UNIVERSITY OF CALIFORNIA,
IRVINE

No-Instruction-Set-Computer (NISC) Technology Modeling and Compilation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Mohammad Reshadi

Dissertation Committee:
Professor Daniel Gajski, Chair
Professor Rainer Doemer
Professor Fadi Kurdahi
Professor Alex Nicolau

2007

The dissertation of Mohammad Reshadi
is approved and is acceptable in quality
and form for publication on microfilm:

_____

_____

_____

_____
Committee Chair

University of California, Irvine
2007

To my wife, Bita, and my parents.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Daniel Gajski for his excellent guidance, support, and patience with me. He never hesitated to share his knowledge and experience with me and always created an open environment for discussion and brainstorming. He gave me freedom to explore on my own and test my ideas and always put up with my mistakes. He has been more than an academic advisor to me for which I am forever grateful.

I would also like to thank Prof. Rainer Doemer for his support and fruitful discussions as well as serving on my committee. I also like to thank Prof. Fadi Kurdahi and Alex Nicolau for serving on my committee and their insightful comments for improving this work.

I am very grateful to other NISC team members especially Bita Gorjiara and Jelena Trajkovic for their helps and contributions, for finding the bugs in the compiler, and for patiently helping me to resolve any issues. I also very much appreciate all the effort and help from Pramod Chandraiah, Dr. Samar Abdi, and Han-su Cho for developing many of the design examples used in this thesis. I like to also thank Karthik Manivannan for his helps with the GCC front end.

I like to thank Dr. Andreas Gerstlauer and Gunar Schirner for not giving up in our long discussions that most often ended on a high point and new ideas to think about. I am grateful to all my lab mates in the ESMG group for their friendship as well as their encouraging and constructive critiques especially in seminars and presentation dry runs. I also like to thank Prof. Nikil Dutt, Prof. Prabhat Mishra, and Dr. Sumit Gupta from whom, in addition to technical matters, I learned a lot about effective presentation and publication.

My pursuit of PhD has become more fun and effective thanks to the members of Center for Embedded Computer Systems (CECS) who were always there to lend a hand (or an ear!). I am also thankful to the CECS and ICS staff specially Grace Wu, Melanie Kilian, Melanie Sanders, and Kris Bolcer for all their help and support.

Last but not least, I am deeply grateful to my parents and my wife, Bita. My parents made many sacrifices to ensure I get the best possible education and lovingly guided me every step of the way. My wife is the most important reason in the success of my PhD. She has provided me with unconditional love, support, and understanding. I have been very lucky, beyond anything I could have wished for, to have her by my side and also work with her in the same team. It would be impossible for me to express my gratitude towards her and my parents in mere words. I dedicate this thesis to them.

Curriculum Vitae

Mohammad[1] Reshadi

Education

2007 **Ph.D.** in Computer Science, University of California Irvine, USA
2000 **M.S.** in Computer Science, University of Tehran, IRAN
1997 **B.S.** in Computer Engineering, Sharif University of Technology, IRAN

Selected Publications

1. B. Gorjiara, M. Reshadi, D. Gajski, "Chapter 10: Low-Power Design with NISC Technology", J. Henkel, S. Parameswaran, Designing Embedded Processors: A Low Power Perspective, Springer, ISBN: 978-1-4020-5868-4, April 2007.
2. M. Reshadi, P. Mishra, N. Dutt, "Hybrid Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation", ACM Transactions on Embedded Computing Systems (TECS), 2007.
3. M. Reshadi, B. Gorjiara, N. Dutt, "Generic Processor Modeling for Automatically Generating Very Fast Cycle-Accurate Simulators", IEEE Transactions on Computer Aided Design (TCAD), Volume 25, Issue 12, pages 2904-2918, December 2006.
4. M. Reshadi, P. Mishra, N. Dutt, "A Retargetable Framework for Instruction-Set Architecture Simulation", ACM Transactions on Embedded Computing Systems (TECS), Volume 5, Issue 2, pages 431-452, May 2006.
1. M. Reshadi, D. Gajski, "Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-scheduled Horizontally-microcoded Architectures in Embedded Systems", Design Automation and Test in Europe (DATE), April 2007.
2. B. Gorjiara, M. Reshadi, P. Chandraiah, D. Gajski, "Generic Netlist Representation for System and PE Level Design Exploration", International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), October 2006.
3. B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined Datapaths", International Conference on Computer Design (ICCD), October 2006.
4. J. Trajkovic, M. Reshadi, B. Gorjiara, D. Gajski, "A Graph Based Algorithm for Data Path Optimization in Custom Processors",9th Euromicro Conference on Digital System Design, September 2006.

[1] Most people know me with my nick name: Mehrdad Reshadi

5. B. Gorjiara, M. Reshadi, D. Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow", Asia and South Pacific Design Automation Conference (ASP-DAC), Design Contest, 2006.
6. M. Reshadi, B. Gorjiara, D. Gajski, "Utilizing Horizontal and Vertical Parallelism with No-Instruction-Set Compiler for Custom Datapaths ", International Conference on Computer Design (ICCD), October 2005.
7. M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths ", International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), September 2005.
8. M. Reshadi, P. Mishra, "Memory Access Optimizations in Instruction-Set Simulators ", International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), September 2005.
9. M. Reshadi, N. Dutt, "Generic Pipelined Processor Modeling and High Performance Cycle-Accurate Simulator Generation", Design Automation and Test in Europe (DATE), March 2005.
10. B. Gorji-Ara, P. Chou, N. Bagherzadeh, D. Jensen, M. Reshadi, "Fast and Efficient Voltage Scheduling by Evolutionary Slack Distribution", Asia and South Pacific Design Automation Conference (ASP-DAC), January 2004.
11. M. Reshadi, N. Dutt, "Reducing Compilation Time Overhead in Compiled Simulators", International Conference on Computer Design (ICCD), October 2003.
12. M. Reshadi, N. Bansal, P. Mishra, N. Dutt, "An Efficient Retargetable Framework for Instruction-Set Simulation", International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), October 2003. **Best Paper Award**.
13. M. Reshadi, P. Mishra, N. Dutt, "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation", Design Automation Conference (DAC), June 2003.
14. S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R.K. Gupta, A. Nicolau, "Dynamic Common Sub-Expression Elimination during Scheduling in High-Level Synthesis", International Symposium on System Synthesis (ISSS), October 2002.
15. D. Rahmati, A. Salimi, M. Reshadi, Z. Navabi, "Handling Complex VHDL Semantics with an OO Intermediate Format", IEEE Canadian Conference on Electrical & Computer Engineering (CCECE), May 2001.
16. B. Gorjiara, M. Reshadi, M. Fakhraie, "GeReDiF: Using XML as a Structured Data Format in Grid Applications", IEEE International Symposium on Cluster Computing and the Grid (CCGrid), May 2001.
17. M. Reshadi, B. Gorji-Ara, Z. Navabi, "Portability and Security, All in CHIRE File System", Hardware Description Language Conference (HDLCon), February 2001.
18. M. Reshadi, A. M. Gharehbaghi, Z. Navabi, "AIRE/CE: A Revision Towards CAD Tool Integration", International Conference on Microelectronics (ICM), November 2000.
19. M. Reshadi, B. Gorji-Ara, Z.Navabi, "HDML: Compiled VHDL in XML", VHDL International Users Forum (VIUF), October 2000.
20. M. Reshadi, A. M.Gharehbaghi, Z. Navabi, "Intermediate Format Standardization: Ambiguities, Deficiencies, Portability issues, Documentation and Improvements", Hardware Description Language Conference (HDLCon), March 2000.

Abstract of the Dissertation

No-Instruction-Set-Computer (NISC) Technology Modeling and Compilation

by
Mohammad Reshadi
Doctor of Philosophy in Information and Computer Science
University of California Irvine, 2007
Professor Daniel Gajski, Chair

Due to the productivity gain of using software in the design of embedded systems, processors are increasingly used in these systems. Embedded processors often run only one or a few applications in the life-time of the system. Therefore, they can be customized for the target applications and significantly improve the quality of the embedded system in terms of cost or other constraints such as performance, and power consumption. Instruction-based architectures limit the customizations because: (a) hardware designer is limited by instruction coding, size and complexity of the decoder; (b) compilers can support certain class of instructions and hence instructions cannot be very complex; and (c) manually updating compilers to incorporate the custom

instructions is not practical and developing compilers that automatically utilize hardware customizations through new custom instructions is very complex.

On the other hand using technologies such as High Level Synthesis (HLS) is not always possible because the traditional HLS techniques can only support relatively small applications. Also they do not give enough control to the designer over the quality of results. Additionally, the main interdependent subtasks of HLS, i.e. resource allocation, operation scheduling, and resource binding, are already too complex themselves and hence adding new constrains such as design for manufacturability to them is not practical.

In this thesis we present a new design approach called NISC (No-Instruction-Set-Computer) Technology. In NISC, the datapath and controller are generated in two different phases. First the datapath is generated or selected from a database based on the application behavior. At the core of NISC technology, there is a *cycle-accurate compiler* that maps a given application directly on a given datapath and generates the control words (CWs) that control the datapath resources in every clock cycle. The NISC architecture style is similar to the old nanocode machines. However, instead of using nanocodes inside the process for implementing the microcodes and in turn instructions, in NISC the nanocode (CWs) are directly used to program the datapath.

NISC simplifies customization and allows designer to fully control design quality. NISC simplifies ASIP (Application-Specific-Instruction-Processor) approach by removing the complex task of finding and designing "most profitable" custom instructions. In NISC only the datapath needs to be specified and NISC compiler generates code *as if* each basic block of the program is executed with one custom instruction. On the other hand, NISC improves resource constrained HLS techniques by

adding the connectivity constraints, on top of the traditional resource constraints, into synthesis process. This enables the designer to control every thing in datapath including wires, which are becoming increasingly more critical in newer technologies.

To realize the NISC Technology design flow, several challenging categories of problems must be solved. Mainly we need:

1. Techniques for efficiently designing and customizing a datapath for an application
2. Techniques for efficiently compiling any application on any given datapath
3. Techniques for efficiently synthesizing a controller from the output of compiler and then generating synthesizable code for different target implementations.

