Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Programa de Maestría en Ingeniería Electrónica

A methodology for the synthesis to logical netlist of an ASIC

para optar por el título de
Magister Scientiae en Ingeniería Electrónica
énfasis en Sistemas Empotrados

con el grado académico de
Maestría

Mauricio Gurdián Murillo

Cartago. Abril 26, 2017

I declare that this thesis document has been made entirely by my person, using and applying literature on the subject, and introducing my own knowledge and experimental results.

In the cases I have used literature, I proceeded to indicate the sources by the respective references. Accordingly, I assume full responsibility for this thesis work and the content of this document.

Mauricio Gurdián Murillo

Cartago. April 26, 2017.
Céd.: 5 0366 0140

Instituto Tecnológico de Costa Rica
Electronics Engineering School
Master's Thesis
Evaluation Committee

Master's Thesis presented to the Evaluation Committee as a requirement to obtain the Master of Science degree from the Instituto Tecnológico de Costa Rica.

Evaluation Committee Members

M.Sc. Pablo Mendoza Ponce
Reviewer

M.Sc. Jeferson González Gómez
Reviewer

M.Sc. Jorge Castro-Godínez
Director

The members of the Evaluation Committee certify that this Master's Thesis has been approved and that fulfills the requirements set by the Electronics Engineering School.

Cartago. April 26, 2017

Abstract

The advances in technology for manufacturing ASICs allow more features to be added. As result, and depending on the architecture of the ASIC, more functional blocks do exist to support such additional features. This imply requiring more resources to synthesize each functional block into a logical netlist.

As the physical design process is completed by a third party, reducing the time to deliver the complete set of synthesis files is critical for the project, so that the engineers can start the quality checks of each netlist earlier than the schedule, and the final product can be both completed and released on schedule.

This work describes a methodology that automatically executes the synthesis flow of RTL code to logical netlist on each block that forms an ASIC. It helps keeping a better traceability of changes through the milestones in a project.

A simulator of the methodology was implemented in `Perl` to validate that the complete synthesis runtime of an ASIC is improved, compared against a serial flow approach. Consequently, the time to synthesize the complete set of functional blocks is speedup 8.8 times.

Keywords: ASIC, Register Transfer Level, logical netlist, synthesis

Resumen

Los avances en la tecnología en la fabricación de ASICs permiten que más características sean agregadas. Como resultado, y dependiendo de la arquitectura del ASIC, más unidades funcionales existen para respaldar tales características. Esto implica que más recursos sean necesarios para sintetizar cada unidad funcional en un netlist lógico.

Como el proceso de diseño físico es completado por un tercero, reducir el tiempo de entrega del conjunto completo de archivos de síntesis es crítico para el proyecto, de modo que los ingenieros puedan iniciar las revisiones de calidad en cada netlist con anterioridad al programa, y que el producto final pueda ser a la vez completado y entregado en la fecha prevista.

Este trabajo describe una metodología que ejecuta automáticamente el flujo de síntesis del código RTL a netlist lógico sobre cada bloque que forma un ASIC. Ayuda a mantener una mejor trazabilidad de cambios a través de las fechas clave de un proyecto.

Un simulador de la metodología fue implementado en `Perl` para validar que el tiempo total de síntesis de un ASIC es mejorado, comparado contra un enfoque de flujo en serie. Consecuentemente, el tiempo para sintetizar el conjunto completo de unidades funcionales en un chip es mejorado hasta 8.8 veces.

Palabras clave: ASIC, Transferencia a nivel de registros, netlist lógico, síntesis.

I dedicate this work to my girlfriend Raquel Arias, whose words of encouragement kept me in the path for finishing this master thesis. I also dedicate this work to my parents, who have supported me throughout the process. I also dedicate this work to my teammates from the physical group at HPE, their words of encouragement are also appreciated.

Acknowledgments

I am deeply grateful to my thesis mentor Jorge Castro. Without both his help and guidance, this work would not have been finished. Furthermore, I would like to thank the Research and Development management team at Hewlett-Packard Enterprise Costa Rica, for encouraging to take and finish the master thesis.

Mauricio Gurdián Murillo

Cartago, April 26, 2017

Contents

List of Figures

List of Tables

List of abbreviations

Abbreviations

ASIC	Application Specific Integrated Circuit
RLS	RTL to Layout Synthesis
RTL	Register Transfer Level
VCS	Version Control System
VLSI	Very Large Scale Integration

Chapter 1

Introduction

The advance in circuits manufacturing has enhanced the density of transistors for a given area, allowing the design of larger chips with even more features. However, time to market does not increase in correlation with the size of the chip. As part of that market necessity, creating an Application Specific Integrated Circuit (ASIC) begins with the system level design and the microarchitecture planning. Within Integrated Circuits (IC) companies, management propose requirements to fit into an industry segment.

Alternatively, in an academic organization, a research team extracts a set of requirements for their project based on their research goals. With that in mind, Figure 1.1 shows a high level flow that summarizes the process of creating an ASIC, which begins with the architectural requirements. However, this high level flow only gets up to the logical netlist part as final product, since the remaining steps to complete an ASIC are normally developed by third parties, hence they are out of the scope.

An ASIC is normally divided in several functional blocks. Each block is designed to achieve specific tasks. As the technology enhances the density of transistors for a given area, more features are designed to fit in, so the size of an ASIC increases, as well as the amount of functional blocks that form the ASIC.

As creating the set of logical netlists is the final product, having them ready to be delivered to a third party requires the synthesis of all the main blocks. Each block passes through the synthesizing process which requires a set of CAD licenses that translate RTL description to a gates representation. An ideal scenario is to have a higher or equal amount of licenses than the total amount of synthesizable blocks. However those CAD licenses are expensive, hence the amount of licenses is limited.

In order to synthesize the total amount of functional blocks, it is necessary to have a mechanism that automatically process the synthesis flow for each functional block. This work proposes a methodology to overcome such goal by taking into account the CAD license limitations.

When the agreements of a system design are set, as depicted in Figure 1.1, architectural

requirements are documented to let engineers work on generating the hardware description code for each functional block, but the ASIC may not be fully specified because the features, requirements, and architecture are constantly adjusted on the fly. When this happens, RTL code also changes and the synthesis flow needs to be restarted on any affected functional block, as Figure 1.1 shows it. The team in charge of the synthesis flow is able to restart the synthesis jobs on these affected blocks, making it to be on the critical path for the project when deadlines approach.

By synthesizing the RTL code into actual representations of electronic devices, the RTL model is mapped into a gate-level netlist at a target technology. It could be also defined as the process that takes both RTL model and hardware library components as inputs to create a flattened model, which must be logically equivalent to the RTL model.

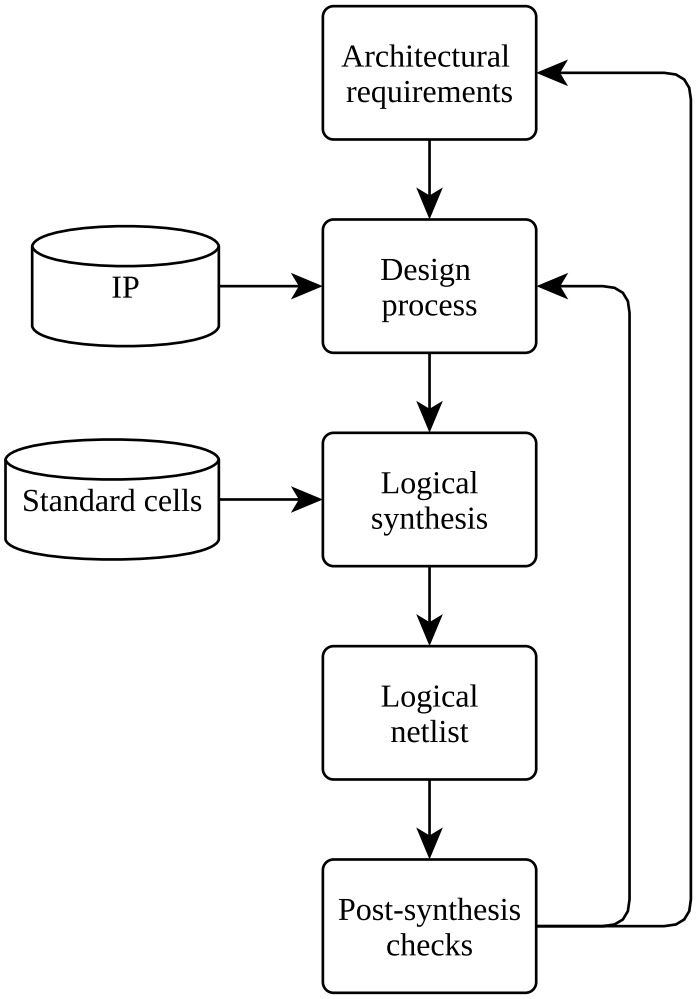

Figure 1.1: ASIC design and synthesis process

It is worthy to mention that this development flow of an ASIC also includes constant feedback between designers and verification engineers, in order to converge into the completion and validation of each main feature of the ASIC. At the same time, an automation

group of engineers develops required tools to gather all the necessary information, so that the ASIC can be assembled physically. This data gathering is changing frequently due to feedback received between validation and design. These tools come in handy to work as interfaces between the project source files and the synthesis CAD tools.

These CAD tools eventually take all the hardware code for every block in order to translate a logic model into a gate model, and provide quality of results, such as estimated design area, timing, and total amount of cells. Normally, they follow a methodology to overcome the complications that involve managing a complex system composed of several logical blocks. These complications arise due to the dependencies among blocks and the limitations that the working infrastructure naturally suffers, such as the amount of available CAD tool licenses and workload resources management. Usage of these CAD tools which automate circuit synthesis have gained high importance, and are widely used in the industry. Some of these commercial tools offer a reference flow, such as the Reference Methodology that Synopsys provides. These scripts are a starting point for developing product-specific flows [7].

