FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Generation of Reconfigurable Circuits from Machine Code

Nuno Miguel Cardanha Paulino

Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major em Telecomunicações

Supervisor: João Canas Ferreira (Assistant Professor)

Co-supervisor: João Cardoso (Associate Professor)

June 2011

Abstract

This work presents a system for automated generation of a hardware description that implements a dedicated accelerator for a given program. The accelerator is run-time reconfigurable, named the Reconfigurable Fabric (RF) and is tailored to perform computationally demanding section of the analyzed program. Previously available information regarding CDFGs (Control and Data Flow Graph) is treated with the developed toolchain in order to generate information that characterizes this RF, as well as information used to reconfigure it at runtime. The RF may perform any of the given CDFGs it was tailored for, and is expandable to variable depths and widths at design time. The RF is organized in rows with operations in a grid like structure. Any operators may be connected to any operation inputs within the RF and likewise any outputs may be connected to inputs of following rows. The number of input operands and results is also design time parameterizable. The RF reutilizes hardware between its mapped CDFGs. The developed toolchain also generates the communication routines to be used at run-time. The system is triggered transparently by bus monitoring. Speedups vary according to communication overhead and the type of graph being computed, ranging from 0,2 to 65.

Acknowledgments

I would like to thank my supervisor, João Paulo de Castro Canas Ferreira, for his guidance and insight and João Bispo, whose own work allowed the completion of mine.

I am grateful to my colleagues for their support during the development of this project.

Lastly, I thank my parents for an entirely different genre of support altogether.

Nuno Paulino

In the begining there was nothing, and it exploded.

Terry Pratchett

Contents

List of Figures

Listings

List of Tables

Acronyms and Symbols

FPGA	Field Programmable Gate Array
DFG	Data Flow Graph
CFG	Control Flow Graph
CDFG	Control and Data Flow Graph
CCA	Configurable Compute Array
DIM	Dynamic Instruction Merging
AMBER	Adaptive Dynamic Extensible Processor
WCLA	Warp Configurable Logic Architecture
SCLF	Simple Configurable Logic Fabric
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction-set Processor
GPP	General Purpose Processor
RF	Reconfigurable Fabric
RM	Reconfiguration Module
CS	Code Segment
FSL	Fast Simplex Link
BRAM	Block RAM

Chapter 1

Introduction

The power and performance walls of current processing architectures are becoming a relevant issue as the present day technologies are steadily hitting their maximum performance point [2]. As size decreases and the density of circuits increases there is no effective way to dissipate heat and ensure better power efficiency. Multiple core CPUs are the current solutions, being the easiest architecture to expand upon. However, they will eventually meet the same difficulties. So, the solution is to design new computing architectures.

Dynamic reconfiguration is a growing research field due to its promising results, and although the notion of configurable hardware has existed since the 60's [3] the creation of a fully autonomous and runtime reconfigurable system has yet to be achieved.

Amongst the several approaches to dynamic reconfigurable systems the main objective of all remains the same: how to configure flexible, custom created, hardware at run-time to optimize computing efficiency of the overall system? The task of custom designing hardware manually, ASICs or ASIPs, is arduous in itself, so the difficulty rests on finding a consistent, scalable and flexible methodology or runtime algorithm that could, potentially, generate hardware as efficient as a custom design.

So, the difficulties of the reconfigurable systems approach start at the very beginning, when the attempt is made to try and identify when and at what level of software execution the reconfiguration effort should be made. The following step is to determine how that possible hardware structure should be generated and interconnected. This would have to be done in such a manner that does not compromise the already present hardware, and, ideally, that allows interfacing with other static hardware in the system. If possible, the reconfigurable hardware would also allow for maximum transparency from the point of view of a software toolchain, making any changes to compiling tools and APIs unnecessary.

The now more widespread use of high level software programming languages, with libraries for use directly on microprocessors (with no Operating System) add to the task of identifying software execution flow during runtime. Although the full tool flow of software programming

(compiling, assembling and linking) results in uniform binary code regardless of the toolchain used, a potential difference of execution structure exists between microprocessor architectures, programming languages and compilers used, which could result in different efficiencies in the detection of the data flow in a program.

Several approaches, at several levels, have already been studied and successfully implemented with very promising results, both in computing efficiency as well as power consumption. These approaches, although effective, are mostly proofs of concept, and so, they are not feasible for common use and commercial deployment. Even the few commercial attempts made have had little market penetration and the implementations based on FPGA exclusive architectures have been academic only.

This project aims to design another proof of concept based on the most interesting aspects of these implementations while also adding a different approach to reconfiguration methodology.

Specifically, this project is oriented towards the automatic, runtime, generation of dedicated hardware from machine code in a single FPGA (Field-Programmable Gate Array). Possible implementation alternatives are explored further in the document, however, a generic objective is the acceleration of computationally intensive cycles in programs by utilizing hardware that is run-time reconfigurable. Ideally, the transparency from the point of view of the software toolchain will be close to total.

Since FPGA technology will be the development platform targeted for this project, it will be given the most relevance in terms of research concerning dynamic systems throughout this document. However, other technologies exist and have been used to test implementations of systems with the same objectives of performance and cost presented by dynamic hardware.

1.1 Dissertation Structure

In this document, a small contextualization of the theme under study has been presented. Following are 6 other chapters.

Chapter 2 details the state of the art, contains a concise comparison between computing technologies and an analysis of related works. In chapter 3 the objectives and some preliminary designs are presented. In chapter 4 the currently implemented system is detailed, followed by chapter 5 which explains the toolflow designed to support the system. Chapter 6 presents the obtained results and, finally, chapter 7 contains a small description of possible future modifications off the implemented system.

Chapter 2

State of the Art

The most relevant technology in the field of reconfigurable computing is the FPGA [4]. It is the most flexible in terms of reconfiguration and can be used to perform either standalone computing (FPGA based systems) or hybrid computing (standard GPPs in parallel execution with FPGAs).

Hybrid computing allows for a softer transition to reconfigurable systems, from the point of view of higher level development. However, their full potential is bottlenecked by the system they are coupled too (if the FPGA is implemented as an expansion IO card in the system). The bottleneck is reduced if the FGPA fabric is implemented directly in-chip, along with the CPU, which is know as a hybrid-core.

Hybrid-cores contrast with heterogeneous computing. In this last method of computing, a single, fixed instruction set processor executes the desired application while resorting to dedicated hardware peripherals for acceleration. With hybrid-cores, the idea is to extend the instruction set to two (potentially more) in a single memory addressing space by coupling co-processors to the system One example of this architecture is the commercially available Convey HC-1 [5], which has a GPP coupled with a reconfigurable co-processor based on a Xilinx Virtex 5 FPGA. However, this kind of approach, and without going into much detail, may involve modifications in the software toolchain or introduces new SDKs to develop configuration data for the co-processor: "(...) a Personality Development Kit (PDK) provides a toolset that allows users to develop their own personalities for specific applications" [6]. This is out of the scope of this document.

Fully FPGA based computers may provide a greater degree of scalability and the bottlenecks found in the current computing architectures maybe reduced, depending on the interconnections between FPGAs. The downside is the inherent necessity of shifting development to a domain that is purely hardware centered. This not only implies a longer development time but also the introduction of new development software as well as requiring specialized knowledge in the field of hardware design, which is something that the currently settled development chains are resistant too.

Looking more closely at embedded systems, there are several technologies relevant to the process of creating dedicated hardware for a specific application: ASICs, ASIPs or FPGAs. Being that the purpose is reconfiguration, the focus of this document is on the latter, however, a small

technology comparison is as follows. How these technologies relate and have been, or can be, reused for reconfiguration is also briefly described.

- GPPs (General Purpose Processors): like those commercially available for use in personal computers. GPPs are easy to deploy and their toolchains are standardized. However, they have a static instruction set, and are made to be coupled to a heterogeneous system. Thus, they fail to be application specific and comparatively with other computational approaches they perform worse and have increased power consumption.

  Implementing one of these processors on an FPGA is a possibility when creating a reconfigurable system, the choice depends on the functional structure desired for the system.

- ASICs (Application Specific Integrated Circuits): as the name implies, they are circuits designed to efficiently perform a particular task. They have the greatest development cycle and are not reconfigurable. However, their performance may be as good as the technology and their design allows. The goal of reconfigurable systems is to shorten development time of systems tailored for a specific application while aiming for the efficiency provided by ASICs.

  Essentially, reconfigurable systems seek to either create or utilize dedicated hardware at runtime, be it more or less fine grained. Granularity is the measure of reconfigurability and the behaviour of a dynamic system is based not only on this characteristic but several others.The relevant ones depend on the application in question, however, a few are: reconfiguration overhead, interconnectivity and changes to toolflow. This last characteristic is specially relevant. Depending on the reconfigurable architecture chosen there may be a need to change to software toolchain. Ideally, no change would be required and any program written in conventional programming languages would be possible to execute and be accelerated on the reconfigurable system.

- ASIPs (Application Specific Instruction-set Processor): a GPP-type architecture whose instruction set is tuned to the application. This presents the same disadvantages as an ASIC but is more permissive in terms of flexibility while being less permissive relative to an actual GPP.

  Creating several ASIPs, as soft-core processors, at runtime in a reconfigurable system is also an alternative. Of course this implies properly identifying the necessary instruction set, and seeing as though processors execute instructions involving reading and writing from memory, such accesses would also have to be managed. There would also have to exist, potentially, communication between processors if, for instance, one program is distributed amongst several processors. This adds to the difficulty of the profiling of the application and its hardware mapping.

2.1 Related Work

As previously stated, this project is focused on a reconfigurable system on a single FPGA. Similar projects, all from academia, are reviewed and their most similar characteristics to this project's goal are critically analysed in the following section.

Generally speaking, implementations of reconfigurable architectures can be characterized based on a few basic properties. Among these: the level of coupling to a GPP, if any, the granularity of its reconfigurable hardware, the type of hardware in the reconfigurable fabric (combinational only or sequential), the structure of the reconfigurable hardware itself, whether or not memory operations are allowed (dynamic memory allocation, pointer operation or access to data memory for read-/write), capacity, method for Data flow Graph (DFG) and Control Flow Graph (CFG) detection and mapping [7] and, most importantly, the practical speedup of execution attained and Instruction Level Parallelism (ILP, the amount of instructions that can be executed in parallel at any stage of the reconfigurable hardware).

Some approaches prefer to start the reconfiguration effort at source code level [8, 9, 10, 11], yet others prefer to diminishing the influence on the software toolchain and perform the optimization at binary level, arguing that supporting standard binary level is also an added bonus to the acceptance and flexibility of these systems.

In one way or another, these implementations seek to attain automatic instructionset extension [12]. ASIPs, being the halfway point between ASICs and GPPs, shorten design time for application specific implementations. However, they are still a deviation from the standard design flows applied today, so, the automatic creation of an instruction set would greatly increase field flexibility and make the development times shorter for application specific circuits as well as increase performance.

2.1.1 Warp Processing

Warp Processing is an FPGA based runtime reconfigurable system [13, 14, 15, 16] that involves binary decompiling, which begins with the detection of cycles [17] in the program, known as profiling. A dedicated module performs the profiling and several others decompile the running binary into high level cycles which are then mapped into the remaining FPGA fabric by custom CAD tools. The target FPGA is a custom built device with a simpler interconnection architecture designed to simplify the execution of the CAD tools. The software execution is then shifted to the mapped hardware for those identified sections, the operands of the instructions being fetched via a Data Address Generator. The system is, thus, fine grained and loosely coupled and is fully FPGA based, containing a MicroBlaze soft-core processor. Although effective and completely transparent to the programmer, Warp Processing has its limitations. For instance, it only detects small loops in the running program. Identifying and mapping more complex hardware for more complicated loops would greatly increase the effort of the SoC CAD tools as well as increase mapping time. Also it does not support floating point operations, pointer operations or dynamic memory allocation. Another disadvantage is the inability to explore parallelism of execution between the

generated hardware and the soft-core processor seeing as tough the binary is altered, and it also does not allow for multiple soft-core processors taking advantage of the reconfigurable fabric.

Despite this, a MicroBlaze based warp processing system was successfully implemented in [18] which allowed multiple processors to utilize the available reconfigurable fabric, although introducing hardware overhead for each additional desired processor. The capability of having a multiprocessor system coupled with any sort of hardware accelerator is non trivial as it raises several issues, such as the management of access to the reconfigurable fabric and compatibly mapping one or more dataflows in the fabric.

Already mentioned and worthy of note in this implementation, and discussed in detail in [14, 19, 20] is the simplified configurable fabric (SCLF) utilized to speedup the mapping and algorithms of the on-chip CAD tools.

The SCLF itself is a network of switch matrices interconnecting several small LUTs which implement all the operations of the mapped hardware. Alongside this interconnection network are registers to store results, additional hardware to manage memory access and a 32 bit MAC (multiplier accumulator). The addition of this piece of dedicated hardware derives from the common occurrence of these operations in embedded systems, thus making it more effective to implement in a hardwired manner, instead of in the SCLF. The entire set of modules is denominated WCLA.

As for the CAD tools, having a full chain of synthesis and place & route tools on-chip would be demanding in terms of memory and execution time. Thus, in the cited paper, a set of simplified mapping tools, ROCPAR, was developed for the WCLA described in related papers. This tool suite simplifies already existing algorithms by allowing only small software kernels to be mapped and assuming that only a limited set of more typical dataflows will be present in said kernels. A detailed look at the routing tool within this suite is published in [19]. In this implementation the SCLF is modified to contain flip-flops within the LUTs of the fabric, and they may either be bypassed or used to construct the desired hardware.

Although an advantage from the point of view of hardware generation efficiency it introduces a constraint on the system, being that the mapped hardware will be generated in such a manner that compatibility with this reconfigurable fabric is possible. In other words, the full potential for optimization is perhaps lost by not utilizing a fully blank FPGA fabric. However, doing so would be a computationally demanding task. In fact, the authors state that their tool suite generates results that are less efficient than more complex placement and routing algorithms that are ran offline. However, the obtained routing results are still competitive with the VPR algorithm [21] and they also present a comparison testing between the entire ROCPAR tool suite and the Xilinx ISE. Even though the computational effort and memory demands are greatly reduced, the requirements are still up to 8MB of memory to execute these synthesis tools. Also, the SCLF introduces considerable hardware overhead for connections between LUTs, which also represent not only pipeline delay but also static power consumption.

For the benchmarked applications (several from the EEMBC and Powerstone suites) the average speedup attained was 5.8, with a reduction of power consumption of 57% on average.

2.1.2 Thread Warping

In [22, 16] a multi threaded approach to warp processing was taken. The implicit parallelism found in threaded applications makes them prime candidates for hardware acceleration. However, threads are handled by operating systems, so it becomes necessary to develop a radically different architecture that supports communication with the OS running on the GPP. Also, the OS has to be made aware of the new resources the FPGA circuits represent so that it can map thread executions on it as it would map them on other computing cores. For this purpose, and API was developed to allow for this interface.

Specifically, the thread warping mechanism comes into play when there isn't a sufficient number of micro processors available to execute the queued threads (for a mono processor layout, this means more than one thread). The on-chip CAD tools, now heavily modified, monitor the OS's thread queue for any opportunities of optimization. Once a thread is identified and as not yet been optimized, it is processed by the CAD tools to generate a corresponding circuit. The original binary may or may not be updated, depending on whether the thread was implemented only partially in hardware, or fully. The generated circuit is then stored, in case it is unmapped but needed again if its corresponding thread is queued for execution at a later point.

Added to the effort of binary decompilation, detection of software kernels, synthesis and mapping is added the step of properly identifying memory accesses between threads as to avoid violation of resource accesses or the creation of race conditions. However, the authors identified regular patterns of resource access between threads and perform a reduced number of memory reads to feed the instantiated accelerators. Still, additional hardware is required such as DMAs (up to one per accelerator or accelerator group) and OS support is required in order to synchronize thread queueing. Several other disadvantages and limitations exists, however they are mostly OS oriented and as such, out of the scope of this paper.

Still, despite all the current limitations and the great amount of alterations needed in the fabric and the functioning of the CAD tools, speedups averaging 169 times were obtained with thread warping.

A small summary of the Warp Processing system found in [15], along with the CCA architecture discussed in the following section.

2.1.3 AMBER and CCA

Besides warp processing, many other implementations utilize application profiling. Generally, profiling involves monitoring instruction memory for backwards branches in an attempt to identify controlled loops and determining which of those loops should be mapped into hardware. The decision criteria for the choice varies, being it the number of occurred backwards branches, the type of dataflow detected in that code segment or the existence of supported instructions. Often, higher level compilers will unroll explicitly stated loops and thus the loop control structure is lost at the binary level. One of the implemented techniques in the papers presented by Vahid et al. [16] was the recovery of this information at profile time to better utilize the hardware acceleration.

Another was operator strength promotion, which recovers multiplications that were previously optimized into a series of additions and shifts targeting the compile time architecture. Of course this is only relevant because the architecture of the WCLA already contains a 32 bit MAC. In a different architecture the optimal situation may have been the processing and hardware mapping of the compiler simplified multiplication, seeing as tough it may allow for ILP in a reconfigurable fabric. Considering this, an obvious optimization to be done at graph detection level is the transformation of complex expressions into simple sequences which are mappable and parallelizable.

With AMBER, presented in [23, 24, 25, 26], a profiler design is utilized alongside a sequencer which stores previously created microcode for the developed accelerator and initiates its execution by comparing the current program counter (PC) with stored information. The dynamic hardware consists of a reconfigurable functional unit, RFU, which is controlled by configuration bits to perform a given operation. The RFU is configured whenever a Basic Block (a sequence of instructions delimited by branches), detected by the profiler, is executed over a determined threshold. Note that this implies the proper detection of the Basic Block's dataflow graph and control flow graph at profile time. This is also true for the Warp Processing architecture which requires this information to synthesize circuits. In the case of the RFU, the same information is utilized to map the data flow directly into the existent pipeline of the RFU. Internally, the RFU is composed of several FUs, that can perform logical and mathematical operations, interconnected by multiplexers controlled by configuration bits. The data is feed into the RFU by direct access to the register file of the GPP it is coupled too. Also note the number of inputs and outputs as well as the number of FUs the authors claim to be optimal for the MiBench test suite (8 inputs, 6 outputs and 16 FUs). It provides a rough measure of the necessary amount and characteristics of reconfigurable hardware to have available in order to map the kind of graphs obtained in embedded applications.

It allows for fast configuration switching. However, it binds the accelerator to the processor, in a 1:1 relation. If several processors were to be embedded, and communicating, the hardware overhead for each HW acceleration architecture might become considerable. An improvement would be to have reconfigurable fabric shared by all running soft-cores in the FPGA, and somehow control and multiplex the access to the mapped hardware. This would allow for multiple CPUs to share already implemented hardware, assuming compatible clocks and the possibility to account for delicate timing issues in cross-processor communication. This does not exclude the possibility of keeping recent and most used configurations for fast switching. This however, would be more difficult to implement due to the nature of the interconnections from the GPP to the RFU. The RFU is integrated in the GPP as another element of its functional pipeline. Therefore, such a system is not discrete and portable in such a way that can be applied to closed commercial soft-core processors such as the MicroBlaze and PowerPC. Also, the result of a particular sequence of instructions is fetched from either the accelerator or the output of the GPP pipeline, this means that, in case of a Custom Instruction being executed the GPP will still be processing and there will be no possibility of exploring parallel execution. Also, as mentioned, the coarse grain nature of the RFU makes it impossible to map all the detected Basic Blocks, thus reducing the potential for optimization. On the other hand the temporal overhead for mapping is severely reduced when

compared, for instance, with Warp Processing.

The authors utilize the nomenclature Custom Instruction to refer to a graph that can be mapped on the RFU and further detail on their treatment in [27]. A temporal partitioning algorithm is used to break down data flow graphs and transform them into mappable segments. Another way to solve the issue of mapping large optimizable segments into a smaller reconfigurable pipeline was a small architectural change. By Connecting the FUs in the RFU in a spiraling fashion, a longer pipeline was achieved, making the system more flexible [24]. This of course introduces more complexity in the mapping algorithm. Further work on this architecture produced a heterogeneous RFU [28]. The reason for this modification was the fact that small groups of instructions (two or three) instructions are often executed in a particular order. That is, many of the RFU configurations were similar (at least for the selected benchmarks) and so the level of generalized interconnection permitted by the architecture was superfluous, introducing propagation delay due to multiplexers. Thus, by replacing the simpler FUs in the fabric with more dedicated ones, able to perform more that one operation in sequence, they eliminated the need to insert the connection multiplexers that were present in the homogeneous RFU layout. The removal of their propagation delay allowed for a faster completion of the mapped instructions, which means a higher speedup. Also, the configuration overhead decreases, as is expected of a layout that is essentially more coarse grained. Despite the speedup and mapping percentage gains claimed to be obtained with this approach, note the additional effort of properly identifying subgraphs and their associated mapping and routing on the non uniform RFU. In fact, several modifications had to be introduced to the graph discovery and mapping tools, as well as storing more detailed graph information in memory to allow the level of efficient mapping obtained on this more restrictive architecture. A further modification was the merging of configuration data for CIs with similar, or equal, flows, which reduces memory requirements. This provides an interesting measure of how fine or coarse grained a reconfigurable architecture needs to be in order to allow speedups on the kind of embedded applications in the MiBench suite (which is intended to be a good representative of embedded applications in general).

A very similar system is the CCA presented in [29, 15]. A reconfigurable array of FUs coupled to an ARM processor. In short, the detection of CFGs and DFGs is done by delimiting the code region to be mapped to hardware by custom inline assembly instructions. This, though it diminishes the transparency level from the point of view of the software toolchain (even more so because the compiler is further modified to reorganize the code to ease the replacement phase at runtime), greatly decreases the effort of run-time application profiling and its associated hardware overhead. Note that the identification of data and control graphs in said delimited regions is done at compile time, which implies an even greater modification of the toolchain. However, a dynamic discovery mode is also supported, in which the graphs are detected at run time. The dynamic graph discovery method is based around the rePLay framework presented in [30], which identifies large clusters of sequential instructions as atomic frames. This is heavily based on trace caches and instruction reuse, which fall out of the scope of this architecture review. However, as a small note, the execution of these frames is controlled by branches whose validity is asserted during frame execution, if the assertion fails, the frame is discarded. The notable difference in this approach is the usage

of these frames to discover several subgraphs that are mappable within one larger control flow, i.e. one large frame may not be itself mappable (unsupported operations or to much data dependence) but may contain several subgraphs that are. In contrast, the AMBER architecture begins reconfiguring its hardware only by a threshold of execution of Basic Blocks, which limits its application range. In other words, the CCA detects graphs by utilizing the trace cache principle (an optimization technique already widely implemented in commercial GPPs) and AMBER only by branch detection. Either approach involves dynamically altering the microcode instruction stream during execution of the program, although at different stages in the GPPs pipeline. These instructions will themselves configure the CCA and provide it with the sequence of instructions and data to perform the calculations contained in the associated graph. Therefore, it is also a form of binary translation. In detail, the CCA is a triangularly shaped set of FUs, much like the AMBER accelerator architecture. Significant differences are the graph discovery methods and the supported operations of the FUs in the reconfigurable arrays. In the CCA architecture, two types of FUs were chosen, one for 32 bit addition/subtraction and another for logical operations, with no support for multiplication or shift operations. This decision was motivated by an empirical analysis that indicated that for the most part (over 90%) the detected graphs could be executed without resorting to such operations. Another key difference lies in the internal connections allowed between FUs, and the number of inputs and outputs. The CCA is less flexible, disallowing the connection of FUs on the same level and possessing only 4 inputs and 2 outputs (this decision however was based on previous studies [7] that indicated that a larger size would bring little advantage if memory operations were not supported). Thus, the CCA approach was claimed to be able to map 82% of the subgraphs discovered by their dynamic graph discovery and the AMBER approach 90.48%. Consider however the previously stated point in the differences in graph discovery, which may influence the results of the mapping as different graphs are discovered. A more valid comparison would be the speedup attained: 10% average for AMBER and 26% for CCA. Although the CCA was apparently more restrictive in its reconfigurable hardware, its better performance may be justified by its more comprehensive treatment of the detected graphs and the detection itself. Note however that this complex algorithm for graph analysis incurs in a large memory overhead. In these two architectures memory operations are disallowed because they cannot be mapped into their respective reconfigurable hardware. This is in contrast to the DIM Reconfigurable System, summarized next. Both the CCA and the AMBER architecture permit a certain level of instruction parallelism within their own reconfigurable units, dictated their width.

2.1.4 DIM

The DIM Reconfigurable System [15, 10] also works based on a reconfigurable array and a binary translation mechanism from which the system gets its name. Essentially this binary translation is DFG detection and transformation into configuration for the array. A distinguishing characteristic is that this transformation occurs in parallel to program execution. The GPP accesses instruction memory to execute the program and concurrently the DIM system accesses the same memory to identify mappable instruction sequences. Similarly to the CCA architecture, this allows for

detection of more heterogeneous code regions to map, not being limited to very specific kernels of execution.

Like previous approaches, the moment to switch to hardware execution is indexed by the GPPs PC, and also, the data to be operated on is fetched directly from the register file. This last feature is only possible, in any implementation, because of the tight coupling between the custom hardware and the GPPs pipeline, if such was not the case, a different method would have to be used to fetch operands, such as the Warp Processing approach. However, that approach is limited to sequential or regular memory accesses, whereas the DIM system is capable of accessing random addresses at runtime. By feeding the load/store units with addresses calculated by ALUs in previous rows a memory access becomes possible at any point of array, allowing for mapping of graphs that include these instructions . This also implies support for pointer operations, with the system being able to read and write from and to memory positions not known at compile time.

The reconfigurable array is also tightly coupled to the GPP as another pipeline element. Like warp processing, the optimization is totally transparent to the software toolchain, and unlike the previously presented implementations its greater coarse granularity, much like the heterogeneous architecture for the AMBER processor, allows for less configuration data and its quicker generation with smaller hardware overhead and memory requirements. By being coupled to the processor pipeline, and seeing as though the configuration of the array is controlled by the PC, there are 3 available clock cycles (derived form any GPPs pipeline) before the data reaches the array. In case this is not enough (if there are too many operands to fetch), the processor will be stalled, however, if three cycles suffice there will be no additional delay in the pipeline. This is unlike the CCA architecture in which the configuration is given to the array via the bits in the micro operations themselves along the GPPs pipeline.

In terms of structure the DIM array it is composed of a set of uniform rows and columns containing a number of ALUs that can perform standard mathematical operations, a lesser number of dedicated multipliers and the load/store modules. Although floating point operations are not supported, they could easily be added as another FU in the array, seeing as tough operations with variable latencies are supported. Interconnection, like other implementations, is done with multiplexers, which chose the inputs from the register file or from the previous row in the array. Propagation of results from one row to the next and so forth without operating on the values in the context bus (the bus that carries the values of the graphs throughout the array) is also possible, to permit the reuse of values in operations further down the line without the need of additional memory writes and reads. This also means that only the last value pertaining to a particular target register is actually written to that register, seeing as tough any intermediate results will have been handled inside the array.

The speedups obtained with the DIM architecture were measured in several configurations regarding the size of the reconfiguration cache (which stores configurations ready to be applied when indexed by the PC) and the GPP it was coupled too (simulations done with the Simplescalar Toolset). A speedup of 2.5 on average is claimed, as well as 55% energy savings on average.

2.1.5 Chimaera

A much earlier system was the Chimaera [9] [31], reconfigurable array. The Chimaera architecture is also tightly coupled to the processor, getting the inputs from the register file. Unlike other tight coupled solutions, the Chimarea has a LUT based reconfigurable array, in which the interconnections are made with control muxes. It is most similar with the SCLF in that regard.

A difference however, is the capability to partially reconfigure its RFU, row by row. Each row has its own configuration and can fetch its own operands from registers that shadow the register file or from preceding rows. Seeing as though an operation is composed of one or more rows in sequence, not all the rows in the array are necessary to implement short operations. So it is possible to have the RFU ready to perform a variable number of operations without needing to reconfigure. Of course this involves checking all the mapped operations for each RFU operation encountered in the instruction stream but this overhead might be smaller than the constant reconfiguration overhead one would encounter if several operations were to be performed alternatively for a long period of time. Dataflow detection and the creating of configurations for the array is performed at compile time, meaning an alteration of the toolchain is required. Specifically a specialized compiler and linker that generated and place instructions and configuration data relative to the RFU in the resulting binary. The detection and treatment of the instruction stream as well as management of the RFU is done by additional logic that is integrated with the GPP so it can redirect and coordinate execution.

The configuration overhead is quite large, needing 1674 bits per row. Implementations such as the CCA required only 245 for the entirety of their largest array. Additionally, each RFU operation permits only nine inputs, one output and no memory operations. The multiplicity of operations that can be mapped on the RFU will diminish this overhead over time however.

2.1.6 GARP

The GARP architecture [8] has the GPP control the reconfigurable hardware directly, as well as permitting some control in the opposite direction, such as the array stalling the processor and requesting memory accesses. So, several instructions had to be added to its set (the base ISA being that of a MIPS-II) such as instructions that allow moving data in and out of the array into its own register file. The system also requires modification of the toolchain, adding an auxiliary program to generate a configuration for the compiler and modifying MIPS assembler to support the new instructions.

The configuration program works as a compiler for a dedicated language that specifies array configurations. In other words, there is no automatic graph discovery, and the optimization effort is manual.

