possible

CiM Computation-in-Memory

CLiMA Configurable Logic-in-Memory Architecture

CMOS Complementary Metal-Oxide Semiconductor

CnM Computation-near-Memory

CPU Central Process Unit

CwM Computation-with-Memory

DExIMA Design Explorer for In Memory Architectures

DFG Data-Flow Graph

ESL Electronic System-Level

FDLS Force Directed List Scheduling

FDS Force Directed Scheduling

FSM Finite-State Machine

HDL Hardware Description Language

HLL High Level Language

HLS High-Level Synthesis

II Initiation Interval

IR Intermediate Representation

ISA Instruction Set Architecture

JIT Just-In-Time

LiM Logic-in-Memory

LS List Scheduling

LUT Lookup Table

MII Minimum Initiation Interval

OpenMP Open Milti-Processing

OS Operating System

RC Resource Constraint

RCA Ripple Carry Adder

recMII Minimum Recurrence Constrained Initiation Interval

resMII Resource Constrained Minimum Initiation Interval

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SDC System of Difference Constraint

SSA Static Single Assignment

TC Time Constraint

WAR Write After Read

WAW Write After Write

Bibliography

[1] M.D. Godfrey and D.F. Hendry. "The computer as von Neumann planned it". In: *IEEE Annals of the History of Computing* 15.1 (1993), pp. 11–21. DOI: 10.1109/85.194088.

[2] R. Stanley Williams. "What's Next? [The end of Moore's law]". In: *Computing in Science Engineering* 19.2 (2017), pp. 7–13. DOI: 10.1109/MCSE.2017.31.

[3] Giulia Santoro. "Exploring New Computing Paradigms for Data-Intensive Applications". PhD thesis. Politecnico di Torino, 2019. URL: http://hdl.handle.net/11583/2737673.

[4] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. "New Logic-In-Memory Paradigms: An Architectural and Technological Perspective". In: *Micromachines* 10.6 (2019). ISSN: 2072-666X. DOI: 10.3390/mi10060368. URL: https://www.mdpi.com/2072-666X/10/6/368.

[5] Andrea Coluccio, Marco Vacca, and Giovanna Turvani. "Logic-in-Memory Computation: Is It Worth It? A Binary Neural Network Case Study". In: *Journal of Low Power Electronics and Applications* 10.1 (2020). ISSN: 2079-9268. DOI: 10.3390/jlpea10010007. URL: https://www.mdpi.com/2079-9268/10/1/7.

[6] Milena Andrighetti et al. "Data Processing and Information Classification—An In-Memory Approach". In: *Sensors* 20.6 (2020). ISSN: 1424-8220. DOI: 10.3390/s20061681. URL: https://www.mdpi.com/1424-8220/20/6/1681.

[7] Loris Mendola. "DExIMA A synthesis tool and performance estimator for Logic-in-Memory architectures". MA thesis. Politecnico di Torino, 2021. URL: https://webthesis.biblio.polito.it/17852/.

[8] A. Marchesin. "Octantis - A High-Level Explorer for Logic-in-Memory architectures". MA thesis. Politecnico di Torino, 2020. URL: http://webthesis.biblio.polito.it/15852/.

[9] Alfred V Aho et al. *Compilers: Principles, techniques, and tools second edition*. Pearson Education Addison Wesly New York, 2007. ISBN: 0321486811.

[10] Wikipedia. *Compiler*. URL: https://en.wikipedia.org/wiki/Compiler.

[11] Yao-Wen Cheng Wang Laung-Terng Chang and Kwang-Ting. *Electronic Design Automation - Synthesis, Verification, and Test*. Elsevier, 2009. ISBN: 978-0-12-374364-0.

[12] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[13] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. URL: https://llvm.org/pubs/2002-12-LattnerMSThesis.html.

[14] LLVM Language Reference Manual. *LLVM Documentation*. URL: https://llvm.org/docs/LangRef.html.

[15] Clang Compiler User's Manual. *Clang Documentation*. URL: https://clang.llvm.org/docs/UsersManual.html.

[16] LLVM's Analysis and Transform Passes. *LLVM Documentation*. URL: https://llvm.org/docs/Passes.html.

[17] The LLVM Target-Independent Code Generator. *LLVM Documentation*. URL: https://llvm.org/docs/CodeGenerator.html.

[18] Razvan Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[19] S. GOVINDARAJAN. "Scheduling Algorithms For High-Level Synthesis". In: *Technical paper* (1995). URL: https://ci.nii.ac.jp/naid/10014961176/en/.

[20] P.G. Paulin and J.P. Knight. "Algorithms for high-level synthesis". In: vol. 6. 6. 1989, pp. 18–31. DOI: 10.1109/54.41671.

[21] P. G. Paulin and J. P. Knight. "Scheduling and Binding Algorithms for High-Level Synthesis". In: *Proceedings of the 26th ACM/IEEE Design Automation Conference*. DAC '89. Las Vegas, Nevada, USA: Association for Computing Machinery, 1989, pp. 1–6. ISBN: 0897913108. DOI: 10.1145/74382.74383. URL: https://doi.org/10.1145/74382.74383.

[22] J. Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation". In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006, pp. 433–438. DOI: 10.1145/1146909.1147025.

[23] Daniel D Gajski et al. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[24] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System synthesis with VHDL*. Springer Science & Business Media, 2013.

[25] R.A. Bergamaschi et al. "Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 5.1 (1997), pp. 82–100. DOI: 10.1109/92.555989.

[26] B. Ramakrishna Rau. "Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 63–74. ISBN: 0897917073. DOI: 10.1145/192724.192731. URL: https://doi.org/10.1145/192724.192731.

[27] S. Katkoori and R. Vemuri. "Scheduling for low power under resource and latency constraints". In: *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 2. 2000, 53–56 vol.2. DOI: 10.1109/ISCAS.2000.856256.

[28] S. Amellal and B. Kaminska. "Scheduling of a control data flow graph". In: *1993 IEEE International Symposium on Circuits and Systems*. 1993, 1666–1669 vol.3. DOI: 10.1109/ISCAS.1993.394061.

[29] Stephanie Soldavini, Sonia López Alarcón, and Marcin Łukowiak. "Using Reduced Graphs for Efficient HLS Scheduling". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9181274.

[30] G. Ramalingam et al. "Solving Systems of Difference Constraints Incrementally". In: *Algorithmica* 23.3 (Mar. 1999), pp. 261–275. ISSN: 1432-0541. DOI: 10.1007/PL00009261. URL: https://doi.org/10.1007/PL00009261.

[31] *LP Solve*. URL: http://lpsolve.sourceforge.net/5.5/.

[32] *LP Solve API Reference*. URL: http://lpsolve.sourceforge.net/5.5/lp_solveAPIreference.htm.

[33] Zhiru Zhang and Bin Liu. "SDC-based modulo scheduling for pipeline synthesis". In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 211–218. DOI: 10.1109/ICCAD.2013.6691121.

[34] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. "Modulo SDC scheduling with recurrence minimization in high-level synthesis". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927490.

[35] B. Ramakrishna Rau. "Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 63–74. ISBN: 0897917073. DOI: 10.1145/192724.192731. URL: https://doi.org/10.1145/192724.192731.