Developing those interface tools between the RTL code and CAD tools can be classified into two main approaches. First approach is to have a fully working flow to build an ASIC which normally goes from having the hardware description code, down to the route and placement of each cell, inside the designated area (or floorplan). This kind of flow is normally known as *RTL to layout* flow (RLS flow) [5]. Part of this flow requires specific quality checks such as adhering to timing constraints for each corner case, design rules checks to avoid having short circuits on metal layers, layout to schematic checks to validate that the logic behavior matches the physical hardware, logic equivalence checks to determine that the RTL code matches the physical netlist, and clock domain crossing checks to validate that multiple frequency data paths are in harmony.

Executing each one of the previous quality checks requires having a set of automated tools as well. They are a combination between commercial tools and local scripts, developed by using programming languages such as Perl or Tcl. These tools do both the analysis and data reporting related to the design. Based on such reports, failing or missing items are detected, such as timing issues or missing expected library components. Having a completed layout requires many RLS flow iterations to fix every remaining issue.

The second approach is related to the case where a third party is hired to do the "placement and route" part of the flow. Unlike the RLS process, the workflow is only composed of the RTL designing process for the ASIC and the synthesis processes for each module, in order to create sets of logical netlists which are then delivered to the third party so that they finish remaining RLS stages. The quality checks mentioned above are also performed on these logical netlists. They are called logical netlists because they don't have any physical boundaries. Yet they are still pseudo-physical netlist, but more logical oriented since connection paths are ideal. These netlists models are then taken as inputs to the remaining RLS stages (floorplanning, placement, clocking, routing), plus all the vendor standard cells and IP databases, to create preliminary versions of ASIC's layout.

Information given as part of the deliverable to the third party, normally includes a summary of design statistics (such as gate and RAM count and timing information for each data path group) in order to keep track of these metrics through milestones. Feedback about the quality checks is eventually received. Any existing issue should be fixed so that the next deliverable is healthier than the later.

1.1 Objectives and document structure

The main purpose of this work is to propose a methodology to automatically execute the individual synthesis flow on a full set of functional blocks that form one or multiple ASICs, in order to deliver a set of logical netlists by reducing the total time for completion.

To achieve the main goal, and in order to execute the synthesis on a functional block, a framework is responsible of managing and serving each of the execution stages, as described in Section 2.1. This framework implementation executes all the different stages of the workflow, which guarantee establishing efficient pre-synthesis stages, by reducing the introduction of human errors, and providing dynamism to the design flow [1]. Based on the individual synthesis flow, a new flow is developed in order to execute multiple individual synthesis in order to have every block of a chip eventually synthesized.

The challenge is to find an optimal way about how to manage the workspace environment, as well as tool's licenses, repositories, and handle execution order of blocks based on their runtime. To that end, the following objectives need be accomplished:

1. Design an execution flow for multiple synthesis jobs for all modules that form an ASIC.
2. Adapt a constraint mechanism to handle synthesis tool's licenses.
3. Propose a repository management for handling ASIC synthesis results.
4. Propose an issue criteria order for the synthesis of each functional block.
5. Evaluate execution-time metrics with a post-process for complete synthesis and compare it against a sequential synthesis.

Figure 1.2 shows the dependency diagram of the above objectives. At first half of this investigation, only three most left items boxes were developed. Among these objectives no dependencies exist, but they are mandatory in order to achieve the intended execution flow for synthesizing all modules that form one or more ASICs. These three independent objectives are found in Section 3. Despite they are independent, they are also mandatory in order to obtain a mechanism for executing the synthesis flow on each of the blocks in an ASIC, to achieve tracking, execution order and efficient runtimes. On the second half of this research, the execution flow on an ASIC was developed and tested by a program

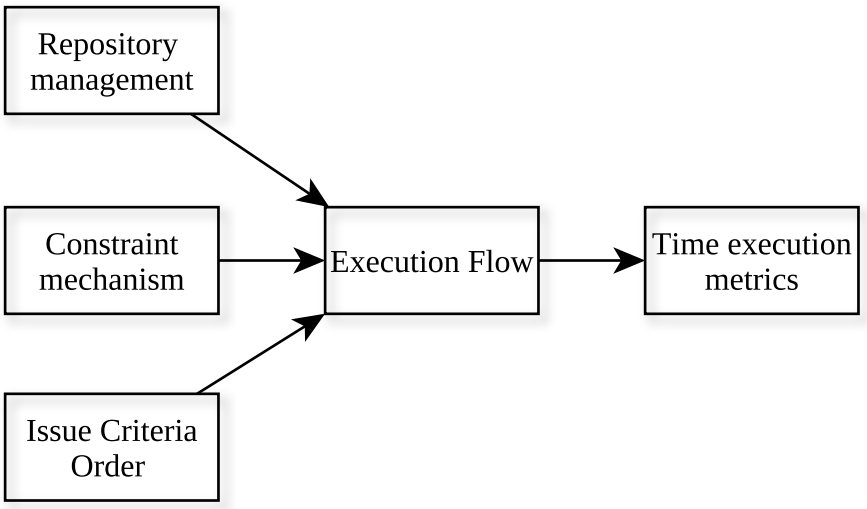

Figure 1.2: Methodology dependency workflow

which simulates the creation of the workspace, the pre-synthesis stages runtime, and the synthesis runtime, for each block that forms an ASIC.

Chapter 2 presents in detail both of the synthesis approaches: individual synthesis at section 2.1 and a full ASIC synthesis approach in section 2.2. Chapter 3 details each of the three independent objectives on different sections that can be found at section 3.1 for data management, section 3.2 explains how to handle jobs, and the dispatch order in section 3.3. Finally the explanation of the complete execution flow is found at section 3.4. Chapter 4 explains how the execution flow was programmed to obtain synthesis time metrics between different sort mechanisms in a chip. Finally, chapter 5 presents the analysis of the obtained results.

There probably exist methodologies that also explain an ASIC synthesis flow, which are used by different tech companies, so it is likely that due to confidential matters these methodologies are unknown. A method to generate the synthesis of all blocks on one or more ASICs was not found in the researching of the state of the art, since this is more an industry workflow.

Chapter 2

Synthesis methodologies

Completion of the RTL design of an ASIC will not be accomplished in one pass. Verification will find logic and timing errors. Designers have to fix these errors by changing their RTL code, and hence the netlist. This netlist uses the standard cells and library components to create a first physical representation of the design. RTL code is an abstract description of logic operations for circuits using hardware description languages, such as Verilog or VHDL.

Logical netlists contain the gates from the standard cell library. This library is a collection of generic logic gates and different transistor sizes in a specific process technology. Logic synthesis uses the standard cell libraries to convert the RTL model to specific standard cells model. Standard cells provide the generic logic functions, such as inversion, NAND, Flip Flops, etc. The standard cells are used in both the logic design and place and routing design. Design and characterization of standard cells determine the quality of the design.

With the increase in complexity on VLSI designs, more features are added and there is a higher density of transistors in areas which are decreasing as the technology is enhanced. As ASICs become larger, hierarchical flows come in handy. A hierarchical flow is the methodology which partitions the chip into functional blocks. The advantage of using this methodology is that each block will be smaller. It is a bottom-up procedure so that optimization and debug on each block are easily handled. On the other side, this methodology requires having even more design resources and CAD tools licenses for managing each of the design stages for multiple blocks.

Due to the fact that similar methodologies are probably confidential since they are related to the industry, and also because they require licenses for CAD tools which might be expensive, this work proposes a methodology to synthesize all main blocks that form an ASIC. Section 2.1 explains the synthesis process for an individual block, based on a previous work [1]. Section 2.2 provides the explanation of how the individual synthesis processes are handled together in order to generate the gate level models for all the blocks that form an ASIC.

2.1 Individual synthesis methodology

Logic designers need to verify that they are achieving requirements, and converging to buildable parts of the chip. This imply achieving specific timing requirements to verify that the data is meeting the system clock period; size in area so that designs fit in the estimated die area. Each person in charge will eventually synthesize their modules to determine the size and correctness in physical terms. To achieve this, a set of automated tools run the synthesis flow.

Basically, each essential part of the per module synthesis flow is considered a "stage". This works as a serialized flow: the output files from a previous stage are the input arguments to the following stage, as shown in Figure 2.1. Results for each stage are created based on their configuration files and their corresponding input files. This workflow allows adding new stages as needed, and each stage can be customized according to specific requirements.

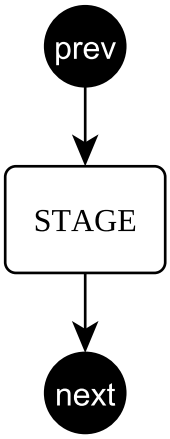

Figure 2.1: An execution stage

Each stage has self-revision mechanisms to ensure both that all input files actually exist, and also that obtained results are error free. Based on the framework explained with detail on [1], and as shown in Figure 2.2 the per-module synthesis flow stages are:

- Simulation stage: All the files that belong to a module are pre-compiled and syntactically revised. The output of this stage is a report which contains the list of all the input files required.

- RTL gathering stage: Based on the reports generated by the previous stage, all the required source files that form each module are gathered into an RTL source directory. With this, the synthesis flow has a module-unique RTL input folder separated from the pool of code. This provides traceability of the code that eventually generates the synthesis results.

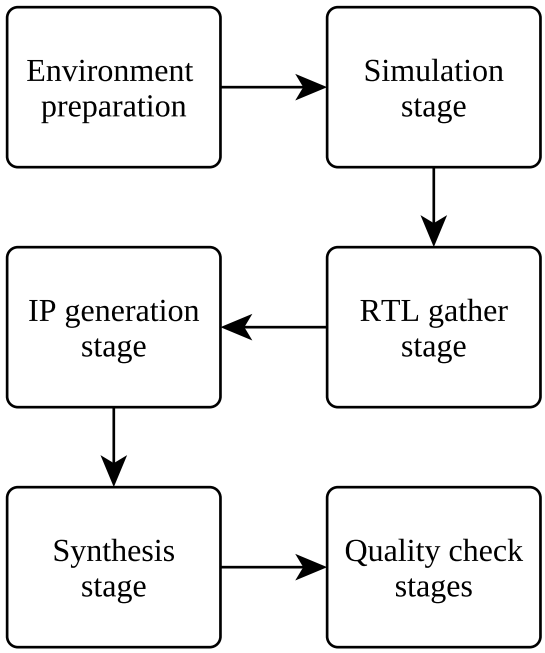

Figure 2.2: Synthesis flow

- Intellectual Property (IP) generation stage: Since the third party uses specific sub-blocks of IP, such as memories when they build and finish the modules in the following RLS flow stages, it is required to have approximate models of these IP blocks, so that the synthesis tools assemble each IP block with pre-built models, generated by this stage. Usage of pre-built models helps to reduce synthesis runtime and they improve output results such as timing. This stage is in charge of generating those pre-built models, which are added into the RTL stage source directory. Thus, next stage synthesizes by using pre-built IP models instead of behavioral RTL models.