The reconfigurable fabric consists of CLBs organized in a matrix, quite similar to ones found in FPGA architectures. Each row contains one specialized control block that serves as the interface between the array and the GPP. These blocks can cause interruptions of execution and memory transfers to or from the array. Besides these, a larger number (16 + 7 per row, the 16 being aligned

with the processor data word) of logic blocks implement the actual arithmetic and logic. Like Chimaera, several configurations can be in the array at once as one operation may not need to utilize all the available rows. Like an FPGA fabric connections between adjacent horizontal and vertical CLBs are allowed (which permits carry logic), additionally, 4 buses carry data in and out of the array and also serve to supply reconfiguration bits, a separate wire network is used to interconnect CLBs. Each block can individually be set up to serve as a pair of LUTs, a dedicated 3 input adder/subtractor/comparator, shifter or a 4-way multiplexer. Though multiplication is not supported directly, it can be more easily mapped by the 3 input adders. The previously mentioned memory accesses are performed trough a memory bus and by forwarding control signals form the operating CLBs to the control CLBs. The array views the memory structure as a GPP would, supporting caching and page faults.

Of note are the separate clock domains, the processor clock and the array clock which operates is sequential logic. This clock is active for a specified number of clock cycles, this value is determined at the start of each array instruction, meaning that when no instruction is being performed there is no clock being fed to the array, avoiding useless propagation of data. Also, the array's clock counter serves to coordinate access to its results and to check when reconfiguration is allowed.

To reduce the configuration effort, the wire network that interconnects does not need any configuration. Instead, it is organized in such a fashion that only one CLB can drive a particular wire and, thus, all the others can read it. The network is then broken into wire segments, to allow different data contexts in the fabric. So, to compare with the Chimaera architecture, configuring each row of the GARP fabric requires 1536 bits, roughly the same. However, since the CLBs are more coarse grained, the number of rows needed to implement a custom instruction may be smaller.

Chapter 3

Design Goals and General Approach

3.1 Objectives

Generally speaking, the final objective would be to develop a system that is capable of automatically generating a functional hardware description, given information regarding the data and control flow of a program. With that description information, map one or several statically held hardware elements which perform computations to a portion of FPGA fabric. Afterwards, configure the placed hardware by routing its operands and results through a simpler set of control bits that dictate the interconnections of the selected library elements. Thus creating a Reconfigurable Fabric (RF), which performs calculations that are equivalent of those found in the originating software.

The more adequate manner in which the elementary operations are to be stored and how the description is to be constructed is also a target of study, as it dictates the underlying architecture. Regarding this aspect, the Reconfigurable Fabric should be viewed as peripheral on the GPPs bus, thus making the access to/from the fabric much more transparent and standardized. Since the MicroBlaze (MB) soft-core processor will be utilized, using an FSL (Fast Simplex Link) connection is also an option. Added to this, allowing access to the fabric by multiple soft-core processors would be a secondary objective, as well as exploring parallelism between the GPP and the fabric, parallelism in the fabric itself and sharing mapped resources between graphs.

3.2 Design Rationale

There are several reasons for this choice of functionality. For instance simply to test of a different approach and to develop a system of mixed granularity and more flexibility that could adapt to different embedded application requirements on the field. The usual operations in embedded systems do not require fine grained application, however, by simply editing the libraries in the reconfigurable module, more or less granularity could be attained. Also there is yet to be a system that can map all the desired code regions to hardware without considerable hardware and temporal overhead.

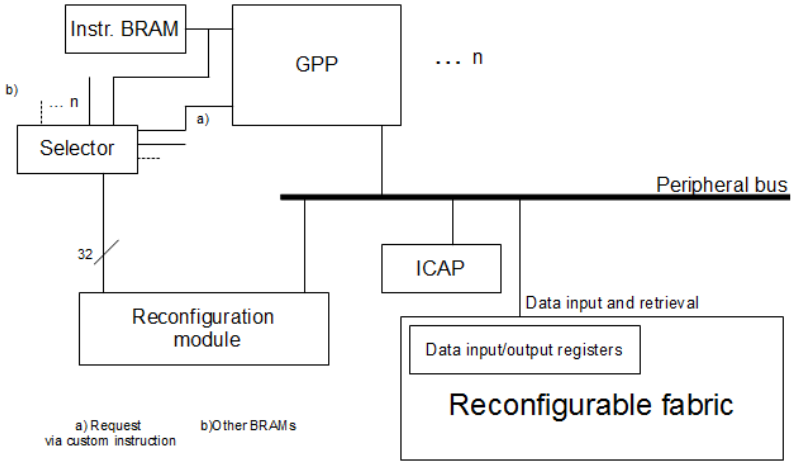


Figure 3.1: System Overview

Such an architecture would also allow for loose coupling to any kind of GPP, so long as the respective instruction set is supported by the reconfiguration module. Although this introduces greater communication and hardware overhead, as well as a greater temporal overhead in terms of mapping, the advantage is the application flexibility and the potential to have a completely standalone reconfigurable fabric that can be applied transparently. Also, the loose coupling doesn't require altering the GPPs pipeline, ideal for closed soft-core processors, and doesn't require several reconfigurable fabrics and associated hardware to allow acceleration for several embedded soft-cores in the same FPGA.

This would allow the system to steer away from any form of binary translation or transformation of any kind besides the interpretation of the instructions read from the GPP. This is because binary decompilation is computationally expensive, and a very robust decompilation mechanism would have to be put into place in order to ensure proper functioning (which is very time consuming). Also avoiding the alteration of the original binary is also somewhat desired to ensure that the GPP can continue to execute the program normally in case the reconfigurable fabric is fully mapped, or in use by another GPP.

To properly test the system and compare the results generated by it and non-accelerated alternatives, the utilized benchmarks will depend on the final status of the developed accelerator regarding its support for DFGs and CFGs (CDFGs).

3.3 Preliminary Proposal for System Architecture

The following section details some preliminary approaches to the layout of a Reconfigurable Fabric and its interface with the remaining system modules. These layouts were developed with a system that utilized an ICAP peripheral in mind. Available in some FPGAs, ICAP allows for editing the FGPA's run-time configuration based on partial bitstreams that target a specific area of the FPGA.

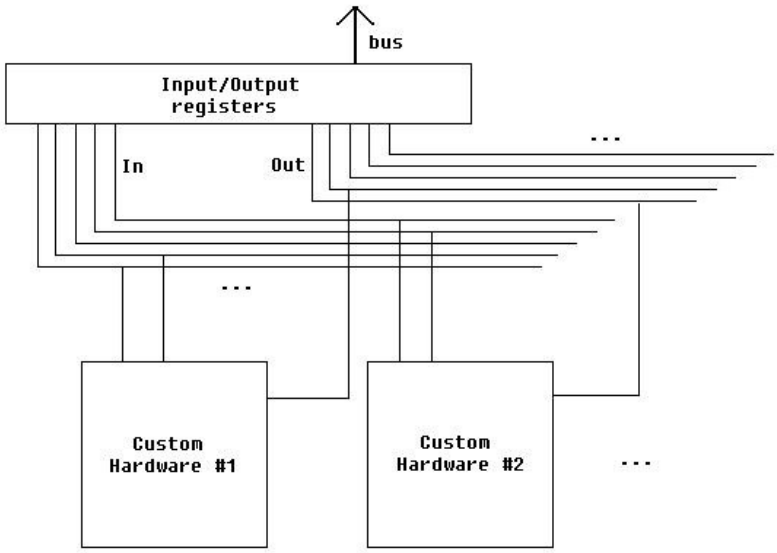


Figure 3.2: Fabric Alternative 1, simple horizontal wiring

The reconfiguration module would be able to access a limited number of Block RAMs (BRAMs) depending on implementation. The proposed architecture for access to instruction memory would make the GPP triggering the reconfiguration process transparent to the reconfiguration module. Once the bitstream is constructed the ICAP module would map that into the reconfigurable fabric (the ICAP module may be connected to a different bus [32]). Also, the reconfiguration module would hold information about the already mapped hardware, stored bitstreams, construct graphs by reading instruction memory and may be responsible for fetching data to feed to the fabric. The GPP would have to run a single thread application with no operating system. Additionally, either the GPP would be required to have at least two custom assembly instructions to send signals to the reconfiguration module in order to instate graph discovery for a region of code encapsulated by those instructions, or the reconfiguration module would itself monitor the instruction bus and contain algorithms that performed the task of finding appropriate regions online. Still to determine is the method of data input into the fabric without involving direct access to the register file. Something similar to the Warp Processing architecture would be ideal, with direct access to data BRAMs. However, this was functional in that design because the data accesses were sequential. In this case an input of n from any memory positions are desired. So, it is necessary to store information about the memory addresses associated with each custom hardware module as well as the PC that triggers its execution.

Regarding fabric architecture, a rigid layout is difficult to envision without knowledge of the method for data retrieval and storing of other configuration data. However a few approaches are conceptualizable.

One alternative for the fabric layout is as presented in figure 3.2. Assuming ease of bistream concatenation and mapping, as well as ease of connection of the custom hardware outputs to the static fabric interconnection network, the fabric would be greatly simplified. Horizontal wires would run at the top of the fabric either providing inputs or fetching outputs from the custom

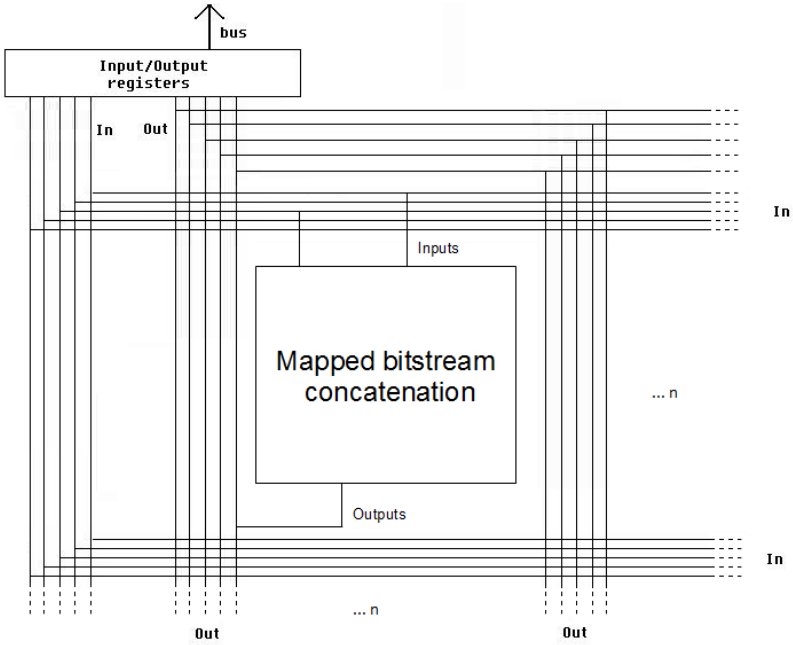


Figure 3.3: Fabric Alternative 2, wire matrix

hardware. The interconnection overhead would be minimal, and ideally with no delay. The inputs/outputs would be fetched/put into/from a register bank that would be seen, from the point of view of the peripheral bus, as a simple memory device. Of course this would limit the amount of operands that could be set as inputs and, consequently, the number of custom hardware blocks placed (as they could produce only as many outputs as the registers allow). However, previous approaches show that, for embedded applications, the number of inputs and outputs settles at relatively low values (around 9 to 4 inputs and 1 to 4 outputs depending on graph discovery and architecture). So, adding this limitation should not be restrictive. The outputs would appear in the output positions a determined number of clock cycles later. To synchronize with the GPP, a clock inhibiting mechanism can be used (i.e. stall the processor).

In case a more complex fabric is required, an approach such as the one in figure 3.3 would be a possibility, dividing the area in cells. Although a decrease in the simplicity of the fabric, it may facilitate the interconnections to the input/output registers. However, this would limit the area permitted for each custom hardware module. One alternative is to replace each said module with a simple library element instead, which would eliminate the need to concatenate bitstreams, and interconnect each element with a switch matrix similar to the SCLF. Of course this would, in turn, limit the area of the library element to map to each cell.

To support clocked custom hardware, a clock signal is required. Since different custom hardware modules will have different library elements the maximum clock permitted may vary. A simplification, as opposed to having several clocks, is to have a clock generator, as in figure 3.4 for the whole fabric. This clock would be regulated by the maximum delay found in the mapped elements (adjusting at every mapping).

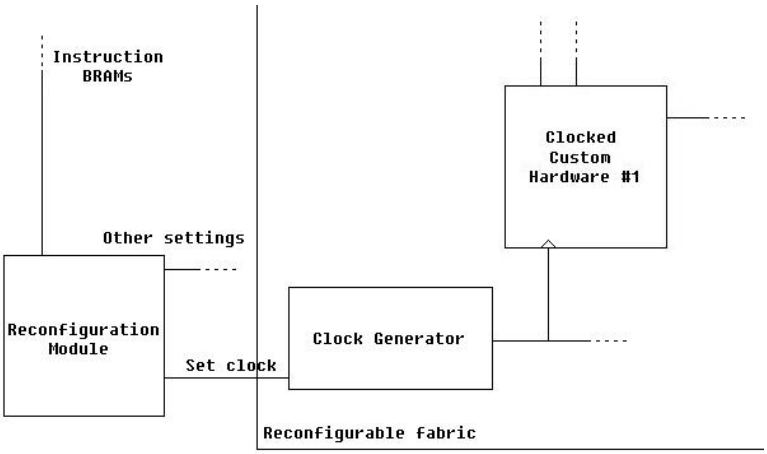


Figure 3.4: Clock generator, would supply clocked hardware elements

3.4 Graphs

As stated in previous chapters, constructing CFGs and DFGs is a necessary step in order to implement a data-path. Even without the notion of parallelism, it is necessary to identify the input data of a particular region of software along with the operations performed upon it in order to design hardware that replicates that behaviour.

To specify, CFGs dictate a flow of control operations whose results control execution of loop based operations. DFGs represent solely the data-path itself, or in other words, useful data. Operands and results propagate through them according to the CFG which also dictates when and at which point the computation for that DFG is completed. Both will be generically referred to as graphs from now on.

The graphs that will be target of study will derived from computational loops. In other words, constructs such as *for* and *while*, or equivalent, result in cyclical computations upon data that, if repeated intensively, will become the most time consuming operations of a program. Control structures such as *if* or *case* (which, at assembly level, are branch instructions), delimit regions of code in segments called Basic Blocks, which will compose the control represented by CFGs. So, a graph may have one or more points where its execution is completed.

In section 4.3 are presented the considered approaches for an architecture capable of implementing the computations and control flows of graphs of this kind.

3.4.1 Graph Characteristics

In an abstract fashion such as the example in figure 3.5, a graph may be represented by placing operations in individual rows, each row containing operations that may be executed in parallel and which propagate their results to any following operations spanning either one or more rows. In the example given, no manner to control the execution is represented, i.e. a CFG.

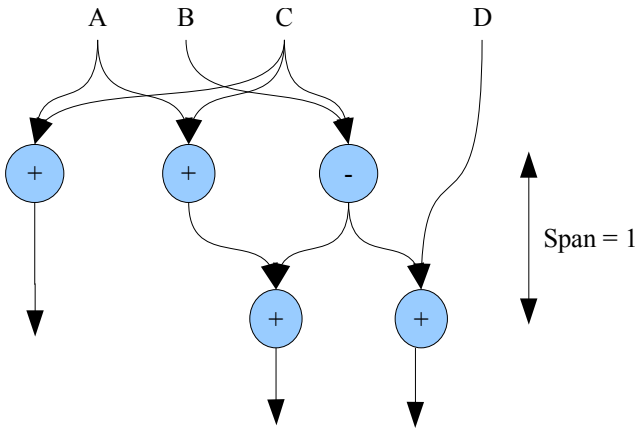


Figure 3.5: An example Data Flow Graph detailing data dependencies and operations

Logically, the type of operations found within the graphs, their data dependencies, the amount of parallelism and the amount of input data itself directly dictates the type of hardware more appropriate, or strictly necessary for their support, as was seen by the implementations summarized in chapter 2.

Adding to these, there is also the presence of load/store instructions (memory access), the span of connections, the manner in which data is feed into the graph at each iteration and whether or not the graph contains more than one exit point. If a graph contains one exit point, it is considered atomic, if several, it is named non-atomic. So, four possible combinations exist considering the exit points and the absence or presence of memory operations. Of course the remaining graph characteristics make for many more types of graphs, but these two especially create combinations which require a more sophisticated hardware layout in order to support them (though the span of connections introduces similar considerations). Also, non-atomic graphs may also be treated as atomic iterations. Further detail on this is found in section 3.5.2, in which the types of graphs thus supported by this implementation are explained. Section 3.5.3 details why some graphs where not considered.

Regardless of necessities found at hardware design level, from a purely conceptual point, these types of data flows are good candidates for optimization for the sheer level of instruction parallelism they possess. The higher the parallelism of a graph the more potential is present for acceleration. For instance, in a MicroBlaze processor, a machine cycle (i.e. one assembly instruction) may take as long as 3 clock cycles (although an instruction is completed every clock cycle due to the processor's pipeline). If the example in figure 3.5 represented a group of MicroBlaze instructions, they would take a total of 8 clocks cycles to execute in the best case scenario. That is, without intermediate operations to move values to and from registers and assuming that the instructions themselves (*opcodes*) were readily present in local memories or cache. On the other hand, a parallel execution by an architecture capable of being feed the input values A, B, C and D simultaneously would complete execution in only 2 clock cycles. In truth, many more communication delays are introduced, as will be show later, but acceleration is still possible.

3.5 Graph Extractor

In order to generate sets of graphs such as these, a Java implemented Graph Extractor [33] was utilized, which also provides information about the CPU registers involved in the operations.

In related works, graph detection was done at either online or offline time, with more or less intrusive approaches. Generally, online approaches observe the running program and, as such, extract graph information from low level execution, offline approaches attempt to generate the same information from high level source code. Here however, the analysis is performed offline and at low level.

The Graph Extractor analyses the instruction stream of a program running over a Microblaze simulator. So, what is observed is the sequence of instructions that the simulated program would perform at runtime. From there it is possible to extract information regarding repetition of particular segments of code delimited by branch instructions, i.e. Basic Blocks, to determine data dependencies and, also, to determine a particular repetitive pattern of a number of these Basic Blocks.

Essentially, the Extractor works with three types of instruction trace units [34] [35]. They are: the instructions themselves, the BasicBlocks and SuperBlocks.

SuperBlocks consist of sequences on BasicBlocks which, at runtime, contain only forward jumps. To clarify, a BasicBlock is delimited by a branch, and the branch destination is either dictated at compile time of by the operators passed to the branch. During runtime (in simulation) if a sequence of BasicBlocks is detected in such a manner that none jumps to an address lesser than its own (i.e. backwards), those BasicBlocks can be grouped into a SuperBlock. So the SuperBlock is a structure that can only be constructed at runtime.

In order to detect a graph, these instruction trace units, each of its own granularity (SuperBlock being the largest), can be grouped into a MegaBlock, which is a repeating pattern of the selected trace unit. Analyzing instruction stream in order to identify a pattern results in a large working set of data. Hence the notion of the SuperBlock.

So, a MegaBlock may be constructed from a trace of SuperBlocks simply by stipulating that the starting address of each SuperBlock (which is the address of first BasicBlock which composes it) is now an identification. This way a repeating pattern SuperBlocks can be found. Thus, a MegaBlock (graph) may be constructed by analysing a much smaller group of identifiers, the SuperBlock addresses, instead of the complete instruction stream.

The output generated by the Graph Extractor is then utilized by the tools explained in chapter 5. The output files themselves are explained in the following section.

3.5.1 Graph Extractor Output Files

The Extractor generates several output files per graph it detects. These files detail the operations, connections and Microblaze registers involved in the execution of the graph. It is capable of detecting both atomic and non-atomic graphs and displaying the information accordingly, as well as performing a number of other instruction stream analysis tasks.

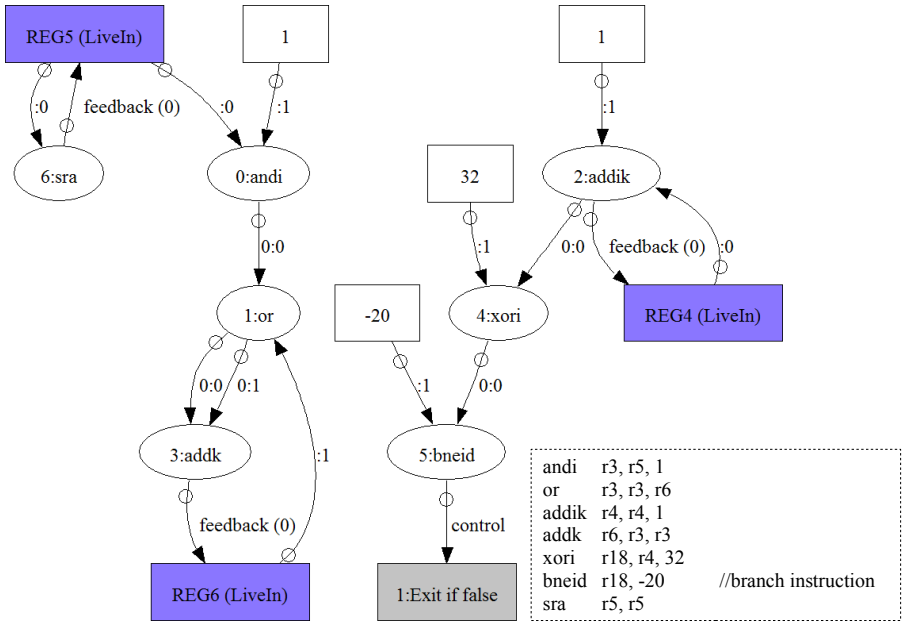


Figure 3.6: An example of a graph that was built by assembly analysis, along with the originating code

Besides these files, which are inputs to the following tools in the toolchain, the Extractor generates graphical representations of the graphs. Such an example is the graph in figure 3.6.

This graph already presents MicroBlaze instruction set instructions and their connections, as they were determined by analysis (although still detached from any hardware structure). The origin of the input data is already identified to be a group of registers from the MB register file (one register per 32 bit operand). Likewise, so is the destination of the outputs, although those connections are not represented (for clarity). The output registers need not be as many as the input registers and there is no direct relation between the two. Any result of any operation may be placed at any output register. This graph representation contains both the DFG and the CFG for the cyclical computation it represents. The righthand operations are performed only to have their results checked by the branch operation (which is an exit of the graph), thus triggering its completion. Although in this example the only data inputs originate from register file of the MB, the already mentioned memory accesses are much more commonly found (as data intensive loops require more operands to be moved and result in more values being altered). In many instances, some operations are executed while having one of their operators constant through all iterations, i.e. those operators are either compile time constants or observed to be constant during instruction stream analysis. So, support for setting these values in hardware will also be necessary (as they cannot be retrieved from the processors register file like the remaining inputs). The graph is also atomic, having only one exit point. Also show in this representation, the execution of the graph continues after each iteration by propagating the results of iteration n to iteration $n + 1$ (i.e. connection of one or more of its outputs into the inputs of the operations in previous rows). Thus, eventually, a value will be fed to the control instruction that will end the execution.

```
--Megablock Stats--
#iterations:29
#original instructions: 7
original instructions x iterations: 203

--Liveness Analysis--
3 live-ins (REG4, REG5, REG6)
5 live-outs (REG18, REG3, REG4, REG5,
REG6)

--Misc--
Does not have memory store instructions
Does not have memory load instructions
#Side-exits:1
startPc:0x880001A0
CPL (AtomicGraph):0
CPL (NonAtomicGraph):0

--Exit Addresses for NonAtomic Graphs--
Exit1:0x880001BC
```

Figure 3.7: Example Stats File - These output files contain information about what processor registers are involved in the graph and where, in memory, the graph is located.

The data in this representation is also kept in the two files utilized by the developed tools. They are the graph Stats File as exemplified in figure 3.7, and the graph Operations file, in figure 3.8. These two examples of these files explained here are relative to the graph presented previously in figure 3.6.

Concerning the graph Stats file, it presents a listing of the input and output registers of the MicroBlaze, that is, the registers that contain the input data to be given to the RF and the registers to which the output data of the RF will be stored too. It details also the presence or absence of both load or store instructions and, importantly, the starting PC of the graph. The previously mentioned PCs the Injector reacts to are these extracted values that indicate where, in memory, the repeating instruction pattern begins to occur.

Related to this parameter are a few hardware design choices that are not immediately apparent. As stated before, a MegaBlock is a sequence of SuperBlocks, so, as an example, consider SuperBlocks named *A*, *B*, *C* and *D*. Now consider any two sequences of these identifiers which start at the same identifier, for instance, *A-B-B-C* and *A-D-D-C*. These two sequences would form, in turn, a sequence of instructions expressable as a graph, and feasible for implementation. However, the Injector contains only a graph table which allows it to associate each PC with an ID and trigger the functioning of the system for that graph. In this case, both graphs would have the same starting PC, as they start at the same block. So, no obvious solution is present as to how to distinguish between graphs at runtime utilizing only the memory address present on the instruction bus. More data would be required at runtime, and that would be the sequence of the SuperBlock's PCs themselves, i.e. a detection of the sequence *A-B-B-C* or any other in question. In section 4.2.1, a

```
OP:2
operation:addik
level:1
numInputs:2
inputType:livein
inputValue:REG4
inputType:constant
inputValue:1
numUsedOutputs:1
outputId:0
outputFanout:1

EX:5
operation:bneid
level:3
numInputs:2
inputType:internalValue
inputValue:4,0
inputType:constant
inputValue:-20
numUsedOutputs:0
exitCondition:false
exitId:1
```

Figure 3.8: Example Operations File - Excerpt from a Operations file, displays operation and connection information as well as to which GPP registers the outputs should be redirected

possible hardware support to handle this issue is briefly discussed.

Also in the Stats file, is the number of total instructions that would be required in software to execute the graph. From these values a rough estimate of performance increase can be derived as will be shown later.

Regarding the file detailing the operations of the graph, it is a simple listing of MicroBlaze instructions detailing the instruction itself, where its operands originate from, the number of useful outputs from that operation and the fanout of each output of an operation. The inputs of an operation can either come from the input registers themselves, from outputs of other operations or are constant. Operations of type *EX* are branches, and, as such, the exit points of the graph. They contain a extra field indicating if the branch is to be taken if the condition expressed by it is either true or false. For instance *branch if greater or equal* or *branch if not greater or equal*, thus allowing for any combination expressable in software. From this information, the placement of operations and routing of operands and results in the selected architecture is performed, as will be shown in section 5.1.

3.5.2 Supported Graph Types

The implemented system was constructed in order to support automated hardware description for graphs such as the one in figure 3.6. In short, this graph is atomic, and possesses no memory accesses, receiving and outputting all its data back into the register file of the MB. These are the kinds of graphs the implemented architecture and toolflow is capable of executing in custom made

hardware. However, non-atomic graphs are also supported by treating them as atomic iterations, equal to the concept of frames used in the rePlay framework [30]. In short, a non-atomic graph may end its execution at any of the intermediate exit points it contains, thus prompting the recovery of the output data at that point and the return to an appropriate position in code memory. However, the entire iteration may simply be invalidated, returning to the very start of the graph.

So, as is currently implemented, if any branch triggers, the iteration of the graph is aborted, and execution is returned to the beginning of the corresponding software region while returning the results of the previous iteration. The software execution would continue normally from that point and branch out at the same branch instruction that had caused the hardware to complete its execution.

This will become clearer once the hardware structure is presented but to summarize, the current architecture does not support memory accesses and supports atomic graphs and graphs with multiple exit points treated as atomic iterations.

That leaves three possible combinations of graph types that, although considered, were deemed more appropriate for later design iterations. They are explained in the following subsection, subsection 3.5.3.

3.5.3 Unsupported Graph Types

Unsupported characteristics of graphs are, as mentioned, memory accesses and graphs with multiple exit points (in which intermediate results may be recovered).

Regarding the memory accesses, the reason as to why they are made more difficult to support is the very nature of the typical processor and memory structure. Such as a von Neumann architecture, as is the case for the system utilized for development. Unlike other operations found within graphs, such as additions and logical operations (exclusive ors, barrel shifting, etc.), memory reading and writing are, obviously, not mathematical in nature, and require accessing and external peripheral, i.e. a memory. When a processor wishes to read data or to fetch instructions from memory, it requires support for pipeline stalling to account for the access delay and dedicated interfaces to communicate with memory controllers. So, to store and retrieve data from a reconfigurable fabric this behaviour would have to be mimicked (as it is with any memory accessing peripheral). The consequent problem is developing a hardware structure flexible enough so that it can both permit memory access and maintain a coherent flow of data by controlling execution of operations in a much more strict way and so that it can also be easily scalable.

Implicitly, this forces the internal architecture into something considerably more rigid, and data output and input points would have to be defined. To clarify, the point of execution in which the graph might necessitate to store or retrieve a value from memory could be any, and so, hardware to execute it would have to be prepared to properly wire such data to and from any random location (i.e. operation) within the graph.

Additionally, consider a graph driven from a high level loop that retrieves information based on the value of a data pointer. Such an access pattern might be irregular, and so, determining what memory positions to access would not be trivial without information contained in the running low

level code. However, the DIM Reconfigurable System deal with this problem efficiently, at the cost of having LSUs (Load and Store Units) at all execution levels.

As for non-atomic graphs, several things would be required for their support. As stated before, a graph is composed of operations either derived from online or offline analysis, and the control structures associated delimit regions where the execution either continues or stops in order to, alternatively, execute something else. In terms of code, this corresponds to Basic Blocks which are all executed in sequence until a branch condition is such that the execution stops. So, when executing graphs in hardware, and the execution terminates normally at any one possible point, it is necessary to know from which processor instruction to continue execution in software. In other words, upon returning from hardware execution the processor now contains updated data, and must start executing instructions from a point where context is properly maintained. This implies keeping track of memory positions associated with each possible branch, and also to know which results to return to the processor at each of those branches. That is, a branch located in the middle of the graph will most likely prompt a return for data found at that point in the execution. Supporting the simultaneous connection of any operation output directly to the outputs of the reconfigurable fabric is, most likely, not trivial to manage.