In this thesis we focus on the compilation problems to enable practical use of NISC IPs in a system. We mainly address: modeling of datapath for compilation, scheduling algorithm for compilation, interrupt support in the statically-scheduled pipelined NISC components, and low-level programming in C language in the absence of assembly. Finally we show how different communication interfaces and protocols can be added and used in a NISC component. At the end, we present results that show efficient and fast compiler as well as significant quality improvements for presented experiments. A working compiler incorporating all of the solutions in this thesis, along with the experiments and other NISC toolsets is available for public use from NISC website http://www.cecs.uci.edu/~nisc/. An online version of the tools can be also directly accessed at this website.

Chapter 1. Introduction

Rising const and complexity of systems on one hand, and more constrained requirements on the other hand, demand design approaches that not only increase the productivity of designers but also provide better ways of controlling and improving the quality of final results. An obvious way of increasing productivity is raising the abstraction level. As a result, designers are increasingly looking into implementing their algorithms from a high level description (such as C or other high level languages) rather than describing them directly at RTL (Register Transfer Level). When using a high level description of an algorithm, one of the obvious implementation choices is to use a processor and compile the algorithm to the instruction-set of that processor. This provides tremendous productivity but does not allow the designer to optimize quality metrics (such as performance, area, power …) the way it is possible with a RTL design. On the other hand, manual RTL implementation can improve quality but at the cost of increased design time and hence reduced productivity.

Ideally, the designer must be able to describe all blocks of the design in C, for example, and be able to achieve the desired quality and meet the required constraints. Two different technologies, High Level Synthesis (HLS) and Application Specific Instruction-set Processors (ASIP), have tried to achieve this goal (Figure 1.1). HLS tries

to improve the productivity of RTL design by directly converting high level C description in to an RTL Hardware Description Language (HDL). However, HLS techniques offer a one-direction path from application (C code) to implementation (RTL). They also typically can only support a sub-set of language features and can handle small application sizes. Additionally, the designer cannot directly correlate the effect of application modifications to final implementation quality metrics such as area, power, clock frequency, routable layout …. Therefore, the designer can only rely on try and error and guess work for improving the quality. Because of this major drawback, result quality of HLS tools is significantly lower than manual RTL. On the other hand in ASIP, a base-processor is "extended" to support application-specific custom instructions. Finding proper custom instructions is a very challenging and time-consuming task. Additionally, since the base-processor is always extended, the designer must always pay for all of the resources of the base-processor even if the application does not use all of them. As a result, today still designers have to choose between high productivity and high quality but cannot achieve both simultaneously.

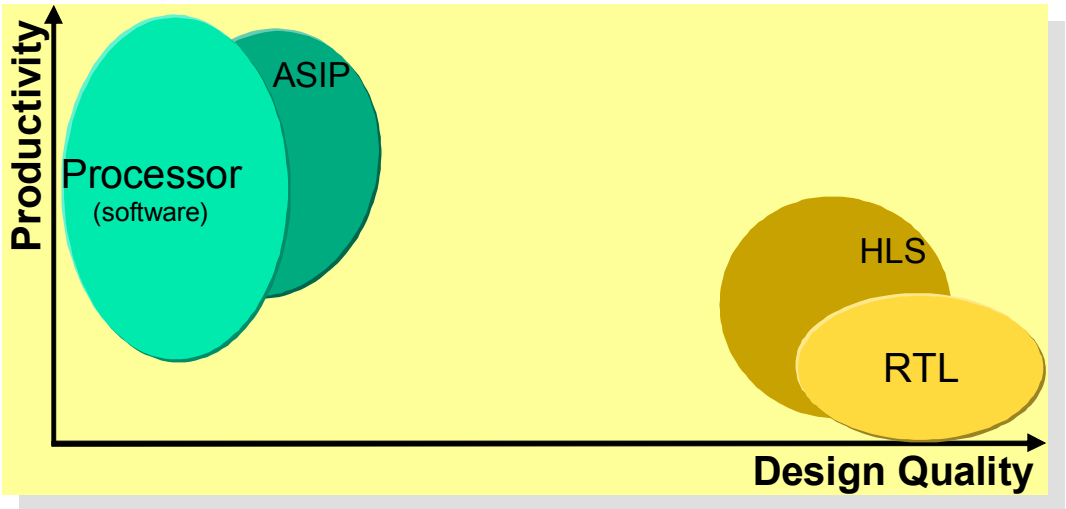


Figure 1.1. Designer Productivity vs. Design Quality

Another problem with ASIP and HLS is the discontinuity in the spectrum of design complexities that the two technologies can be applied to. In other words, if the designer has a mid-complexity design that can be implemented by both ASIP and HLS, then to try both technologies, the designer must use two distinct set of tools, skills, and design flows. This practical limitation prevents efficient design space exploration. Typically, the building blocks of an application have different levels of criticality in the overall quality. For example some blocks may consume more power or area than others; or the overall performance of the design may depend on some blocks more than others. The designer would want to spend more time on optimizing more critical blocks (depicted in Figure 1.2). However, with the current technologies, the designer should first decide upfront with blocks are implemented in software and which blocks are implemented in RTL. Then proportionally, the same amount of time should be spent on all RTL blocks, and similar degree of quality loss will appear in software blocks. If in later stages of design, a block must be moved from one domain (e.g. software) to another (e.g. hardware), then all developments efforts on that block will be inapplicable in the new domain.

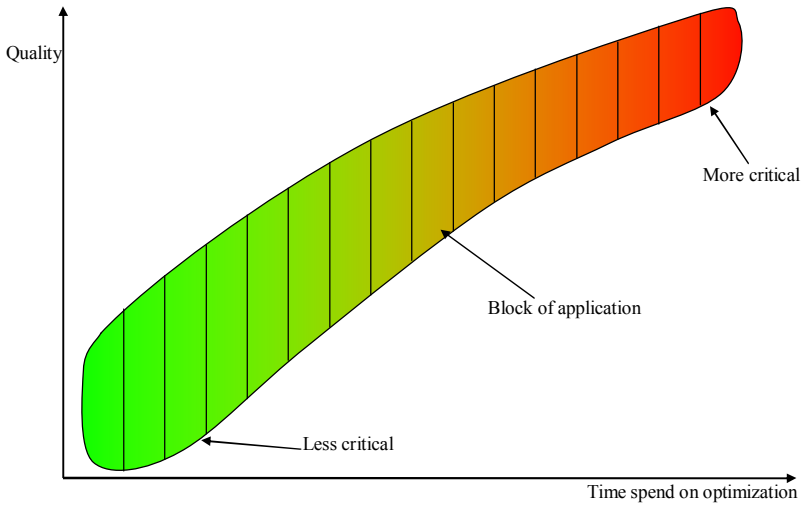


Figure 1.2. Desired proportion between optimization effort and criticality

In this thesis, I present an alternative design approach called No-Instruction-Set-Computer (NISC) Technology to address the above problems. In the rest of this chapter, I first explain the limitations of ASIP and HLS in more details and then present an overview of the NISC technology and the corresponding design approach. Chapter 2 describes the NISC architecture and design flow in more details. Chapter 3 shows how we model NISC and Chapter 4 presents the compilation algorithm. Chapter 5 and Chapter 6 explain how low-level programming and interrupt are supported in NISC while Chapter 7 shows how to use these features handle timed behaviors and add different communication interfaces to NISC. Several experimental results are presented in Chapter 8. Chapter 9 concludes the thesis and presents some future directions for extending the work of this thesis.

1.1 From High Level Synthesis (HLS) to NISC

Traditional High Level Synthesis (HLS) techniques [19][56][26][54][14][15][46] take an abstract behavioral description and generate a register-transfer-level (RTL) datapath and controller. Traditional HLS includes three main tasks: resource allocation, operation scheduling, and resource binding. These tasks are interdependent and different researchers have suggested different heuristics that perform then in different orders. Typically, first operations are scheduled based on some resource constraints, then proper number of functional units and storages are allocated, and finally operations are bound to functional units and variables are bound to storages. Afterwards, the datapath is generated by connecting storages and functional units to ensure that in each cycle the scheduled operation has access to corresponding storages for reading its input operands and writing its result. While most HLS techniques use list-based scheduling [19] and perform

allocation and binding separately, some approach, such as [13] and [44], try to perform scheduling, allocation and binding simultaneously using integer linear programming or branch-and-bound algorithms. Although they may achieve optimal results, complexity restrains the practical applicability of such approaches.

The generated datapath is in form of a netlist and must be converted to layout for the final physical implementation. Lack of access to layout information limits the accuracy and efficacy of design decisions (or optimizations) during synthesis. For example, applying interconnect pipelining technique is not easy during scheduling, because wire information is not available yet. Many researchers ([37][69][20][33][72][22]) have also attempted to incorporate layout information in the synthesis process, especially in scheduling. However, similar to traditional HLS, these approaches generate the datapath after scheduling and therefore they can only predict or estimate layout information during scheduling. However, usually these techniques are inaccurate and are at best only applicable to a specific manufacturing technology. It is also possible to refine the design after generating the layout. In this case, since the physical parameters can be calculated or estimated from the generated layout, the results will be more accurate. However, possible optimizations that use these physical properties have very limited applicability. This is because they are applied to a generated design after scheduling and the optimizations must always maintain the validity of the schedule. For example, applying interconnect pipelining is only possible if the affected states have enough slack time so that schedule can be modified locally while maintaining its validity [34][64][60][17][7][16][29][28]. In other words, aggressive and efficient optimizations are limited after generating the datapath because they might invalidate the schedule.

On the other hand, to give the designer more control over quality of generated results, resource constrained approaches can incorporate limitations on the type and number of resources that are allowed in the design and are given as input to the system. In some cases [15][27] number and type of multiplexers and buses can also be specified. But the designer cannot control input how components are connected as a constraint to these systems. No HLS technique considers interconnects as constrain.