- Synthesis stage: This stage finally takes all previous results from each stage to start synthesizing each module. Commercial synthesis CAD tools available are offered by Synopsys, Cadence, Xilinx, and Altera. Open-source frameworks for Verilog RTL synthesis tools are also available, such as Yosys [3]. Design compiler from Synopsys is the framework of choice when the synthesis stage is executed [1]. The final output of this stage is the logical netlist, as well as a set of quality reports.

- Quality checks stages: After synthesis, RTL and netlist models have to be logically verified. RTL source files produced during the RTL stage are compared against the flattened model (netlist). There should not be any mismatches between designs. Another quality check is the clock domain crossing check. If the module has multiple clock domains, there should not be any synchronization issue when crossing the boundaries between clock domains.

Stages ranging from environment preparation up to IP generation are called pre-synthesis

stages.

In order to prepare the synthesis workspace, an auxiliary framework is in charge of creating the physical environment for the current module. This environment has the directories with source files for each stage of the synthesis flow.

There is another framework in charge of executing the source files for each stage of the synthesis flow, to generate output results [1]. This framework executes stage by stage whenever the previous stage has finished successfully. Upon failure, it ends the flow execution and reports the error. The framework has a mechanism to report the execution duration for each stage to track metrics of how long the current module takes to execute each of the synthesis flow stages. This framework is designed in such a way that reports are saved in the corresponding stage directories.

2.2 Full ASIC synthesis methodology

All the modules that form the ASIC need to be synthesized. Synthesizing each of the modules one by one would involve a great effort. It could also lead to making mistakes in the process that can be critical for the project. Automating this process is helpful to handle huge amounts of data for an ASIC, since it is normally composed of anywhere between ten and one hundred modules. Furthermore, this improves the confidence in the design team since it assures quick completion of the ASIC synthesis so that the overall results are reviewed timely, for each build iteration.

Large projects are normally managed under version control systems to take advantage of the multiple options that they offer to create branches, as Figure 2.3 shows for instance. Each branch recreates a picture, or a snapshot, of the whole data in a specific timestamp and it allows to measure both the project progress and the quality of data.

In an ASIC project, when a milestone approaches, a branch is created and linked to the milestone. If the synthesis build results are finished satisfactorily, this branch is then declared as the milestone branch. Every netlist model and its metrics are then delivered to the third party that completes the following RLS flow stages.

Working with snapshots is useful since each snapshot represents all the data at a specific time. This data includes every RTL model and every tool for that specific moment. Based on that data, the individual synthesis methodology is executed on each module. Eventually, all ASIC core modules are synthesized.

The moment a snapshot is declared a milestone snapshot, its data is then pushed into the main repository. By doing this, the official snapshot data for that milestone is saved. This is done to backup milestone data. Based on the saved milestone snapshot, a new set of results can be recreated without further issues. Every tool and RTL model was already saved and they exist for that milestone epoch.

Furthermore, if synthesis is not successfully completed for a certain module (due to RTL

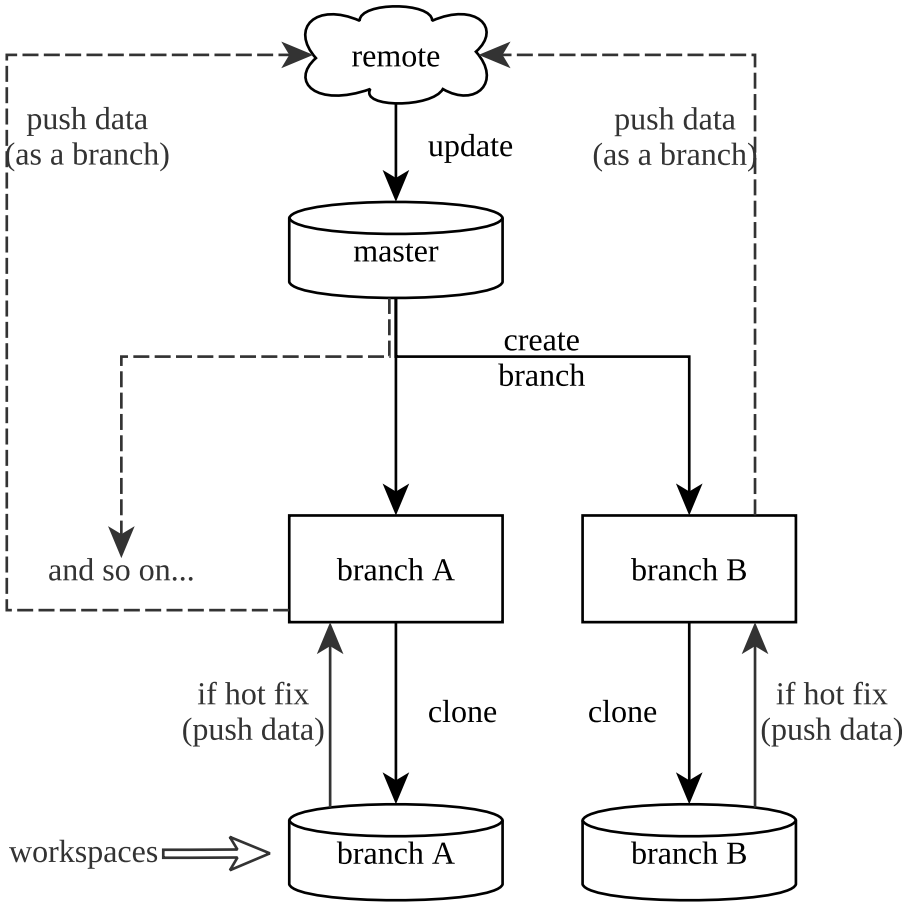

Figure 2.3: Snapshot workflow

modeling issues, for instance), a patch fix (or hot fix) can then be included into the RTL source directory to execute the individual synthesis process for the broken module. If the hot fix actually settles the synthesis, related fix files need to be pushed into the branch that is linked to the snapshot directory. Then, if this branch is the milestone snapshot, the hot fix should also be pushed into the milestone branch, as Figure 2.3 summarizes.

Chapter 3

Methodology for synthesizing a chip

There are two potential options to execute the synthesis flow for each module. First, a loop that executes the synthesis flow framework module by module, serially. This methodology might work efficiently only if the total number of modules is a small (between 5-10 modules for instance). And each module is relatively small, so the synthesis process does not take much time (several minutes for each module). In this case, in less than a few hours the synthesis for all the blocks is ready. This approach results productive in cases where resources limitations exist, such as very limited amount of CAD licenses, or lack of high end processing machines.

A drawback with that methodology is that it does not work effectively in cases where larger numbers of blocks exist. Modern ASICs are bigger and more complex, and the total time to deliver a complete milestone drop would be the sum of the time that each individual module takes to pass throughout the synthesis flow stages. For instance, a big block can take up to 24 hours to finish synthesizing. And other 10 blocks can take one hour each to complete their synthesis. Total time to wait until the entire synthesis of all blocks is complete can take at least 1 day and 10 hours, plus the overhead of the pre-synthesis stages.

Parallelism can be the best approach to run the synthesis for the full set of modules that form a chip, in order to save time and get partial results faster. However, there are many restrictions that arise through the process, like the amount of available CAD tool licenses. Full usage of licenses is avoided, to let other designers to keep working on their module completeness.

Another potential restriction is the licenses starving problem, which can be overcome by using workload management platforms. A commercial solution offered is Platform LSF by IBM [4], or as an open source alternative OpenLava [6]. By using these platforms, management and queuing for synthesis jobs can be achieved. The approach is that whenever a license is released, a new synthesis job could be started.

Based on the individual synthesis work flow defined in section 2.1, and taking advantage of the fact that the flow is divided by stages, a full synthesis methodology can be developed

by running each of the functional blocks in parallel processes.

3.1 Data management under repositories

It is common that in any ASIC projects the related information is stored in data repositories, in order to use version control over the files. A version control system (VCS) records changes to these files over time, so that any version can be recalled later if necessary. The main advantage of using a VCS is that you can easily recover files in case they are broken or lost. It also allows to revert either files or even the project, to a desired previous stage. Furthermore, you can compare changes over time between files, or which user is responsible for eventual modifications.

A popular VCS tool was a system called RCS, which works by keeping patch sets (difference between files), so that it can recreate any file and how it looked like at any point by adding up all the patches. Then there is the Subversion system, also highly known as SVN, which is a version control system designed to be better than CVS (Concurrent Version System). CVS is also a version control system, which can record the history of source files and documents. Perforce is another VCS but proprietary. Its database is pre-configured and self-installed, and it uses a distributed version control model, or it can take a centralized approach as well.

Collaborating with other developers on other systems is a problem to deal. For overcoming that issue, Centralized Version Control Systems were developed. These systems, such as CVS, Subversion and Peforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. The problem of having a centralized approach is that if the server goes down, nobody can collaborate at all or save versioned changes. Then, if the central database is corrupted, and there isn't proper backup, the entire information and history of the project is lost except the data at people's local machines.

On the other hand, there are Distributed Version Control Systems, such as Git, Mercurial, Bazaar, where clients don't just check out the latest snapshot of files. Every user is actually fully mirroring the repository. With this approach, if any server dies, any of the client repositories can be copied back to the server to restore it.

For the purposes of this project, the VCS used is Git. Git stores and thinks about information very different than other systems, such as SVN or Perforce, even though they have similar interfaces. The major difference between Git and other VCS, is the way Git thinks about data. Other systems store information as a list of file-based changes over time. Git thinks its data like a set of snapshots of a miniature file system. Everytime a commit is done, Git takes a picture of all the files at that moment and stores a reference to that snapshot. In order to be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file that it has already stored. Git would be like a stream of snapshots.