Chapter 4

Prototype Organization and Implementation

The previously outlined functionalities and architectural layouts were not all put into place in the working version of the reconfiguration system.

However, and although it differs from the defined preliminary approach, mainly in terms of architecture, it still covers the main objective of generation of reconfigurable circuits from machine code for an FPGA target.

The implementation alternatives are further explained in the following sections but, in short, the different outlined approaches were developed mainly due to consideration of what tasks were and were not appropriate for online and offline execution. This, coupled with tool flexibility and ease of development led to a few distinct layouts.

In general terms, the implemented tool flow allows for the analysis of a given program (compiled for an embedded environment) and extraction of graph information from that program. With that information a combined hardware description based on Verilog parametry and language constructs is generated. Along with that, information regarding the runtime configuration of the Reconfigurable Fabric (RF) is created along with assembly level code that permits writing to and reading from the RF. This RF allows for execution of several graphs, although only one at a time, according to its current runtime configuration. Both the toolflow and the capabilities of the RF and the method for its description are explained in chapter 5 and section 4.3 of this chapter respectively. In order for the system to function, no alteration of the running binary is necessary, there is a single interfacing point between the GPP and the entirety of the reconfigurable system that can be easily placed or removed, as it's interfaces, and the interfaces of all system modules, are standard bus connections. So, the modules of the system retain a considerable level of transparency, allowing for their individual replacement or altering of their interfaces without compromising system functionality. This leaves room for several possible alterations with potential performance improvements as explained in chapter 7.

The development platform was an FPGA development board. The were no hard requirements for the platform except for one: support for ICAP so as to allow runtime reconfiguration of the

conceptual Reconfigurable Fabric. So, the selected platform was a Digilent Atlys which is built around a Xilinx Spartan-6 LX45 FPGA. In this board there are two external memories present, a non volatile flash memory and a volatile DDR2. As will be understood later, these two memories dictate much of the system layout. Attached to this platform are the standard development environments for designing soft-core processor systems, namely, Xilinx's ISE Design Suite. Included in this toolkit is Xilinx's Platform Studio (XPS), which is an embedded processor design tool. It allows for system design by interconnecting desired IP (Intellectual Property) cores and allows for integration with software development, automatically generating a bitstream which contains both the hardware design and the software application properly initialized in the processors associated program memory. Also in the tool suite there is Xilinx's PlanAhead environment which allows for manual placement of modules within the FPGA. This feature seemed promising in regards to one of the initially considered approaches as is explained in the following sections.

4.1 Architecture Overview

The architectural modifications that were made to the preliminary design, in an early stage after testing and choosing development platforms, were determined by what toolflows were available and what were the limitations of these tools. A more detailed study into what was and was not feasible for implementation with said tools and platform dictated these modifications. However, the basic functioning remained the interpretation of the instruction stream of a GPP and, with appropriate treatment of that information, generate reconfiguration data for a module capable of altering its internal operations, thus producing data for a particular software intensive kernel.

To reiterate, the preliminary design described a system in which a GPP would execute code located in BRAMs (Block RAMs). This code would have to be altered at a few set points, that would have to be manually determined, with custom instructions that would delimit a code region to be analyzed and mapped to hardware. The analysis would begin by capturing the instructions being read into a reconfiguration module via a tap in the GPPs instruction bus. This runtime analysis would determine DFGs and CFGs and associate these operations to previously stored bitstreams, each representing an operation of finer or coarser granularity. These would then be merged to form a final bitstream that would be mapped, via a PLB (Processor Local Bus) ICAP peripheral, onto the Reconfigurable Fabric (RF). In addition to this, data regarding the currently mapped operations and their connections would have to be kept in this reconfiguration module and it would also have to intervene the next time the GPP began to execute the now mapped hardware, shifting its execution from software to the RF.

Upon further inspection, several implementation difficulties, and some concepts left vague, around this design lead to the modifications and developments which are explained in the following sections.

Section 4.1.1 details a first iteration and section 4.1.2 presents final adopted architecture overview and its functioning.

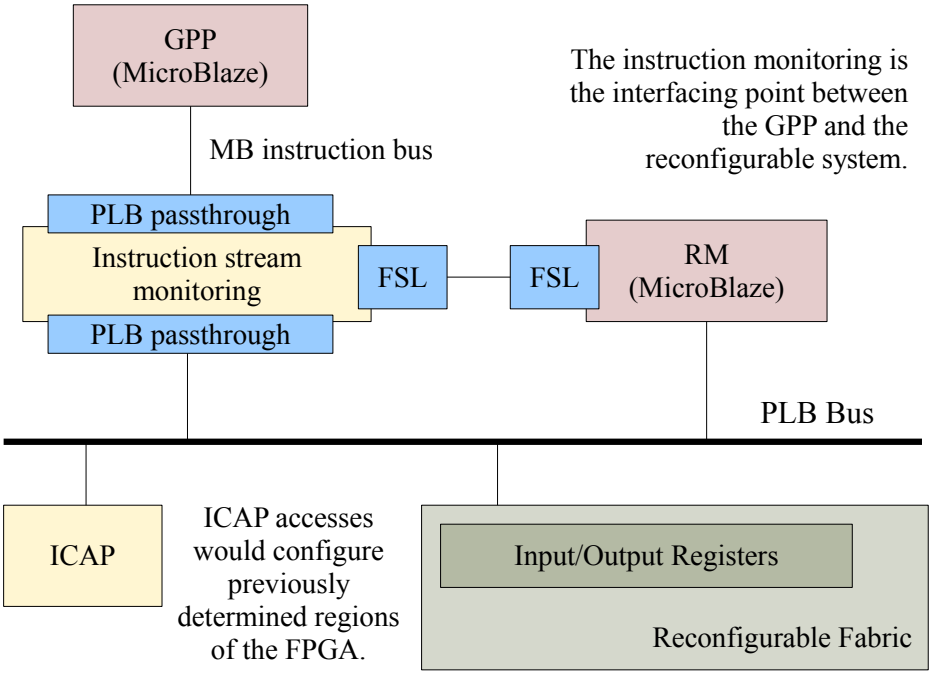


Figure 4.1: Initial Architecture

4.1.1 Initial Architecture

This initial layout was similar to, but more defined, than the preliminary designs.

First, the location of the benchmarking code for the GPP was changed, due to code size. Initially conceptualized to be in BRAMs some benchmarks proved to large for the available capacity of these memories, which are available within the FPGA. So, the code to be optimized was relocated to an external RAM. This relocation implied a different interface for instruction stream monitoring. The monitoring and control of the instruction stream had to be moved to the GPPs instruction bus that accesses external volatile memories (containing the application loaded by a bootloader), its PLB interface.

The second crucial change was the method through which portions of code are identified as good candidates for dedicated hardware, and how this dedicated hardware is produced and configured. Initially, the idea of a tap into the GPPs instruction bus was proposed so the instruction stream could be monitored and analyzed in real time, thus producing equivalent hardware. However, several setbacks quickly appear with this method. Analysis of the instruction stream is an algorithm intensive task (potentially more so than the program which is being targeted for optimization). So, being that this kind of analysis could only be performed by a soft-core processor (or a similar method through which implementation of synthesis tools could be supported) each instruction read by the GPP would have to be captured into a second MicroBlaze and properly interpreted and inserted into a CDFG being built at run time. This would of course imply the buffering of the captured instructions (perhaps into unpractical sizes) and maybe a faster clock frequency for this processor alone in an attempt to diminish delay and buffer size.

Simply put, the overhead of the complete task of analyzing the instruction stream, constructing hardware for those computations, mapping it to the reconfigurable fabric, and, from that point on, intersecting to GPP at the proper moment at which to use that hardware becomes to large to be acceptable as an online task. So, many functionalities were distributed amongst system modules, with most being changed to offline tasks, leaving as the only online functions the reconfiguration of the fabric with offline generated information, the intervention to switch execution to hardware and the actual GPP initiated communication with the fabric so as to utilize it.

Thus, the tasks set to be offline are performed by a toolkit developed to extract DFGs and CFGs from a particular compiled program (ELF file) and generate hardware and reconfiguration information. The toolkit, its functions and the generated information are explained in detail in chapter 5.

So, the approach was changed to an architecture that could interface with the GPPs instruction stream and, by monitoring it, detect the start of regions of code previously transfered into hardware by the offline tools. This allows for the reduction of connections between the GPP and the modules responsible for reconfiguration (i.e. the system elements become more transparent than the initial approach). The module responsible for interfacing with the instruction bus, later developed as the PLB Injector, communicates with a soft-core which performs reconfiguration tasks (namely the reconfiguration of the RF with tool generated information) by FSL (Fast Simplex Link).

The reconfiguration module (RM) was chosen to be a Microblaze for the ease of debug found at software level, although, as will be explained, it could easily be removed from the final design without loss of functionality.

The following subsection explains the further changes made to this architecture upon a second iteration and the data flow and functioning of the system. The most significant change was the complete abandonment of the ICAP method of configuration. This choice is explained in section 4.3 detailing the internal composition of the Reconfigurable Fabric, which details the gradual shift to an architecture that does not need the kind of capabilities that ICAP provides. Any alterations performed over the reconfigurable lead, of course, major changes in how reconfiguration information is generated and how the overall system works, as it is the main module of the design.

4.1.2 Current Architecture

The currently implemented system has an architecture that is represented by figure 4.2. This final implementation retained the basic functional layout that was aimed for by the architecture presented in the previous subsection. However, as stated, the key differing point is the lack of any ICAP peripheral, as such functionalities became unnecessary for the chosen RF architecture. Also, it permits that the system be implemented in any FPGA target that does not support this feature.

So, the system is now composed of considerably discrete elements: the PLB Injector, which monitors and alters the contents of the GPPs instruction bus and is further explained in section 4.2; the GPP itself, a regular MicroBlaze soft-core; the RM, a MicroBlaze utilized for reconfiguration

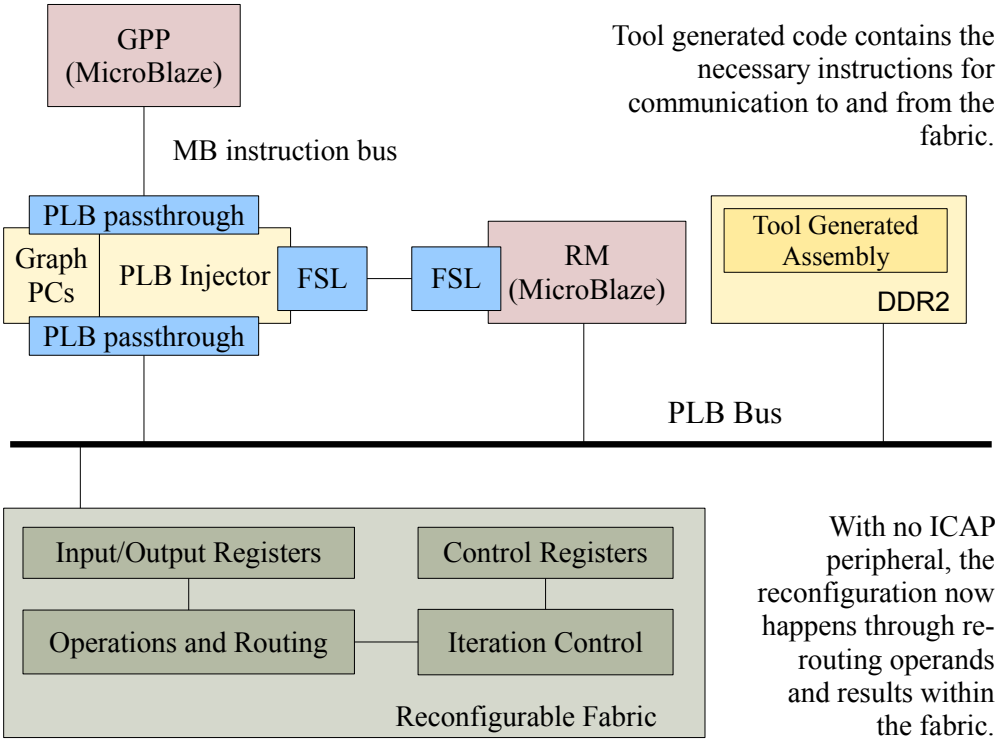


Figure 4.2: Current Architecture

tasks; segments of tool generated code placed in DDR2 memory which are explained in section 5.2; and the RF itself which functions solely through memory mapped registers and whose final adopted architecture is described in subsection 4.3.3. The system is built around the PLB bus, utilizing only standard interfaces. The code to optimize is placed in flash memory and loaded into DDR2 at boot.

Since every single instruction to be executed by the GPP has to be monitored in order to for the system to be aware of its current state, the use of cache had to be disabled. If the GPP fetches a number of instructions into cache, it will later consult these memories to retrieve the instructions and, so, they will not pass through the bus monitoring peripheral, the Injector.

The functioning of the system is as follows, assuming that the starting point is one where all the configuration information has been generated and all graphs of interest have been constructed as hardware. The GPP begins execution of the software bootloader present in its local code memories (BRAMs) in order to load the desired program into volatile a external memory (DDR2) from the flash memory. Simultaneously, the RM performs a similar operation, copying to known DDR2 memory positions segments of code that include operations that store/load values from/to the GPPs register file to/from the RF. To clarify, these Code Segments (CS) are also tool generated and held statically in the RM's BRAM program memory. They are written to be executed by the GPP in replacement of the code it would normally execute to perform the computations now mapped to hardware (each graph being associated to a particular segment of tool generated code) and are further explain in chapter 5.

After the GPP has loaded the program, it then executes it as it normally would, without interference until the PLB Injector stalls it by injecting into the instruction bus a branch that maintains the GPPs PC at the same value. The moment where the stall occurs is dictated by an internal graph table that associates a graph ID to a specific memory address. The memory addresses contained in this table are those that indicate the start of a block of code who's operations have been mapped into the RF, therefore, the Injector stalls the GPP thus beginning the process of utilizing the generated hardware. Simultaneous to the stall, it communicates the graph ID to the RM via FSL. The RM the consults the information statically held in its own code regarding the configuration for that particular graph. It then reconfigures the fabric so it performs the given graph. The reconfiguration information is, amongst others, the routing setup of the operations (which outputs connect to which inputs). The RM then responds, via FSL, with Microblaze instruction set instructions that will, when placed in the instruction bus by the Injector, cause the GPP to branch to a memory position that contains the tool generated code segment that communicates with the fabric. From this point, neither the Injector nor the RM are required to intervene. The code now being executed loads the operands contained in the register file to the memory mapped input registers of the RF followed by a start signal. While the RF is operating the GPP checks for a completion flag. When done, the GPP retrieves the results to its register file, and then returns to its previous location in the original program code via a branch back that is part of the code segment. The program execution continues as normal, now that the values in the register file are such that the branches delimiting the code blocks mapped to hardware will fail, i.e. the graph will not execute in software.

This way, the intervention of the reconfigurable system happens in a very punctual manner and in a completely transparent way to the processor and its internal register values (no internal modifications are necessary).

4.2 The PLB Injector

As explained, a method was required to tap into the GPPs instruction stream in order to have that information redirected to a module responsible for performing reconfiguration tasks. Since it was stipulated that the GPP would be accessing code in external memory, this module needs to function as a passthrough for the Microblaze's PLB instruction bus.

So, this peripheral has two PLB ports, one serving as a slave and another as a master. The Injector acts as a regular PLB interface from the point of view of the GPP, permitting this processor to connect its master IPLB (Instruction PLB) interface into the Injector's slave port as it would connect it to an actual bus. The master port of the Injector then connects to the bus itself. While it allows for the bus signals to pass unaltered, it captures them in order to send them to the RM for processing and will also alter the instruction being returned into the GPP by the bus.

While initially this module was designed to only retrieve instructions from bus, its functionality was quickly expanded to also alter the instruction stream once the system architecture attained a more solid design. Since the complete system aimed to not alter the running binary, there was no evident way to trigger the use of the RF after it had been prepared for use. So, the Injector permits

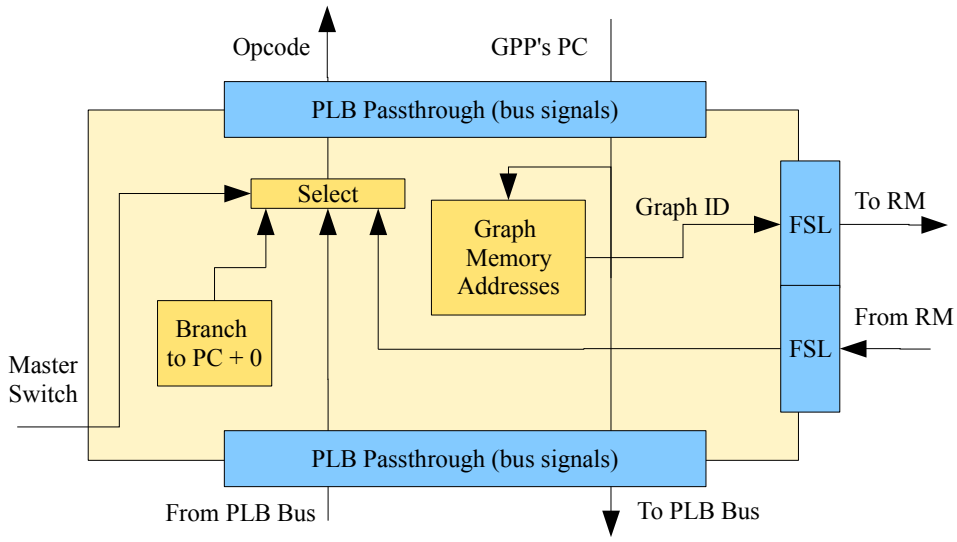


Figure 4.3: PLB Injector - While waiting for a response from the RM, the Injector maintains the state of the GPP by branching it to the same memory position. An external Master Switch allows to completely disable or enable the reconfigurable system. If disabled, the Injector acts like a completely transparent passthrough. The Graph Memory Addresses are values specified at synthesis time.

this behaviour by altering the instruction stream in order to make the GPP jump to a predetermined memory position that holds a Code Segment previously loaded into the RAM that allows for communication to and from the fabric.

In short, the Injector contains a table of Program Counters (PCs), or in other words, memory addresses, that are associated to the beginning of regions of code that were translated into graphs and successfully mapped to hardware. So, it is the task of the RM to, from a specific PC received by its interface with the Injector, reconfigure the RF to perform the operations that correspond to that graph, before replying to the Injector with a specific, previously calculated memory position, to which the GPP must branch.

This communication overhead from the Injector to the RM, adding to this processor's software delay plus the reply back to the Injector is far too great to be performed during the time it would take for one instruction to be read into the GPP (i.e. several instructions would pass during that time), and a loss of execution context would occur (the values in the GPPs register file would be altered). So, when it is necessary for the Injector to wait for a reply from the RM it is capable of stalling the GPP by altering the instruction into a branch to the same line (PC = PC + 0) before the actual instructions is read into the GPP. The interface with the RM is done by a point to point connection implemented through the FSL interface, which allows for very fast communication.

There is, however, the issue of two or more graphs having coinciding memory addresses, and, as such, creating ambiguity as to which graph is to be executed in hardware. This was previously mentioned in section 3.5.1 and now that the Injector has been explained the nature of the problem becomes apparent. For this reason, the Injector also performs detection of branch instructions. This feature was developed for pattern detection in order to allow for the identification of graphs

at runtime by determining which pattern of repeating SuperBlocks (or a trace unit of another granularity) was occurring. Although conceptually functional, it was not utilized because it demanded further changes in the hardware layout (coupled to the necessity of knowing the starting PCs of SuperBlocks in order to detect that type of trace units). So, for the current time, the toolkit does not allow for two or more graphs that share the same starting memory position to be implemented for simplicity.

Regardless, the Basic Block Detector would be the apparent solution to this problem, identifying the sequences of SuperBlocks and afterwards communicating to the RM an ID very much in the same manner as the current implementation. This would of course require that the graph iterate in software at least once so that the sequence could be found. Also, the maximum number of SuperBlocks that composed the MegaBlock (i.e. the graph would dictate the maximum size of pattern detection meaning the Basic Block Detector would require more area on the FPGA depending on this factor. For now, this is not being performed, thus limiting the system to one graph per PC merely because of this ambiguity, in terms of the RF, there is no limitation of this nature.

4.2.1 Design Considerations

As implied, the Injector acts as a signal passthrough for a PLB bus. XPS does not have wizard supported creation of modules of this type. So, utilizing the Injector as a peripheral in the XPS environment required a few manual modifications of peripheral descriptions.

Firstly, manual editing of peripheral description files is necessary. The most important file is the MPD (Microprocessor Peripheral Description) file. This file details how the peripheral is viewed by XPS. Several parameters need to be either edited or set. Namely, the peripheral type needs to be a BUS, as the GPPs instruction bus port can only connect to this type of interface. Also the Injector needs to have BUS interfaces itself, one Slave and one Master, to act as a passthrough. To retrieve the signals output by the GPP to the BUS (in order to know what signal inputs and outputs the pass-through needs) the GPPs MPD can be inspected or, alternatively, a custom peripheral with a PLB bus interface can be created the signals can be derived from there. This procedure can also be performed to retrieve the signals necessary for the FSL connection.

By connecting the GPP's IPLB to the Injector's master port any peripherals on the actual PLB bus disappear from the GPPs memory map, in this case, external memory is no longer present. So, software applications can't be compiled and linked to reside in those memory locations. The workaround is a simple, one time, manual editing of the linker script.

Regarding the precise moment in which the Injector alters the instruction stream, it may not be any. One identified situation was the injection of an instruction to branch to the same line (while the Injector is waiting for a replay from the RM) after an *IMM* instruction. This instruction loads a special register with an immediate 16 bit value, and is used before other instructions that require and immediate operands of 32 bits, such as absolute branch instructions. A relative branch (taken to PC plus the lower 16 bits of the branch instruction) becomes absolute if performed after an *IMM*. So, if the Injector began forcing the GPPs PC to the same value by injecting a branch after an *IMM* (which could have occurred randomly depending on the running program), the injected

branch would become absolute and the behaviour would be undefined. No graph, however, was identified to start after an *IMM* instruction.

Another issue is the possibility of a false positive. The memory address of any graph needs only to pass through the Injector in order for it to be detected as such. In some cases, these memory addresses are placed on the bus by the GPP when they will not, in the end, utilize the retrieved instruction. This happens due to the MicroBlaze's delay branches, or even branches without delay, as can be seen in listing 4.1.

```
...
88001318: add   r5, r5, r5
8800131c: bgeid r5, -4          // a backwards branch to 88001318
88001320: addik r29, r29, -1    // while executing this instruction
                                // the value 88001324 is on the memory port
                                // of the bus, through the Injector
88001324: add   r5, r5, r5      // this is the start of the graph
88001328: addc  r3, r3, r3
...
```

Listing 4.1: Injector false positive

Branch instructions may be delayed so that the MicroBlaze may execute the instruction following that branch. While executing that instruction, the processor places a request for the instruction following that on the bus. It will not execute it however, since the delayed branch will now trigger, causing the processor to branch backwards and discarding the instruction fetched from the memory position following a branch (or two memory positions following a delayed branch). The solution is to not only detect when the graph PC occurs, but to also detect if the next memory address the GPP would access would be the one immediately after that. Since this only occurs if the processor is in fact executing the instruction that is the start of the graph, this confirms that the fetching of that instruction (the appearance of the graph PC in the Injector) was not a false positive and that the GPP would be entering the memory region corresponding to the graph.

4.3 Alternative Architectures for the Reconfigurable Fabric

The Reconfigurable Fabric (RF) is the element of the system that produces outputs from given inputs through a set of operations who's layout and interconnections are determined by the description tools and run-time configuration information.

From the start, the RF was to have a standardized memory mapped interface to the PLB bus. In this manner, the GPP may write inputs to the appropriate registers and, by polling a status register, determine the moment at which to retrieve outputs. The memory positions of the input and output registers are generated by the toolkit by starting from the base address of the fabric extracted from the XPS environment.

The three main alternatives for the internal design of this module are presented in the following subsections. They differ on the method through which the fabric itself is reconfigured, on

the flexibility each alternative provides and, consequently, on how the graphs and their configurations would have to be represented as data structures to support operation with each alternative. Common to all the alternatives are a few characteristics that define the fabric, namely, its width (maximum parallelism), depth (maximum execution level), the number of available inputs and output registers and the necessary runtime configuration information.

4.3.1 Dynamic Architecture for the Reconfigurable Fabric

The fully ICAP implementation of the RF was developed with the idea of a mixed granularity fabric in mind, very much like the preliminary proposals. This first approach was designed to work along a fully online system, that would detect graphs and construct a hardware representation at runtime from bitstreams previously stored in the RM memory. The bitstreams would be merged to form a module that contained, implicitly, all the connections between operations and the connection to the fabric's output registers. In other words, this fabric would have only, as static elements, the input and output registers and the necessary logic to control its operation. The remaining space allocated to the fabric within the FPGA would be unprogrammed (meaning blank), and would be the target of reprogramming via ICAP with the generated information.

The RM would have, in static program memory, a library of modules to be matched against the detected graph in a structure such as the following:

```
//module structure
struct module {
        enum module_class mod_class;
                //A_ADD, A_MUL, A_BRA, L_AND, L_ORL, etc
        int *mod_bitstream;
        int mod_blen;
                //bitstream length
        int mod_numins, mod_numouts;
        int commutative;
        int delay;
                //combinational delay
};
```

Listing 4.2: Module Structure

The data structure would maintain information about the type of module (the operations it could perform), its bitstream, the bitstream's length, the number of inputs and outputs of the module and any other data deemed necessary to map the modules at runtime. The types of modules that could be stored in the library could perform any desired operations, from simple additions, multiplications and logical operations to more complex mathematical operations, such as square roots or powers. This was later proven difficult and cumbersome approach by for reasons stated in the following paragraphs, but it was this concept that permitted the approach based on a mixed granularity system. An important field was thought to be the combinatory delay that particular bitstream represented. As explained in the preliminary design, it was conceptualized that the fabric would have a software controlled clock to maximize its frequency to the point allowed by

the mapped operations. Hence the need for the delay of each brick, the maximum delay would also have to be computed at map time. However, even synthesis tools predict these maximum limits with difficulty, which makes this conceptual feature unlikely to be functional (unless very conservative calculations are made).

During online detection of graphs the RM would construct a software representation of the hardware to be mapped by performing the necessary parallelism and data dependency discovery. However, this type off mapping would imply knowledge about both the data organization of a bitsteam and knowing how to extract their information in such a way as to create a final bitstream containing the concatenation off all individual parts plus their connections. The additional reasoning behind this is the hopeful reduction of control bits. If bitstream tools allowed, concatenation of several circuits could provide a much faster way to interconnect operations within the Reconfigurable Fabric (RF), eliminating the need for a high number of configuration bits needed for interconnection multiplexers or other devices and also removing their delay and reducing the area required. Although the method of altering information of the stored bitstreams so their placement on the FPGA changed to the desired position (i.e. to an appropriate position within the fabric) was relatively straight forward, the main problem was assuring the routing could be properly and efficiently performed. Specifically, how to generate and maintain information regarding the current wiring in the FPGA? This is, in fact, the most complex task that has to be performed by commercial synthesis tools. Performing routing for several mapped bitstream concatenations (each representing a graph) requires that no wiring is crossed (as the FPGA is single layered) and adding to that there was no apparent way to treat for the multiple drive that each graph would impose on the output registers. Although in an offline environment this could be treated with high impedance signals and choice of output via selection bits this appeared non-trivial for this architecture.

So, a different concept, which abandoned the process of merging bitstream information, was created in an effort to standardize the connections between operations and to/from the output and input registers. Its conceptual architecture is presented in figure 4.4.

The RM would still maintain a library of bitstreams corresponding to elementary operations (from now on called bricks) and would now simply perform several ICAP accesses to map each one and its connections. This of course makes the system inherently fine grained. This approach would rely on ICAP's minimum permitted granularity of reconfiguration to create a grid like structure in which to place operations. So, the signal routing would be done implicitly, that is, bricks would need to have their outputs and inputs standardized to allow for removal of one, placement of other, while assuring that the signals still propagated properly. In truth, a routing effort is also necessary, by creating bricks that serve as passthroughs to lead the wires to appropriate places.

When the detection of graphs shifted to an offline task, this approach, as well as the previous, remained valid, simply relying on graph information already in memory, one that described the position and type of modules to map as determined by the offline graph analysis tools. Information such as the following would be produced to instantiate all the bricks composing a graph:

```
        //graph1
2       int graph1_nummodules = n;
```

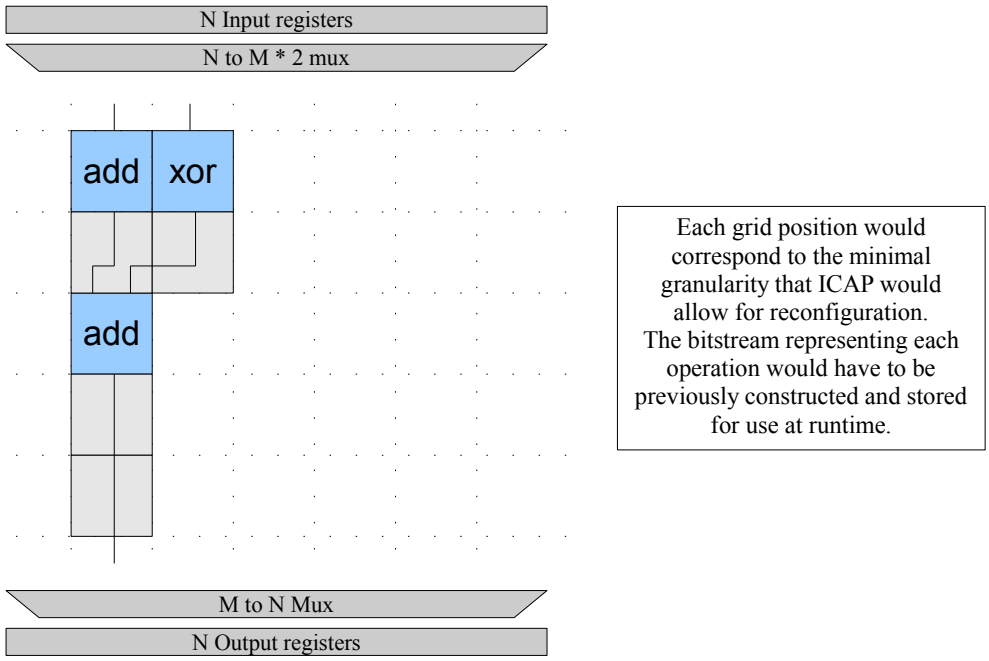


Figure 4.4: ICAP based fabric

```
int graph1_moduletype[n] = {A_ADD, A_MUL, etc..}
int graph1_module_placement[n*2] = {x1, y1, x2, y2, etc..}
        //paired values of frame offset
        //relative to fabric start (upper left corner)
```

Listing 4.3: C Level Graph Instantiation

The RM would utilize this information to configure the RF at runtime with the appropriate bitstreams in the specified locations, or would reutilize already mapped bricks. The moment of reconfiguration would happen by detecting when the GPP was about to execute a particular block of code that the RM would know, thanks to the statically held tool generated information, how to map to hardware.