In the newer technologies, wires contribute significantly more so the overall delay, power consumption, area, and complexity of the design. The wire parameters directly depend on the geometry of the layout. The layout information is also needed for Design-For-Manufacturability (DFM). The growing complexity of new manufacturing technologies demands synthesis techniques that support DFM. However, the interdependent scheduling, allocation and binding tasks in HLS are too complex by themselves and adding DFM and accurate layout estimations will add another degree of complexity to the design process. This increasing complexity requires a design flow that provides a practical separation of concerns and supports more aggressive optimizations based on accurate information.

If we consider the progress of resource-constrained HLS, we can see that researchers have always tried to improve the quality of results by increasing the designer's control over how the final datapath should look like. In the past the effect of wires on the quality was negligible compared to that of logic resources (i.e. functional units, storages, multiplexers, and some how busses). In the newer technologies, it is necessary to consider both resource and *connectivity* constraints for efficient and high quality synthesis. But what does connectivity-constrained means? It means in addition to the

number and type of resources, the way these resources are connected to each other is also inputted to the synthesis. In other words, the datapath is specified, and the synthesis tool must map the application (C code) to the given datapath and generate the controller. This brings us to the NISC Technology. Figure 1.3 shows a design flow in which generation of datapath and controller are completely separated. This separation has several positive effects including:

- It enables iterative design and quality improvements through refinement.
- DFM and other layout optimizations can be handled independently.
- Accurate layout information can be used by scheduler and other synthesis phases.

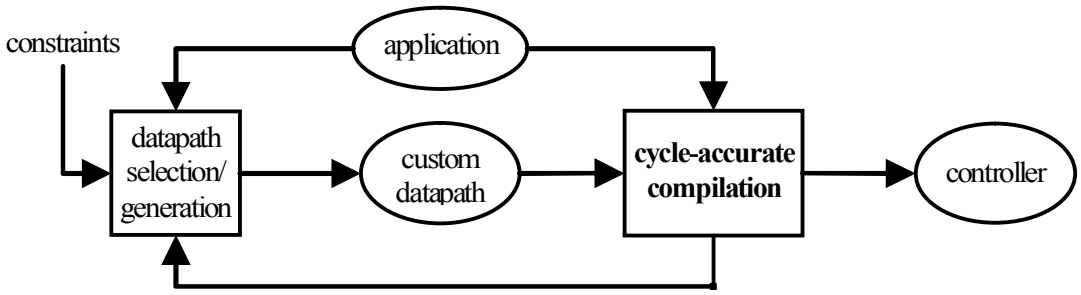


Figure 1.3. Complete separation of datapath and controller in the design flow

In the design flow of Figure 1.3, first the datapath is designed and remains fixed during compilation. Then the controller is generated by mapping (scheduling and binding) the application on the given datapath using a new *cycle-accurate compiler*. This compiler combines HLS, Application Specific Instruction set Processor (ASIP) design, and retargetable compiler techniques.

In some aspects, the proposed design flow is similar to the compilation of applications for processors because in both cases the datapath is fixed during the mapping process. However, traditional compilers rely on instruction-set (or microcode) to abstract out the functionality of processor's datapath and assume that the processor translates such

abstractions to proper control signals. In our approach, the *cycle-accurate compiler* directly maps the application on the given datapath by (1) binding operations, storages, and interconnects, and (2) scheduling the control signal values of datapath components in proper clock cycles. Therefore, it has complete fine-grain control over datapath and can achieve better parallelism and resource utilization. Since we do not use predefined instruction semantics, we call the result architecture No-Instruction-Set-Computer (NISC). In this thesis, I present the solutions to some of the challenging problems involved in realization of such cycle-accurate compiler.

1.2 From ASIP to NISC

Performance of applications can be improved by exploiting their inherent horizontal and vertical parallelism. Horizontal parallelism occurs when multiple independent operations can be executed simultaneously. Vertical parallelism occurs when different stages of a sequence of operations can be overlapped. In processors, horizontal parallelism is utilized by having multiple functional units that run in parallel and vertical parallelism is utilized through pipelining.

Currently, in VLIW processors, the compiler controls the schedule of parallel independent operations (*horizontal control*). However, in all processors, the compiler has no control over the flow of instructions in the pipeline (*vertical control*). Therefore, the vertical parallelism of the program may not be efficiently utilized. In Application Specific Instruction-set Processors (ASIPs) [38], structure of pipeline can be customized for an application through custom instructions.

In ASIPs, functionality and structure of datapath can be customized for an application through custom instructions. At run time, each custom instruction is decoded

and executed by the corresponding custom hardware. Due to practical constrains on size and complexity of instruction decoder and custom hardware, only few custom instructions can be actually implemented in ASIPs. Therefore, only the most frequent or beneficial custom instructions are selected and implemented. Implementing these custom instructions requires: (a) designing custom hardware for each instruction, (b) implementing an efficient instruction decoder, and (c) incorporating the new instructions in the compiler. These steps are complex and usually time consuming tasks that require special expertise. Furthermore, in all processors, no matter how many times an instruction is executed, it always goes through an instruction decoder. The instruction decoder consumes power, area, and complicates the controller as well.

Typically, ASIPs rely on retargetable compilers that automatically incorporate the custom instructions into the compiler by using a processor description captured in an Architecture Description Language (ADL) [52][71]. All retargetable compilers rely on high level instruction abstractions to *indirectly* control the datapath of the processor. They always assume that the processor already has a controller that translates the instructions into proper control signals for the datapath components. In behavioral ADLs, the processor is described in terms the behavior of its instructions. These ADLs are usually very lengthy because they have to capture all possible configurations of instructions. Furthermore, since no structural information is available in the ADL, the quality of automatically generated RTL (if any) for the processor is very low. Structural ADLs try to improve the quality of generated RTL by capturing the controller, instruction decoder and datapath of the processor. Capturing the instruction decoder significantly complicates these ADLs. Additionally, extracting the high level instruction behaviors

from these ADLs for the compiler is very complex and can only be done for limited architectural features.

Getting a fixed architecture model as input is a common assumption in retargetable compilers, mostly used for ASIPs. But usually in these compilers the architecture model is described in terms of instructions, which is a much higher level of abstraction than the structural details of the architecture. UDL/I [25] is an HDL that captures the architecture at the Register-Transfer (RT)-level. A target specific compiler can be generated based on the instruction set extracted from the UDL/I description. However, UDL/I cannot support architecture with any instruction level parallelism. Compilers such as RECORD [58][59] and CHESS [32] use a structural description of architecture but still need to extract the higher level instruction information for using in the compiler. The RECORD compiler extracts behavioral model of instructions from MIMOLA HDL [51][63]. They assume a Horizontal Microcode Architecture (HMA) [66] with single cycle operation. They process the structure of the datapath from destination storages towards source storages to extract valid register transfers (RTs). After analyzing the controller, they reject illegal RTs that do not correspond to an instruction, and use the remaining RTs in the compiler. This approach was suitable for architecture implementation but had two drawbacks: (a) they did not support pipelined datapaths or multi-cycle units, and (b) the designer had to describe the controller explicitly. The CHESS compiler uses the nML language [2] to extract the instruction-set graph (ISG) that captures structural resources in the architecture that are used by each instruction. In both of these approaches, (a) the controller and instruction decoder must be explicitly specified in the input format, and (b) the compiler must analyze the controller to extract the instruction behaviors. Hence, not

only the input descriptions of these approaches are very complex, it is very difficult to extract the instructions as well. As a result, supported architectural features are limited. Similar to MIMOLA, the TIPI (Tiny Instruction-set Processors and Interconnect) [68] targets statically-scheduled HMAs with single-cycle instructions. The main difference is that instead of relying on specification of the controller, the TIPI uses the speciation of non-deterministic atomic actions on architectural state and outputs. While MIMOLA uses binary decision diagrams (BDDs) [57] to extract the valid instructions, in TIPI they extract the instruction-set as a set of *operations* and *conflict table* from the programmability constraint descriptions using Boolean satisfiability (SAT) algorithm. Cycle-accurate simulator and HDL generation from TIPI has been reported, but it does not have a compiler and all programming must be done manually. Retargetable approaches such as LISA [4][48] and EXPRESSION [3][53] use a behavioral instruction description mixed with structural architecture information and mainly focus on code generation and simulation. Absence of implementation details in the input description of these techniques degrades the quality of their recently added HDL generation. Tensilica [70] has a set of extensible processors and uses a proprietary language called TIE to describe the new custom instructions. This language is only for generating the RTL of the custom instruction rather than automatically detecting the usage of the custom instructions in the program by compiler. The programmer should explicitly use the custom instructions in the program in order to utilize them.

Before RISC processors become popular, microcode processors [1][66] were extensively studied for several years. Microcodes are mainly used inside processors for implementing complex instructions or for controlling programmable coprocessors such as

PICO [36][61] and ARM OptimoDE [45], [35]. In many processors, the instruction does not operate directly on the internal resources, and instead is decoded to a sequence of microcodes. In some machines (Motorola 68000, [62]) the microcode instructions are also translated to a sequence of *nanocode* commands, which are in fact the Control Words (CWs) directly controlling the datapath resources in every cycle [30][67]. In such machines, microcode and nanocode programs are manually developed and stored in a ROM in the processor.

Instead of hard-coding the nanocode into the processor, if we expose them for programming the datapath, then we will have the most flexible way of controlling the resources of datapath and also support far more complex datapaths than what HLS techniques can generate. Nanocode exposes all structural details of datapath, therefore manual programming at nanocode level is not practical, and also the compilation techniques must be upgraded to deal with all of the low-level structural details of the datapath. Furthermore, if we develop such a compiler in a way that datapath structure and timing details can be described and used as input to the compiler, then we can have a design flow in which datapath is specified and application is mapped directly on the datapath. This idea leads us to the NISC Technology and the cycle-accurate compiler at the core of it.