The basic GIT workflow goes like this: Files are modified in the working directory. Files are staged, adding snapshots of them to the staging area. Files are committed, which take the files as they are in the staging area and stores that snapshot permanently into the GIT repository.

3.1.1 Data structure

The individual synthesis methodology explained in Section 2.1 was developed to work by stages. This framework is responsible of managing and issuing each of the execution stages, as described in [1]. The complete synthesis methodology for an ASIC as proposed in Section 2.2 should be designed in a way that it can assure a correct management on the project GIT repository data structure, and it has to be able to synthesize each of the major block layouts, by automatically executing the individual synthesis flow on the full set of functional blocks that form one or multiple chips, in order to deliver the final set of logical netlists to the third party, who is in charge of concluding the RLS process.

3.2 Handling synthesis tool's licenses

The framework that executes a synthesis job uses underneath a Load Sharing facility (LSF), such as the platform LSF from IBM. This platform is a software which is industry-leading and enterprise-class that distributes work across existing heterogeneous IT resources to create a shared, scalable, and fault-tolerant infrastructure that delivers faster, more reliable workload performance [2].

LSF provides a resource management framework that takes job requirements, finds the best resources to run the job, and monitors its process. All of this according to host load and site policies.

Some Platform LSF terms definition for an overall understanding:

Job: A job is a unit of work that is running in the LSF system. It is a command that is submitted to the LSF for execution. LSF schedules, controls and tracks the job according to configure policies.

Job Slot: A job slot is a bucket into which a single unit of work is assigned in the LSF system.

Resources: Resources are the objects in the cluster that are available to run work.

cluster: A cluster is a group of computers (hosts) running LSF that work together as a single unit, combining computing power, workload and resources. A cluster provides a single-system image for a network of computing resources. An LSF cluster manages resources, accepts and schedules workload, and monitors all events.

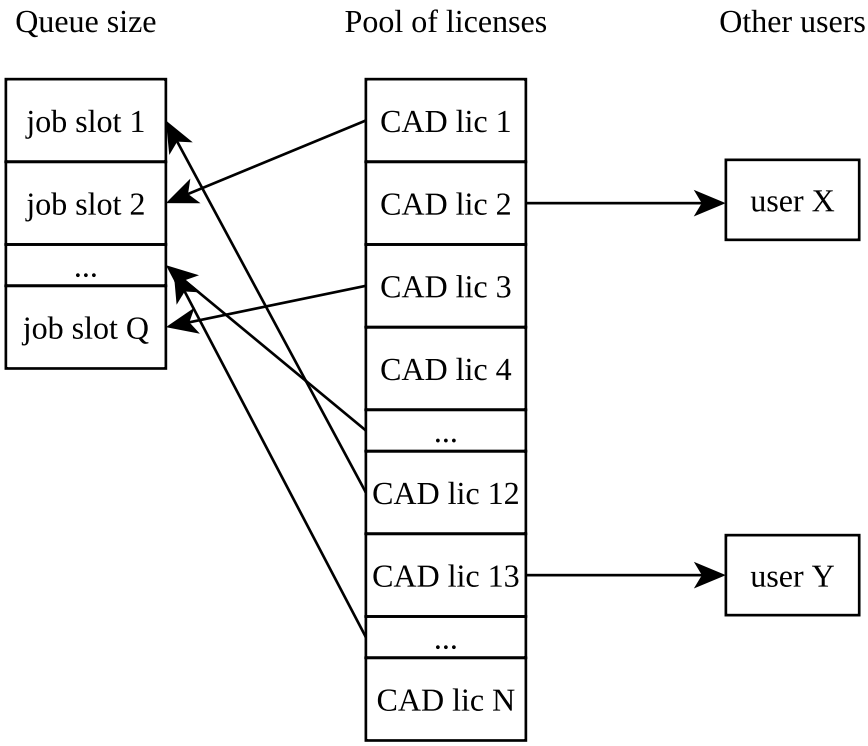

Figure 3.1: CAD license assign to queue and users

queue: A cluster-wide container for jobs is called a queue. All jobs wait in queues until they are scheduled and dispatched to hosts.

The complete synthesis methodology for an ASIC as proposed in Section 2.2 should be designed in a way that it can correctly use the platform LSF for synthesizing all of the blocks that belong to an ASIC, to avoid drying up the amount of CAD tool licenses available for any other user, since each job slot will consume one available CAD tool for synthesizing any block, as shown in Figure 3.1. To deal with that, the usage of a queue that is addressed directly to the methodology is helpful to handle a maximum amount of synthesis jobs. For instance, a queue can be sized to have only 8 job slots, so that CAD licenses are always available from the pool of total licenses when other users require to work on its synthesis experiments.

Another approach instead of using queues is a more aggressive method: Dispatching synthesis jobs as long as CAD licenses are free. Of course, the aggressiveness level can be reduced if scheduling policies are implemented at LSF level (a task normally done by the LSF administrator). Also, when dispatching synthesis jobs from the full ASIC methodology, a buffer needs to be used to limit the number of busy licenses. The buffer insures that a pool of slots and licenses are kept free so that user jobs in other queues can start with no issues.

3.3 Issuing criteria order for synthesis processes

When doing individual synthesis for each major block, the data of how long does synthesis take for each of them is known. Based on that data, bigger blocks (more time consuming) are dispatched first into the LSF. By doing this the total time of a complete synthesis for an ASIC is the sum of the highest block synthesis runtime, plus the remaining runtime for blocks that were waiting for dispatching, for free job slots and free licenses. Overhead from pre-synthesis stages is also taken into account for gathering the total amount of runtime that all ASIC blocks take to fully synthesize.

Table 3.1: Example of synthesis runtime per block (in minutes)

Block name	**CPU time**
MAC	745
SoC	102
Memory controller 1	88
Arbiter 1	85
Memory controller 2	24
Fabric controller	15
Crossbar 1	3
Crossbar 2	2
Crossbar 3	2
Total	1066

Table 3.1 is an example of actual synthesis runtime for blocks that form a Fabric chip. If that list of blocks are dispatched serially, the total amount of time to have all synthesis results completed will be 1066 minutes plus the overhead. In other words, total time to wait for this ASIC to be synthesized by a serial approach is approximately 17 hours.

Instead of dispatching table 3.1 blocks serially, they can be issued in the same order as they are in the table (longest-job first). For example, if MAC block is started first, it will take 12 hours to be ready. At the same time, each of the other blocks are dispatched and serviced as long as free job slots and licenses are free. So assuming a queue of 8 job slots, blocks from SoC up to crossbar 2 are all executed at the same time. In the moment a synthesis jobs finishes, crossbar 3 will be attended, as shown in Figure 3.2.

With this, instead of waiting 17 hours to fully synthesize this example chip, it can be ready in 12 hours at most, saving up to 5 hours of runtime, since the addition of all blocks except MAC (321 minutes in total), is lower than the 745 minutes that it takes to run only MAC block. In other words, runtime is reduced about 30%.

Another approach is to run shortest jobs first. Blocks with lowest runtime are issued first. This issuing method helps in obtaining results faster since smaller blocks finish first. The disadvantage in this case is that the total synthesis runtime for all blocks is delayed by the longest runtime.

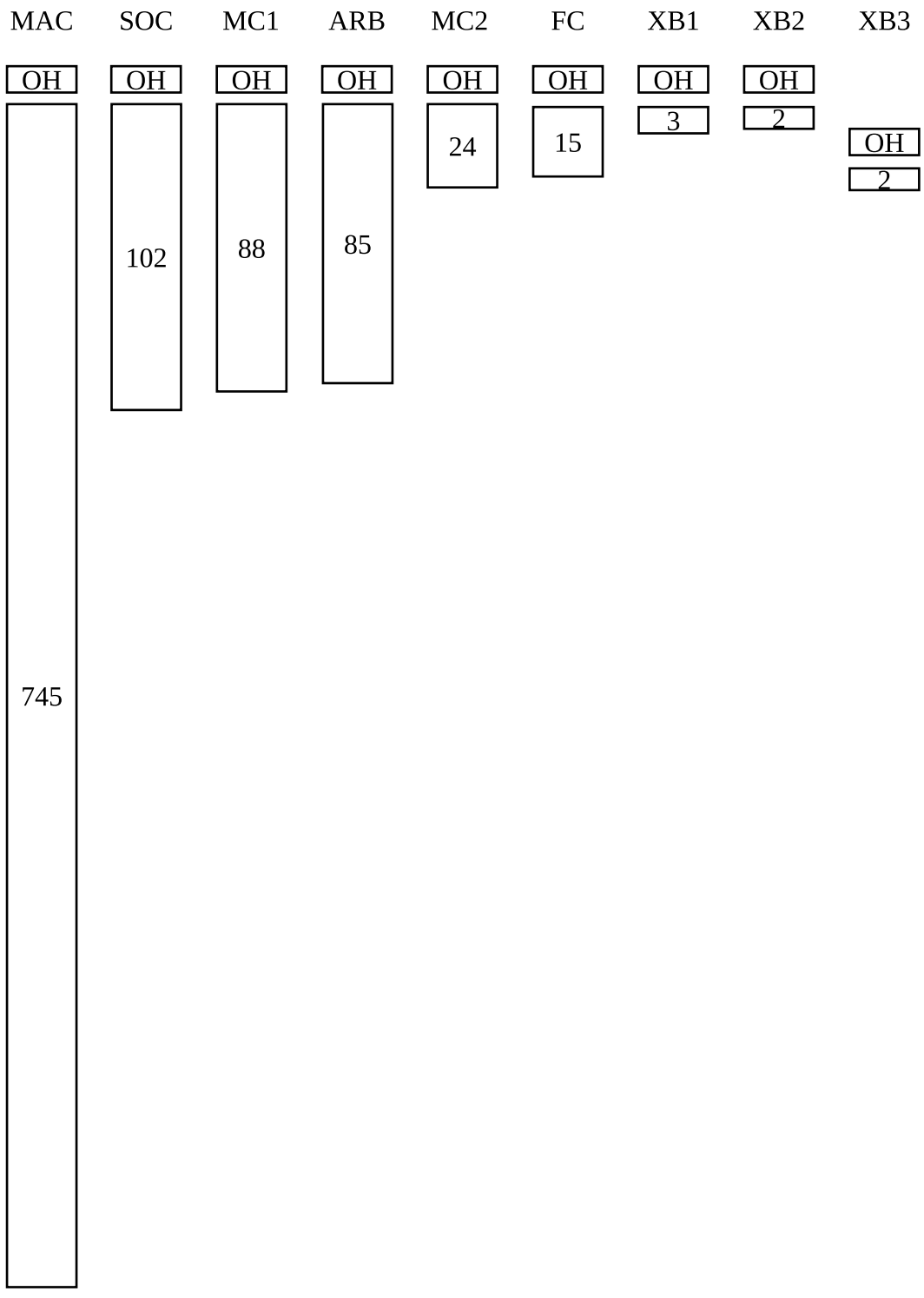

Figure 3.2: Jobs and synthesis time - Longest job first scenario

Continuing with the assumption that the size of the queue is 8 slots, the total synthesis runtime is the sum of the shortest job plus the longest job:

$$XB3 + MAC = 2min + 745min = 747min$$

For this example, issuing by shortest jobs first does not hurt the total synthesis runtime since the smallest blocks are synthesized in a few minutes.