However, regardless of where the graph detection was performed, this type of mapping would require, as was implied, knowledge about both the data organization of a bitstream file and knowing how to extract information from said files in such a way as to create a file containing the concatenation off all individual parts plus their connections. Adding to that, the access time to the ICAP peripheral would represent a considerable overhead, and the protocol messages used to communicate with this module would have to be implemented as well.

However, it would not be feasible to assume that any one operation could be synthesized and successfully placed in a region of the FPGA that ensured it stayed within the minimal granularity of the ICAP access (and some type of operations utilize dedicated resources in the FPGA, which is not homogeneous, hence the loss of the notion of a standardized brick). But the most problematic issue was, once again, ensuring that the inputs and outputs of each brick were located at a correct position, which proved rather difficult to accomplish, and impossible without specialized

toolflows. To clarify, whereas the original routing problem present in the merged bistream scenario was related to knowledge of how to route a signal throughout the entire fabric at runtime, this is relative to the location computed (in synthesis time), of a single module's inputs and outputs in order to ensure that they overlap when placed adjacent to another module. These are the kinds of features available with Xilinx PlanAhead, the manual placement tool with which it is possible to create more intuitive placement restrictions such as the ones required for this approach. While initially deemed ideal for development of this type of layout for the RF there were impediments to the use of the features it provides. A Module-Based Partial Reconfiguration toolflow allows for specification of modules that are meant for later addressing over ICAP and reconfigured. So, their input and output ports must be, and can be, locked in position via boundary modules named Bus Macros. However, one bus macro would have to be placed at each border of each reconfigurable module (which in this design would be the bricks themselves). It is easy to conclude that a large number of bus macros would be necessary to do this (to cover each grid border), which would create a large spatial overhead and greatly increase the difficulty of automatizing the generation of the fabric module itself in development time. Additionally, Module-Based Partial Reconfiguration was not supported on Spartan-6 targets.

Also, this approach would require larger initial temporal overhead and would offer nothing after an extended period of time (after all graphs had been identified and mapped). For instance, to map the example given in figure 4.4, seven accesses to the ICAP module would have been required, that is, a larger number than the actual operations (and still not accounting for routing to input and output registers). The system would reconfigure one brick at a time, that being the minimal reconfiguration granularity permitted by each ICAP access, and, as such, the overhead would be too great (although one time only). For these reasons, and also because graph identification and treatment is being done offline, it would not be reasonable to follow this approach. The mapping and routing algorithms would also have been, possibly, more complex to implement.

Common to both alternatives, knowing the absolute fabric position for each system iteration (i.e. each alteration and consequent synthesis) would be required. Automating the propagation of this information amongst tools might not have been trivial and, generally, it would strays from known, more linear, toolflows.

Still, in an effort to reduce the use of bus macros and also the number of accesses to ICAP necessary due to the need of passthrough bricks, the following redesign was created.

4.3.2 Partially Dynamic Architecture for the Reconfigurable Fabric

This approach was meant to simplify the effort found in interconnecting bricks through the previously described ICAP method. In this design, represented in figure 4.5, ICAP would still be used to map only the operations themselves. The routing would be done by means of crossbar connections between each row of operations. This would also greatly simplify the mapping effort, i.e. the absolute location of each brick would lose relevance as the crossbar would be able to connect any of the outputs of a row to any of the inputs of the next row. So, only the vertical position of a brick remains relevant, as data precedences must be maintained. The routing information would

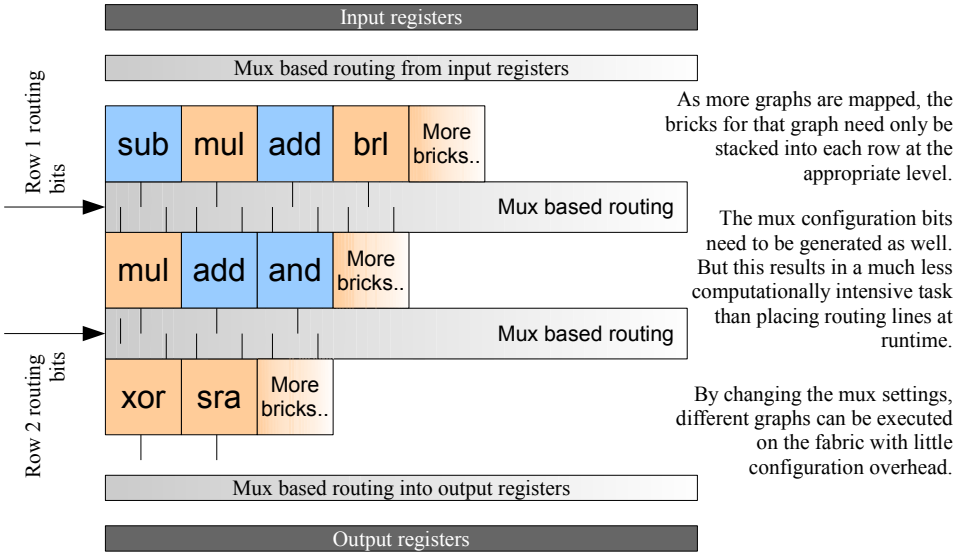


Figure 4.5: Partially ICAP based fabric

then have to be generated differently, consisting in a set of control bits for the crossbars instead of locations for bricks solely dedicated to wiring.

The bricks would be maintained in a runtime library by the RM in the same data structure as with the previous design, however, no placement information would have to be generated by the tools.

Each row of the fabric would then start from the initial state of having no bricks mapped and as graphs were detected the needed bricks would be placed in the appropriate row. The RM would maintain information of the currently mapped bricks which would allowed them to be reused from graph to graph. In other words the looser mapping constraints and the very nature of how the entire graph is constructed and routed (i.e. not in a closed, software computed, bitstream) would permit the reuse of individual bricks between graphs, seeing as though only one graph is executing at any one time. In other words, the graphs can be matched in terms of necessary hardware, or, to formalize, Graph Matching can be performed. This would also be possible with a Fully Dynamic approach, but it would complicate routing every time more graphs that reutilized bricks were mapped. Additionally, the number of Bus Macros required would equal the number of rows of the fabric plus two (for interfacing with input and output registers), a large reduction comparatively with the Dynamic design in subsection 4.3.1.

However, interconnection of operations that span more that one row would still have to be done with passthrough operations, thus increasing the number of ICAP accesses beyond the number of actual operations (but still a more reduced number than the Dynamic design).

The main concerns with this approach were the size occupied by the crossbars on the FPGA (being N to M muxes, their size grew at an elevated rate, for instance, 2560 LUTs required for a 12 to 16 demultiplexer) and, as with the fully ICAP based fabric, the specialized tool flows necessary to work with partial bitstreams and the method through which those bitstreams are allowed to be

placed and properly routed (i.e. bus macros).

To solve the problem regarding the size of the crossbars an architecture that only permitted connections from the output of a brick to the brick bellow or to the two adjacent to that one was considered. The reasoning was that N to M connections might not have been frequently required, being possible to construct the graph even limiting the connections supported. However, this implied that the effort would be less focused on the data routing itself and would be, once again, split to the placement as well (as bricks may need to be adjacent so that connection is possible). Although not invalid, it was a far to restrictive approach in terms of possibly supported graphs. Also, the crossbars would also need to have an upper limit of inputs as well as outputs, as these characteristics that cannot be changed at runtime. So this means that, while graph discovery was occurring, there would come a point were the width of the fabric would be filled to the maximum supported limit. Then, either no more graphs could be mapped or some bricks would have to be changed, which introduce the need to remap graphs once again if needed creating a larger temporal overhead if many switches occurred.

Regardless, the lack of support for partial reconfiguration based projects on the target platform, the cumbersome design flow, and the spatial and temporal overhead were the motivators for a design who's generation was shifted to offline tools. Adding to that, was the seemingly non trivial matter of how to control execution in a fabric structure that could place operations

4.3.3 Semi-Static Architecture for the Reconfigurable Fabric

This was the final iteration upon the internal design of the fabric and, consequently, on the method through which its description and reconfiguration information is generated. Considering the unreasonable overheads and design effort involved with the dynamic based approaches, coupled with already present shift of graph detection to offline time, the elaboration of the RF itself may also be moved to an offline task. The rationale, as was previously mentioned, is that there would be little advantage to having a system whose capabilities were based on reconfiguring portions of the FPGA whose final composition had already been dictated and would not be susceptible to change. Although a complete runtime reconfiguration system would be, conceptually, the most versatile, several impediments hinder its design. And even if such was not the case, a system with such a level of flexibility is only justified in environments of quickly changing computational demand, as is not the case for embedded systems who could benefit greatly for dedicated acceleration hardware without forcing a design flow through the costly and long lasting steps involved in hardware design.

4.3.3.1 Fabric Description

Unlike the other approaches, this fabric was fully written in HDL, but in such a manner that its heavily parameter based design allows for automatic, tool performed alteration. Specifically, it can be expanded in both width and depth (with some limitations of practical nature), the necessary bricks are instantiated and correctly placed at synthesis time, all the inputs, outputs, control

registers and routing registers as well as necessary wiring for all signals is created solely by resorting to Verilog constructs and parametry. Namely, *generate* loops instantiate all the necessary logic according to information contained in parameters and parameter arrays that are generated by the placement and routing tools. In the same manner, all wires and instantiated and assigned values from the memory mapped registers or a bit selection is performed on large arrays of bits on a particular range in order to direct them to or from the correct modules. A Verilog parameter is a numerical value that utilized to control certain characteristic for hardware instantiation by the synthesis tools. Parameters may be passed into modules at instantiation time in order to alter port widths (number of bits) and other aspects.

So, even though this fabric is considerably more static than a dynamic approach, the toolchain created permits rapid description of any variation of this layout, instantiating a piece of dedicated hardware for a compiled program in a way no different from creation of a standard user peripheral. The complete toolflow and its outputs are explained in chapter 5;

Most relevantly, this method solves all the previous problems of placement and routing, as the RF is a hardware module completely within the standard hardware design flow for FPGAs. The description of the fabric is done by altering header files containing the previously mentioned parameters that describe the fabric in its entirety. As an example, the following is an array of parameters that specify the operations themselves, for a small fabric, along with some others that specify basic characteristics:

```
        parameter NUM_IREGS             = 32'd4;
        parameter NUM_OREGS             = 32'd5
                //nr of input and output registers
        parameter NUM_COLS              = 32'd5;
        parameter NUM_ROWS              = 32'd3;

        parameter [ 0 : (32 * NUM_ROWS * NUM_COLS) - 1 ]
                ROW_OPS = {
                                //this is the top of the fabric
                        `A_ADD, `A_ADD, `A_BRA,
                        `L_AND, `PASS, `PASS,
                        `A_SUB, `B_NEQ, `NULL
                                //this is the bottom
                };
```

Listing 4.4: Verilog Parameter Arrays

Besides these, the tools generate many more arrays and parameters to fully characterize the RF.

Thus, another feature of this design is the ability to instantiate bricks in which one of the operands is constant. As show before in the graph descriptions in section 3.4, many graphs contain operations in which one operator is constant. With the previous approaches for the fabric architecture this had not been addressed. One alternative would have been the creation of a register bank of constant values for use or either alter the bitstream of each brick to include that constant value within the brick. However, these ideas would have been met with the same difficulties that led

to the abandonment of those fabric designs, namely, how to properly and quickly route operands and results. With a offline created fabric, the flexibility of description allows for this feature in the same manner that the bricks themselves are instantiated. Two parameters are passed to each brick detailing whether or not an operation has two variable operators or only one, along with the value of the constant operator in the latter case. It is also possible to have a brick that either operates on two variable inputs, or with only one input and a stored constant value, which is also dictated by parametry (this type of brick is possible to instantiate, although it was not utilized for reasons later explained).

Also, Graph Matching is also performed at a tool level, when the fabric description is constructed. That is, the necessary hardware is minimized to the essentials to perform all desired graphs. So, the fabric supports execution of several graphs (as many as the tools process successfully and in a number that will not result in an RF too large to place in the FPGA, although conceptually any number).

An important aspect is support for feeding the fabric with a different clock from that off the PLB bus. Currently, the RF is receiving its clock from the bus interface, but the internal operations within the fabric are, in some cases, considerably superior to the frequency of the bus. So feeding a higher clock would result in higher acceleration. However, clock synchronization would have to be considered in order to maintain hardware coherency. One solution is to have the clock of the fabric set to a multiple of the bus clock, but this is not always achievable.

As was stated in section 3.5.2, non-atomic iterations are not supported, as well as graphs which contain memory accessing operations. So, despite functional, there is room for improvement and in chapter 7 some alterations to this layout are discussed that might provide support for these features at a later iteration.

4.3.3.2 Fabric Structure

So, structure wise, this RF is composed of the same kind of elementary operations, or bricks, in a grid like layout. Arranged horizontally, on the same row, are operations that have no data dependencies. The results of each row are propagated to the next via switchboxes that allow for N to M connections.

As with the Partially Dynamic design, routing operands and results through a distance that spanned more that two rows was addressed. Although in that approach the problem was solvable, a passthrough (or a chain, if the span was of several levels) operation to propagate the data would occupy the same minimal granularity of ICAP as any other brick, which is wasteful in terms of space for a simple wiring. With a statically coded fabric the problem disappears. Along with it, the issue of maintaining each brick within that same minimal granularity is also solved, as it no longer applies. While in previous approaches the bricks were elements whose bitstreams were statically held, they are now simple HDL modules which will be matched against the instructions found in a extracted graph by the tools.

Also supported by the fabric are variable number of inputs and outputs from a brick. Meaning that expansion for other kinds of operations beyond those implemented now would be simplified.

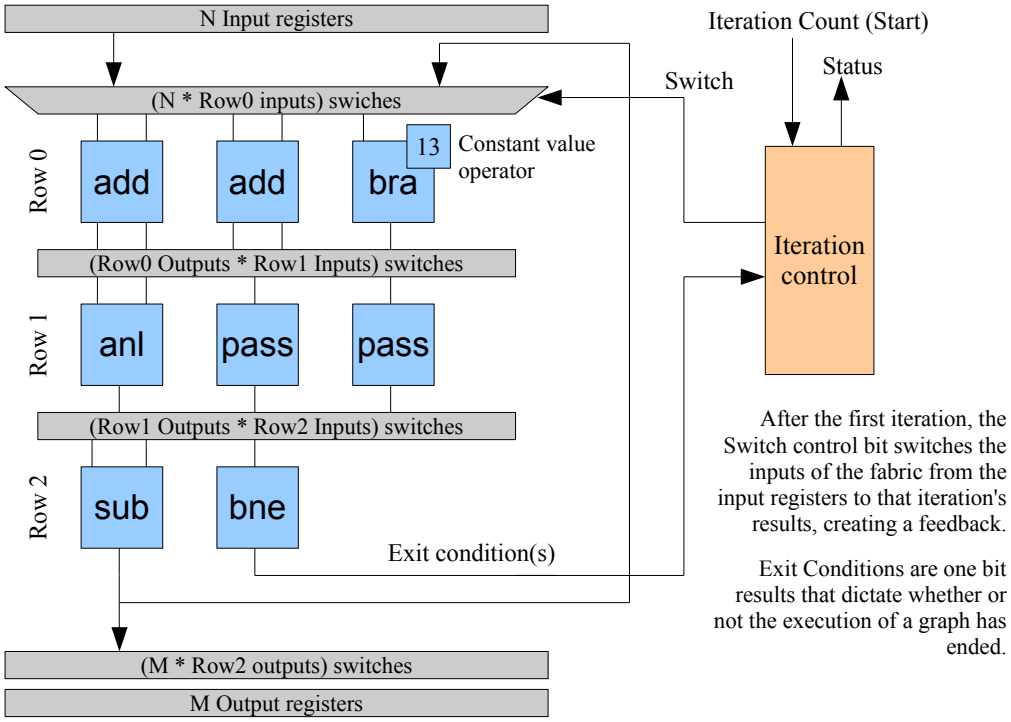


Figure 4.6: Semi-Static Fabric - The switchboxes function by receiving their selection bits from the memory mapped routing registers, the bricks have a variable number of inputs and outputs and may be reused between graphs

The toolkit also takes variable inputs and outputs into account while generating routing information. Related to this, and importantly, the fabric also supports operations that modify the GPPs *Carry* bit. To be more correct, the carry result (in 32 bits to standardize connections) is propagated to the outputs registers, or to wherever it is necessary, as with any other 32 bit result. Similarly the upper 32 bits of a multiplication may also be treated this way. The code that the GPP executes in order to retrieve results from the fabric is capable of verifying the value of the output register containing the carry result and setting or clearing the GPPs *Carry* bit accordingly.

Each row registers its results (including passthroughs), meaning that a full iteration through the fabric consists of a number of clocks equal to its depth. At the end of the first iteration the results are fed back into the fabric, causing a cyclical flow of data that will terminate when one of the exits conditions is true. Note however that, although each row consists of a register stage, the fabric is not pipelined. Because the next iteration depends on data retrieved from the previous, it is impossible to have the fabric filled with useful data and producing one iteration result at every clock. Thus, the number of clocks that it takes to complete execution is the number of iterations times the depth of the fabric. This carries a penalty to smaller graphs, as they will be subject to a depth equal to that of the deepest graph (which dicatates the depth of the RF), slowing down their execution.

Flexibility wise, as is implied, the fabric may execute any number of graphs as there are possible combinations of operator routing. Of course the useful ones are those that correspond to the routing configuration generated by the tools that perform graph analysis. Switching configuration

from one graph to another will take as much time as is required to write to all routing registers. An approximate formula for this is show in the following section.

Once routing is done the fabric can be used to perform computations of a graph corresponding to that routing scheme. Since one iteration is completed in as many clock cycles as the RF's depth, the total number of clock cycles required to execute a graph, assuming the fabric is already routed, can be estimated, roughly, by the expression given in equation 4.1. Let T_{FC} be the total number of clock cycles, N_{Itrs} the number of iterations the graph will perform and $Depth_F$ the depth of the fabric.

$$T_{FC} = Depth_F \times N_{Itrs} \tag{4.1}$$

Adding to this is the access time the GPP is subject to by communication through the bus. This depends on how many operands that particular graph requires to be loaded into the fabric and results to retrive. These overheads are explained in section 6.1.

As will be shown later, the instructions the GPP executes to communicate with the fabric themselves introduce delay if repeated in great numbers since they are in external memory. In the previous equation, the delay of communication is expressed as a function of the number of inputs and outputs times the number of PLB access clocks. However, the number of Microblaze Instructions that need to be performed in order to write and read those values is greater than the sum of inputs and outputs of the graph (explained along with the description of the tools in section 5.2).

Note that the synthesis was only performed in Xilinx ISE, and with no other tools such as MentorGraphics Precision or Altera Quartus. The constructs and hardware instantiation loops might not be fully portable from tool to tool, even tough the utilized syntax is within Verilog specifications.

4.3.3.3 Switchbox Routing

Regarding the switchboxes, they retain a crossbar-like structure to facilitate tool development. As mentioned in section 4.3.2, limiting the interconnection scheme constraints the placement of bricks, and may reduce reutilized resources between graphs (as bricks can no longer be placed at any horizontal position). So a generalized approach was taken. The tools were written to later allow for definition of placement constraints, which facilitates further iterations on the hardware architecture regarding this issue.

The switchbox itself is a simple module that receives a set of bits allowing it to chose which input to place at each output. The inputs of a switchbox are the outputs of the bricks present in the row preceding it, and its outputs are, in turn, the inputs of the row of bricks following that switchbox. The number of bits necessary to control the switchbox is, therefore, dependant on the number of inputs and outputs. The total number of routing registers necessary is dictated by the total sum of the bits needed to control each switchbox and these are, in turn, determined by the width of the fabric (i.e. the maximum number of outputs to choose from), as expressed in

	Nr. Selection Bits				
Total Inputs	2	3	4	5	6
15	2	2	3	3	4
30	3	4	5	6	7
50	4	6	7	9	10
65	5	7	9	11	13

Table 4.1: The number of routing registers necessary is a direct function of the maximum selection width and the number of brick inputs within the fabric.

equation 4.2. Let Max_{bits} be the maximum number of bits needed to represent the widest row, $Total_{ins}$ the total number of brick inputs, $Total_{bits}$ the total number of bits needed and $Nr_{Routeregs}$ the number of routing registers required.

$$Total_{bits} = Max_{bits} \times Total_{ins} \tag{4.2}$$

$$Nr_{Routeregs} = (Total_{bits} + 32 - 1) \div 32; \tag{4.3}$$

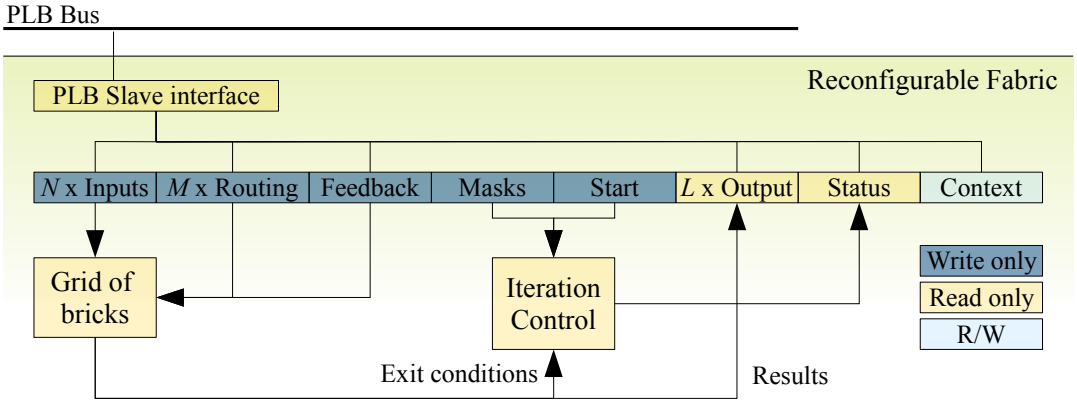
So, the routing information for all levels is concatenated across all registers, or in other words, a single register does not necessarily contain routing bits for a single level, it may contain information for any number of levels.

To clarify, the maximum number of selection bits considered is determined by the maximum number of outputs of all rows (i.e. find which row has the maximum number of outputs to choose from and the number off selection bits for all rows is computed from that). Of course this causes that switchboxes with less inputs (in other words, placed after a row with outputs less than the maximum number) have superfluous selection bits, but it was a required workaround to some lack of flexibility present in the Verilog language (as were several others). Some considerations regarding this can be found in section 4.3.3.6.

Mentioned before was the need to have the Injector stall the GPP while the RM reconfigured the fabric for a graph. As is obvious at this point, the greater the number of routing registers the longer the access time from the RM to the PLB bus in order to write to these registers. So, it cannot be assured that the RM will reconfigure the fabric quickly enough as to immediately have the Injector branch the GPP to the Code Segment for that graph, thus the need to have the GPP wait. This reconfiguration time is, of course, one of the overheads of the system, all of which are presented in section 6.1.

Regardless, table 4.1 details some possible combinations of input and outputs within the fabric (between rows) that lead to different cases of required registers.

Clearly, as the necessary number of routing registers increases, the more delay the reconfiguration of the fabric introduces. For a program that requires constant switching between reconfigurations, this might be harmful to the speedup, or even result in a slowdown. The final result would depend on the size of the graph as well, as there is a trade off that involves checking if a graph



The number of memory mapped registers varies according to the graphs from which the RF was built.

Figure 4.7: Memory Mapped Registers

is worthwhile to be implemented in hardware. If too small, the communication overhead would exceed the original software computation time.

A simple solution is found however. By replacing all the routing registers with a simple graph selection register, the reconfiguration time for a graph of any size would be constant. The RF itself would hold a look up table with the necessary configuration bits instead of having and external source provide them. Tool-wise, the generated information would be the same and would be provided as parameters, similar to all the other arrays that describe the RF, at synthesis time. Although this was implemented, it was not deeply tested, with the current configuration remaining for development purposes. However, relocating the routing information to within the RF itself would eliminate the possibility of creating a great number of graphs by changing the routing register to whichever value was desired. From a practical standpoint, this seems useless, however, if the system were to be expanded to include online functionalities such as the discovery of more graphs, new routing information would have to be created and could not, at that point, be inserted into the RF if the routing registers were not visible from the bus.

4.3.3.4 Memory Mapped Registers

The memory mapped registers the fabric utilizes are detailed in Figure 4.7. Being that the fabric is custom designed for each set of graphs it can execute, the number of input and output registers vary, along with the number of routing registers necessary to configure the connections to and from the operations. The remaining registers are static in number, being implementation independent. Input and output registers have straightforward functions, the values need to be written to the inputs prior to commencing the calculations and the results are read from the outputs once they are concluded. The routing registers are filled with values generated by offline tools and so, like the inputs, they merely need to be written to before calculations begin. No online computation for routing values is performed. A detailed look into a routing register can be found in section 5.1. The feedback register serves the same purpose, however, it routes only the results contained in the

		Input Registers				
Output Registers	Nr. Sel. Bits	7	8	9	10	11
4	2	14	16	18	20	22
8	3	21	24	27	30	33
9	4	28	32	36	40	44

Table 4.2: Number of bits necessary for feedback routing for a given number of inputs and outputs.

last row of the fabric back into the first. The reason as to why this routing information is held separate from the rest was for simplicity of tool design and RF design. Of course this limits the number of outputs and inputs of the fabric, as it has to be able to feed any output into any input and a single register (32 bits) will not suffice for all the configuration bits necessary over certain numbers, as seen in table 4.2.

To clarify, a combination of inputs and outputs is supported as long as the number of bits required to represent all output registers multiplied by the number of input registers does not exceed 32 bits. This, of course means, there is a dependency of values between the two limits (16 inputs limit the system to 4 outputs for instance). Still, this was deemed irrelevant considering that the typical number of inputs and outputs the extracted graphs presented was far bellow these values. Both the contents of the routing registers and the feedback register are wired into the appropriate switchboxes, the parametry of the fabric being able to dictate which bits within the registers correspond to which switchbox.

The Masks register is utilized to enable or disable certain exit points within the fabric. To clarify, if several graphs are utilized to built the fabric then all the operations composing the graphs will be present in the fabric. Along with each graph, there is at least (and, in an atomic graph, at most) one operation which outputs a 1 bit result indicating the termination of the graph, i.e. an exit point. So, while data is propagating through the fabric following certain connections as to perform a particular graph, random data will also be entering all the operations whose results are not being considered. The exit conditions however, may result in false positives with this random data. So, masks are necessary to consider only the exit points pertinent to that graph. Since only one register is used for masking, and since only 1 bit is required to activate or disable an exit point, only 32 exit points are supported in the fabric. It is a limitation, but easily expandable and not very restrictive for purposes of testing as the number of exits for the graphs extracted from the tested programs proved to be bellow this limit. Like the routing, the masking options could be parameters feed at synthesis time, with all the same advantages and consequences.

The Start register is merely used to signal the fabric to start executing the graph for which it was routed for, using the data now present in the input registers. In truth, it serves another purpose, the value loaded into the Start register will be the maximum number of iterations the fabric performs before aborting. In other words, although the fabric computes until one of the exit points is true, that being its normal execution, a maximum number of iterations was included as a failsafe in case the fabric goes into an undesired cyclical state in which the data never changes (i.e.

Status Register						
31 - 6	First fail	Control Exit	Aborted	Busy	Switch	Graph done

Figure 4.8: Semi-Static fabric Status Register

no exit condition is ever true). Although in theory this should never happen, as the tools generate the correct hardware description and configurations, this feature was left in.

The Status register contains bit level information, that is, control and status bits. Figure 4.8 details the contents of the Status register. Only 6 control bits are necessary, bits 31 to 6 are reserved. *First fail* indicates if the graph has completed execution normally in its first iteration. This is a necessary verification because completion of a graph in this condition is a special case. If it occurs, the GPP cannot retrieve the results from the output register, as they have no valid data since no iterations were performed on the RF.

Not mentioned before to keep the functional description of the system clear, the RF returns the results of the iteration before the last. This is because the computations that are part of the CFG are performed alongside the DFG in the fabric. If the graph contains only one exit point, this is not problematic. For graphs with multiple exits points treated atomically this is not the case. Since the system branches the GPP back to the start of the memory region containing the graph instructions, the processor will not have the correct data to skip execution of the graph in software (i.e. trigger a branch), as the retrieved values are those resulting from executing the graph up to an intermediate point. In other words, executing the graph in hardware up to the last iteration would be supported if multiple exits were also supported. In this implementation, the RF returns the results of the penultimate iteration. Thus, the GPP returns to software, executes the graph instructions and ends execution of that graph at the correct branch (the same one that caused the RF to finish execution).