The NISC cycle-accurate compiler generates code *as if* each basic block of program is executed with one custom instruction. A basic block is a sequence of operations in a program that are always executed together. Ideally, for each basic block we should have one instruction that reads the inputs of basic block from a storage (e.g. register file) and computes the outputs of basic block and stores them back. The large number of basic

blocks in a typical program prevents us from using an ASIP approach to achieve the above goal. To solve this problem, in NISC instruction decoding is moved from hardware to the compiler. In ASIP, after reading the binary of a custom instruction from memory, it is decoded into a set of control words (CWs) that control the corresponding custom datapath and executes the custom functionality. Instead of having too many custom instructions and then relying on a large instruction decoder to generate CWs in hardware; in NISC the CWs are generated in compiler by directly mapping each basic block onto the custom datapath. Therefore, the compiler can construct unlimited number of custom functionalities utilizing both horizontal and vertical parallelism of the input program. If the datapath is designed to improve the execution of certain portions of program, the NISC compiler will automatically utilize it. Since the compiler is no longer limited by the fixed semantics of instructions, it can fully exploit datapath capabilities and achieve better parallelism and resource utilization.

1.3 NISC Technology

It was explained in the beginning of chapter that there is a gap between designer productivity and design quality that can be achieved with current technologies. There is also a disconnect in the spectrum of design complexities that the ASIP and HLS technologies can be used for. The goal of NISC technology is to fill this gap and provide a technology that is applicable across a wide range of design complexities (Figure 1.4).

The core idea in NISC is to specify the datapath and map the application directly on the datapath to generate the controller. In contrast to ASIP, the complexity of datapath and final design can be reduced and customized to match exactly the requirements of the application. In HLS, designer needs guess work, implicit tricks, or coding styles in the

input source code in order to control non-functional parameters such as area, power, and clock frequency. Such non-functional parameters cannot be explicitly controlled in the high level C code which only captures the functionality of the design. Instead, in NISC, the functionality is captured in standard ANSI C (or potentially other similar un-timed high level languages), and other non-functional parameters are captures and controlled via the datapath description.

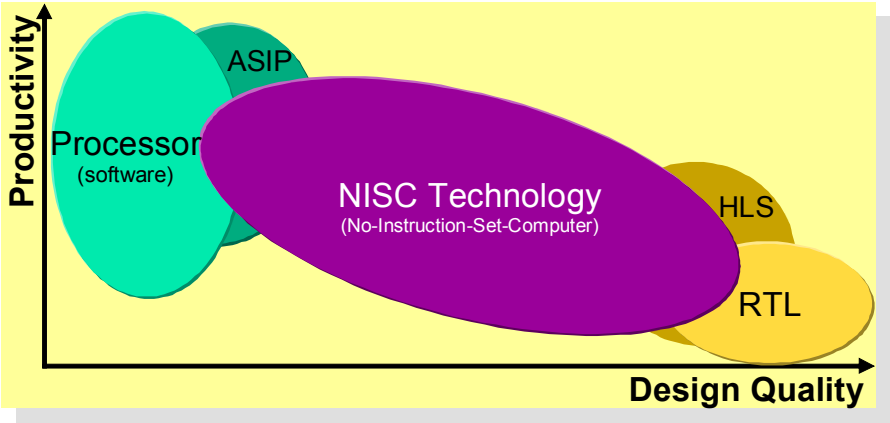


Figure 1.4. NISC: filling the gap between designer productivity vs. design quality

As we mentioned in Section 1.2, NISC architecture style can look like a processor except that instruction decoder is removed and its job is moved from hardware to compiler. Over the past years, the trend of processor design has been to give compiler more control over the processor. This is more evident in transition from CISC (Complex Instruction Set Computer) to RISC (Reduced Instruction Set Computer) and from Superscalar to VLIW (Very Long Instruction Word) processors. While in CISC complex functionalities could be executed with complex instructions; in RISC the compiler uses simpler instructions to execute those complex functionalities in software. Similarly, while in RISC and superscalar machines operations were scheduled in hardware, in VLIW machines schedule of operations is determined statically by compiler in software. Increasing the role of compiler and its control over the processor has several benefits:

- The compiler can look at the entire program and hence has a much larger observation window than what can be achieved in hardware. Therefore, much better analysis can be done in compiler than hardware.
- More complex algorithms (such as instruction scheduling, register renaming) can be implemented in compiler than in hardware. This is because first, the compiler is not limited by the die size and other chip resources; and second, compiler's execution time does not impact the application execution time. In other words, compiler runs at design time, while hardware algorithms run during application execution.
- The more functionality we move from hardware to compiler, the simpler the hardware becomes, and the less the runtime overhead is. This has a direct effect on area and power consumption of the circuit.

In No-Instruction-Set-Computer (NISC) technology, compiler not only constructs functionalities and schedules operations, it is also responsible to decode operations into control words that control the hardware and execute the program. In NISC, the compiler determines both the schedule of parallel independent operations (horizontal parallelism), and the logical flow of sequential operations in the pipeline (vertical parallelism). The compiler generates the control words (CWs) that must be applied to the datapath components at run time in every cycle. In other words, in NISC, all of the major tasks of a typical processor controller (i.e. instruction decoding, dependency analysis, and instruction scheduling) are done by the compiler statically. Since, in NISC, the compiler decides what the datapath should do at every clock cycle, we call it a *cycle-accurate compiler*. The NISC cycle-accurate compiler compiles the application directly to the

datapath. It can achieve better parallelism and resource utilization than conventional instruction-set based compilers.

NISC technology can also help low-power application-specific processor design, because: (a) the compiler-oriented control of the datapath, inherently minimizes the need for runtime hardware-based control, and therefore, reduces the overall power consumption of the design; (b) NISC technology allows datapath customizations to reduce total number of cycles and therefore total energy consumption. The extra slack time can also be used for voltage and frequency scaling, which result in more savings; and (c) NISC does not limit the number of custom functionalities that can be implemented on its datapath because instead of using custom instructions and then relying on the decoder in hardware to generate the control signals, in NISC the compiler generates the control signal values.

Of course, moving functionality from hardware to compiler means that the compiler becomes more complex and new problems and challenges must be solved. In the rest of this thesis, I present the main new challenges in a compiler for NISC Technology and present my solutions.

1.4 Contributions of this thesis

The NISC idea was first introduced as the single, necessary, and sufficient computation component for design of systems-on-chip [18]. In NISC design approach, the datapath is specified and the controller is generated by mapping the application directly on the datapath without using any instruction-set. To realize the NISC idea, several challenging categories of problems must be solved. Mainly we need:

- Techniques for efficiently designing and customizing a datapath for an application
- Techniques for efficiently compiling any application on any given datapath
- Techniques for efficiently synthesizing a controller from the output of compiler and then generating synthesizable code for different target implementations.

In this thesis I mainly focus on clarifying the details of the NISC architecture as well as its compilation problems. The goal of my research was to clearly define what goes into the hardware and what must be done by compiler, and then implement a practical compiler that runs fast enough while supporting the features necessary for designing a NISC component to be used in a system. Accordingly, I have identified and addressed the following problems:

1. I have defined what exactly the execution semantics of NISC architecture is, how to model the architecture, what information must be captured in the model to enable compilation, and how the control bits are organized in the control word.
2. Once we have the model, in addition to the standard compilation techniques, we also new compilation algorithm that can incorporate all low-level structural details of the datapath. I present a scheduling and binding algorithm that supports operation parallelism, pipelined/multi-cycle operations, operation chaining, and heterogeneous pipeline and data forwarding.
3. It is important to enable the designer to access some low-level resources directly from the C code. This is needed for example for accessing specific registers or ports in the NISC datapath, or accessing functional units whose functionality cannot be represented by any of the operators in the C (or other high level) language. To solve this problem in processors, programmers use assembly

instructions. However, NISC does not have instructions and hence cannot have assembly. Therefore, I present an alternative mechanism that allows low-level programming in the C language.

4. A NISC must support interrupt. However, since NISC is a statically-scheduled nanocode machine, it cannot be interrupted arbitrarily. I propose a safe and low-overhead mechanism for implementing the interrupts.
5. To use NISC in a system, it should be able to communicate with other components and communication protocols. For some protocols, NISC must adhere to a cycle-accurate and timed behavior. Since C and other high level languages are un-timed, we need to find a way to support timed behavior without requiring any language extensions. I show how the presented techniques in this thesis are sufficient enough for describing different communication protocols.

As the result of this PhD thesis a working cycle-accurate NISC compiler has been developed. However, to actually use the NISC technology and evaluate its different aspect several other tools in addition to the compiler were necessary. There tools have been developed through collaboration of several people and are now available for download from NISC Website [47]. Furthermore, the development of each experiment and benchmark, from converting the standard C code to an embedded implementation down to implementing the final results in hardware, involved several people. Some of the reported results in this thesis, such as area and clock frequency numbers, are not a direct output of the compiler but are included form other people's work in order to provide more accurate analysis and comparison.

Chapter 2. NISC Architecture

A NISC architecture is composed of a datapath and a controller. The datapath of NISC can be simple or as complex as datapath of a processor. The controller drives the control signals of the datapath components in each clock cycle. These control values are generated by the NISC compiler. These values are either stored in a memory or generated via logic in the controller. Both the controller and the datapath can be pipelined. Figure 2.1 shows a sample NISC architecture with a pipelined datapath that has partial data forwarding, multi-cycle and pipelined units, as well as data memory and register file.