In next chapter a larger chip is used as example. Instead of using a total of 9 block, that chip is formed by 23 major blocks, which their runtimes are bigger than the chip of table 3.1. Furthermore, additionally to the longest and shortest job issuing methods, three randomly sorted list of blocks are tested throughout the methodology to compare them against both the longest and shortest mechanisms.

3.4 Execution Flow

Figure 3.3 summarizes the flow of the methodology. First, all local variables are defined, such as the root directory, the repository directory, the target directory, and so on. Good definition of variables on a program of such complexity is vital to understand it later, and to debug potential issues with the methodology.

A set of options is configured to demonstrate all the possibilities that the methodology program can do. These options include selection flags for which chip is being built, which milestone the construction of the snapshot is related, custom input files to process different blocks from the standard input file. To run only a sub-set of stages (such as either pre-synthesis only, or avoid post-synthesis stages). Anyway, having a proper set of options helps the final user to understand better what to do with the program.

Depending on which chip is built, the program selects the configuration files for it and processes the environment to leave it configured to work on the selected chip.

Once the environment is configured, the next step is to update the main repository. This is done in order to work on all recent changes from everyone involved in the project. After updating the repository, a branch is created based on the main repository. This snapshot is the reference data base to build the entire chip.

Of course, if a branch with the same name was previously created, the methodology should abort since it can't work on a previously existing snapshot. This security check avoids overwriting an existing workspace. As suggestion, the branch name can be attached with the current date, so that every time a snapshot is generated, a completely different name is created.

At this point, the branch exists abstractly only at the main repository. It needs to be cloned into a physical location in order to execute the synthesis flow. This cloning process is executed by the methodology as well.

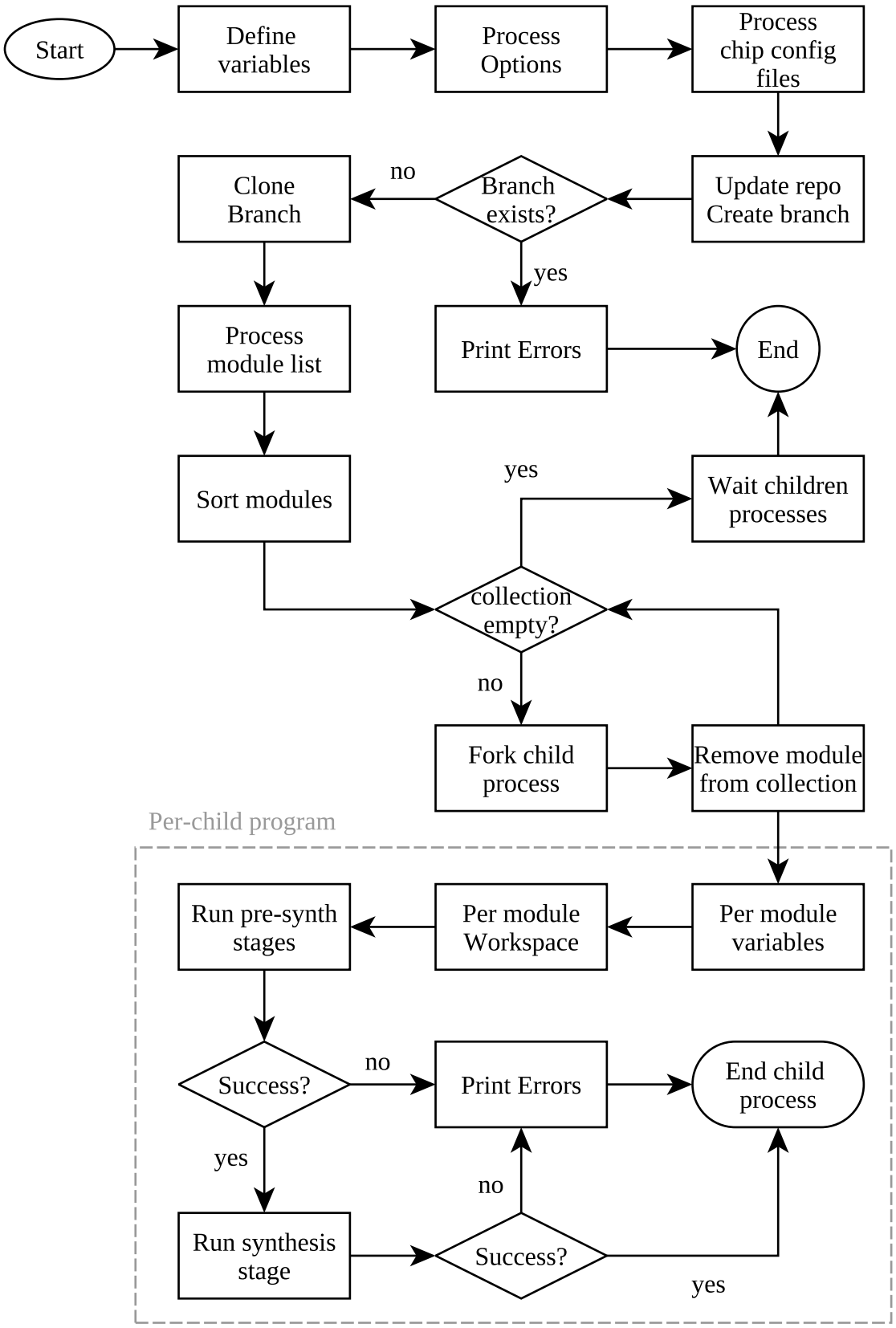

Figure 3.3: Full execution flow

Now that a physical location to work on already exists, it is time to process the list of modules that forms the chip. This list is defined by a configuration file which has the list of modules, with no specific order.

Also, each module has as known data, how much time it takes to pass through the individual synthesis process. So the list of modules is processed then by a sort mechanism which depending on the sort type, it will sort the list of modules based on their synthesis time.

The types of sorting that the mechanism supports are:

- Longest jobs first.
- Shortest jobs first.
- Random sort.

The methodology starts iterating over the sorted list of modules. For each module, a fork system call is used to create a new process running the same program at the same point. The process ID number is returned to the parent process. So that the parent waits for the children process to finish. Eventually, the parent process waits for all the children processes to finish.

The children processes run the *Per-child program* (discontinued grey line box) shown in figure 3.3, which both the per-module and workspace are set. The methodology in this case runs the pre-synthesis stages. If none of the pre-synthesis stages fail, it continues with the synthesis process. When either of the stages fail, the flow will print the Error messages and the child process is then finished. If all stages are completed successfully, the methodology reports success state for each stage and then the child process is finished.

In a real case scenario, all of the pre-synthesis stages for each module are eventually running in parallel. But when the first modules arrive to the synthesis stage, depending on how many slots the LSF queue has as available, the synthesis process are then attended.

Continuing with the assumption that the queue size is eight slots, the first eight jobs to get into the synthesis stage are served. Remaining jobs are queued, and the LSF handles them in a *First In First Out* approach. Eventually, eight synthesis jobs are running in parallel, and the remaining jobs are waiting for available slots. Parent process waits for its children to finish. Once all children processes are done, the parent process terminates the main program.

In this chapter, each section detailed what it is needed to bind the proposed methodology. Section 3.1 explained several tools to handle data under repositories. This is helpful to keep track of changes by using snapshots. Then, Section 3.2 details the platform that handles the workload. Section 3.3 showed by an example how the issue criteria order affects the total synthesis runtime. Finally, Section 3.4 presented the methodology to synthesize all blocks that form an ASIC.

In next chapter, a simulation program is used to validate the proposal. As this methodology involves having actual CAD tools and a Load Share Facility, the program only simulates those synthesis times and LSF overhead times. The three sorting types are tested. Furthermore, different constrained queue sizes to handle synthesis jobs are used.

Chapter 4

Simulator

A simulation tool is developed due to the fact that the real environment cannot be used for academic purposes. The real environment is highly based on Figure 3.3.

This chapter explains the program or solution developed to simulate the proposed methodology. Based on 3.4 section, and as figure 4.1 shows, the program defines all local variables, processes user specific options, processes the module list which is based on a bigger chip, shown at table 4.1. The sorting mechanism takes that module list and based on the sorting type selected by the user, it sorts the modules either on longest jobs first, shortest jobs first, or randomly.

Then, based on that new sorted module list, the program iterates over those modules, creating a new child process for each module that runs the *Per-child program*.

The constrained queue defines how many synthesis jobs can run at the same time. This helps to avoid consuming all the CAD licenses, such as Design Compiler licenses, a tool that synthesizes RTL to standard cells. For instance, if a company only has 20 CAD licenses for synthesis, and all modules from a chip need to be synthesized, having this constrained queue configured at LSF level to only 8 slots as previous examples, allows to run only 8 synthesis jobs at the same time.