This means that if the first iteration triggers an exit, no useful data is to be read from the RF, in such a case, the GPP does not need to execute the code which retrieves the values of the output registers, merely returning to software.

Control Exit indicates normal termination of execution. It is the bit that the GPP polls while waiting to retrieve outputs. *Aborted* is related to the previous feature described for the Start Register. This bit will be set when the fabric executed beyond the maximum number of iterations. The *Busy* bit is set while the RF is processing, similarly, the *Graph done* bit is clear during that time. Both switch values at the end of execution.

Finally, the Context Register is associated with the situation that justifies the existence of the *First fail* bit. Explained in more detail in the chapter describing the toolkit, this register holds the original value of one of the registers in the GPPs register file, as the latter needs to be used for instructions that load and store values to from and to the fabric. In short, it is an auxiliary register.

In a particular case, two graphs may share exactly the same routing, in which case only the masking register needs altering, in order to choose which exit point is to be considered. Such cases occur when very similar, or identical, cycles in high level code differ only on the stopping condition, i.e. *greater or equal than* or simply *greater than*.

4.3.3.5 Supported Operations

Currently the RF contains the following individual modules that correspond to the bricks. The relationship between each module and each MicroBlaze instruction is nearly 1:1. The following listing is a *C* file which is part of the Graph2Bricks tool explained in section 5.1, but its contents may be presented here simply for the purpose of listing the supported operations.

```
#define NUM_MODULES 21
enum module_class {
        UNKN,
        PASSTHROUGH,
        A_ADD,  //all adds
        A_ADDC, //all adds with carry
        A_SUB,  //all subtractions
        A_MUL,  //all multiplications
        A_BRA,  //barrelshift right arithmetic
        A_SRA,  //shift right arithmetic
        L_SRL,  //shift right logical
        L_XOR,  //logical xor
        L_AND,  //logical and
        L_ORL,  //logical or
        L_BRL,  //barrelshift right logical
        L_BLL,  //barrelshift left logical
        B_EQU,  //branch if equal to 0
        B_NEQ,  //branch if not equal to 0
        B_BGT,  //branch if greater than 0
        B_BGE,  //branch if greater or equal to 0
        B_BLT,  //branch if lesser than 0
        M_STO,
        M_LD    //memory operations (not implemented)
};
```

Listing 4.5: Operations Supported by the RF

Some trivial operations which are very similar to the ones in this listing are not currently implemented simply because none of the considered benchmarks contained these operations, such as a barrel-shift left arithmetic. Since both the fabric and the tools are prepared to deal with variable inputs and outputs adding another type of brick should be a simple modification granted that the operation does not present any kind of special behaviour such as modifying SPRs (Special Purpose Registers) on the GPP (actually, one of these is supported, the recovery of the carry, but that implied modifications elsewhere as is show later in section 5.2).

4.3.3.6 Design Considerations

The RF is functional within its limits. However, a few design difficulties were found during the development of the fabric. Related to this, the fabric does possess some limitations in terms of description at a source code level.

All are relative to the flexibility of the Verilog language, and were worked around by generating a number of auxiliary parameter arrays.

Tied to the problem following explained is the notion of array or port in HDL design. Either are considered to be a bus of bits of any desired size. However, when we wish to represent, for instance, two numbers of 32 bits, an array of 64 bits is utilized, and the desired numbers are accessed by addressing the upper and lower 32 bits as is the wanted case. These are called flat arrays, in which several values are treated as a larger, wider, value. Note that support for multidimensional arrays exists. A multidimensional array would be addressed in much the same way as an array is accessed in a language such as *C*. For instance, *array[0]* and *array[1]* (each being 32 bits), as opposed to *array[31 : 0]* and *array[63 : 32]*. However, these arrays cannot be passed from module to module, and are overall, counter-intuitively, more restrictive. In fact, the common practice is to utilize flattened arrays, and perform appropriated addressing.

So, the most significant limitation in Verilog's hardware instantiation loops that is, in this case, related to array handling, was the lack of auxiliary variables to control bit selection. For instance, consider a Verilog *generate* loop, which is a construct such as this:

```
genvar k;
generate
for(k = 0; k < 3; k = k + 1) begin: gen_example
        wire [7:0] outs;

        assign outs = inputs[ 8 * (k + 1) - 1 : 8 * k ];
                //consider a previously existing signal named inputs
                        //[ 7 : 0 ]     for k = 0
                        //[ 15 : 8 ]    for k = 1
                        //[ 23 : 16 ]   for k = 3
end
endgenerate
```

Listing 4.6: Verilog Generate Example 1

This is a simplified example of the type of construct present in the HDL description of the fabric. It illustrates the main issue found with language flexibility. The *genvar k* is a variable that controls the instantiation loop. No other variables are supported to control other aspects of the instantiation. In this case, *outs* needs to be assigned a particular bit range from one, larger, input array. If the assignment is regular, the construct is functional. That is, if we wish to assign to the three instances of the wire arrays named *outs* (loops instantiate wires, or signals, of the same name by adding a prefix to that name) same sized parts of the input value then one control variable is enough to determine which bits to retrieve, as the progression is linear. However, consider a case in which we would wish to assign to several signals different sized selections of one, larger, input signal. This involves two things, knowing how large the selection if for each assignment (size of the bit range), and knowing where that assignment starts. It is this last necessity that causes a problem, as the linear increase of a variable such as *k* will not result in irregular selection ranges. To clarify:

```
parameter params[3] = {32'd2, 32'd4, 32'd1};
        //number of bytes, consider this a paramter array generated
        //by tools

genvar j;
generate
for(j = 0; j < 3; j = j + 1) begin: gen_example_2
        wire [ 8 * (params[k] + 1) : 0 ] outs;
                //wiring for param[k] bytes

        //this would be required
        assign outs = inputs[ 8 * params[k] - 1 + STARTING_BIT
                                                : STARTING_BIT ];

        //how to compute STARTING_BIT?
        //STARTING_BIT = STARTING_BIT + params[k];
                //a declaration such as the one above, or any
                //variation on it, is not supported
                //regardless of the datatype of STARTING_BIT
                //(genvar, integer or parameter)
end
endgenerate
```

Listing 4.7: Verilog Generate Example 2

The number *STARTING_BIT* would have to be derived from the cumulative number of bytes already assigned in previous iterations, and there is no valid manner in which to compute these values in synthesis time. The constructed workaround was to generate these values as separate parameter arrays, which in the presented case would be the following:

```
parameter comulative_params[3] = {32'd0, 32'd2, 32'd6};
        //in this manner, they could be addressed by a genvar
        //which has linear progression and the correct, non linear values
        //could be retrieved
```

Listing 4.8: Verilog workaround arrays

This kind of irregular array addressing is found throughout many connections and wirings within the fabric. For instance, directing inputs to into a particular row, in which each brick accepts a different number of inputs, and the addressing would have to be something such as: the first 32 bits for the first brick, followed by the next 64 bits for the second brick and again 32 for the last. As stated before, the tools generated many parameter arrays in order to describe the entirety of the fabric. Four of these arrays serve the purpose of a workaround around this issue.

Also, as stated before, the number of bits utilized to perform selection utilizing the switchboxes will always be dictated by the maximum number of outputs on all rows. It is due to similar issues that this choice was made, as each row would indeed have its own number of selection bits, and properly wiring the correct parts of an array would involve a much more complex bit selection

from the originating 32 bit routing registers, as all the routing information is placed across all registers as was show previously.

So, the manner in which the fabric is described does present some design issues, and might introduce similar problems in the future. Specifically, if further iterations are to be performed in order to support more complex graphs and different kinds of execution control. This includes the already mentioned non-atomic graphs and memory access. The type of description being utilized may not be flexible for such alterations and, also, future changes to the Verilog standard and consequently the synthesis tools might compromise the validity of the written code.

However, more significantly, the heavily parametrized approach was an attempt to describe a module in a manner slightly atypical in comparison to standard hardware design. In other words, current hardware description languages might not be, natively, geared to this kind of design.

In retrospect, a safer approach might have involved generating actual Verilog code via custom tools. The reasoning for the implemented description was a diminishing of deviations from the standard hardware toolchain, and, in fact, to determine whether such design flows were appropriate for designing this manner of reconfigurable modules.

Chapter 5

Implemented Tools

The hardware layout detailed in section 4.1.2, along with the RF design presented in 4.3.3 were created to work upon graphs as those briefly presented in section 3.4.

So, in order to fully arrive at a functional system it is necessary to start from the abstract graph structure and construct a flow that permits describing the necessary hardware, its configuration and means of communication. Each developed tool has a considerable degree of transparency and flexibility and the near entirety of the toolchain is aided by scripts that automate calls and file handling.

This chapter presents the utilized and developed tools that implement the required steps in order to achieve a functional reconfigurable system.

5.1 Graph2Bricks

The conversion from Graph information to the hardware representation presented in the description of the RF in section 4.3.3 is done by this tool, which is implemented in *C*.

It's inputs are the Extractor generated files regarding operations, their inputs and connections, as exemplified in figure 3.8. It generates both the Verilog parameter description of the fabric as well as the graph table required by the PLB Injector. Also, it calculates the contents of the routing registers for each graph that it processes, for use by the RM.

5.1.1 Main features

Graph2Bricks works on one graph at a time, but is capable of keeping context between calls. The context of execution comprises the current status of the fabric as well as routing information. To clarify, upon one call of the program with one input graph, brick placement and switchbox routing information is generated for that graph alone. Upon the next call, the tool considers the already computed status of the fabric in mapping the subsequent graphs, and also may need to recompute the values of the routing registers as will be explained bellow.

To clarify, the term *mapped* in previous chapters referred to the hardware itself, already implemented in a particular position in the fabric. In this context, a mapped operation refers to the information currently kept in the tool generated files for later implementation.

To keep this information, a generation file is created and later updated at each call with information regarding the bricks, their positions, the types of inputs, the grid's current depth and width, amongst others. The following is an excerpt of this file:

```
Generated graphlayout files: 1
Number of ops on grid: 36
Number of reused bricks on grid: 0
Number of inputs: 3
Number of outputs: 6
Grid depth: 6
Grid width: 7
Numexits: 1
Current grid ILP:  5, 6, 7, 6, 6, 6
```

Listing 5.1: Graph2Bricks Generation File

This file also keeps the routing information of all already processed graphs. The need for this will be better understood after the explanation regarding the generation of the routing register values presented in section 5.1.2. Related to this, the program is capable of rejecting the processing of input files that do not meet certain criteria, for instance, the existence of unsupported operations in the input files, or the execution is aborted if it is determined that the feedback routing exceeds the supported number of bits. In these cases the Generation File is not updated.

The Generation File is, however, only an auxiliary data container. So, one of the actual useful outputs of this program are files such as this:

```
//routing regs for graph0
int graph0routeregs[NUMROUTEREGS + NUMFEEDBACKREGS] =
                        { 0x14080200,  0x35415210,  0x2c6880bb,
                                        0x18561688,  0x3511ac2c, 0x119};
//routing regs array
int *graphroutings[1] = {graph0routeregs};

//masks for branches
int branchmasks[1] = {0x1};
```

Listing 5.2: C Level representation of routing registers

This is a file containing the values of the routing registers and masking register for a single graph. This is the *C* code that the RM utilizes to reconfigure the fabric. How the values of the routing registers are computed is explained further, in section 5.1.2. When generated, this file will contain as many routing register arrays as graphs, each array with the same number of elements, and the masks arrays will hold one value per graph. Besides this file, Graph2Bricks generates a Verilog header containing information as presented in example code 4.4. This file, containing all the bricks necessary to perform the graphs the program has processed, is what is included in hardware synthesis. Besides this file, the program also outputs hardware descriptions of the same

nature solely for the graph under analysis. In other words, it both produces the sum hardware description as well descriptions for each graph.

The graph table for the PLB Injector is an equally simple file with a Verilog parameter array detailing the starting memory addresses of the graphs:

```
1       localparam NUM_PCS = 3;
        localparam [ 0 : 32 * (NUM_PCS) - 1] GRAPH_PCS = {32'h880001ec
3                                                       , 32'h880012e4
                                                        , 32'h8800138c};
```

Listing 5.3: Verilog Program Counter array for the Injector

These are the addresses, specified at synthesis time, which will cause the Injector to trigger the process of utilizing the RF, as was explained in the overview of the current system in section 4.1.2. In this example, three addresses are contained within the Injector, meaning that three graphs where mapped to hardware by the tools, and will trigger the functioning of the RF once the GPP reaches the respective memory positions.

Another auxiliary file that is created contains simple environment information to be passed to the next tool in the chain, described in section 5.2. It contains the number of input and output registers, as well as the number of routing registers along with the base address of the fabric. The need for these values will be later explained.

The program is able to parse MicroBlaze operations that correspond to the supported brick types found in listing 4.5 but is not restricted to that set. In terms of flexibility, Graph2Bricks was written to permit quick expansion to other Instruction Set Architectures (*ISA*), requiring only adding the instructions composing that instruction set to header files. The program supports selection of one of the supported architectures upon call (although only MicroBlaze was utilized for development). So, in the same manner, adding bricks types (i.e. new operations) to the application is equally simple.

To support this, the tool was written to be able to parse operations with any number of inputs or outputs and route any output to any input. In that sense, it is not strictly bound to the hardware architecture of the RF. Regarding constant operators, it is also able to output information that configure the bricks so that the proper operator is taken as a constant (for instance, in a subtraction operation, creating a brick that either implements *a* - *constant* or *b* - *constant*).

Regarding the relationship between the parsed MicroBlaze instructions and the bricks, it is not necessary 1:1. That is, each instruction does not necessarily imply a brick (hardware module) that matches it and only it. Some generalization was attempted, and achieved up to a point. Any type of addition at MicroBlaze level (add with carry, add without carry, add with immediate value or register value) can be performed by the same addition module in the fabric. However, some operations are to specific to generalize. Still, this does not compromise flexibility in any respect. Overall, the entirety of the application is written in considerably discrete modules, made as transparent and independent as possible.

5.1.2 Generating Routing Information

As mentioned, the RM is the module of the system that contains information regarding the routing of the fabric. This information is utilized to configure the fabric at any moment where a particular graph is to be executed. Graph2bricks being the tool that works upon information regarding operation connections in order to place them, becomes also the appropriate place within the toolflow to generate these routing values.

To better understand the layout of the routing information within a register, consider the example graph in figure 5.1.

In this figure is represented a possible interconnection of operations within the RF. The routing values to be generated for this, or any graph, include the connection of input registers to the top of the fabric, connections between rows and the connections to output registers. As stated before, the widest number of outputs dictates the number of bits used to perform a selection. In this case, row 1 has the most outputs, 5. So, 3 bits would be required to represent a range from 0 to 4. Each group of 3 bits is referred to as a *block*.

The total number of inputs in the fabric is 14 accounting for the output register (bricks with a constant operator do not have their second input represented, but it is necessary to count it). So, the total number of bits required is 3 times 14, totaling 42, which means 2 routing registers are needed. The registers are represented with the LSB at the right, and each block is represented in decimal notation. The numerical value of the block corresponds to the output identification of the previous row, from the position of the block itself in the register are derived the input identifications for that row. To clarify, the switchbox in *Row 0* would be feed the first 6 blocks and attribute to its output nr. 4 input nr. 2.

As the figure shows, some bits of the registers are not used, marked as *don't care*. In *Register 1*, these bits correspond to the most significant bits of the register that were not utilized, as only 42 are required. In *Register 0*, the 2 most significant bits of the register are also not utilized since the size of the block is 3 (only 10 groups of 3 bits fit 32 with 2 bits remaining). As for sixth and seventh blocks of the same register, these would correspond to the first input *anl* and the second input of *bra*. Although not used (as the brick is operating on a constant value) as Verilog does not allow for a module to have a variable number of ports (i.e. existence of non-existence of the port based on a parameter). So, the ports must exist, but are left unconnected, which is a design hindrance related to the manner utilized to describe the RF.

In terms of multiple calls to this tool, maintaining current information about brick placement is fairly obvious. It is required to re-utilize bricks between graphs and to produce a final representation of hardware for all the graphs. The reason for maintaining routing information as well now becomes apparent. If the number of selection bits required for the graph increases, by increasing of the maximum number of outputs in any row, then the routing tags will have to be placed in different locations in the routing registers. In fact, a larger number of registers is likely to be required. So, the solution is to store this information in a structure that is abstract from the routing registers but in fact contains the same information and to recalculate the routing registers at every

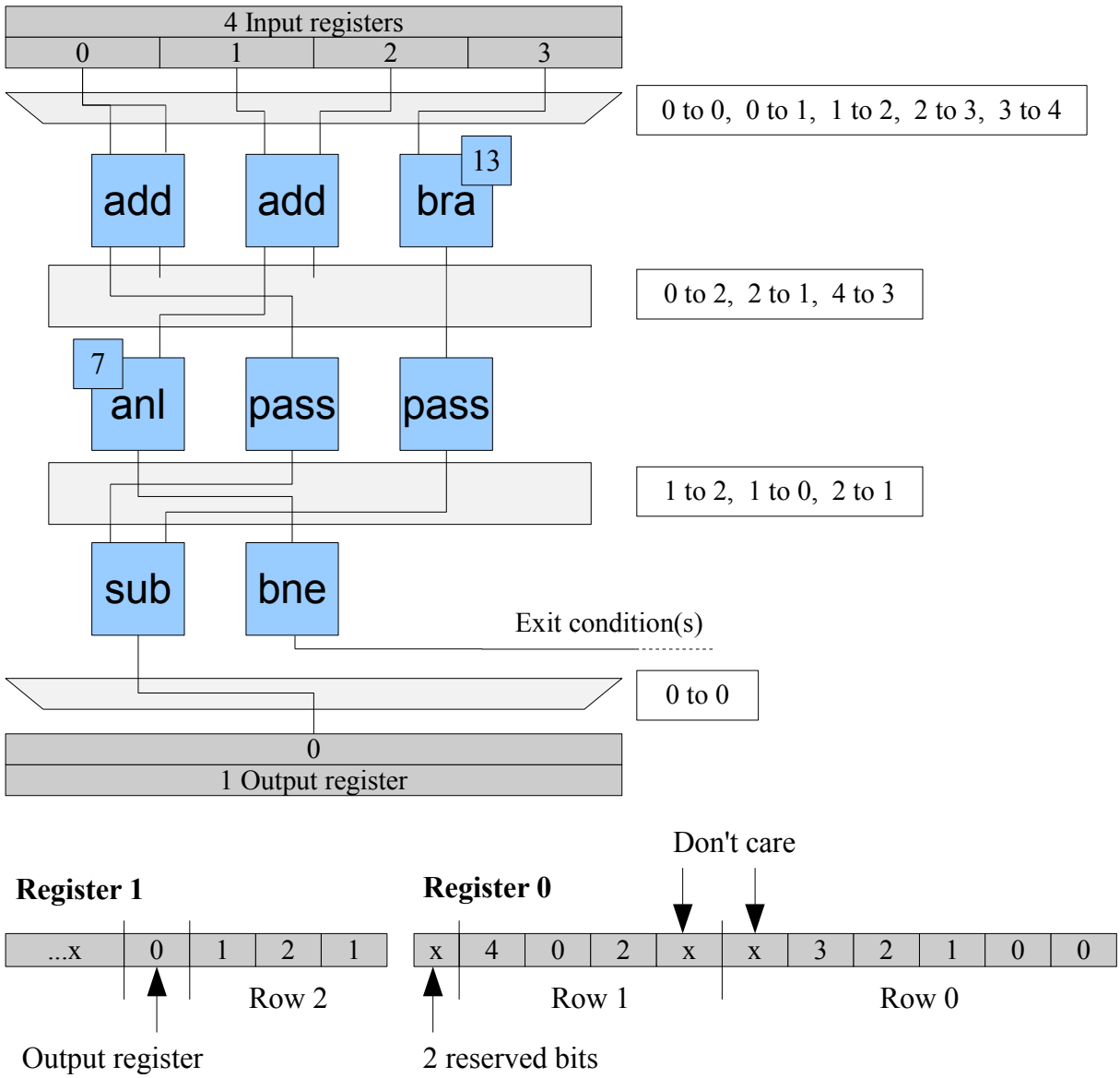


Figure 5.1: Routing Example - Inputs and outputs are numbered left to right, branches have individual numbering, as they are not treated in the same manner.

call, if necessary.

```
//represents a routing of an input to an output in a generic fashion
struct routing {
        int     fromX,  //upper level X coordinate, output
                toX;    //lower level X coordinate, input
        int     route_lvl;
                        //what level is this referring too
};
```

Listing 5.4: Abstract routing structure

So the alteration of the fabric's width creates the need for re-routing, however, the fabric may also be increased in depth if the tool is called with a graph with greater depth than those before. In such a case, the fabric will now need to propagate the data of smaller (less deep) graphs downwards in order to feed them back or route them to the output registers. In order to do that more passthrough operations are required to propagate the signals downwards. So, in terms of routing, more registers will be required and their value will have to be computed. Consequently, more information will have to be stored in the Generation File for all the runs of the tool.

In fact, one of the issues with the fabric's structure is the number of passthroughs required even for simple graphs. As was seen before in the dynamic approaches to the fabric architecture, placement of passthroughs would exceed placement of the operations themselves. The same is true here, the passthrough placement creating a pyramid of passthroughs, each lower row requiring more than the previous (to guide the new results of each new row and the previous ones back to the input of the fabric). Although the passthrough operation itself is only wiring and so does not introduce any logic, all the operations in the fabric are registered so for an elevated number of passthroughs an equally elevated number of registers to hold their outputs at each row. This effect can be seen, for instance, in figure 5.7.

5.1.3 Constraints and Optimizations

In the same way that it is expandable to other architectures (i.e. processors), the tool has a flexible system for constraint specification. The two sets of constraints considered for generating a description for the developed hardware architecture are relative to the operations themselves, and the placement of those operations. Although both adding a new *ISA* or altering the constraints requires recompilation, the modification effort and time is small and punctual.

Regarding operation constraints, they are all related to the capabilities of the fabric themselves. The operations composing a graph are parsed and data structures are initialized with all relevant data. Upon parsing, the program verifies a set of conditions that must be met if the graph is to be expressed as hardware. For now, the considered restrictions on operations to be mapped have been already presented while describing the hardware. The currently utilized constraints are the maximum supported of inputs and outputs and whether or not that parsed instruction, part of the considered *ISA*, has an equivalent hardware operation available (i.e. a brick). Initially, a restriction that only allowed one output per brick was in place at fabric level, that is, in its HDL description.

However, the fabric was later further developed to potentially support any number of inputs and outputs, so the constraints regarding this could be lifted. However, the constraint system can also be used not only to generate information which is within the capabilities of the fabric, but also to dictate restrictions that might be wanted simply for design purposes. That is, restricting the maximum width wanted, or not allowing particular types of operations for reasons of available resources or area.

Without going into needless detail, restrictions are held in an array of functions, each function being a restriction:

```
1        //constraint check
         int (*op_constraint_funcs[NUM_OP_CONSTRAINTS])
3                        (struct operation *currentop) =
                                 {op_numinputs, op_numoutputs, op_validclass};
```

Listing 5.5: Graph2Bricks Constraints

So, the addition or removal of constraints from the constraint vector could be easily implemented by call time switches once a large enough library of constraints warranted such a feature.

Regarding mapping constraints, the utilized system is the same, and the current constraints are actually better regarded as optimizations, although any could be added that acted as a placement constraint. As stated during the description of the RF in section 4.3.3, and also in the sections describing the non implemented approaches, two or more graphs can be overlapped in terms of operations. In other words, they can be matched. So, since the implemented fabric is capable of routing any output of a row to any input, its a simple matter of verifying the current state of the fabric as to find operations already mapped to be re-utilized if needed, or possible. So, one of the optimizations performed when placing bricks, is the reuse of already mapped bricks. This way, the necessary hardware is reduced. The program attempts to reuse bricks as much as possible. As presented before, bricks can either receive two variable inputs or one variable input and a constant input. A brick with 2 variable operators can be reused without limit for graphs that utilize the operation it implements, however, a bricks with an defined constant value (set by a previously parsed graph) can only be reused between graphs that utilize that same value. This is also supported even if the constant operator is operator A instead of operator B, but only if the operation is commutative. If these conditions are not met, a new brick is required. Naturally, the same addition brick for instance, cannot be utilized to perform two distinct additions by the same graph, seeing as though that addition is either in parallel or on another level.

In the previous chapter, in the description of the adopted RF, it was mentioned that a brick could also operate with both two variable operators and with only one operator and a set constant value, selecting one or another functioning at run-time. That hardware feature ended up not being utilized because this tool does not yet generate that configuration information. Implementing it would simply require further software iterations and add nothing to functionality, contributing only to a reduction of fabric size. So, Graph2Bricks can also activate or deactivate the use of double typed bricks. This is an example of editing mapping constraints. Regardless, having double typed

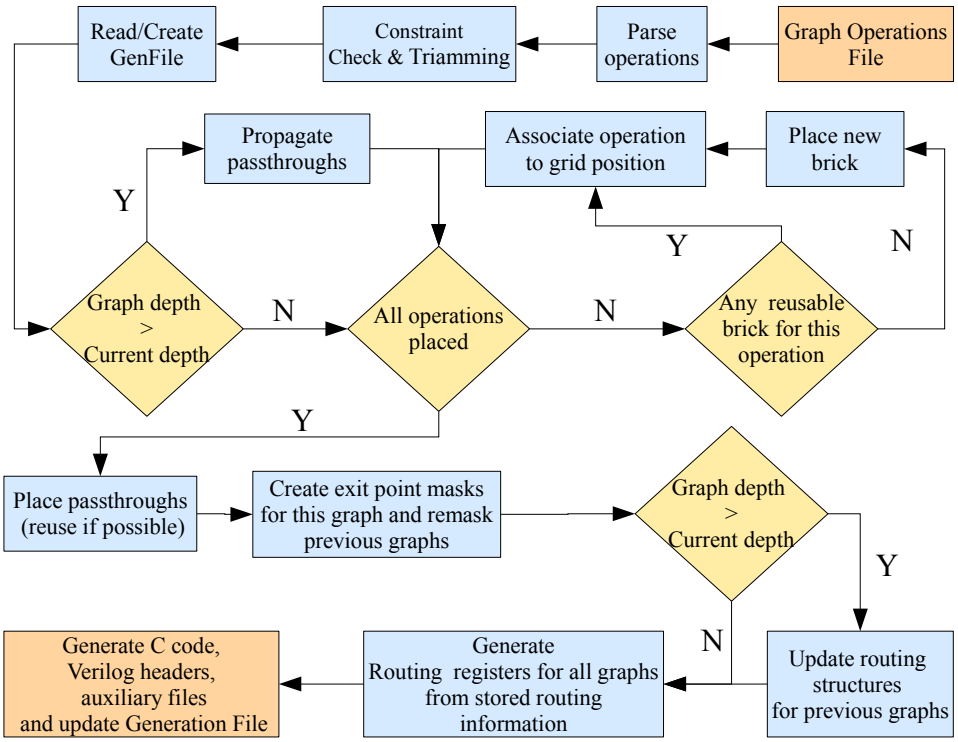


Figure 5.2: Graph2Bricks FlowChart - A summary of the tasks the program performs. Generation and verification of routing information and placement of passthroughs was one of the most time consuming features to implement.

bricks would have allowed to reduce space, but would have introduced reconfiguration delays and additional memory mapped registers to configure, at runtime, the operating mode of the bricks.

Another aspect regarding operation mapping is relative to passthroughs. Passthroughs are operations in which the output is equal to the input. They are required to wire operands that span more than one row, which is a very frequent characteristic of graphs. So, they are the only bricks on the grid which are not derived from the instructions themselves, but from their connections. Like all the other bricks, their outputs are registered. So, one option was included that dictates a small aspect of passthrough placement. Consider a situation where two operations on one row require the same output from one brick two rows above. One option would be to place one passthrough in the intermediate row for each brick in the lower row. The other, is to place only one, and then feed the two bricks the output of that one passthrough. Although seemingly the same, the difference lies in the hardware behaviour. Whereas placing two passthroughs creates 2 32 bit registers (the outputs) each with a fanout of 1 (to the brick below), the second alternative creates only 1 32 bit register with a fanout of 2 (to both bricks). This option was introduced in order to test if there was any observable trade off between area and altering of the fanout of the registers in terms of clock frequency.

Also, after verification of operation constraints, the program performs some trimming optimizations, removing needless information from the parsed operations. For instance, the second operand of a branch instruction, which is the relative jump value of the branch operation. This has

no equivalency in terms of hardware, as the return to software is handled by the code generated by the Graph2Hex tool explained below.

So, to reiterate, the flowchart in figure 5.2 of this program summarizes, in a general fashion, the steps performed to generate the described information.

5.2 Graph2Hex

While Graph2Bricks generates information regarding the hardware description, this program acts as a simple assembler that generates communication routines. As with Graph2Bricks, the program can be easily adapted to any ISA and keeps execution context between calls, although it only needs to store a minimal amount of information as the graphs do not influence each other's communication routines.

Since no knowledge of the internal connections of the operations is necessary, this tool only requires inputs regarding which GPP registers are to be loaded to to fabric, and to which registers the results are to be recovered too. So, the input files to this tool are ones such as the example in figure 3.7. Graph2Hex also requires an output file from Graph2Bricks that contains the base address of the fabric and the number of input and output registers. This ties in as to why both tools cannot work in parallel, as is displayed in the toolflow in section 5.3. In order for this program to know the number of input and output registers on the fabric, Graph2Bricks must be run first in order to determine these numbers (which are attained after the fabric is described by processing the outputs of the Graph Extractor). The numbers are required in order for Graph2Hex to know the addresses to write to and read from while assembling the code.