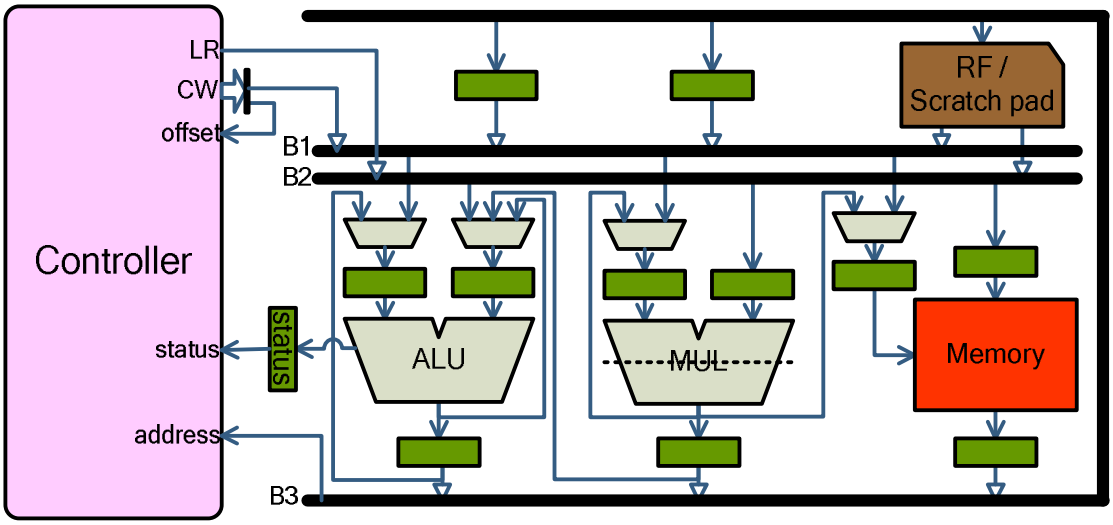


Figure 2.1. A sample NISC architecture

The controller has a fixed template and implements one FSM produced by compiler. The FSM must be as simple as possible to provide maximum flexibility and control over datapath for the compiler. Changes in the datapath are automatically incorporated by the compiler, but changes in the controller may require changing the compiler itself. As we mentioned in Chapter 1, the goal of NISC is to enable customizations to exactly match the architecture to the requirements of the application. Therefore the architecture features must be controlled by adding/removing components to/from datapath, and the controller must be kept as simple as possible without imposing any minimum complexity overhead.

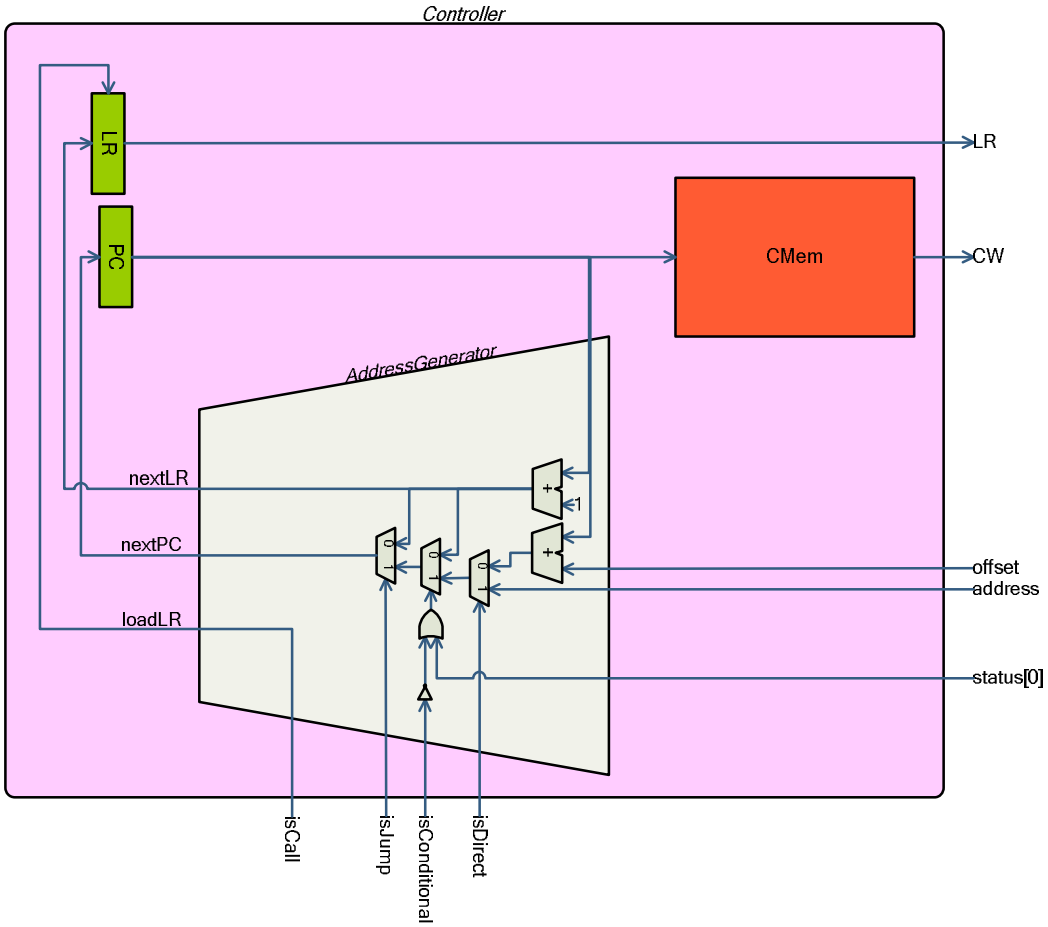
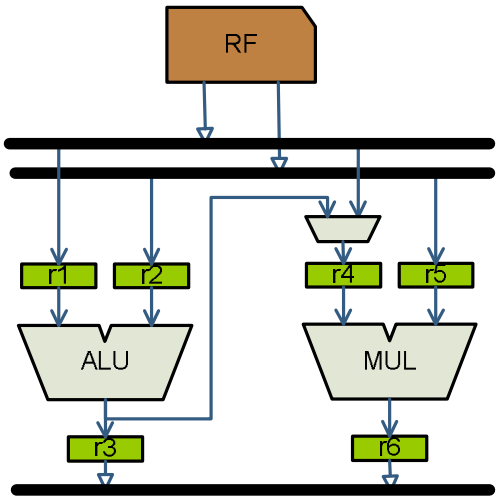


Figure 2.2. A typical NISC controller

For small size programs, the control values can be generated via logic in the controller, for example using flip-flops, etc. For larger applications, or for enabling

reprogrammability, a memory based controller can be used which stores the control words in some sort of memory. Figure 2.2 shows a typical NISC controller which is composed of a Program Counter (PC) register, an Address Generator (AG), a Control Memory (CMem), and a Link Register (LR). In each cycle, a new control word appears on the *CW* port of the controller. A control word contains one or more constant fields that are used as constants in the datapath or as offset for a jump. The controller itself has some control bits that come from CW. For example in Figure 2.2, control bits *isJump, isConditional, isDirect*, and *isCall* come from CW. The LR register stores the return address when executing a function call. In the prolog of every function, the LR value is read and pushed on the stack, and then in the epilog of that function, the return address is popped from stack and passed as a direct jump (i.e. *isJump=1* and *isDirect=1*).



d=(a+b)×c
Assume:
a=RF(1), b=RF(2), c=RF(3), d=RF(4)

Cycle	CW
1	r1=RF(1); r2=RF(2);
2	r4=ALU(+,r1, r2); r5=RF(3);
3	r6=MUL(*,r4, r5);
4	RF(4)=r6;

Figure 2.3. Statically scheduled control words

To execute the program on a given datapath, the corresponding control words are statically scheduled at compile time. Figure 2.3 shows the scheduled control words for executing a simple expression on the given datapath. The actual control words contain control bits that configure the corresponding resources to do the required task, e.g. set the

load of a register to load a new value in a certain cycle; or select a certain operation from a functional unit in a certain cycle.

In presence of controller pipelining (e.g. PC and Status registers in Figure 2.2 and Figure 2.1), the compiler should also make sure that the branch delay is considered correctly and is filled with other independent operations. Detail of the compilation algorithm is presented in Chapter Chapter 4.

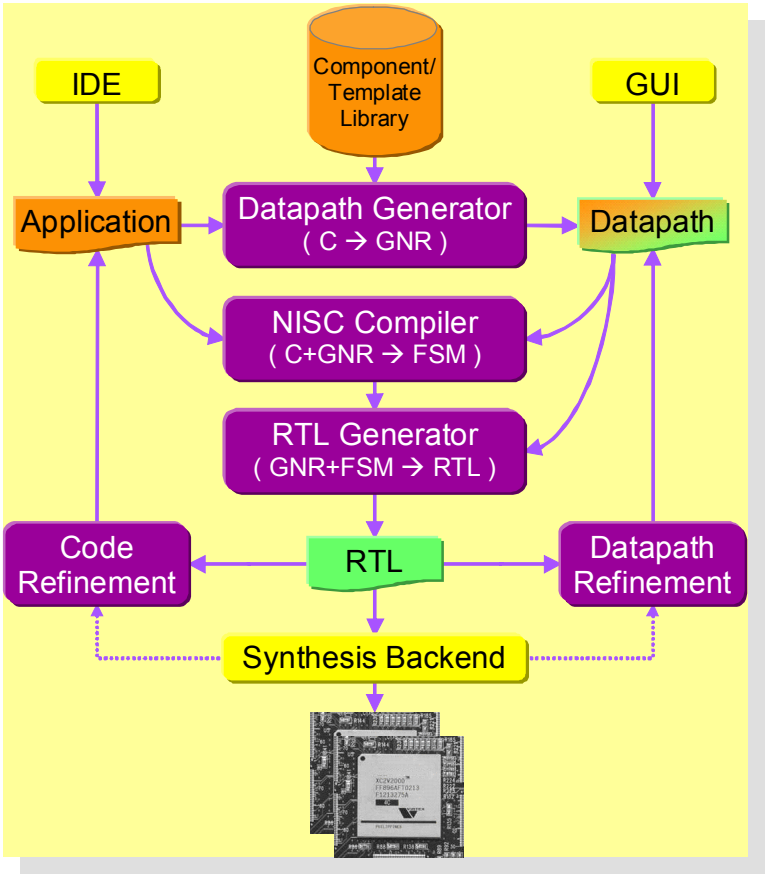


Figure 2.4. NISC design flow

Figure 2.4 shows the design flow for designing a custom NISC for a given application. In NISC, the controller is generated after compiling the application on a given datapath. Therefore both the application and the datapath description are considered input to the NISC cycle-accurate compiler. The datapath can be generated

(allocated) using different techniques. For example, it can be an IP, reused form other designs, or specified by the designer. The datapath can also be generated automatically based on the application behavior. The datapath is captured in the GNR (Generic Netlist Representation) format [12] which describes the datapath as a netlist of components and assigns different attributes to each component. A component in datapath can be a register, register-file, bus, multiplexer, functional unit, memory etc. The functionalities of components are associated with timing information of corresponding control values.