After the per module variables have been defined, each child process simulates the pre-synthesis stages (a simple `sleep` function). Then, once that part is completed, the program simulates the synthesis stage (by running again a `sleep` function). The execution of the synthesis stage is done only if the constrained queue is not full. If the queue is already full, the child process keeps polling the size of the queue until a slot is available. Once the constrained queue has an available slot, the synthesis stage is executed (or simulated). Finally, after the module is completed, the child process is ended.

Parent program waits for all of its children to finish. Each child process that arrives to the synthesis stage will take an empty slot from the constrained queue. The size of the constrained queue is defined by the user through the options. Once all available slots are taken, remaining jobs will wait in another queue for an empty slot. To simulate this queue

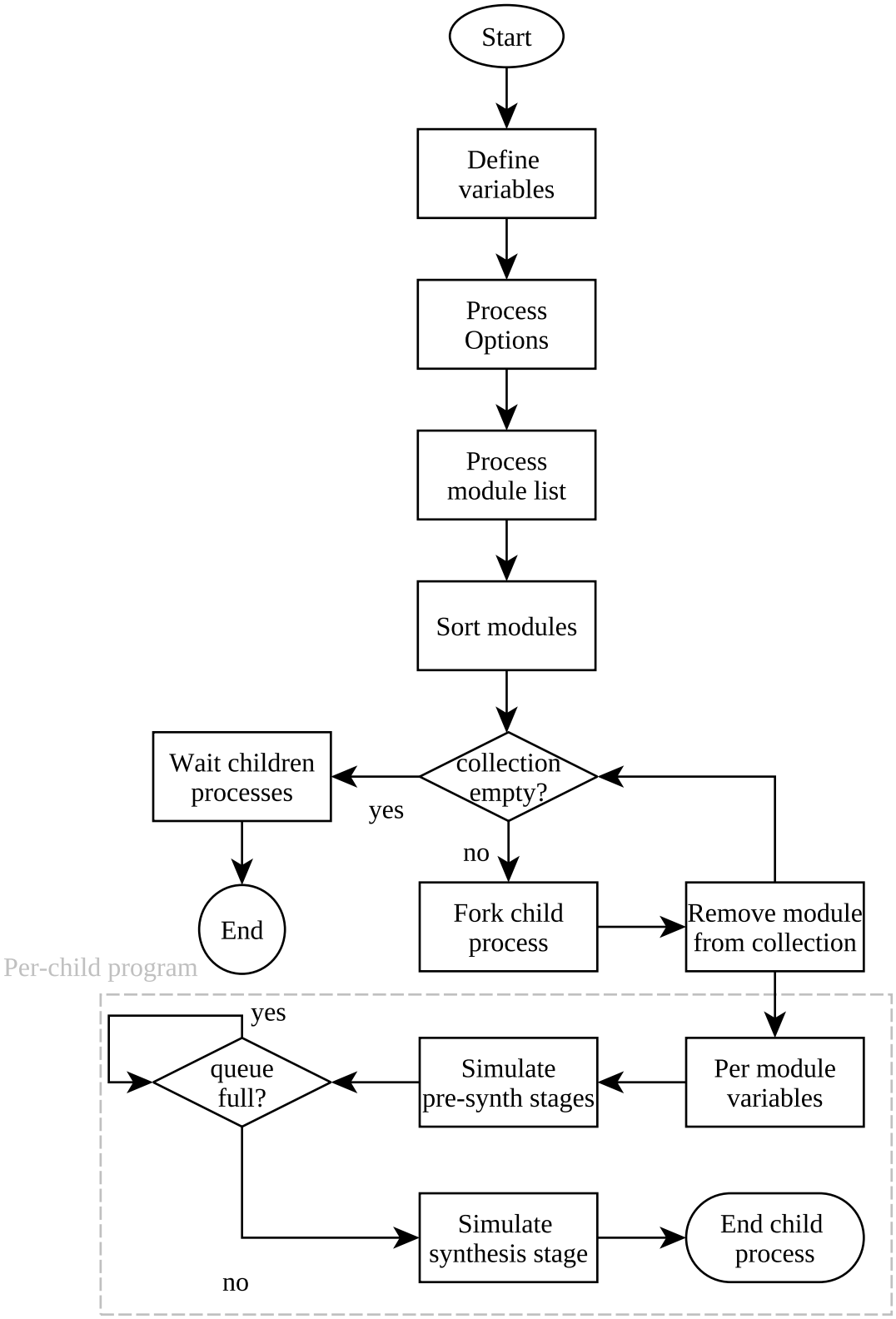

Figure 4.1: Tool execution flow

approach, a shared variable between processes is either decremented or incremented. So if the current value of the constrained queue is lower than the maximum size of the queue, a new synthesis job is dispatched from the waiting queue.

Several constrained queue sizes were tested. The intention of this is to probe that the bigger the size of that queue, the better the total runtime (lower total runtime). So that each sorting type was simulated on each queue size.

Also, since the LSF does not support such feature a queue size cannot be changed dynamically. The size of a queue is defined by the LSF administrator, and it is a static value.

The program has an option to define the size of the constrained queue. When selecting a queue size of 1 slot, the approach becomes serially. It is the same as running module per module, with the advantage that is executed automatically. By increasing the size of the constrained queue, the total runtime decreases. Eventually, the size of the queue is irrelevant to reduce the total runtime, since the longest job determines the total runtime of the chip.

At the beginning of the simulation, first jobs see empty slots, so the constrained queue is filled with every new incoming job. Eventually, that queue becomes full.

The program simulates the Load Share Facility (LSF). It uses an array to simulate the queue that contains all the waiting synthesis jobs. It also uses a counter that is either incremented or decremented by each thread to know the current size of the constrained queue. This constrained queue simulates for example, the part where the LSF allows only 8 synthesis jobs at the same time. So that every time a new job is launched, it will check the value of that constrained variable, and if it is lower than the constrained queue size, a job is submitted. Then when a job has finished, it decrements the constrained queue size variable so that the waiting processes, which are polling that variable, will see eventually an available slot.

4.1 Main program

The program is coded in `Perl` language since it is the preferred language due to previous expertise. Although the simulation program was coded in `Perl`, it is recommended to use another programming language with better parallel processing management, such as `Python` for example.

CPAN module `IPC::Shareable(':lock')` allows `Perl` to tie a variable to shared memory to easily share the contents of that variable between `Perl` processes [8].

By using that module, a set of variables are marked as shared so that each child process can either access to the content or modify the value of these variables. Among these shared variables, a counter that is either incremented or decremented by each thread simulates the current amount of running synthesis jobs on the constrained queue. This

variable allows to know whether a new synthesis job can be submitted or not.

Another shared variable is the waiting queue. This variable is an array that contains all submitted jobs, but not necessarily the synthesis running jobs, because these jobs are already at the constrained queue. The first element on the waiting queue is the next job that should be attended.

4.1.1 Help menu

The following code presents the help menu of the program, which is called `chip_builder_2_0.pl`

```
chip_builder_2_0.pl [OPTIONS]

[OPTIONS]
-h      | help                  Print this screen
-l      | longest               Longest jobs first
-s      | shortest              Shortest jobs first
-r      | random                Random order
-q      | queue     <queue>     Queue size. By default is 8.
-e      | seed      <seed>      Seed to random order mechanism. By default is 4.
-i      | iter                  Iteration number
-d      | debug                 Displays additional execution information
```

Options `-l`, `-s`, and `-r` are mutually exclusive, and at least one of them is mandatory. Sort mechanism is related to those options. If either `-l` or `-s` are used, option `-e` is ignored, since a seed is not needed to sort by either longest or shortest jobs first.

Option `-i` is used to indicate the iteration number. This is useful for creating several iterations on any sorting mechanism, so that the output file shows the iteration number and the program does not overwrites a previous existing report.

Some usage examples:

- Constrained queue is 16 slots wide. Sorted by longest job first. Debug option activated.
  `$ chip_builder_2_0.pl -l -q 16 -d`
- Constrained queue is 8 slots wide. Sorted by shortest job first. Debug option activated.
  `$ chip_builder_2_0.pl -s -q 8 -d`
- Constrained queue is 4 slots wide. Jobs are sorted randomly, and based on the same random order, two iterations are executed by using the same seed.
  `$ chip_builder_2_0.pl -r -e 3 -i 1 -q 4`
  `$ chip_builder_2_0.pl -r -e 3 -i 2 -q 4`

Once the program processes the options and defines all local variables, as well as the output report file, it generates a hash structure based on the list of blocks and their synthesis runtime. This hash contains as keys every block, and each key has as value the related individual synthesis runtime.

After this, that hash structure is then sorted depending on which option the user has chosen. The sort mechanism creates a pair structure that a `foreach` loop iterates over, creating a child process for each block. This block is then pushed into the waiting queue. Each children process start the individual synthesis flow simulation on each block. Once the synthesis simulation finishes, the child process is ended.

Next the `foreach` loop, the main program waits for all children processes to finish. Once all children processes are completed, main program reports that it has finished, and closes the output report file with the information of the simulation.

4.2 Input files

The sorting mechanism needs to know how much time each block takes to pass through the synthesis flow individually. Sort mechanism takes as input the following file:

`$TOOL_DIR/inputs/synthesis_runtimes.txt`

This file is formatted in the following way, to match the program regular expression which parses:

```
block_name;time
```

Table 4.1 shows the list of blocks and their synthesis runtimes.

For simulation purposes, the time is taken as seconds instead of minutes as in real life. Example chip from table 4.1 takes if run serially, about 5 days to be fully synthesized (or 2 hours in simulation time).

Since an input file is loaded with the data into the simulator, if the data of one or several functional block change, the new total runtime information related to the change can be then obtained again by running the simulator with the updated input file. Changes in the input file can also be linked to different ASICs.

4.3 Subroutines

Not all the processes are done by the main program. There are subroutines which provide special functions separated from the main program.