5.2.1 Main features

As an output, the program generates several files, one per graph, that contain the communication with the RF via the PLB bus, utilizing the MicroBlaze's load and store instructions to write and read from the fabric as well as some other auxiliary instructions. One example of this output is as seen in figure 5.3 (instructions omitted for brevity). These routines are referred to as Code Segments (CS).

Graph2Hex first generates code that saves the value of one of the GPPs registers to the RF's Context Register. This is necessary because one register of the GPP will have to be utilized in order to perform the required memory loading and storing instructions in a more efficient manner. As explained before, the *IMM* instruction is used to allow the following instruction to work with immediate values of 32 bits.

So, an absolute value operation requires 2 instructions, one loads the *IMM*, and the second is the instruction itself which contains the lower 16 bits (the *IMM* is then cleared, needing to be reloaded). Its quick to conclude that, for instance, 5 registers to load to fabric would result in 10 instructions if absolute loads were used. So, relative loads and stores are being used, in which only 1 instruction is required per load/store by keeping the upper 16 bits in one of the registers in the register file.

Save context of MB register (needed if 1st iteration fail): 0x880f0064: 0xb000c4e0 0x880f0068: 0xfba00050 **Load const live-ins:** **Load live-ins:** 0x880f006c: 0xb000c4e0 0x880f0070: 0xfba00004 **Set address offset:** 0x880f0074: 0xb000c4e0 (...) **Load itercount(start signal):** 0x880f0080: 0x23a0ffff (...) **Wait for fabric:** 0x880f008c: 0xb000c4e0 (...)	**Restore live-outs:** **Set address offset:** 0x880f009c: 0xb000c4e0 (...) **Restore live-outs that holds carry:** 0x880f00a8: 0xb000c4e0 (...) **Recovering last live-out:** 0x880f00c0: 0xb000c4e0 (...) **Jump back:** 0x880f00c8: 0xb0008800 0x880f00cc: 0xb80c12e4 popcnt8-2-stats.txt Percentual gain (instructions reduced too): 45.000000

Figure 5.3: Graphhex File - A simple sequence of instructions that write all inputs to the RF and later recovers them. Allows for recovery of values into carry. These instruction sequences are named Code Segments, and each represents one communication routine with the RF.

Still, the need for the Context Register is only justified by coupling the previous explanation with the fact that the fabric may conclude execution at the very first iteration. Thus, the contents of the register used to as part of the instructions would be lost upon return to software, and a loss of execution context would occur. Maintaining the original value in the Context Register allows for its recovery.

Following this, the instructions that copy the contents of the appropriate GPP registers to the RF, as interpreted from the Stats file, are written. Instructions to send the start signal and to poll Status Register follow. After the execution is completed, the output registers of the RF are copied back to the destination registers of the GPP. In the example given, there is also code to retrieve a carry result. While the load instruction provided by the MicroBlaze ISA allows a value from the RF directly to the register file, the *Carry* bit of the processor is held in a special register which is bit addressable. So code must be generated to check for the value present in the output register, and set or clear the *Carry* bit.

The file also outputs a gain factor. This is the ratio of instructions that the GPP would perform, at most, in software execution, versus the number of instructions needed to communicate with the fabric. This does not include the time required for hardware execution, it is merely a measure of reduction in terms of MicroBlaze instructions.

Related to this, in section 4.3.3.2 a formula for estimating the execution time of a graph was introduced, equation 4.1. Now understanding the entirety of the communication routine, this can be adjusted to include the equations found in section 6.1, relative to system overheads.

Currently, Graph2Hex generates instructions that the MicroBlaze executes through the PLB bus, i.e. memory loading a storing. The fact that the Code Segments are in external memory introduces considerable overhead. One possible adaptation would be to have the system function in a one to one connection from the GPP to the RF, detailed in section 7.1.

5.3 Toolflow

The complete toolflow of the system is as detailed in figure 5.4. These are the steps necessary to describe an RF and generate all the necessary configuration information.

As a final clarification as to how to tools connect the following is a short description of the flow.

Firstly, the code must be imported into the XPS development environment in order for the compilation tools to link the program into the appropriate memory positions, resulting in an *ELF* file properly placed in memory (in this case, the program is placed at the start of the DDR2 RAM). Now the *ELF* may be passed through the Graph Extractor, which will generate the files presented above regarding the graphs (which are identified as being in DDR2). Graph2Bricks and Graph2Hex may now be run over the previous output. Due to the dependency of the latter on the number of inputs and outputs provided by the former, Graph2Bricks must be ran first. An alternative would be to have an intermediate tool to generate that information for both, making their executions independent. This does not compromise any functionality however. These two tools will then generate all the necessary hardware information.

With the Verilog headers now generated, hardware synthesis can be performed for the RF and the Injector, resulting in netlists for both peripherals. The assembly code to be executed by the GPP is ran through an auxiliary script that places it in C containers so it can be included by the RM and copied to DDR2. The final system may now be generated, resulting in a bitstream ready to be transfered to the FPGA.

In order to execute the segment of the toolflow containing Graph2Bricks and Graph2Hex, several auxiliary scripts were created that automate the calls to the programs and generate other auxiliary files, such as the *C* files utilized by the RM (containing the CSs). These scripts consult the program's directory for input files and call the tools for each input file. At the end of all executions the outputs are copied to other folders as to permit the execution of following tools. Encapsulating these scripts is a single script. So the output of the system, up to the point of the hardware descriptions and header files, can be generated by a single run of a script, assuming that appropriate input files were placed in the tools directories.

Although the toolflow starts at source code, no tool performs a static analysis of the source code, as the Extractor receives the instruction stream from a simulator. The need to start from the source code of the applications appears because the program needs to be properly linked into the address of the external memory so as to have the tools pick up the correct addresses in turn. However, if the program to be ran is small enough to fit in BRAMs this problem may not appear. Since the memory addresses of the BRAMs for any processor in the system start at zero, an *ELF* previously linked will most likely have been linked from this address onwards as well, maintaining coherence. If this this is not the case, there is no way to relink an executable *ELF* file, thus the flow must start from the source code.

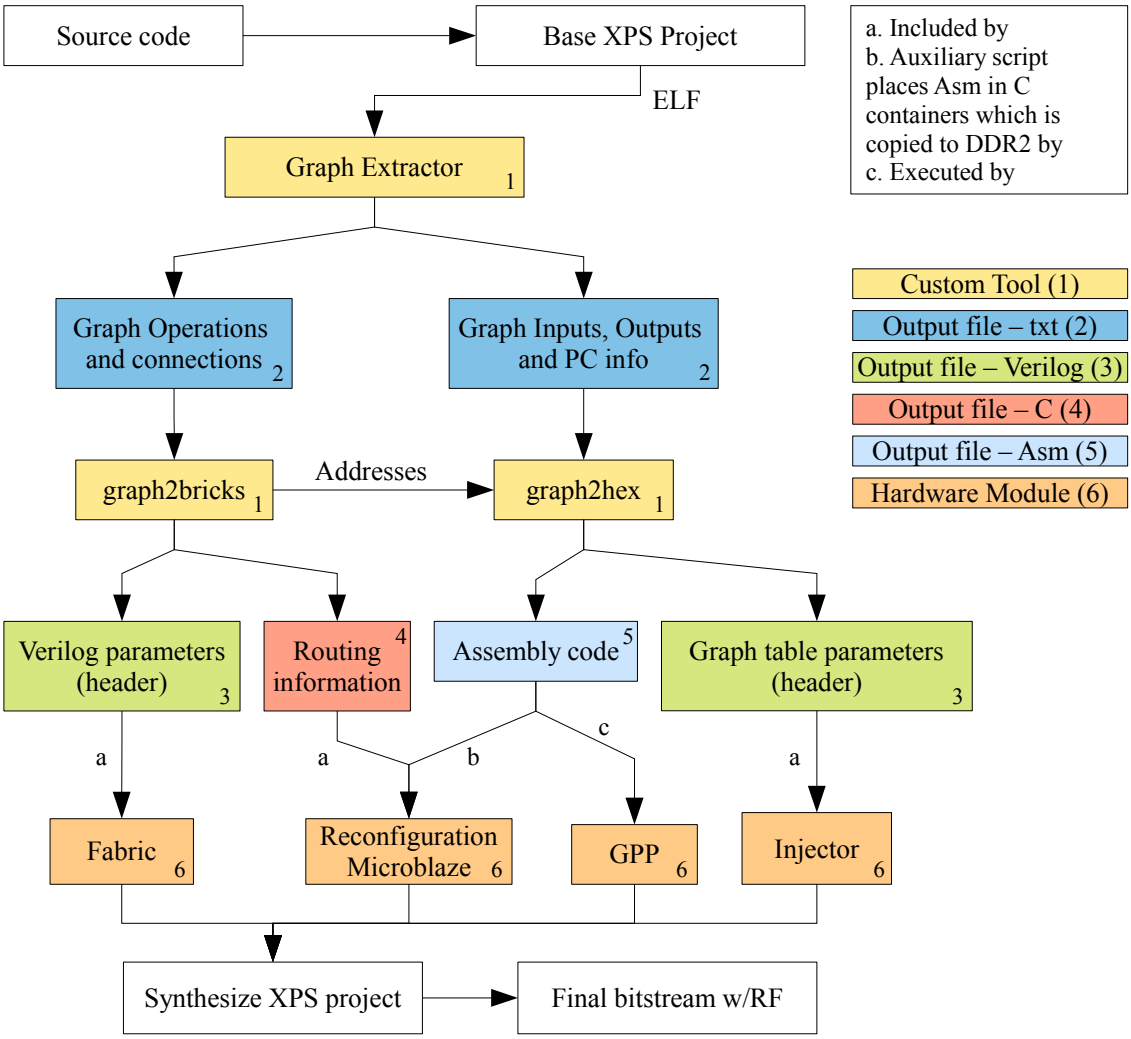


Figure 5.4: Complete Toolflow Diagram - these are the necessary steps to arrive at a functional reconfigurable fabric.

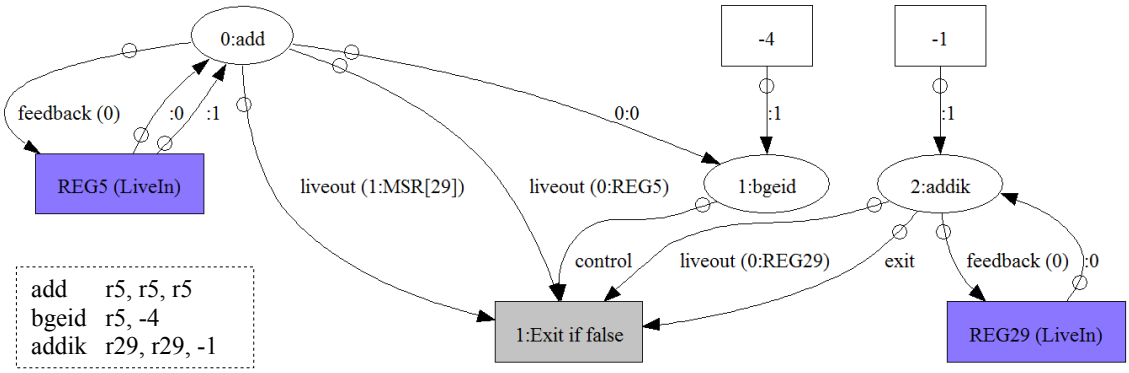


Figure 5.5: Example Graph - a small example graph to demonstrate the output of the tools

5.3.1 Toolflow Example Output

The following is the result of passing the Graph Extractor outputs relative to the simple graph in figure 5.5 through the explained tools. The CS to perform communication for this graph is in figure 5.6. A graphical representation of the resulting fabric and it's routing registers is in figure 5.7. The Verilog parameters that describe the fabric are represented in listing 5.6 and the related address table for the Injector is as presented in listing 5.7.

```
//Verilog fabric parameters:
parameter MAX_WIDTH              = 32'd4;
parameter NUM_COLS               = 32'd4;
parameter NUM_ROWS               = 32'd2;
parameter NUM_IREGS              = 32'd2;
parameter NUM_OREGS              = 32'd3;
parameter NUM_ROUTEREGS          = 32'd1;
parameter NUM_FEEDBACK_REGS      = 32'd1;

parameter [ 0 : (32 * TOTAL_EXITS) - 1 ] NEGATE_EXITS    = {32'd1};
parameter [ 0 : (32 * NUM_ROWS) - 1 ]    ROW_NUMINS      = {32'd4, 32'd4};
parameter [ 0 : (32 * NUM_ROWS) - 1 ]    ROW_NUMOUTS     = {32'd4, 32'd4};
parameter [ 0 : (32 * NUM_ROWS) - 1 ]    ROW_NUMOPS      = {32'd2, 32'd4};
parameter [ 0 : (32 * NUM_ROWS) - 1 ]    ROW_NUMEXITS    = {32'd0, 32'd1};
parameter [ 0 : (32 * NUM_ROWS * NUM_COLS) - 1 ] ROW_OPS = {
                        //this is the top of the fabric
                    `A_ADD, `A_ADD, `NULL, `NULL
                    , `B_BGE, `PASS, `PASS, `PASS    };
                        //this is the bottom

parameter [ 0 : (32 * NUM_ROWS * NUM_COLS) - 1 ] INPUT_TYPES = {
                    `INPUT_ONLY, `CONST_ONLY, `NULL, `NULL
                    ,`INPUT_ONLY, `INPUT_ONLY, `INPUT_ONLY, `INPUT_ONLY };

parameter [ 0 : (32 * NUM_ROWS * NUM_COLS) - 1 ] CONST_VALUES = {
                    32'h0, 32'hffffffff, `NULL, `NULL
                    ,32'h0, 32'h0, 32'h0, 32'h0      };
```

Listing 5.6: Example Graph Parameters - some content of this file is still omitted for brevity

```
1        localparam NUM_PCS = 1;
         localparam [ 0 : 32 * (NUM_PCS) - 1] GRAPH_PCS = { 32'h88001314};
```

Listing 5.7: Example Graph Table of PCs

Save context of MB register (necessary if 1st iteration fail):	**Return if First fail true:**
0x880f0000:imm -15136	0x880f0048:imm -15136
0x880f0004:swi r29, r0, 40	0x880f004c:lwi r29, r0, 40
Load live-ins:	0x880f0050:imm -30720
0x880f0008:imm -15136	0x880f0054:brki r0, 4884
0x880f000c:swi r29, r0, 4	**Restore live-outs:**
Set address offset:	**Set address offset:**
0x880f0010:imm -15136	0x880f0058:imm -15136
0x880f0014:addi r29, r0, 0	0x880f005c:addi r29, r0, 0
0x880f0018:swi r5, r29, 0	0x880f0060:lwi r5, r29, 28
Load itercount(start signal):	**Restore live-outs that holds carry:**
0x880f001c:addi r29, r0, -1	0x880f0064:imm -15136
0x880f0020:imm -15136	0x880f0068:lwi r29, r0, 24
0x880f0024:swi r29, r0, 20	0x880f006c:bnei r29, 12
Wait for fabric:	0x880f0070:msrclr r29, 4
0x880f0028:imm -15136	0x880f0074:bri 8
0x880f002c:lwi r29, r0, 36	0x880f0078:msrset r29, 4
0x880f0030:andi r29, r29, 4	**Recovering last live-out:**
0x880f0034:bnei r29, -12	0x880f007c:imm -15136
Check for exit status:	0x880f0080:lwi r29, r0, 32
0x880f0038:imm -15136	**Return Jump:**
0x880f003c:lwi r29, r0, 36	0x880f0084:imm -30720
0x880f0040:andi r29, r29, 32	0x880f0088:brki r0, 4884
0x880f0044:beqi r29, 20	
...	executable-4-stats.txt
	Percentual gain (instructions reduced too): 55.555557

Figure 5.6: Example GraphHex - communication routing for this graph from the GPP to the RF. The instructions have been decoded into their original mnemonics for clarity

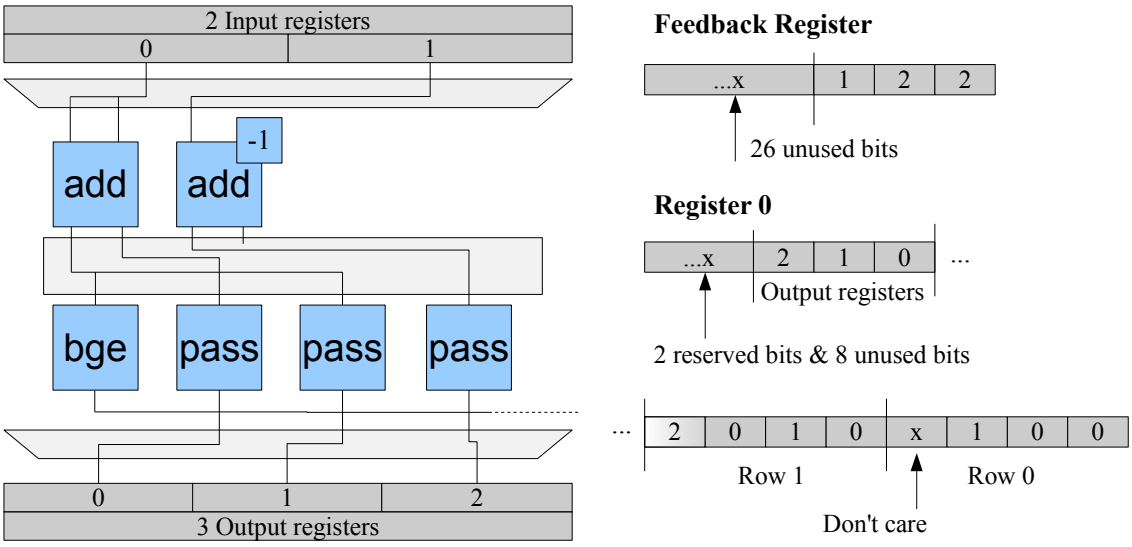


Figure 5.7: Example Graph Layout - resulting hardware layout and routing information

Chapter 6

Results and Conclusions

The implemented prototype was tested with 6 simple benchmarks to provide a proof of concept of the entire architecture and to observe the behaviour of the system in terms of speedups. As was stated before, the Injector allows the disabling of the entire acceleration system via a switch, and so, the benchmarks were ran with the system deactivated and then activated. The architecture was as explained in section 4.1.2.

Since the RF did not permit memory operations, the working set of graphs was somewhat reduced. Thus, each benchmark was based on a simple loop or two nested loops that performed operations on single variables (i.e. no array accessing). The benchmarks utilized only contained, usually, one graph useful for implementation due to their simplicity. These graphs were found encapsulated within a function call. So, the results presented in section 6.2.1 are for one graph per benchmark. In order to test the functionality of an RF implementing several graphs, a benchmark was written that included calls to all the functions of previous benchmarks (*merge*). In other words, *merge* contains 6 graphs which were sucessfully translated into a hardware description. These results are presented in section 6.2.2.

Five of the utilized benchmarks were generic routines, *Even Ones*, *Hamming*, *Count*, *Pop Count* and *Reverse*. The last is a benchmark taken from the SNU Real-Time Benchmarks suite [1], namely, *Fibonacci*. In appendix A an excerpt of code from each benchmark and the graphical representation of the implemented graph for that benchmark are presented along with detailed result tables for each one. All the benchmarks had changeable parameters that allowed for testing the same benchmark for a different number of calls of the graph, for instance. These can be better understood by consulting the code found in the referenced Appendix. A call of a graph is understood to be either the execution of the code from which the graph was derived, if used in a software context, or the utilization of the RF to perform that graph, if used in this context.

The tested graphs are quite similar amongst each other, due to the current status of development of the prototype, but they still provide a measure of speedup and prove the transparency of the system as well as the functioning of the toolflow. One detected graph was functionally supported but not tested as the amount of passthroughs required to route it exceeded FPGA resources. Altough passthroughs are registered, they could be implemented as simple wiring, as the RF does

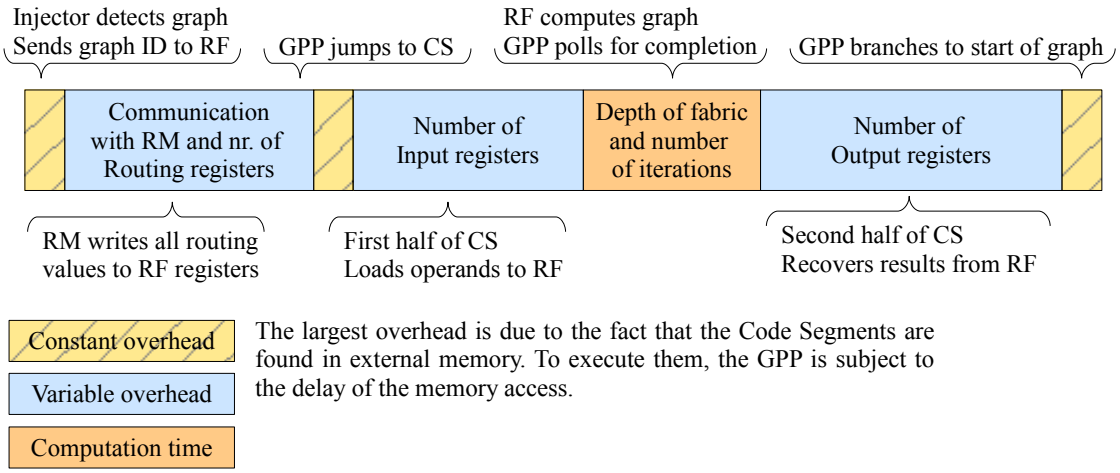


Figure 6.1: System Overheads - The variable overheads are those that could be improved with some alterations to the system. The factors that contribute to each segment of time are detailed within the corresponding box (not to scale).

not act like a pipeline and data is only retrieved after a number of clocks corresponding to an iteration. This was not tested however.

6.1 Causes for Overhead

To better understand the comparative results in the next section, figure 6.1 summarizes, once again, the functioning of the system while representing the overheads it is subject to.

Although these overheads greatly add to the total time required to run the program via acceleration and, so, lower the achievable speedups, the factor that should be considered for comparison is the computation time within the fabric. It is this time that is a measure of the gain achieved by automatically detecting and generating a hardware description for graphs. Of course a reduction of the overhead is important, and manners through which it can be reduced are later discussed in section 7.1.

Regarding the computation time of the graph itself, it is as expressed by equation 4.1. So, if all overheads were to be eliminated this would be the true factor of speedup for the system. Of course, the speedup would be proportional to the parallelism possible, for a given graph.

Now that previous sections explained the functioning of the system, the remaining time can be expressed by the following equations.

The routing overhead is a direct function of the number of routing registers present in the system, so, this overhead can be expressed by equation 6.1. Consider that each access utilizing the PLB bus (to write to each register) can be as long as 23 clock cycles, expressed as N_{Ac} (worst case scenario as measured with ChipScope Analyzer, a signal analysis tool). Let Nr_{RR} be the number of routing registers and T_{CR} the total number of clock cycles this overhead introduces.

$$T_{CR} \simeq N_{Ac} \times Nr_{RR} \tag{6.1}$$

Adding to this is the time the RM requires to execute its own program. Considering it is found in BRAMs, this time is negligible relative to the given equation.

The overhead caused by loading and retrieving values from the RF is relative to the Code Segment itself. It is a direct function of how many instructions compose that CS. The CSs are in external memory and so must be fetched (in fact, the approximate 23 clock cycles required for an access over the PLB bus were measured from an access to external memory). So, consider the previous variables and let N_{CSInst} be the number of instructions that make up the Code Segment and T_{CS} the total number of clock cycles.

$$T_{CS} \simeq N_{CSInst} \times N_{Ac} \tag{6.2}$$

So the complete time that is required to perform a graph in hardware is the sum of the PLB access time for writing all the inputs to the fabric and reading the outputs plus the time it takes for the computations themselves to be performed within the fabric, adding to the constant overheads which can be neglected.

So, the total time is as expressed by equation 6.3.

$$T_C \simeq T_{CR} + T_{CS} + T_{FC} \tag{6.3}$$

6.2 Comparative Results

The objective of the system was the acceleration of detected graphs via custom hardware description. So, as stated, the comparative factor of interest is the computation time within the graph to determine the gain derived from parallelism. However, the system does suffer from considerable overhead, as is shown later in table 6.6. So, to have a good term of comparison for the speedup obtained by the system, it will be compared with a reference system composed solely by a Microblaze Processor, running a benchmark located in external memory, with data and instruction caches enabled (with 2Kb of size) as well as a barrel shifter and multiplier.

Since there was no immediate way to measure the actual computation time within the RF in runtime, the following values are derived from equation 4.1 found in section 4.3.3. Unlike the other formulas that calculate overhead, this formula is not affected by any estimation errors, and the actual values of computation within the fabric may actually be derived by simulation alone.

The execution times were extracted via a timer peripheral added to the system. The segments of code that were translated into graphs were encapsulated between a call to start the timer, and a call to stop it. These calls introduce further constant delay as show later in appendix A. The timer returns the number clock cycles it has counted, so, all the values relative to hardware and software execution in the following tables are expressed in this unit. The measured values were retrieved via a UART peripheral.

Table 6.1: Result excerpt for RF system versus a Cache enabled system

	Even Ones		Hamming		Fibonacci		PopCnt(inner)	
Calls	100	400	100	400	100	400	100	400
HW	121575	483243	117249	465992	142552	753468	121590	483261
SW(cache)	24087	95494	24106	95506	36058	563308	29775	118275
Speedup	0,2	0,2	0,21	0,2	0,25	0,75	0,24	0,24

(a) benchmarks with 1 parameter

	Count		Reverse	
Calls of Graph	100	400	100	400
HW	112930	448761	125848	500427
SW (cache)	7084	27484	27106	107495
Speedup	0,06	0,06	0,22	0,21
	I = 8		I = 32	

	PopCount	
Iterations	128	512
HW	2210	3285
SW (cache)	1139	3837
Speedup	0,52	1,17
	Bits = 1	

	Count		Reverse	
HW	121528	483204	136604	543475
SW (cache)	14085	55489	52695	209894
Speedup	0,12	0,11	0,39	0,39
	I = 18		I = 64	

	PopCount	
HW	4963	8486
SW (cache)	4627	17693
Speedup	0,93	2,08
	Bits = 3	

(b) benchmarks with 2 parameters

6.2.1 Results for an RF implementing a single graph

Versus a cache enabled software-only system the RF prototype achieves worse results in terms of speedup in most cases, although it evens out the overheads in some. The overhead introduces the most time when the graph repeats a large number of times and an equal number of communication routines are needed to utilize the RF.

The four benchmarks in table 6.1a were tested varying the number of calls of the function containing the graph. At every call, the system was triggered to utilize the RF. For *Even Ones*, *Hamming* and *PopCount (inner)* the speedup is constant regardless of number of calls. Regarding this last benchmark, it contains an nested loop. Both the inner and outer loop were detected as separate graphs and both were tested, the refered table containing the results for the inner loop.

The graphs for these three benchmarks iterate a number of times that is constant from call to call (it is a constant value built-in the fabric itself). So, more calls add the same amount of time to both the hardware and the software, keeping the speedup constant. Unlike these, *Fibonacci* contains a graph whose number of iterations is dependent from one of the input values. In this case, that input value is the number of the call itself. That is, the graph iterates once in the first call, twice in the second and so forth. Since a complete iteration completes within the fabric quicker than at software level, with a sufficient number of iterations, the overhead introduced by communication begins to even out. For a total of 400 calls that results in 80200 iterations through the fabric. For each call, the time spent in the fabric increases with the number of iterations while the communication time remains constant. Thus, the total sum of the overheads, for all calls of the

Table 6.2: Result excerpt for RF system versus a Cache disabled system

	Even Ones		Hamming		Fibonacci		PopCnt(inner)	
Calls	100	400	100	400	100	400	100	400
HW	121575	483243	117249	465992	142552	753468	121590	483261
SW	502760	2008477	502792	2008501	754830	11873707	623027	2489356
Speedup	4,14	4,16	4,29	4,31	5,3	15,78	5,12	5,15

(a) benchmarks with 1 parameter

	Count		Reverse	
Calls of Graph	100	400	100	400
HW	112930	448761	125848	500427
SW	144267	574461	566026	2261542
Speedup	1,28	1,28	4,5	4,52
	I = 8		I = 32	

	PopCount	
Iterations	128	512
HW	2210	3285
SW	19819	76501
Speedup	8,97	23,29
	Bits = 1	

	Count		Reverse	
HW	121528	483204	136604	543475
SW	291875	1164937	1105908	4420991
Speedup	2,4	2,41	8,1	8,13
	I = 18		I = 64	

	PopCount	
HW	4963	8486
SW	92697	368024
Speedup	18,68	43,37
	Bits = 3	

(b) benchmarks with 2 parameters

graph, grows linearly while the computation time grows according to an arithmetic series.

Table 6.1b presents some results for the benchmarks in which two parameters were varied. One is the number of calls, the second (indicated below the tables) is a parameter that alters either the characteristics of the graph (as is the case for the outer loop of *PopCount*) or alters the number of iterations per call of graph. Further detail on this is found in appendix A. *Count* and *Reverse* present similar behaviours. As with the results in table 6.1a an increase in number of calls maintains the speedup constant for the same reasons. Likewise, increasing I increases the number of iterations (which equal I) and, so, an increase in speedup is also verified. For the *PopCount* benchmark the table presents the results slightly differently. From this benchmark was extracted a single graph that encompassed it's entire nested loop. So, the graph is called only once but, like *Fibonacci*, with a variable number of iterations. Altering its *Bits* parameter, unlike other benchmarks, alters the depth of the fabric. When *Bits* = *3*, the fabric is 9 rows in depth and contains 20 operations. The fact that the communication routine is only called once shows in the number of clock cycles required to compute the graph in hardware. They are much lower than the remaining benchmarks. In fact, due to this, *PopCount* evens out relative to software execution for a much smaller number of iterations performed in the RF. *Fibonacci* only achieves a speedup of 0,75 for 400 calls of the graph (80200 iterations) while *PopCount* demonstrates a speedup of 1,17 for only 512 iterations even in the smallest version of its graph.