The GNR description of the datapath and the high level description of the application (e.g. C code) are then given as input to the NISC compiler. The NISC compiler, maps the application directly on the given datapath and generates a Finite State Machine (FSM) that determines the behavior of datapath in each clock cycle. The NISC compiler applies a combination of traditional compilation algorithms as well as HLS techniques, specifically resource scheduling and resource binding. At the end, the complier generates the contents of data memory (if any) and also uses the FSM to generate the stream of control values. The RTL generator, first synthesizes a controller from the output of compiler, and then uses the datapath information to generate the final synthesizable RTL design (described in Verilog). This RTL is then used for simulation (validation) and synthesis (implementation). After synthesis and Placement and Routing (PAR), the accurate timing, power, and area information can be extracted form the layout and used for further datapath *refinement*. For example, the user may add functional units and pipeline registers, or change the bit-width of the components and observe the effect of modifications on precision of the computation, number of cycles, clock period, power, and area. In NISC, there is no need to design the instruction-set because the compiler

automatically analyzes the datapath and extracts possible operations and branch delay. Therefore, the designer can refine the design very fast.

As a unique feature of the NISC design flow in Figure 2.4 is that it enables the designer to iteratively refine and improve the results (as depicted in Figure 2.5). In this flow, the designer can start with an initial application description and use an initial datapath for executing the application and generate initial results. Then the designer can iteratively modify the application or the datapath and use the NISC toolset to generate a new set of results. An important benefit of this approach is that in each iteration the designer can focus on one quality metric. For example, the available parallelism in the application can be improved in one iteration, the clock frequency of the datapath can be improved in another iteration separately, and then the area of the datapath can be improved in yet another separate iteration. In this way, multi-optimizations can be applied to the design without one optimization complicating another. At the end, from several design points, the designer can select the one that best meets the design requirements.

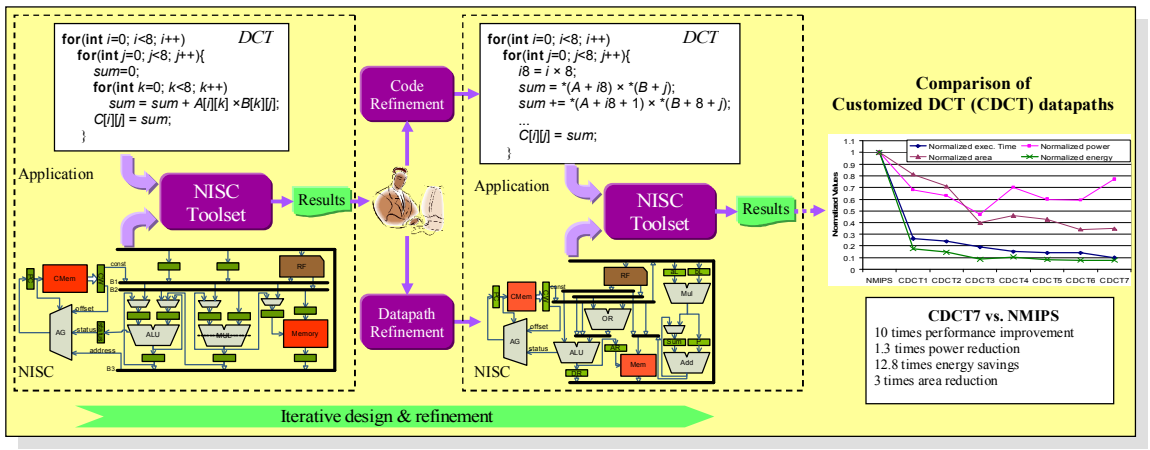


Figure 2.5. Iterative design using NISC Technology

Chapter 3. Modeling NISC architecture for compilation

In NISC, one of the inputs of the compiler is the architecture description. To describe the architecture we need both a modeling approach and a language. The model determines what information about the architecture must be captured and how this information is organized, while the language provides building blocks and composition rules to capture these information in text. Both model and language must cover all of the needs of the NISC toolset (Figure 2.4). Details of such a complete model are outside of the scope of this thesis. A subsystem of multiple NISC components can be described in the GNR (Generic Netlist Representation) [12]. GNR is a multi-aspect description format that is used by all the NISC toolset. GNR is an XML [74] based format and used XML Schema [73] for validation and enforcing the structural rules of GNR. In this chapter I focus on the parts of the model that are needed for compilation (i.e. compilation aspect) and use the GNR syntax to describe the examples[2].

The compiler aspect of the component models must essentially provide operation information of the components as well as their connectivity. In the compiler, we need to

[2] The GNR syntax and its loader is developed by Bita Gorjiara at CECS in UC Irvine.

map the CDFG [6] and variables of the program to the given NISC architecture. The NISC cycle-accurate compiler must schedule operations, bind variables to storages, and bind operations to functional-units. For variable binding, we need the mapping between variable types (e.g. integer, floating point, character, …) and storages that can hold a variable of a particular type. For operation scheduling and binding, we need the mapping between operations and datapath resources that implement a particular operation. The latter mapping, must also provide the timing of the operation on a particular resource as well as the control values (if any) that configure the resource to execute the particular operations. As well as considering these requirements, when developing the model, we should also consider what information must be explicitly captured in the model and what information can be efficiently extracted from the model automatically. During development of the NISC model, the general idea was that we capture the detailed architecture information needed for generating good HDL and then annotate related information that otherwise would be very difficult, complex, time consuming, or even impossible to extract automatically.

In NISC model, each component has a set of input, output, and control ports. The architecture is modeled as a netlist which includes instances of components and connection that connect an output port of one component to an input port of another component. In a hierarchical netlist, a component may also have internal netlist. As such, the connections may also be defined between ports the component and the ports of components in its internal netlist. Each component has a typed associated with it, and this type can be one of: *register*, *register-file*, *multiplexer*, *tri-sate buffer*, *bus*, *functional-unit*, *memory*, *controller*, *module*, and *NiscArchitecture*. *NiscArchitecture* is the top level

component that contains all of the architecture information. It is a special case of a *module*. A *module* is a hierarchical component that can have an internal netlist. A component may provide different *aspects* for different tools. The information of each aspect depends on the component type. The component information required for compilation is stored in it *compiler aspect*.

Note that in an RTL description the above component types are not explicitly differentiated, thus, many synthesis tools must detect component types by relying on the description style, which is difficult to formulate and enforce and whose definition varies from vendor to vendor! This important difference between the NISC model and a structural HDL is the main enabler of the NISC compiler to understand the behavior of the datapath and compile on it.

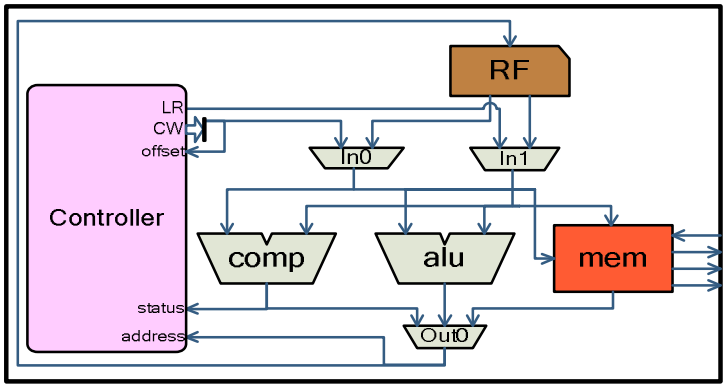


Figure 3.1. Block diagram of a simple NISC architecture

3.1 NiscArchitecture: the top module of design

The *NiscArchitecture* component type is the top module that captures all information about a NISC architecture. The ports of this component are used to connect it to rest of a system. Figure 3.2 shows the GNR description of a simple NISC component shown in Figure 3.1. The datapath of a NISC architecture can have several instances of each component type. A component instance has a unique name and a type name that refers to

a component description in the library. The datapath description also includes clock and control connections (between CW port of the controller and the control ports of components). The GNR parser automatically adds these connections if they are not already specified [12].

```
<CustomIP type="simpleIP">
  <Ports>
    <Clock n="clk" bitWidth="1"/>
    <InPort n="reset" bitWidth="1"/>
    <InPort n="dm_r" bitWidth="32"/>
    <OutPort n="dm_addr" bitWidth="32"/>
    <OutPort n="dm_w" bitWidth="32"/>
    <OutPort n="dm_readEn" bitWidth="1"/>
    <OutPort n="dm_writeEn" bitWidth="1"/>
  </Ports>
  <Netlist>
    <Components>
      <Instance n="controller" type="Controller"/>
      <Instance n="RF" type="RF2x1">
        <SetParam n="BIT_WIDTH" val="32"/>
        <SetParam n="REG_COUNT" val="32"/>
      </Instance>
      <Instance n="In0" type="Mux2"/>
      <Instance n="In1" type="Mux2"/>
      <Instance n="Out0" type="Mux4"/>
      <Instance n="comp" type="Comparator"/>
      <Instance n="alu" type="ALU"/>
      <Instance n="mem" type="DataMemProxy"/>
    </Components>
    <Connections>
      <Conn src="controller" sPort="cw" dest="controller" dPort="offset" extend="signed" s="9" e="0"/>
      <Conn src="controller" sPort="cw" dest="In0" dPort="i0" extend="signed" s="9" e="0"/>
      <Conn src="comp" sPort="o" dest="controller" dPort="status" s="0" e="0"/>
      <Conn src="Out0" sPort="o" dest="controller" dPort="address"/>
      <Conn src="Out0" sPort="o" dest="RF" dPort="w0"/>
      <Conn src="RF" sPort="r0" dest="In0" dPort="i1"/>
      <Conn src="RF" sPort="r1" dest="In1" dPort="i0"/>
      <Conn src="In0" sPort="o" dest="comp" dPort="i0"/>
      <Conn src="In1" sPort="o" dest="comp" dPort="i1"/>
      <Conn src="comp" sPort="o" dest="Out0" dPort="i0"/>
      <Conn src="In0" sPort="o" dest="alu" dPort="i0"/>
      <Conn src="In1" sPort="o" dest="alu" dPort="i1"/>
      <Conn src="alu" sPort="o" dest="Out0" dPort="i1"/>
      <Conn src="In0" sPort="o" dest="mem" dPort="addr"/>
      <Conn src="In1" sPort="o" dest="mem" dPort="w"/>
      <Conn src="mem" sPort="r" dest="Out0" dPort="i2"/>
      <Conn src="" sPort="dm_r" dest="mem" dPort="dm_r"/>
      <Conn src="mem" sPort="dm_addr" dest="" dPort="dm_addr"/>
      <Conn src="mem" sPort="dm_w" dest="" dPort="dm_w"/>
      <Conn src="mem" sPort="dm_readEn" dest="" dPort="dm_readEn"/>
      <Conn src="mem" sPort="dm_writeEn" dest="" dPort="dm_writeEn"/>
      <!--GNR parser automatically adds clock and control connections -->
    </Connections>
  </Netlist>
  <Compiler-aspect defaultDMem="mem" clockPeriod="1" pointerByteSize="4">
    <CwFields n="cwFields">
      <Field n="const0" bitWidth="10"/>
      <CtrlField component="RF" ctrlPort="we"/>
      <CtrlField component="RF" ctrlPort="wa"/>
      <CtrlField component="RF" ctrlPort="ra0"/>
      <CtrlField component="RF" ctrlPort="ra1"/>
      <CtrlField component="alu" ctrlPort="ctrl"/>
      <CtrlField component="comp" ctrlPort="ctrl"/>
      <CtrlField component="mem" ctrlPort="ctrl"/>
      <CtrlField component="In0" ctrlPort="sel"/>
      <CtrlField component="In1" ctrlPort="sel"/>
      <CtrlField component="Out0" ctrlPort="sel"/>
    </CwFields>
    <StackPointer><RegisterFile ref="RF" index="0"/></StackPointer>
    <FramePointer><RegisterFile ref="RF" index="1"/></FramePointer>
  </Compiler-aspect>
</CustomIP>
```