Table 4.1: Total Synthesis runtime (minutes) per module in a chip

#	Block	Synthesis runtime	#	Block	Synthesis runtime
1	egcm	28	2	egtcam	17
3	fefep	861	4	fifep	108
5	flb	171	6	fpg	104
7	gtcam	93	8	hash	84
9	igcm	198	10	mac_fabric	176
11	mac_network	502	12	ms	799
13	ms_mem	29	14	nefep	673
15	nifeb0	498	16	nifeb1	705
17	npg	211	18	rep	98
19	soc	628	20	tmab	524
21	tmma	116	22	tmqb	773
23	tmts	158		**Total:**	7554

4.3.1 gen_hash

This subroutine generates the hash structure based on the input file information. As the input file gives the pair block-runtime, the usage of a hash structure is convenient to pair the key-value with the block-runtime structure. Furthermore, there are `Perl` functions to work on hashes that easily sort the contents of a hash based on the value.

`gen_hash` subroutine opens and parses the input file to create the block-runtime hash, which is a variable that the entire program has access. It finally ends by closing the input file.

4.3.2 sort_mechanism

In this subroutine, and based on the sorting type chosen by the user, the contents of the hash generated from subroutine 4.3.1 are sorted by the runtimes, and located into an array variable. The block-runtime information is addressed into an output file so that the user knows what would be the information that the program will process in the main loop. Finally, the array structure is converted into a pair array structure (like tuples) by using the `Perl` function `pairs`.

Main loop from main program iterates over the pair array structure, so that each pair obtained from the array is then decoupled into both the block variable and runtime variable.

4.3.3 synthesis_simulator

At this subroutine, the program let the user know that a block starts the synthesis flow, specifically the pre-synthesis stages. As figure 2.2 from section 2.1 shows, stages from RTL stage up to IP generation stage are run in this part. Normally, the total overhead on these stages is only a few minutes for even a big block. So the average overhead taken for simulation is 2 minutes average (or 2 seconds for simulation purposes).

After a block has finished its pre-synthesis stages, it enters to the synthesis stage. For this, another subroutine is called to simulate the LSF.

4.3.4 lsf_simulator

This subroutine is the most important part of the program since it decides whether a block can start its synthesis stage. It is in charge of simulating how the LSF enqueues every synthesis job that needs to wait for a free slot at the constrained queue. At the beginning it lets the user know that a job is submitted. It also takes the first element from the queue of jobs. As at the very beginning there are plenty of free slots in the constrained queue, the first block can start synthesizing without any problem.

In an infinite `while` loop, each child process starts checking whether the related block can be synthesized or not. First, all shared variables are locked, then an auxiliary subroutine checks whether a block can be executed by returning a `1` if the block can start the synthesis process, or a `0` if the constrained queue is full so the current job needs to wait until a slot is released.

The auxiliary subroutine called `check_if_service()` does the requirements checking process by polling the current size of the constrained queue. If the current size of the constrained queue is lower than the maximum size of the constrained queue, and if the current job name is equal to the next job that should be attended, the subroutine let the current block to be synthesized. On the contrary, current block has to wait. This is done to let only the next job in queue to be synthesized, and no other jobs to go after the empty slot.

If a block can be synthesized, the next job variables is updated with the next job in the waiting queue, and the program lets the user know that the block started the synthesis process. It also indicates at what time the block has started, and eventually it will indicate when the block has finished the synthesis process. Then, the current size of the constrained queue is increased, and the shared variables are unlocked so that other parallel processes can access them. Then a `sleep` function simulates the synthesis execution of current block. This `sleep` function waits the runtime in seconds of the related block.

Once current block has finished simulating the synthesis process, the program locks the current size of the queue variable and decreases it. Finally, infinite `while` loop is broken and the child process is ended.

If a block does not meet the requirements to be synthesized, all shared variables are unlocked, and the infinite loop keeps running. Each time a block checks whether it can be serviced, all shared variables are first locked so that the related child process can read the current values, and no other child process have access in the meantime. In this way, data coherency exists between the processes.

4.4 Output files

All messages from either the main program or the subroutines are addressed to output files. These files are located at the following location:
`$TOOL_DIR/results/`

Each file is named as:
`${type}_job_first_result_iter_${iter}_queue_size_${queue_size}_2_0.rpt`

Where `$type` is the sorting type (either longest, shortest or random). `$iter` is the iteration number. And `$queue_size` is the size of the constrained queue. Located at same place, another file reports how the blocks were sorted, and it is given by:
`${type}_sort_seed_${seed}_iter_${iter}_queue_${queue_size}_2_0.rpt`

where `$seed` is the seed number, only relevant when the random option is used.

4.5 Running multiple experiments

In order to run multiple experiments, with different sorting types and multiple iterations, the following script is in charge of running all simulations:

`all_queues.sh`

It is a straightforward bash script that executes multiple experiments. This script sets two environment variables for running random experiments, so that the same random order can be used on different queue sizes. When `PERL_PERTURB_KEYS` is set to 0 then traversing keys in a hash structure will be repeatable from run to run for the same `PERL_HASH_SEED`.

```
#!/usr/bin/env bash

cd $TOOL_DIR
export PERL_PERTURB_KEYS=0
export PERL_HASH_SEED=0x01

iter=0
for seed in 1 3 7; do
    ((iter++))
    for queue in 16 8 4 2; do
        ./chip_builder_2_0.pl -e $seed -c -i $iter -q $queue -d
    done
done

for queue in 16 8 4 2; do
    ./chip_builder_2_0.pl -l  -q $queue -i 1 -d
    ./chip_builder_2_0.pl -l  -q $queue -i 1 -d
done
```

By executing the `all_queues.sh` script, three different seeds generate random sorting on the list of blocks. So the first `for` loop iterates over three different seeds. Each seed runs the `chip_builder_2_0.pl` on 4 different queue sizes (2, 4, 8, 16 slots). The same randomly sorted list is repeated on each different queue size experiment. Then, second `for` loop iterates over the same queue sizes but for the longest and shortest jobs first scenarios.

Chapter 5

Results and analysis

The sort mechanism from the simulator generates the following lists shown in tables 5.1, 5.2, 5.3, 5.4, and 5.5.

Table 5.1: Sorted by longest job first

Block	Synthesis runtime
fefep	861
ms	799
tmqb	773
nifeb1	705
nefep	673
soc	628
tmab	524
mac_network	502
nifeb0	498
npg	211
igcm	198
mac_fabric	176
flb	171
tmts	158
tmma	116
fifep	108
fpg	104
rep	98
gtcam	93
hash	84
ms_mem	29
egcm	28
egtcam	17

Table 5.2: Sorted by shortest job first

Block	Synthesis runtime
egtcam	17
egcm	28
ms_mem	29
hash	84
gtcam	93
rep	98
fpg	104
fifep	108
tmma	116
tmts	158
flb	171
mac_fabric	176
igcm	198
npg	211
nifeb0	498
mac_network	502
tmab	524
soc	628
nefep	673
nifeb1	705
tmqb	773
ms	799
fefep	861

Table 5.3: Random sort, first seed

Block	Synthesis runtime
ms	799
flb	171
ms_mem	29
fpg	104
mac_fabric	176
tmab	524
fefep	861
nifeb1	705
egcm	28
tmma	116
gtcam	93
npg	211
rep	98
nifeb0	498
igcm	198
soc	628
hash	84
tmqb	773
fifep	108
nefep	673
mac_network	502
tmts	158
egtcam	17

Table 5.4: Random sort, second seed

Block	Synthesis runtime
soc	628
tmts	158
fpg	104
gtcam	93
rep	98
fefep	861
ms_mem	29
mac_fabric	176
igcm	198
tmma	116
tmqb	773
egtcam	17
ms	799
egcm	28
mac_network	502
flb	171
hash	84
tmab	524
fifep	108
nifeb0	498
nefep	673
npg	211
nifeb1	705

Table 5.5: Random sort, third seed

Block	Synthesis runtime
fpg	104
nifeb1	705
tmab	524
rep	98
ms_mem	29
tmqb	773
mac_fabric	176
hash	84
flb	171
egcm	28
mac_network	502
fefep	861
fifep	108
gtcam	93
egtcam	17
igcm	198
npg	211
tmma	116
tmts	158
ms	799
nifeb0	498
soc	628
nefep	673

Each table represents an input to the main loop that simulates the synthesis flow for each block in the list. The order of execution is given by the same order of each table. Every report generated by the different executions of `chip_builder_2_0.pl` simulator presents incrementally the time in which each block finishes its synthesis job. So that the time when the last block has finished represents how much time the synthesis of all blocks in the example chip from 4.1 take to complete.

Table 5.6 summarizes how much time each sort mechanism takes to complete the synthesis of all modules from table 4.1, depending on the size of the constrained queue. When the queue size is constrained to only 1 slot, there is no difference in the obtained results for total synthesis runtime for each sorting mechanism. Similarly, in the opposite case, when the size of the queue is 16 slots, the differences between the results for each sorting mechanism is not relevant. Hence, either having only 1 slot, or a large number of slots in the constrained queue, as 16 slots, makes the sort mechanism to lose relevance.