Table 6.2 presents the speedups of the hardware execution times versus a cache disabled reference system. The conclusions to be derived from these results are the same as those explained

Table 6.3: Estimated results for a zero overhead system versus a Cache enabled system. HW(RF) is as derived by equation 4.1.

	Even Ones		Hamming		Fibonacci		PopCnt(inner)	
Calls	100	400	100	400	100	400	100	400
HW (RF estimate)	9600	38400	9600	38400	15150	240600	9600	38400
SW (cache)	24087	95494	24106	95506	36058	563308	29775	118275
Speedup	2,51	2,49	2,51	2,49	2,38	2,34	3,1	3,08

(a) benchmarks with 1 parameter

	Count		Reverse	
Calls of Graph	100	400	100	400
HW (RF)	2400	9600	9600	38400
SW (cache)	7084	27484	27106	107495
Speedup	2,95	2,86	2,82	2,8
	I = 8		I = 32	

	PopCount	
Iterations	128	512
HW (RF)	384	1536
SW (cache)	1139	3837
Speedup	2,97	2,5
	Bits = 1	

	Count		Reverse	
HW (RF)	3600	14400	19200	76800
SW (cache)	9885	38685	52695	209894
Speedup	2,75	2,69	2,74	2,73
	I = 18		I = 64	

	PopCount	
HW (RF)	1152	4608
SW (cache)	4627	17693
Speedup	4,02	3,84
	Bits = 3	

(b) benchmarks with 2 parameters

before. The difference lies in the increase of software execution times. Seeing as though the presence of cache accelerates execution by a near constant factor for all benchmarks (approximately 21), the speedup values are affected by the same factor. Now that the GPP is fetching all its instructions from the same memory (DDR), these values begin to show that the speedup is attained by the ratio of CS instructions versus the number of instructions that would be performed in regular software execution. A difference in speedup between *PopCnt (inner)* and the remaining benchmarks is more noticeable in this scenario. Unlike *Even Ones* or *Hamming*, *PopCnt (inner)* results in more operations within the RF for the same depth, giving it a higher count of Instructions Per Clock (IPC) than the remaining benchmarks, as shown in table 6.5.

Table 6.3 contains some of the estimates of speedups that would be attained relative to cache enabled execution if the system had zero overhead. That is, if the communication routines introduced no delay. Chapter 7 discusses ways to diminish their impact.

6.2.2 Results for an RF implementing multiple graphs

The benchmark written to test both the runtime reconfiguration capabilities of the RF and the routing and placement information generated by the tools calls each of the 6 functions of the utilized benchmarks alternatively. In other words, one function which contains one graph is called, and the RF is used to perform it; following that, another function is called, the RF is then reconfigured for that graph and the graph is performed in hardware. Since the 6 functions are being called

Table 6.4: Results for *merge*

n	100	200	300	400	500
HW	882352	1798852	2740767	3715146	4716849
SW	3068240	7611737	13631430	21127314	30099397
Speedup	3,48	4,23	4,97	5,69	6,38

n	100	200	300	400	500
SW (cache)	146044	361494	646952	1002401	1427844
Speedup	0,17	0,2	0,2	0,24	0,3

n	100	200	300	400	500
HW Estimate (fabric)	55950	141900	257850	403800	579750
Potential Speedup	2,61	2,55	2,51	2,48	2,46

alternatively there is a reconfiguration overhead between each utilization of the RF. Each graph is called n times, so a total of $6 \times n$ reconfigurations of the RF are performed.

The benchmark was compiled so has to have each graph iterate the same number of times as the individual results presented previously. *Even Ones*, *Hamming* , *PopCount (inner)* and *Reverse* iterate 32 times per call of the graph, *Count* iterates 8 times. The graph derived from the outer loop of *PopCount* was not implemented. Table 6.4 summarizes the results for this benchmark.

Table 6.5 contains information regarding the amount of bricks within the RF for each benchmark. The tools manage to reutilize a considerable amount of resources when mapping all 6 graphs to the RF. For all graphs, the number of passthroughs required to implement the connections of operands and results exceeds the number of actual operations. However, the RF for all graph manages to reverse that ratio, containing more operations than passthroughs, and avoiding the mapping of an additional 38 passthroughs, as well as an additional 21 operations. As a consequence, the FPGA resources required for the this RF are less than the total sum of the resources required for each individual RF. Relative to that sum, aproximately 50% of LUTs are required and 25% of

Table 6.5: Brick usage and multiple graph RF resource reutilization

Benchmark	OP Bricks	Passthroughs	IPC	Config. Bits	Pass/OP Ratio
Count	6	6	2	72	50,00%
Even Ones	6	10	2	87	37,50%
Fibonacci	6	9	2	87	40,00%
Hamming	6	9	2	81	40,00%
PopCount (inner)	8	7	2,67	84	53,33%
Reverse	7	7	2,33	81	50,00%
merge	19	10	2,08	212	65,52%

	OP Bricks	Passthroughs	Config. Bits
Sum for all benchmarks	39	48	492
Ratio to RF for merge	48,72%	20,83%	43%

Table 6.6: Communication overhead

	HW (RF estimate)	HW	Comm. Cycles	Overhead
count	12000	603793	591793	98,01%
even_ones	48000	603793	555793	92,05%
fibonacci	375750	1017340	641590	63,07%
hamming	48000	582248	534248	91,76%
pop_cnt (inner)	48000	603818	555818	92,05%
reverse	48000	625284	577284	92,32%
merge	579750	4716849	4137099	87,71%

Flip-flops. In terms of routing registers, each benchmark requires 4, and *merge* requires 8, as it is a wider RF. From the registers, not all bits are used (as explained before), so the useful number of bits within the registers for each graph is also considerably low, as the RF is a dedicated description for that graph alone. For *merge*, the required number of bits is also considerably reduced. The depth for all the RFs is 3.

6.2.3 Overhead

Table 6.6 contains some examples of the overhead measured for the tested benchmarks. All benchmarks are presented, as well as *merge*. The overheads are for the case in which the number of calls of the graph is 500. The depth of the RF is 3 for all cases. All the graphs iterate, within the RF, 32 times except for *Count* which iterates 8 times and *Fibonacci* which iterates a variable number of times. The number of estimated clock cycles required to compute the graph in the RF are subtracted from the actual measured value achived, thus attaining the number of clock cycles which correspond to the overhead. Since *Fibonacci* iterates a much larger number of times than the remaining, more time is spent within the RF, diminishing its overhead.

6.3 Conclusions

The benchmarks utilized to test the system were put through the toolchain explained in both section 3.5 and section 5. The previously explained output files and hardware descriptions allowed for the implementation of small, but functional, dedicated hardware peripherals through the use of the standard synthesis tools utilized afterwards, namely, Xilinx ISE and XPS. As is, the current implementation of the toolchain allows for a near automated generation of these hardware descriptions and their configuration data. So, the toolchain produces outputs useful for implementation.

Regarding the architecture itself, a few aspects leave room for improvement or modification. However, it was proven that the layout is functionally sound. With no interference at a software development level it allows for an considerably transparent adaptation of an embedded system to allow for the use of a custom created hardware accelerator, tailored to the target application's most repetitive software kernels. As for the description of the RF, being based on HDL constructs alone allows for further transparency in terms of design, but is perhaps limited by what the current

hardware description languages allow. An advantage of the overall architecture is the relatively loose coupling between system modules, allowing for easy modifications as to perform further development iterations.

Although the implemented graphs utilized to test the system were relatively simple in structure, the computational results were verified to be correct. Also, even though the documented results for the benchmarks were derived from systems in which the RF has one graph alone, it was also tested with 6 simultaneous, computationally useful, graphs. This last test is specially important as proof of both the proper functioning of the routing capabilities and the validity of the routing information as well as the reuse of already mapped resources by several graphs.

Current issues with the system are relative to communication overhead and the support for more complex graphs, possibly including memory access. For a system not coupled to external memories, an interface with other types of memory buses would have to be developed. Related to this, support for cache would have to be added in order to obtain considerably enhanced speedups. These issues are discussed in chapter 7. Another aspect is the fact that many tasks are being performed offline. Although this reduces runtime overhead it does lengthen deployment time. Another issue are the resource requirements of the switchboxes, which were left as crossbars to facilitate development. For a system in which graphs are detected offline, restrictions on connections make sense in order to reduce resources. However, for an online system, in which graphs are not known before they are constructed, a rich interconenction scheme may be required to ensure support for any detected graph. Reduced interconnection capabilities may still be employed, at the risk of inability to map some of the graphs detected at run-time.

Chapter 7

Possible Modifications and Improvements

7.1 Improving the Current System

The current prototype system is functional within the stated limits for graph support. However, there is some room for optimization. The RM and its interfaces, as well as the routing scheme based on visible, memory mapped, registers was left as is to aid in design. But their functions can be relocated and the whole system greatly simplified. Figure 7.2 illustrates this point. Without introducing any modifications, the current architecture is a halfway point to a design that might allow for detection of graphs at run-time, as it is more flexible and the RM may be utilized to perform this detection and generate new CSs and routing information.

The whole system could be reduced to the RF, the Injector, and a modified version of the currently in place bootloader (which loads the program from flash).

The current function of the auxiliary Microblaze is to, at boot, copy the tool generated assembly to DDR2 memory, thus acting as a bootloader of sorts for the Code Segments. It's second and third tasks are the listening for graph requests over FSL and responding with the proper pair of instructions that permit jumping to the address where the CSs are located and, lastly, re-routing the RF at each request. This, however, can be information completely held in hardware and in the Code Segments themselves. The Injector can hold a lookup table matching graph PCs to memory positions of CSs and the bootloader the GPP contains, to copy its program from flash memory, can copy the CSs as well (assuming these were placed in flash). These would hold the instructions to re-route the fabric as well, as they already hold the instructions to load and recover data.

So, to achieve a functional system such as this, virtually no alteration in the toolchain is required. The resulting functional flow would be as such: at boot, the GPP copies the program from flash to DDR, as well as copying the Code Segments to locations known by the Injector (this module must now know them beforehand, as there is no communication between it and any other module); after that, the program may run; when a graph PC is detected, the Injector will branch the execution to a Code Segment; in those instructions will be contained the writing of input values to

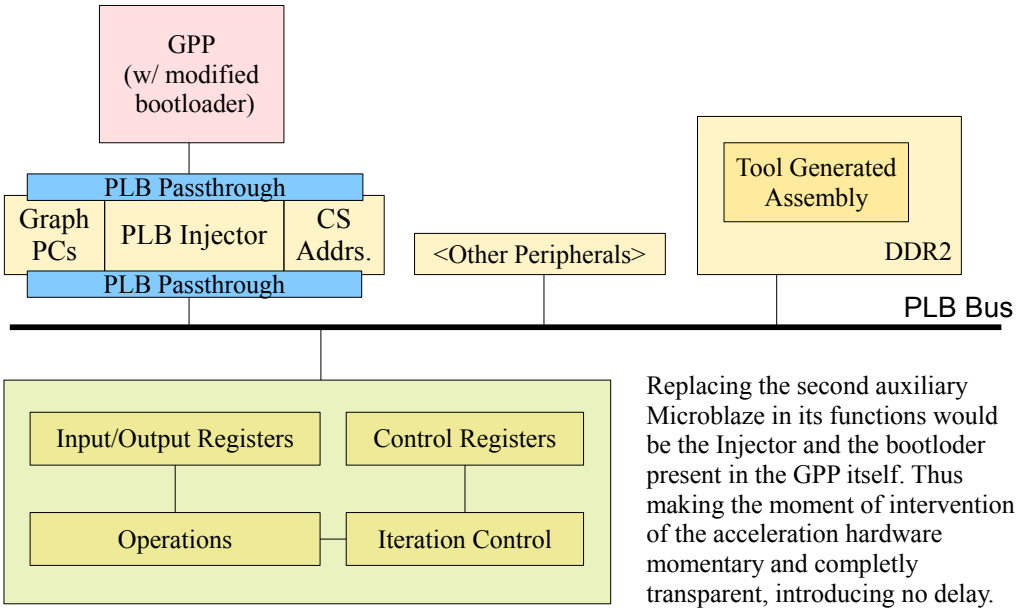


Figure 7.1: Possible Adaptation of Current System - Simple removal of auxiliary Microblaze and minor modifications to the PLB Injector would create a much more efficient and non intrusive system

the fabric, the configuring of routing by writing to routing registers, and the retrieval of outputs as well as the jump back. Also mentioned before, the information provided by the routing registers could be given at synthesis time, reducing the number of memory mapped registers to one graph selection register, which would result in an even smaller reconfiguration time and shorter CSs. The variable reconfiguration overhead associated with re-routing the fabric, would be come constant.

Note that the CSs would now have to be placed in a different location previous to booting. In the current prototype the RM holds the CSs in its BRAMs before copying them to DDR so they are accessible by the GPP. Without the RM, they would have to placed in memory in another fashion. For instance, written to flash along with the program, and copied into DDR by the GPP.

A system such as this would alter the estimates presented previously slightly, as no configuration overhead would be present from the Injector to the RM, from the RM to the RF and finally from the RM to the Injector. The only, more accurately measurable, overhead would be the execution of the Code Segments, and thus, a measure of the speedup can be attained by considering the relationship of the original number of instructions versus the instructions in the Code Segments.

An estimate of the full time it would take for a graph to be completed with this architecture would be as expressed in equation 7.1. Let T_C be the total number of clock cycles and T_{CS} and T_{FC} as computed previously, note that N_{CSInst} now accounts for the additional instructions to have the GPP write a value to a graph selection register.

$$T_C \simeq T_{CS} + T_{FC} \tag{7.1}$$

The total time would be a function only of the computation itself and the communication.

Relative to the interface of the RF, this could also be adapted, although requiring deeper design

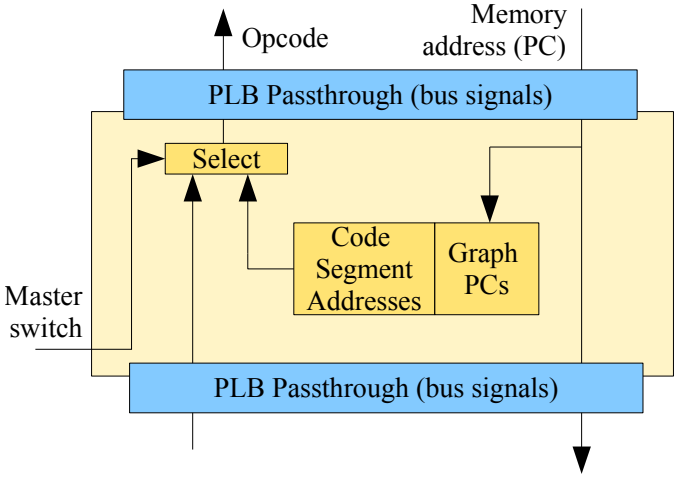


Figure 7.2: Simplified Injector - eliminating the FSL connection and keeping the necessary instructions to be injected in the Injector itself simplifies the system

alterations. As was shown by the equations estimating the PLB access time to the fabric as well as the results, the Code Segments executed make up a great portion of time spent utilizing the RF. Although not much compared to the time of execution of the larger graphs, this time could be reduced by utilizing an FSL interface. However, this modification would limit the number of GPPs utilizing the fabric to 1. This alteration would imply modifying Graph2Hex, as to generate Code Segments that contained FSL writing and reading instructions and would require implementing a protocol based communication between the GPP and the RF, as the FSL is a point to point connection, that is, memory mapped registers would no longer exist (which would also require that all routing information be kept within the RF itself).

As for memory support, the memory operations within graphs would adapt better to the current RF if they could be transformed in a manner that allowed the removal redundant or useless stores and loads or the relocation of these operations to the periphery of the computations (i.e. only at the start and end).

7.2 LMB Injector

The largest delay in the system however is the PLB bus. To specify, the DDR2 memory in which the program code is held must be fetched by accessing this bus, thus introducing great execution delay in the system. Although necessary for large programs, an external memory might be needless if the program's size is reduced enough for local memories. So, in order to support a system based only on these memories, a few more alterations would be required.

One would be the location of the Code Segments in flash as explained before.

Another would be the introduction of the LMB Injector.The developed Injector was designed for the PLB bus, due to the need of containing benchmarks in external memory. However, local memories such as BRAMs are more appropriate for storing programs of reduced size. So, the only alteration required in order to adapt the system is the alteration of the Injector in order to allow

for it to behave as a LMB (Local Memory Bus) passthrough. The LMB is the interface utilized by the MicroBlaze processor to access local memories. Adding to that, the Microblaze only allows for caches in a system with external memories, as BRAMs are themselves fast enough to compete with cache access (caches are in fact implemented in BRAMs). So, without even adapting the system for cache support considerable speedups could be attained.

Still, a tighter coupling between the Injector and the GPP might facilitate the development of a system such as this while also permitting caches. The Injector would instead be placed between the GPP and the cache memories (which are, in turn, connected to any other memories).

7.3 Other Aspects

The following are minor hypothetical modifications to the system with the aim of expanding its functionality. They were not tested nor analyzed in depth but they aim to demonstrate the flexibility of the system in terms of alterations.

7.3.1 Interconnection Scheme

In order to reduce the resources utilized by the switchboxes, the tools could be adapted to generate a row-by-row description of dedicated switchboxes. These would only provide the connections necessary for their respective row. Though conceptually simple, this step would require modification of the parameter-based description of the RF.

7.3.2 Working with Cache

As stated before, data and instruction caches have been disabled for the GPP. All the presented approaches had not considered cache. Regarding the implemented system, the Injector needs to monitor at which point in execution the GPP is, in order to know whether or not it is about to enter a block of code mapped to hardware. Had cache been used, this information might not pass through this peripheral. However, disabling cache results in a performance reduction. So, a workaround to this is to disable the cache around regions of code that are known, by inspection, to contain the mapped graphs (and that will have to pass through the Injector).

The MicroBlaze soft-core processor libraries contain a small set of functions that allow for this behaviour. Such as:

```
#include "xil_cache.h"

Xil_ICacheInvalidate();
Xil_ICacheEnable();

Xil_DCacheInvalidate();
Xil_DCacheEnable();
```

Listing 7.1: MicroBlaze Cache Enabling/Disabling Functions

However, since the measurements of speedup were only performed for those segments of code, as was explained, enabling cache for other regions would only accelerate execution of program regions that would execute as software only. In absolute terms this would, of course, be desired, but the target measurements in this case were the computational times within the RF.

7.3.3 Working with Pre-Compiled ELFs

Show previously in the toolflow of the system, the starting point was the source code of the program to be optimized. The only reason as to why the flow must start from source code is only so as to assure that the resulting ELF is linked to the proper memory locations for the system under design in XPS.

In truth, the application, after compiled, is placed in flash and copied to DDR2 RAM. Currently it is being copied to a starting memory position so as to coincide with the addresses as seen by the ELF (the addresses the application was linked too). However, absolute memory positions are only relevant if absolute jumps exist within the code.

So, seeing as though ELFs can't be relinked with tools such as Object Copy (*objcpy*), a workaround to working with pre-compiled ELFs is to have the Injector correct the absolute memory positions.

To clarify, pre-compiled ELFs are linked to other memory positions, but once the application is copied to DDR2 to a different starting point, the offset will always be the same. Since the only instructions that need correcting are absolute jumps (so as to not have the execution branch to an undesired memory position) the Injector can easily alter the instructions as to correct for this offset.

Appendix A

Detailed Results

This appendix contains the most relevant data derived from testing the system with the previously mentioned benchmarks. The following sections present one benchmark each. For each one, the code originating the graph that was implemented and the graphical representation of the graph itself are presented. Tables relative to each benchmark detail both the measured execution times (expressed in number of clock cycles) and the values derived from utilizing the approximation formulas as given in section 6.1. Values in rows named as *HW Estimate (total)* are computed by equation 6.3.

Also presented are some characteristics of the resulting fabric, such as FPGA resource requirements and the depth of the fabric (the most relevant parameter as it is related to the duration of one iteration).

The obtained values for the estimated hardware execution time are lower than the actual measured value due to the method of measuring and instructions surrounding the area mapped to hardware. While the timer peripheral is being accessed (to deactivate it), the instructions are still being fetched from external memory, thus adding to the measured time. Likewise, the timer is activated before, or outside, loops or calls of functions that enclose the graph so as to avoid intrusion. In the following tables, the estimated number for these instructions is also computed. The values for hardware and software execution are expressed in clock cycles.

A.1 Even Ones

A.1.1 Even Ones Graph and Source Code

```
int evenOnes(int temp, int Num) {
        int Cnt = 0;
        int i;
        //this loop results in a graph
        //the threshold Num was a constant value of 32 for all runs
        for(i=0; i<Num; i++) {
                Cnt ^= (temp & 1);
                temp >>= 1;
        }
        return Cnt;
}
```

Listing A.1: Even Ones Source Code

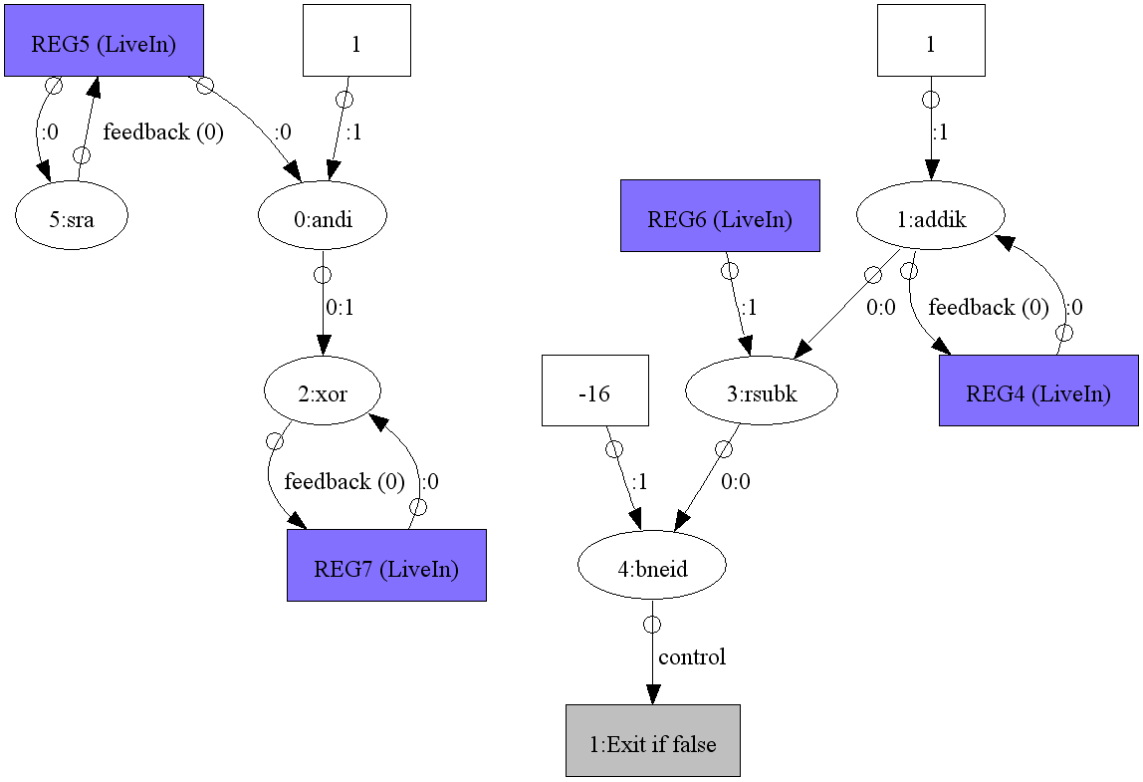


Figure A.1: Tested Graph for Even Ones

A.1.2 Even Ones Result Tables

n	100	200	300	400	500
HW	121575	242136	362690	483243	603793
SW	502760	1004669	1506570	2008477	2510380
Speedup	4,14	4,15	4,15	4,16	4,16

n	100	200	300	400	500
SW (cache)	24087	47891	71695	95494	119283
Speedup	0,2	0,2	0,2	0,2	0,2

n	100	200	300	400	500
HW Estimate (total)	87892	175692	263492	351292	439092
Estimation error	33683	66444	99198	131951	164701
Equivalent Instrs.	14,64	14,44	14,38	14,34	14,32

n	100	200	300	400	500
HW Estimate (fabric)	9600	19200	28800	38400	48000
Potential Speedup	2,51	2,49	2,49	2,49	2,49

Table A.1: Detailed results for Even Ones

Fabric Depth	Max. Freq. (MHz)	Nr. Routing Registers
3	132,83	4

Nr Iterations per call of graph	CS Length
32	34

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	2690
Used	1152	2330	794
%	2,11%	8,54%	29,52%

Table A.2: Fabric Characteristics for Even Ones

A.2 Hamming

A.2.1 Hamming Graph and Source Code

```
int hammingDist(int i1, int i2) {
        int i;
        int result = 0;
        int xor1 = i1 ^ i2;
            sum += (input) & 1;
        //this loop results in a graph
        for(i=0; i<32; i++) {
                result = (xor1 & 1) + result;
                xor1 = xor1 >> 1;
        }
        return result;
}

//which is called "n" times
for(i=0; i<n; i++) {
        result = hammingDist(i, i+1);
        acc += result;
}
```

Listing A.2: Hamming Source Code

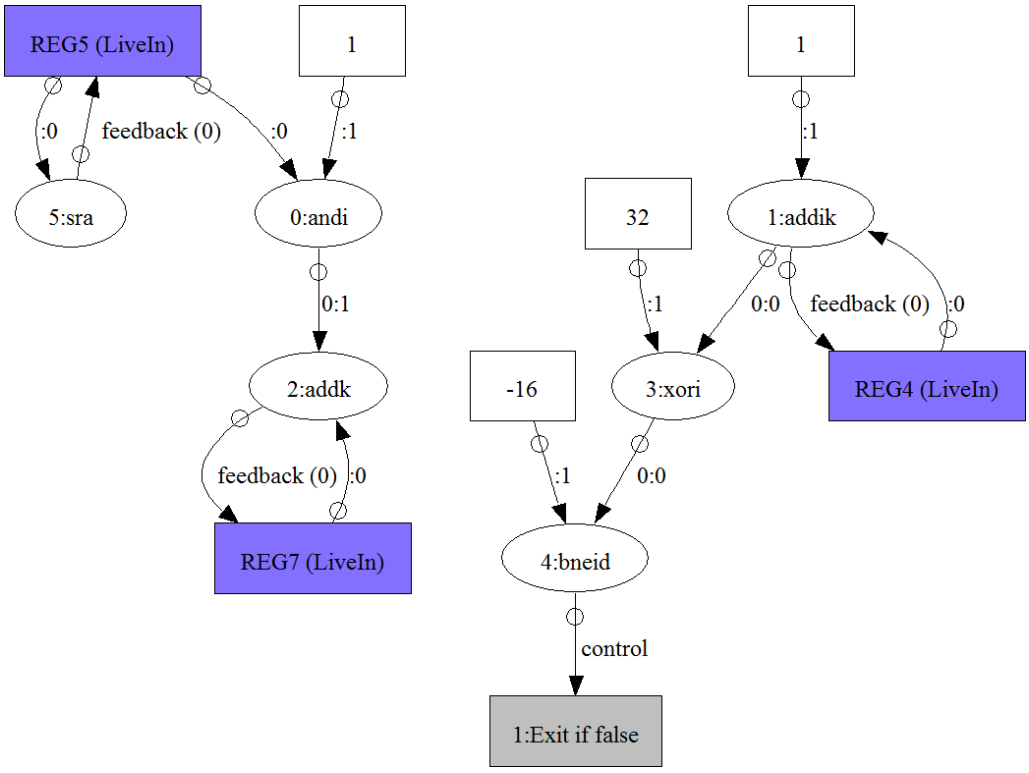


Figure A.2: Tested Graph for Hamming

A.2.2 Hamming Result Tables

n	100	200	300	400	500
HW	117249	233493	349751	465992	582248
SW	502792	1004686	1506598	2008501	2510401
Speedup	4,29	4,3	4,31	4,31	4,31

n	100	200	300	400	500
SW (cache)	24106	47918	71708	95506	119307
Speedup	0,21	0,21	0,21	0,2	0,2

n	100	200	300	400	500
HW Estimate (total)	83292	166492	249692	332892	416092
Estimation error	34049	67093	100151	133912	166248
Equivalent Instrs.	14,76	14,57	14,5	14,47	14,45

n	100	200	300	400	500
HW Estimate (fabric)	9600	19200	28800	38400	48000
Potential Speedup	2,51	2,5	2,49	2,49	2,49

Table A.3: Detailed Results for Hamming

Fabric Depth	Max. Freq. (MHz)	Nr. Routing Registers
3	138,08	4

Nr Iterations per call of graph	CS Length
32	32

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	2022
Used	1086	1738	803
%	1,99%	6,37%	39,71%

Table A.4: Fabric Characteristics for Hamming

A.3 Reverse

A.3.1 Reverse Graph and Source Code

```
int reverse(int Word) {
        int I;
        int WordRev = 0;
        //this loop results in a graph which iterates 32 times,
        //a value of 64 for the threshold was also used
        for(I=0; I<32; I++) {
                WordRev |= (Word & 1);
                WordRev = WordRev << 1;
                Word = Word >> 1;
        }
        return WordRev;
}
//called "n" times (ranges from 100 to 500 were tested)
for(i=0; i<n; i++) {
        acc += reverse(i);
}
```