Figure 3.2. GNR description of the NISC in Figure 3.1

3.1.1 Compiler aspect of *NiscArchitecture*

In addition to the netlist information of the *NiscArchitecture* the compiler may also need extra information that is captured in the compiler aspect of the *NiscArchitecture*. The datapath of NISC can be customized and simplified to exactly match the requirements of the application. Therefore, some of the information in the compiler aspect is optional. If such information is not provided in the model, then the corresponding features will be disabled in the compiler. In the reset of this section, we explain the compiler aspect information of NISC architecture.

3.1.1.1 Main memory

The compiler aspect of the *NiscArchitecture* component, determines one memory component in the datapath as the main memory. However, a datapath can have more than one *memory* component, but the user must directly access the rest (this mechanism is explain in Chapter Chapter 5). All normal memory (load/store) and stack (push/pop) operations are bound to the specified main memory. If the compiler aspect does not specify any memory component as the default main memory, then memory and stack operations will be disabled and compiler will generate error if these operations are detected in the program. Consequently, the function calls will also be disabled because they depend on stack operations.

3.1.1.2 Clock period length

The clock period length must be specified in term of the time unit. This value along with the operation delays is used by the compiler to determine if multiple operations can be chained within one cycle, or if execution of one operation must expand across multiple clock cycles. For example, if the clock period is 1 unit and a multiply operation delay is 2

units, then the multiply is scheduled to last two cycles before the results can be used by another operation or written back to register file.

3.1.1.3 Size of pointers

The size of pointers is used by the compiler when handling memory accesses and pointer calculations. For example, when loading a pointer to pointer variable (e.g. `int** pInt;` in C), the compiler needs to know what kind of load instruction it should use. Also, when performing pointer calculations the pointer size must be considered. Figure 3.3 shows an example of how the point size affects the compiler output.

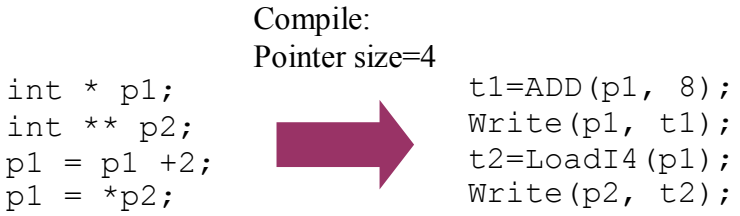


Figure 3.3. Compiling pointer calculations to 3-address code with 4 byte pointers

3.1.1.4 Special registers

To support function calls, the Stack Pointer (SP) and the Frame Pointer (FP) registers must be specified a register or a register-file element. The FP register always points to the start of the stack of a function, while SP register points to the end of the stack. Figure 3.4 shows how these registers are utilized for function calls.

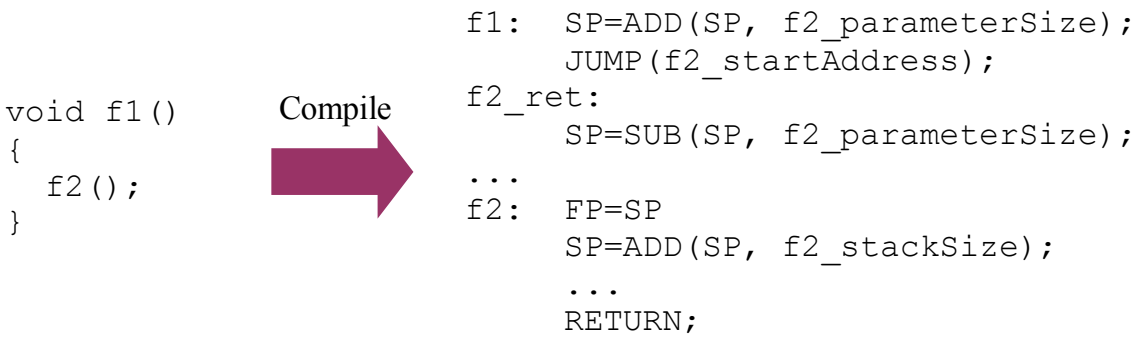


Figure 3.4. Using SP/FP registers in the 3-address code of function call

3.1.1.5 Structure of control words

The structure of the control words determines (a) the bit-width and number of constant fields, and (b) the index of control bits of each component in the control word. The compiler uses this information to convert the FSM into the corresponding control value stream. For example in Figure 3.2, the `cwField` section of the compiler aspect specifies that the lower 10 bits of the control word are used for constant values and the rest of control word is filled with the control signals of the components. Therefore, the corresponding control word looks like Figure 3.5. According to the GNR description of Figure 3.2, the first 10 bits of the CW are first sign extended and then connected to the `In0` multiplexer which provides a constant field for operations with a constant operand.

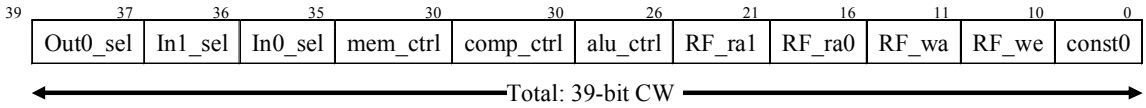


Figure 3.5. Control word structure according to GNR of Figure 3.2

Figure 3.6 shows the structure of the NISC model after control word structure is constructed and proper control connections are added between CW and control ports of the components. A *sign extender* component is also added for accessing constant filed.

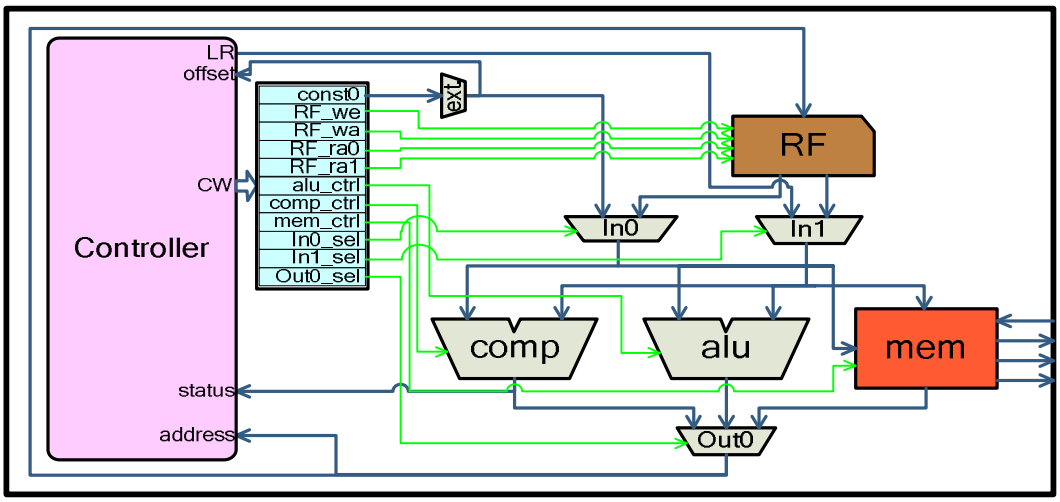


Figure 3.6. NISC model of Figure 3.1 after loading and adding control connections

3.2 Basic components

The basic component types include *register*, *register-file*, *multiplexer*, *tri-sate buffer*, *bus*, *functional-unit*, *memory*, and *controller*. Depending on the component type, the compiler aspect of a component may define one or more machine actions (MA). The machine actions are the very low level functionalities of the components. The compiler constructs different behaviors by composing the MAs and scheduling them in proper clock cycles.