Table 5.6: Total Synthesis runtime per sort mechanism on different queue sizes

Sort Mechanism type	1	2	4	8	16
Longest	7556	3783	1980	1009	863
Shortest	7556	4059	2340	1471	973
Random 1	7556	3781	1947	1204	869
Random 2	7556	3974	2214	1392	868
Random 3	7556	3913	2097	1224	892

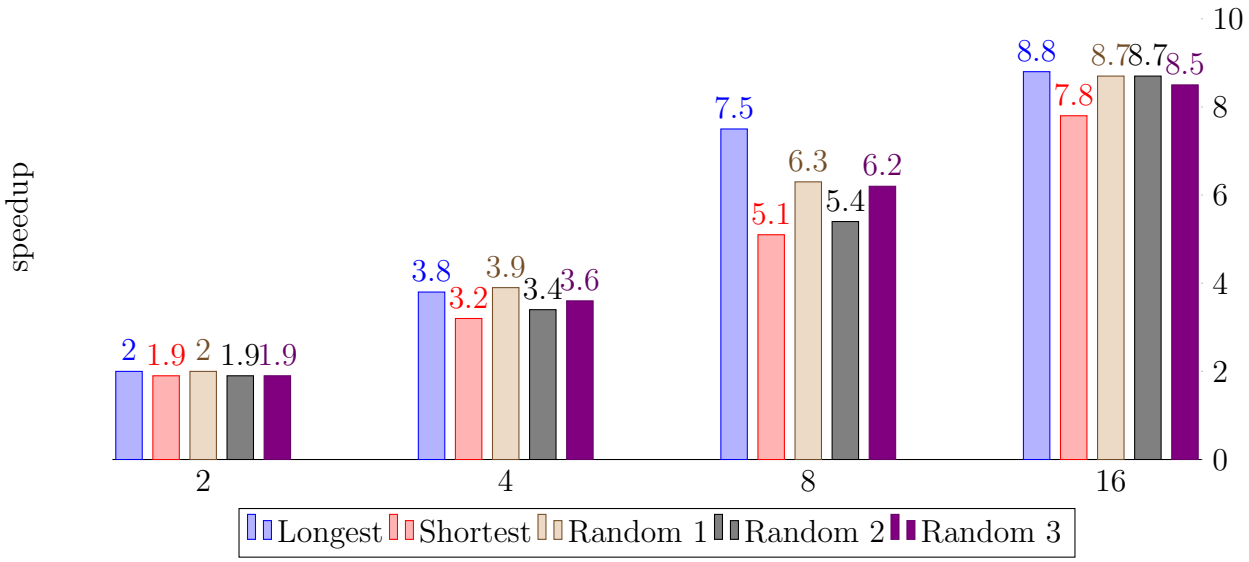

Figure 5.1: Speedup for each sorting mechanism on different queue sizes

However, the higher the size of the constrained queue, the better the total synthesis runtime. Figure 5.1 shows the normalized values, indicating how much the total synthesis runtimes has improved regarding the serialized process (or when the constrained queue size is 1 slot only).

In other words, Figure 5.1 data confirms that having higher sizes of the constrained queue speed up the total synthesis runtime in the example chip, in up to 2, 3.9, 7.5, and 8.8 times with respect to the serialized process, on each constrained queue size respectively. Figure 5.1 plots the speedup of each sort mechanism for every constrained queue size.

The ideal scenario is to have as many CAD licenses as the total number of synthesizable units that form the ASIC. However, by having an infinite number of licenses available in the constrained queue, the total synthesis time is no longer reduced due to the fact that it depends on the longest job synthesis runtime. This is seen at Figure 5.2, where passing from 8 slots to 16 slots the total synthesis runtime is not reduced drastically, as it occurs from 1 to 2, 2 to 4, and 4 to 8 examples. Figure 5.2 also presents how much the runtime is reduced based on the amount of licenses taken by the constrained queue, as

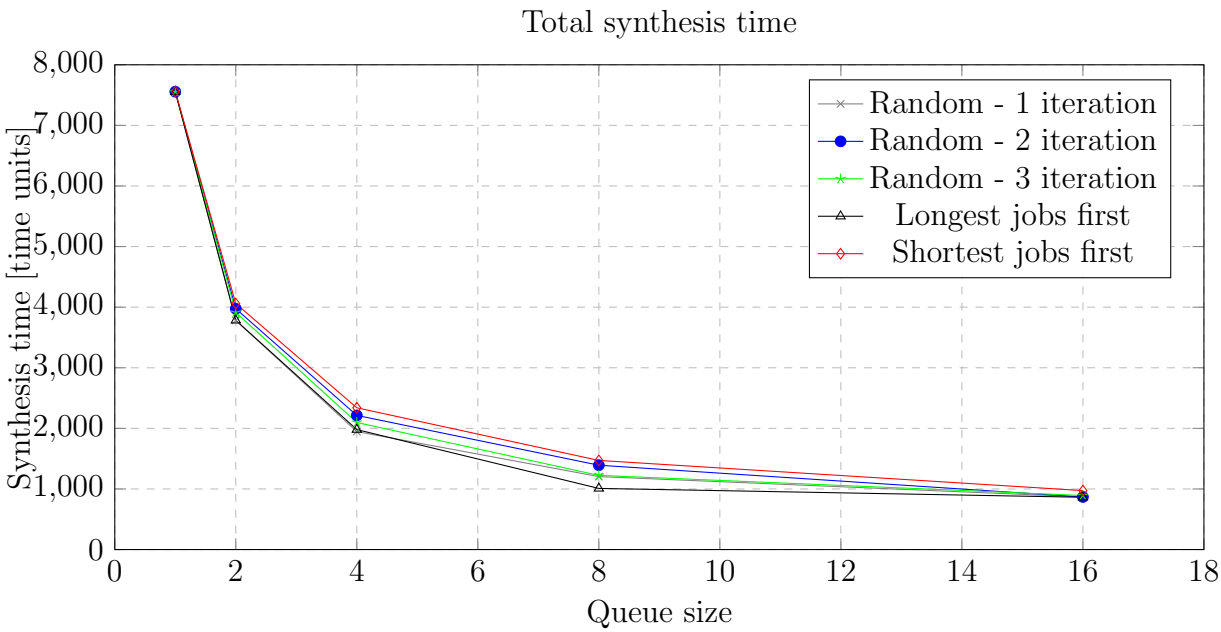

Figure 5.2: Total synthesis time per sorting mechanism on different queue sizes

an exponential distribution pattern.

A higher number of licenses for the constrained queue helps in completing faster the smaller blocks. However, the biggest synthesis job defines the total synthesis runtime in a chip. That biggest runtime can be reduced if the host machine that run the processes is a more powerful computer. Also, synthesis runtime can be reduced by splitting the big block into smaller functional blocks. But as drawback, splitting into more functional blocks could affect both the routing congestion and timing of top level signals at the top level of the ASIC.

Furthermore, the utilization of these CAD licenses is not always 100%. Not all the time ASICs are being synthesized except when milestones approach; the utilization of these CAD tools increases at those dates. Since RTL designers are using these CAD licenses as well (to verify that their designs are meeting quality checks such as timing, area, power) only a fraction of licenses should be taken from the total pool of licenses to build the entire chip, in order to avoid CAD tools starvation from the other members of the team, so that they can keep working on their experiments.

Therefore, based on Figure 5.1 results, taking more than 1 license to synthesize blocks speeds up from 2 times (queue size 2) the total time to build an ASIC, up to 8.8 times (queue size 16) if there are more licenses taken from the pool of licenses. By either taking 4 or 8 licenses from the pool of licenses, this methodology helped in reducing significantly those total synthesis runtimes in an ASIC.

Shortest jobs first approach is the worst approach in all scenarios. Despite the fact that faster modules are finished first, the longer jobs are serviced last, worsening the total synthesis runtime of the example chip. This scenario can be helpful only when results

need to be reviewed sooner.

As Figure 5.1 shows, when the constrained queue is limited to allow only two synthesis jobs at the same time, both the Random 1 and Longest present the best results: 2 times of speedup regarding to having only 1 slot in the queue, respectively. There is a difference in 2 time units only (2 seconds in simulation, 2 minutes in a real synthesis case). Similarly, when the constrained queue is limited to 4 slots, the same sort mechanism types present the best results: 3.9 times and 3.8 times of improvement, respectively.

When limiting the constrained queue to 8 slots, Longest jobs first approach is now the best, followed by Random 1, with 7.5 and 6.3 times the performance improvement, respectively. Finally, when the constrained queue is limited to 16 job slots, the best approach is also Longest, followed by Random 1, with 8.8 and 8.78 of speedup, respectively.

If the ASIC team does not possess that many CAD tool licenses, it is not worthy to grab such many licenses from the pool of total licenses. The gain in total runtime is only 15% by doubling the amount of licenses (8 to 16). Unless there are several ASIC projects running in parallel, each one with tens of synthesizable block units, taking higher fractions of licenses is helpful for completing more blocks in parallel, but increasing the chances that the other users suffer from licenses starvation.

Chapter 6

Conclusions

By using both the snapshot workflow, which helps keeping a better traceability of changes through the milestones in a project, and parallel processing handled by a load share facility such LSF, this research has described a dynamic, flexible and completely automated design workflow methodology, which is able to manage the synthesis of multiple functional modules that form an ASIC.

As the real working environment cannot be used for academic purposes, a simulator of the methodology was implemented in `Perl`. Different types of sorting mechanisms can be tested, and the user can specify different sizes of the constrained queue, allowing the execution of several experiments. Consequently, by using the simulator it was demonstrated that the time to synthesize the complete set of functional blocks is speedup in 8.8 times, compared against a serial execution approach. Longest job first approach possess the fastest total synthesis runtimes, achieved in a constrained queue sized that allows 4 or more synthesis jobs running at the same time.

For future researches, this methodology can be enhanced to support cases where the main functional blocks are hierarchical blocks. In other words, some blocks are that big (millions of gates for instance) that they need to be sliced in smaller pieces of functional sub-blocks, in order to synthesize them separately from the parent block. Then the results of these smaller sub-blocks are fed as inputs into the synthesis processes of the parent block. This approach helps both the total synthesis runtime and resources usage, as memory consumption and CPU runtime.

The consequences of this approach is that there will be more synthesizable units, increasing the amount of blocks that should wait for an empty slot at the constrained queue. The challenge is to find an optimal way to sort the list of modules so the resources can be assigned in an efficient manner, thus reducing the total runtime of the synthesis process for a chip.

Bibliography

[1] Raquel Araya. Mejora en el flujo de diseño del grupo de backend. Grade graduation project, 2014. Unpublished.

[2] IBM Corporation. Foundation - platform lsf, 2013.

[3] Tim Edwards. Yosys open synthesis suite, June 2016. URL http://www.clifford.at/yosys/about.html.

[4] IBM-Systems. Ibm spectrum lsf, high-performance workload management for demanding hpc environments, June 2016. URL http://www-03.ibm.com/systems/spectrum-computing/products/lsf/index.html.

[5] Branimir Malnar. *Synthesis Flow for Designing a High Performance Microprocessor*. 2012.

[6] OpenLava-project. Openlava, open source workload management, June 2016. URL http://www.openlava.org/home.html.

[7] Solvnet. Reference methodology, June 2016. URL https://solvnet.synopsys.com/rmgen/.

[8] Benjamin Sugars. Ipc::shareable, April 2017. URL http://search.cpan.org/~msouth/IPC-Shareable-0.61/lib/IPC/Shareable.pm.