Listing A.3: Reverse Source Code

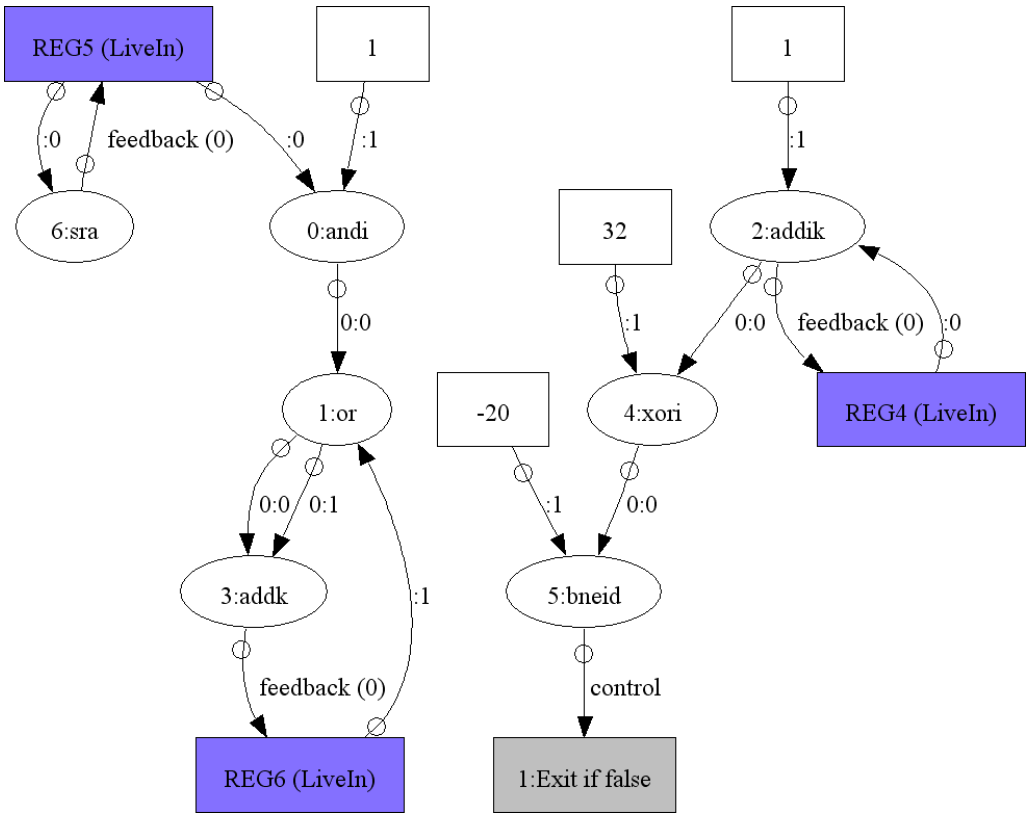


Figure A.3: Tested Graph for Reverse

A.3.2 Reverse Result Tables

N	100	200	300	400	500
HW	125848	250701	375566	500427	625284
SW	566026	1131194	1696364	2261542	2826705
Speedup	4,5	4,51	4,52	4,52	4,52
SW (cache)	27106	53897	80696	107495	134295
Speedup	0,22	0,21	0,21	0,21	0,21
HW Estimate (total)	78296	156496	234600	312800	391000
Estimation error	47552	94205	140966	187627	234284
Equivalent Instrs.	20,67	20,48	20,43	20,39	20,37
HW Estimate (fabric)	9600	19200	28800	38400	48000
Potential Speedup	2,82	2,81	2,8	2,8	2,8

Values for I = 32

N	100	200	300	400	500
HW	136604	272236	407849	543475	679109
SW	1105908	2210922	3315959	4420991	5526024
Speedup	8,1	8,12	8,13	8,13	8,14
SW (cache)	52695	105086	157495	209894	262295
Speedup	0,39	0,39	0,39	0,39	0,39
HW Estimate (total)	78392	156400	234600	312800	391000
Estimation error	58212	115836	173249	230675	288109
Equivalent Instrs.	25,31	25,18	25,11	25,07	25,05
HW Estimate (fabric)	19200	38400	57600	76800	96000
Potential Speedup	2,74	2,74	2,73	2,73	2,73

Values for I = 64

Table A.5: Detailed results for Reverse

Nr. Routing Registers	Max. Freq. (MHz)
4	132,83

I	Fabric Depth	CS Length	Nr iterations per call of graph
32	3	34	32
64	3	34	64

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	2098
Used	1070	1779	754
%	1,96%	6,52%	35,94%

Table A.6: Fabric Characteristics for Reverse

A.4 Fibonacci

A.4.1 Fibonacci Graph and Source Code

For this graph, the number of iterations is dependant on one of the input parameters. So the ratio from overhead to time spent on the RF is variable.

```
int fib(int n) {
        int  i, Fnew, Fold, temp,ans;
        Fnew = 1;  Fold = 0;
        i = 2;
        //a graph with a variable number of iterations per each call
        while( i <= n ) {
                temp = Fnew;
                Fnew = Fnew + Fold;
                Fold = temp;
                i++;
        }
        return Fnew;
}
//called "n" times (ranges from 100 to 500 were tested)
//graph iterates "i" times per call
for(i = 0; i < n; i++)
        acc += fib(i);
```

Listing A.4: Fibonacci Source Code [1]

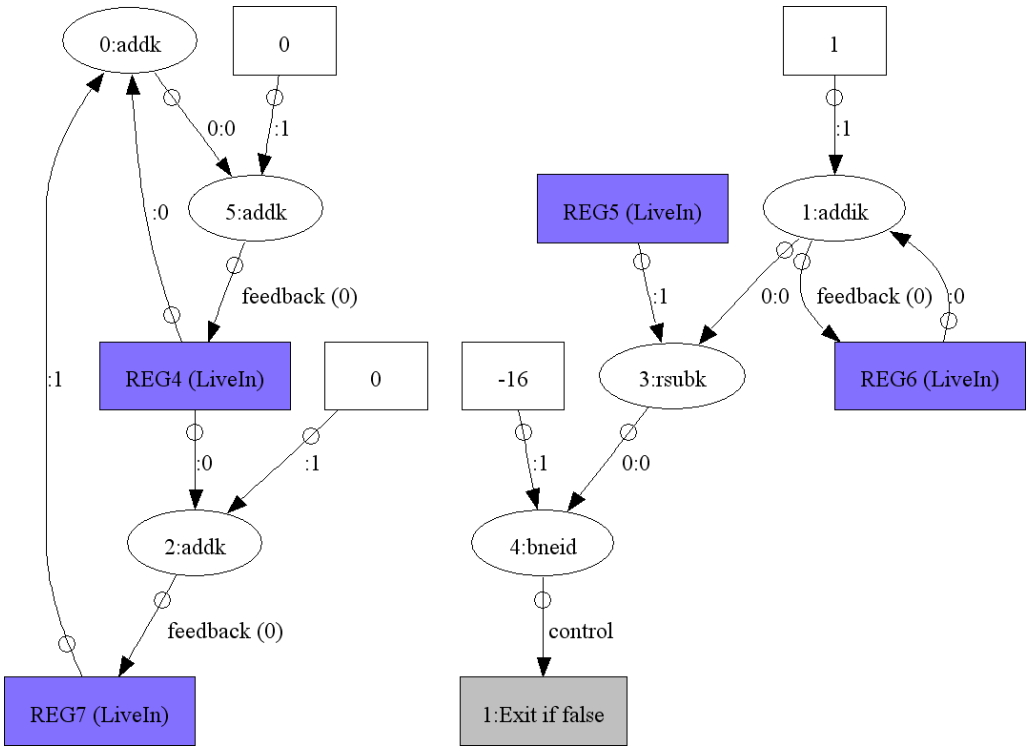


Figure A.4: Tested Graph for Fibonacci

A.4.2 Fibonacci Result Tables

n	100	200	300	400	500
HW	142552	315638	519621	753468	1017340
SW	754830	2984934	6691229	11873707	18532381
Speedup	5,3	9,46	12,88	15,76	18,22

n	100	200	300	400	500
SW (cache)	36058	141809	317564	563308	879066
Speedup	0,25	0,45	0,61	0,75	0,86

n	100	200	300	400	500
HW Estimate (total)	93442	216792	370142	553492	766842
Estimation error	49110	98846	149479	199976	250498
Equivalent Instrs.	21,35	21,49	21,66	21,74	21,78

n	100	200	300	400	500
HW Estimate (fabric)	15150	60300	135450	240600	375750
Potential Speedup	2,38	2,35	2,34	2,34	2,34

Table A.7: Detailed results for Fibonacci

Nr. Routing Registers	Max. Freq. (MHz)
4	121,56

Fabric Depth	CS Length	Nr Iterations per call of graph
3	34	i

n	Nr iterations (total)
100	5050
200	20100
300	45150
400	80200
500	125250

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	2812
Used	1168	2369	727
%	2,14%	8,68%	25,85%

Table A.8: Fabric Characteristics for Fibonacci

A.5 Count

A.5.1 Count Graph and Source Code

```
int count(int Word) {
        int I;
        int NumOnes = 0;
        //the values for the threshold for
        //"I" utilized were 8, 12 and 18
        for(I=0; I<8; I++) {
                NumOnes += (Word >> I) & 1;
        }
        return NumOnes;
}
//called "n" times (ranges from 100 to 500 were tested)
for(i=0; i<n; i++)
        acc += count(i);
```

Listing A.5: Count Source Code

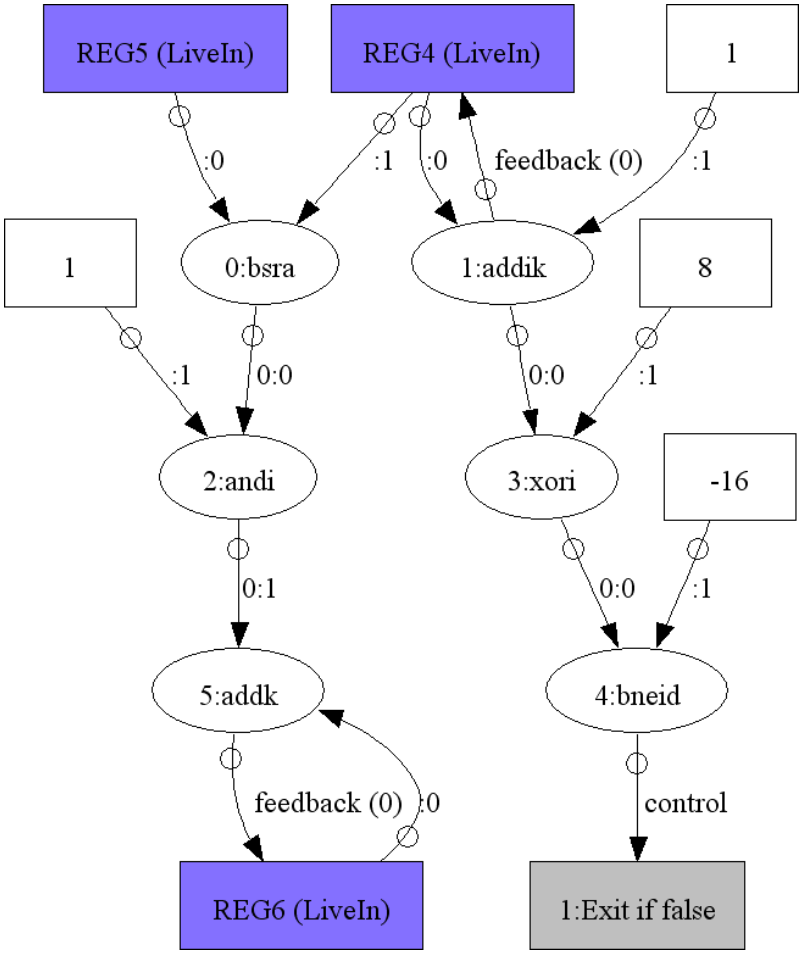


Figure A.5: Tested Graph for Count

A.5.2 Count Result Tables

N	100	200	300	400	500
HW	112930	224867	336816	448761	560707
SW	144267	287649	431058	574461	717873
Speedup	1,28	1,28	1,28	1,28	1,28
SW (cache)	7084	13884	20684	27484	34288
Speedup	0,06	0,06	0,06	0,06	0,06
HW Estimate (total)	76092	152092	228092	304092	380092
Estimation error	36838	72775	108724	144669	180615
Equivalent Instrs.	16,02	15,82	15,76	15,72	15,71
HW Estimate (fabric)	2400	4800	7200	9600	12000
Potential Speedup	2,95	2,89	2,87	2,86	2,86

Values for I = 8

N	100	200	300	400	500
HW	112930	224871	336818	448759	560704
SW	203307	405757	608205	810652	1013112
Speedup	1,8	1,8	1,81	1,81	1,81
SW (cache)	9885	19485	29085	38685	48294
Speedup	0,09	0,09	0,09	0,09	0,09
HW Estimate (total)	77292	154492	231692	308892	386092
Estimation error	35638	70379	105126	139867	174612
Equivalent Instrs.	15,49	15,3	15,24	15,2	15,18
HW Estimate (fabric)	3600	7200	10800	14400	18000
Potential Speedup	2,75	2,71	2,69	2,69	2,68

Values for I = 12

N	100	200	300	400	500
HW	121528	242094	362649	483204	603763
SW	291875	582901	873923	1164937	1455958
Speedup	2,4	2,41	2,41	2,41	2,41
SW (cache)	14085	27885	41689	55489	69285
Speedup	0,12	0,12	0,11	0,11	0,11
HW Estimate (total)	79092	158092	237092	316092	395092
Estimation error	42436	84002	125557	167112	208671
Equivalent Instrs.	18,45	18,26	18,2	18,16	18,15
HW Estimate (fabric)	5400	10800	16200	21600	27000
Potential Speedup	2,61	2,58	2,57	2,57	2,57

Values for I = 18

Table A.9: Detailed Results for Count

Max. Freq. (MHz)
99,3

Fabric Depth	CS Length	Nr. Routing Registers
3	32	4

I	Nr iterations per call of graph
8	8
12	12
18	18

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	1679
Used	926	1433	680
%	1,70%	5,25%	40,50%

Table A.10: Fabric Characteristics for Count

A.6 PopCount

A.6.1 PopCount Graph and Source Code

PopCount contains a parameter that, when altered, creates a differently shaped graphs. This is due to and unrolled loop that appears, and varies in length, according to this parameter. As it can be seen in the benchmark's code, listing A.6, by varying the *bits* parameter, the operations within the inner loop will repeat that number of times. If a small enough number is used the loop is unrolled, and what is detected as a graph is the outer loop. If *bits = 3* this results in the largest graph tested, containing 20 operations. Since this graph is only called once, the number of software instructions that it contains can be easily found in the outputs of the Graph Extractor.

```
        for (i=0; i<N; i++) {
                input = i;
                sum = 0;
                for (j = 0; j < bits; j++) {
                    sum += (input) & 1;
                    input = input/2;
                }
                output += sum;
        }
```

Listing A.6: PopCount Source Code

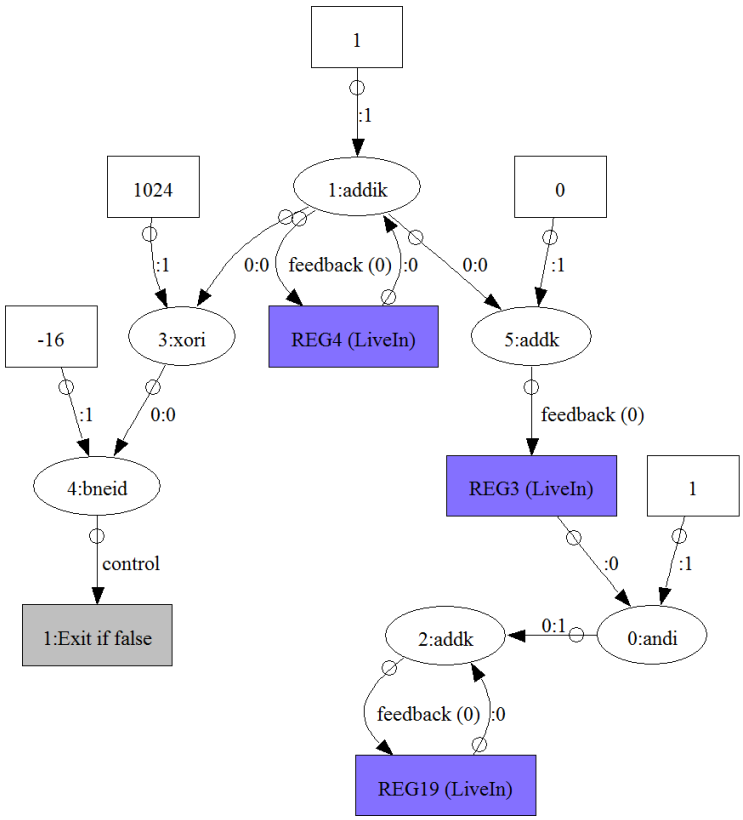


Figure A.6: Tested Graph for PopCount for parameter Bits = 1

A.6.2 PopCount Result Tables

N	128	256	512	1024	2048
HW	2210	2535	3285	4906	7907
SW	19819	38718	76501	152087	303292
Speedup	8,97	15,27	23,29	31	38,36
SW (cache)	1139	2035	3837	7417	14579
Speedup	0,52	0,8	1,17	1,51	1,84
HW Estimate (total)	1189	1573	2341	3877	6949
Estimation error	1021	962	944	1029	958
Equivalent Instrs.	44,39	41,83	41,04	44,74	41,65
HW Estimate (fabric)	384	768	1536	3072	6144
Potential Speedup	2,97	2,65	2,5	2,41	2,37

Values for Bits = 1

N	128	256	512	1024	2048
HW	2728	3483	5091	8105	14246
SW	33314	65705	130494	235056	519196
Speedup	12,21	18,86	25,63	29	36,45
SW (cache)	1797	3333	6405	12550	24847
Speedup	0,66	0,96	1,26	1,55	1,74
HW Estimate (total)	1665	2433	3969	7041	13185
Estimation error	1063	1050	1122	1064	1061
Equivalent Instrs.	46,22	45,65	48,78	46,26	46,13
HW Estimate (fabric)	768	1536	3072	6144	12288
Potential Speedup	2,34	2,17	2,08	2,04	2,02

Values for Bits = 2

N	128	256	512	1024	2048
HW	4963	6095	8486	13103	22337
SW	92697	184477	368024	735137	1469349
Speedup	18,68	30,27	43,37	56,1	65,78
SW (cache)	4627	8979	17693	35091	69914
Speedup	0,93	1,47	2,08	2,68	3,13
HW Estimate (total)	2164	3316	5620	10228	19444
Estimation error	2799	2799	2866	2875	2893
Equivalent Instrs.	121,7	120,83	124,61	125	125,78
HW Estimate (fabric)	1152	2304	4608	9216	18432
Potential Speedup	4,02	3,9	3,84	7,61	15,17

Values for Bits = 3

Table A.11: Detailed Results for PopCount

BITS	Nr. Routing Registers	Max. Freq. (MHz)
1	4	137,26
2	6	132,59
3	11	133,63

BITS	Fabric Depth	CS Length	Nr. SW Instructions (total)
1	3	31	6126
2	6	33	11231
3	9	33	31682

BITS	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
1	900	1356	661 (41,44%)
2	1833	3242	1507 (42,24%)
3	2648	5454	2030 (33,43%)
Total	54576	27288	

Table A.12: Fabric Characteristics for PopCount

A.6.3 PopCount (inner) Graph

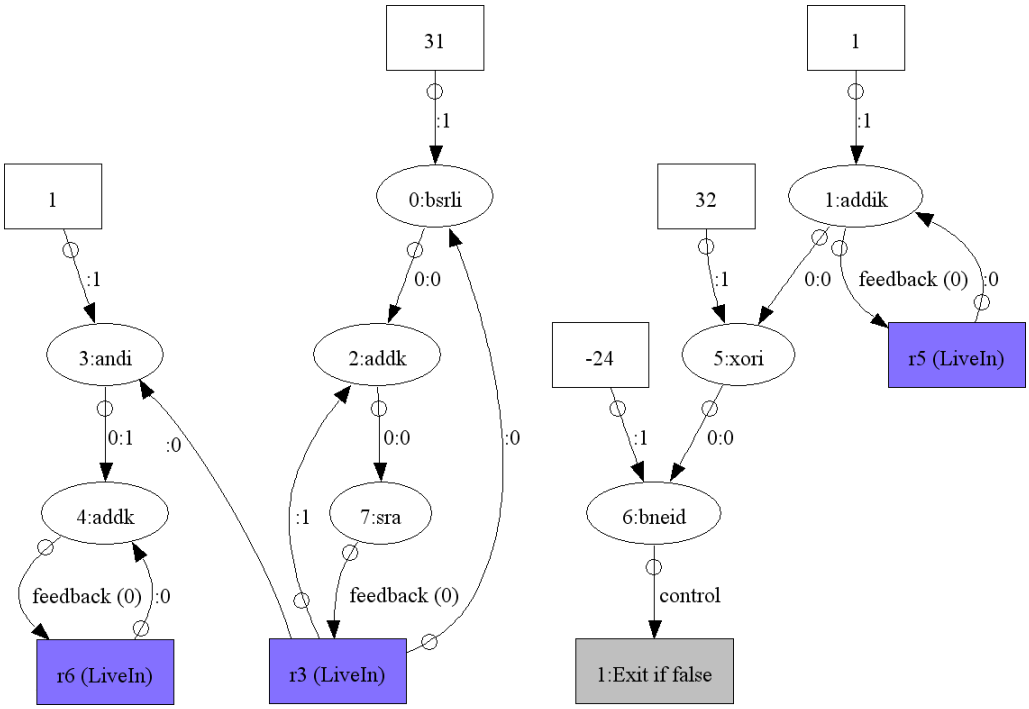


Figure A.7: Tested Graph for PopCount (inner)

A.6.4 PopCount (inner) Result Tables

n	100	200	300	400	500
HW	121590	242158	362713	483261	603818
SW	623027	1245137	1867245	2489356	3111467
Speedup	5,12	5,14	5,15	5,15	5,15

n	100	200	300	400	500
SW (cache)	29775	59275	88775	118275	147777
Speedup	0,24	0,24	0,24	0,24	0,24

n	100	200	300	400	500
HW Estimate (total)	83292	166492	249692	332892	416092
Estimation error	38298	75666	113021	150369	187726
Equivalent Instrs.	16,65	16,45	16,38	16,34	16,32

n	100	200	300	400	500
HW Estimate (fabric)	9600	19200	28800	38400	48000
Potential Speedup	3,1	3,09	3,08	3,08	3,08

Table A.13: Detailed Results for PopCount (inner)

Fabric Depth	Max. Freq. (MHz)	Nr. Routing Registers
3	137,97	4

Nr Iterations per call of graph	CS Length
32	34

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	2071
Used	1059	1757	745
%	1,94%	6,44%	35,97%

Table A.14: Fabric Characteristics for PopCount (inner)

A.7 Merge

A.7.1 Merge Source Code

This benchmarks contains calls to the 6 functions from which the previous graphs are extracted. The implemented graph for *PopCount* was the one derived from its inner loop, and all graphs iterate 32 times save for *Fibonacci*, which iterates a variable number of times, and *Count*, which iterates 8 times. The effect o *Fibonacci's* variable iteration count can be seen in the increase of speedup with the number of calls.

```
        for (i=0; i<N; i++){
                acc += evenOnes(i,32) + count(i);
                acc += reverse(i) + fib(i);
                acc += hammingDist(i, i+1) + popcount(i);
        }
```

Listing A.7: Merge Source Code

A.7.2 Merge Result Tables

n	100	200	300	400	500
HW	882352	1798852	2740767	3715146	4716849
SW	3068240	7611737	13631430	21127314	30099397
Speedup	3,48	4,23	4,97	5,69	6,38

n	100	200	300	400	500
SW (cache)	146044	361494	646952	1002401	1427844
Speedup	0,17	0,2	0,24	0,27	0,3

n	100	200	300	400	500
HW Estimate (total)	509050	1048100	1617150	2216200	2845250
Estimation error	373302	750752	1123617	1498946	1871599
Equivalent Instrs.	162,31	163,21	162,84	162,93	162,75

n	100	200	300	400	500
HW Estimate (fabric)	55950	414900	257850	403800	579750
Potential Speedup	2,61	2,55	2,51	2,48	2,46

Table A.15: Detailed Results for Merge

Max. Freq. (MHz)
85,19

Fabric Depth	CS Length (average)	Nr. Routing Registers
3	33	8

I	Nr iterations per call of graph (average)
100	31,08
200	39,42
300	47,75
400	56,08
500	64,42

	Slice Registers	Slice LUTs	Fully used LUT-FF pairs
Total	54576	27288	6958
Used	1719	6325	1086
%	3,15%	23,18%	15,61%

Table A.16: Fabric Characteristics for Merge

References

[1] SNU Real-Time benchmarks. http://www.cprover.org/goto-cc/examples/snu.html. accessed on 25th June 2011.

[2] Hybrid-core computing: Punching through the power/performance wall. http://www.scientificcomputing.com/articles-HPC-Hybrid-core-Computing-Punching-through-the-power-performance-wall-112009.aspx. accessed on 25th June 2011.

[3] G. Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+ v) structure computer. *Annals of the History of Computing, IEEE*, 24(4):3–9, 2002.

[4] Scott Hauck. The roles of FPGAs in reprogrammable systems. In *Proceedings of the IEEE*, pages 615–638, 1998.

[5] W. Augustin, V. Heuveline, and J.-P. Weiss. Convey HC-1 – The Potential of FPGAs in Numerical Simulation. EMCL Preprint Series, 2010.

[6] J. D Bakos. High-Performance heterogeneous computing with the convey HC-1. *Computing in Science & Engineering*, 12(6):80–87, 2010.

[7] Pan Yu and Tulika Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 723–728, New York, NY, USA, 2004. ACM.

[8] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 12–. IEEE Computer Society, 1997.

[9] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 87–. IEEE Computer Society, 1997.

[10] Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE '08*, pages 1208–1213, Munich, Germany, 2008.

[11] Michael J. Wirthlin and Brad L. Hutchings. A dynamic instruction set computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, 1995.

[12] Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. In *Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '08, pages 209–220, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Warp processing. http://www.cs.ucr.edu/~vahid/warp/. accessed on 25th June 2011.

[14] Roman Lysecky and Frank Vahid. A configurable logic architecture for dynamic hardware/-software partitioning. In *Proceedings of the Conference on Design, automation and test in Europe*, DATE '04, pages 480–485. IEEE Computer Society, 2004.

[15] Antonio Carlos Schneider Beck Fl. and Luigi Carro. *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques: Automatic Acceleration of Software Execution*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[16] F. Vahid, G. Stitt, and R. Lysecky. Warp processing: Dynamic translation of binaries to fpga circuits. *Computer*, 41(7):40–46, July 2008.

[17] A. Gordon-Ross and F. Vahid. Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Transactions on Computers*, page 1203–1215, 2005.

[18] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 18–23, 2005.

[19] R. Lysecky, F. Vahid, and S. X.D Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proceedings of the 41st annual Design Automation Conference*, page 954–959, 2004.

[20] Greg Stitt, Roman Lysecky, and Frank Vahid. Dynamic hardware/software partitioning: a first approach. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 250–255, New York, NY, USA, 2003. ACM.

[21] V. Betz and J. Rose. VPR: a new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, page 213–222, 1997.

[22] G. Stitt and F. Vahid. Thread warping: a framework for dynamic synthesis of thread accelerators. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, page 93–98, 2007.

[23] H. Noori, K. Murakami, and K. Inoue. A general overview of an adaptive dynamic extensible processor. In *Workshop on Introspective Architectures*, 2006.

[24] H. Noori, F. Mehdipou, K. Murakami, K. Inoue, and M. SahebZamani. A reconfigurable functional unit for an adaptive dynamic extensible processor. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, page 1–4, 2007.

[25] H. Noori, F. Mehdipour, K. Inoue, K. Murakami, and M. Goudarzi. Custom instructions with multiple exits: Generation and execution. *IPSJ SIG Technical Reports*, 2007(4):109–114, 2007.

[26] Hamid Noori, Farhad Mehdipour, Kazuaki Murakami, Koji Inoue, and Morteza Saheb Zamani. An architecture framework for an adaptive extensible processor. *J. Supercomput.*, 45:313–340, September 2008.

[27] F. Mehdipour, H. Noori, M. Zamani, K. Murakami, M. Sedighi, and K. Inoue. An integrated temporal partitioning and mapping framework for handling custom instructions on a reconfigurable functional unit. *Advances in Computer Systems Architecture*, page 219–230, 2006.

[28] Arash Mehdizadeh, Behnam Ghavami, Morteza Saheb Zamani, Hossein Pedram, and Farhad Mehdipour. An efficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor. In *VLSI-SoC'07*, pages 151–156, 2007.

[29] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 30–40. IEEE Computer Society, 2004.

[30] Sanjay J. Patel and Steven S. Lumetta. replay: A hardware framework for dynamic optimization. *IEEE Trans. Comput.*, 50:590–608, June 2001.

[31] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 225–235, New York, NY, USA, 2000. ACM.

[32] H. P Rosinger. Connecting customized IP to the MicroBlaze soft processor using the fast simplex link (FSL) channel. *Xilinx Application Note*, 2004.

[33] João Bispo. Megablock tool suite - graph extractor v0.17, May 2011.

[34] João Bispo and João M. P. Cardoso. On identifying and optimizing instruction sequences for dynamic compilation. In *International Conference on Field-Programmable Technology (FPT'10)*, pages 437–440.

[35] João Bispo and João M. P. Cardoso. On identifying segments of traces for dynamic compilation. In *International Conference on Field Programmable Logic and Applications (FPL'10)*, pages 263–266.