There are four types of machine actions: *Read*, *Write*, *Transfer* and *Execute*. The *Read* and *Write* MAs describe access to a registered storage, while *Transfer* describes data movement from one port to another. The *Execute* MA represents the operations that a component can perform. Each MA may have at most one output which is associated with one output port of the corresponding component. Similarly, an MA may have several inputs each of which is associated with an input port of the corresponding component. Each MA also defines the timing as well as the control values of each control port of the corresponding component. Finally, if an *Execute* MA specifies number of pipeline stages, then it is considered as a pipelined operation. Otherwise, if the delay of that MA is longer than the clock period, then it is considered as multi-cycle operation. Basic components are divided into four groups: (a) *register* and *register-file* components can only define *Read* and *Write* MAs; (b) *multiplexer*, *tri-sate buffer*, *bus* components can only define *Transfer* MAs; (c) *functional-unit*, *memory* components can only define *Execute* or *Transfer* MAs; and (d) *controller* components defined a set of predefined *Execute* MAs. For example Figure 3.7 shows the GNR description of a small ALU that implements three operations, hence its control port is two bits wide (supporting up to four

operations). Although RTL generation for NISC is not part of this thesis, it is worth mentioning that, in many cases, the RTL description of a component can be easily generated from the MA descriptions in its compiler aspect.

```
<FU typeName="ALU">
  <Params>
    <Param n="BIT_WIDTH"/>
    <Param n="DELAY" val="1"/>
  </Params>
  <Ports>
    <InPort n="i0" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i1" bitWidth="{@BIT_WIDTH}"/>
    <OutPort n="o" bitWidth="{@BIT_WIDTH}"/>
    <CtrlPort n="ctrl" bitWidth="2" default="00"/>
  </Ports>
  <Compiler-aspect>
    <Operations>
      <Operation n="Add" delay="{@DELAY}">
        <Output port="o"/>
        <Input port="i0"/>
        <Input port="i1"/>
        <Ctrl val="00" port="ctrl"/>
      </Operation>
      <Operation n="Sub" delay="{@DELAY}">
        <Output port="o"/>
        <Input port="i0"/>
        <Input port="i1"/>
        <Ctrl val="01" port="ctrl"/>
      </Operation>
      <Operation n="Not" delay="{@DELAY}">
        <Output port="o"/>
        <Input port="i0"/>
        <Ctrl val="10" port="ctrl"/>
      </Operation>
    </Operations>
  </Compiler-aspect>
</FU>
```

Figure 3.7. GNR description of an example ALU

As an example, assume that the compiler needs to implement an addition between two variables in C code on the datapath of Figure 3.1. For each operand in the DFG of the program, the compiler must use a *Read* MA from register file `RF`, and a *Transfer* MA through multiplexer `In0` or `In1`. Then, an *Execute* MA is scheduled on the `alu` and

result is written back by using a *Transfer* MA through multiplexer `Out0` and a *Write* MA on the `RF`. If the clock period is longer than the sum of all these machine-actions, then the ADD operation in DFG takes one cycle, at most; otherwise it takes multiple cycles.

In addition to *Read* and *Write* MAs, the *register* and *register-file* components specify the variable types that they can store. During compilation, the compiler uses this information to bind local variables to proper register storage. Global variables are bound to the main memory.

3.3 Hierarchical components

The hierarchical components are used to simplify the construction of new components from the available ones. This is done with the *module* component type in the NISC architecture model. For example, consider the full GNR description of a two input multiplexer (`Mux2`) shown in Figure 3.8. We can repeat this whole code to create a four input multiplexer (`Mux4`); however different aspects of the components must be again described for every tool. Another option is to construct a `Mux4` from three `Mux2` components as shown in Figure 3.9 and its corresponding GNR shown in Figure 3.10. At compile time, every time a data transfer is needed from an input port of a Mux4 component to its output port, the compiler schedules several *Transfer* MA trough different internal multiplexers of this module.

For larger multiplexers built using a *module*, the increased number of MAs that the scheduler should schedule for a single data transfer may slow down the compiler. To prevent this negative effect, for frequently used *module* components, we can add compiler aspect that describes the internal behavior of the module and prevent the

compiler from going further down the hierarchy for generating the control bits. Figure 3.11 shows the updated version of Figure 3.9 with the compiler aspect information. Especially for more complex models, this flexibility of modeling allows the designer to choose where to spend the most time to gain the most productivity. The GNRs in Figure 3.9 and Figure 3.11 will lead to exactly same RTL results at the end.

```
<Mux type="Mux2">
  <Params>
    <Param n="BIT_WIDTH"/>
    <Param n="DELAY" val="0"/>
  </Params>
  <Ports>
    <InPort n="i0" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i1" bitWidth="{@BIT_WIDTH}"/>
    <CtrlPort n="sel" bitWidth="1" default="x"/>
    <OutPort n="o" bitWidth="{@BIT_WIDTH}"/>
  </Ports>
  <Annot_verilog>
    <Synthesis topModuleName="Mux2">
      <VerilogParams>
        <Param n="BIT_WIDTH" val="{@BIT_WIDTH}"/>
      </VerilogParams>
      <VerilogCode>
        <File n="Mux2.v"/>
      </VerilogCode>
    </Synthesis>
    <Simulation topModuleName="Mux2">
      <VerilogParams>
        <Param n="BIT_WIDTH" val="{@BIT_WIDTH}"/>
      </VerilogParams>
      <VerilogCode>
        <File n="Mux2.v"/>
      </VerilogCode>
    </Simulation>
  </Annot_verilog>
  <Annot_compiler>
    <Transfers>
      <DataTransfer inPort="i0" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="0" port="sel"/>
      </DataTransfer>
      <DataTransfer inPort="i1" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="1" port="sel"/>
      </DataTransfer>
    </Transfers>
  </Annot_compiler>
</Mux>
```

Figure 3.8. GNR description of a Mux2 multiplexer

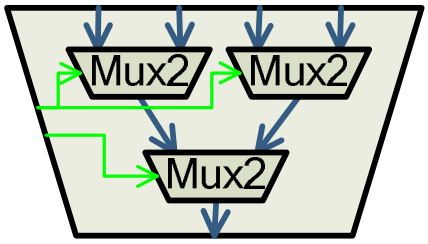


Figure 3.9. Construction of Mux4 from several Mux2 components

```
<Mux type="Mux4">
  <Params>
    <Param n="BIT_WIDTH"/>
    <Param n="DELAY" val="0"/>
  </Params>
  <Ports>
    <InPort n="i0" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i1" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i2" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i3" bitWidth="{@BIT_WIDTH}"/>
    <CtrlPort n="sel0" bitWidth="1" default="x"/>
    <CtrlPort n="sel1" bitWidth="1" default="x"/>
    <OutPort n="o" bitWidth="{@BIT_WIDTH}"/>
  </Ports>
  <Netlist>
    <Components>
      <Instanse n="m0" type="Mux2"/>
      <Instanse n="m1" type="Mux2"/>
      <Instanse n="m2" type="Mux2"/>
    </Components>
    <Connections>
      <Conn src="" srcPort="i0" dest="m0" destPort="i0"/>
      <Conn src="" srcPort="i1" dest="m0" destPort="i1"/>
      <Conn src="" srcPort="i2" dest="m1" destPort="i0"/>
      <Conn src="" srcPort="i3" dest="m1" destPort="i1"/>
      <Conn src="m0" srcPort="o" dest="m2" destPort="i0"/>
      <Conn src="m1" srcPort="o" dest="m2" destPort="i1"/>
      <Conn src="m2" srcPort="o" dest="" destPort="o"/>
      <Conn src="" srcPort="sel0" dest="m0" destPort="sel"/>
      <Conn src="" srcPort="sel0" dest="m2" destPort="sel"/>
      <Conn src="" srcPort="sel1" dest="m3" destPort="sel"/>
    </Connections>
  </Netlist>
</Mux>
```

Figure 3.10. GNR of Mux4 built from several Mux2 components

```
<Mux type="Mux4">
  <Params>
    <Param n="BIT_WIDTH"/>
    <Param n="DELAY" val="0"/>
  </Params>
  <Ports>
    <InPort n="i0" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i1" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i2" bitWidth="{@BIT_WIDTH}"/>
    <InPort n="i3" bitWidth="{@BIT_WIDTH}"/>
    <CtrlPort n="sel" bitWidth="2" default="xx"/>
    <OutPort n="o" bitWidth="{@BIT_WIDTH}"/>
  </Ports>
  <Netlist>
    <Components>
      <Instanse n="m0" type="Mux2"/>
      <Instanse n="m1" type="Mux2"/>
      <Instanse n="m2" type="Mux2"/>
    </Components>
    <Connections>
      <Conn src="" srcPort="i0" dest="m0" destPort="i0"/>
      <Conn src="" srcPort="i1" dest="m0" destPort="i1"/>
      <Conn src="" srcPort="i2" dest="m1" destPort="i0"/>
      <Conn src="" srcPort="i3" dest="m1" destPort="i1"/>
      <Conn src="m0" srcPort="o" dest="m2" destPort="i0"/>
      <Conn src="m1" srcPort="o" dest="m2" destPort="i1"/>
      <Conn src="m2" srcPort="o" dest="" destPort="o"/>
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Transfers>
      <DataTransfer inPort="i0" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="0" port="sel1" /><Ctrl val="0" port="sel0" />
      </DataTransfer>
      <DataTransfer inPort="i1" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="0" port="sel1" /><Ctrl val="1" port="sel0" />
      </DataTransfer>
      <DataTransfer inPort="i2" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="1" port="sel1" /><Ctrl val="0" port="sel0" />
      </DataTransfer>
      <DataTransfer inPort="i3" outPort="o" transferDelay="{@DELAY}">
        <Ctrl val="1" port="sel1" /><Ctrl val="1" port="sel0" />
      </DataTransfer>
    </Transfers>
  </Annot_compiler>
</Mux>
```

Figure 3.11. Mux4 module with compiler aspect to speed up compilation

3.4 Comparison with other approaches

Model based compilation has be the focus of many retargetable compilers. Rather than being fixed for a single processor, the retargetable compilers can generate code for a class of processors usually captured in an Architecture Description Language (ADL)

[52][71]. As we explained in section 1.2, all retargetable compilers use instruction behaviors for compilation. Behavioral ADLs (e.g. LISA [48] and EXPRESSION [3]) explicitly describe the behavior of all possible instructions while structural ADLs (e.g. RECORD [58][59], CHESS [32], and MIMOLA [51][63]) describe the structure of the processor and hence instruction behaviors must be extracted from them.

The proposed model for the NISC architecture has several advantages over retargetable compilation models including:

- Previous models are very lengthy and complex because they should either capture all possible instructions format, or capture the instruction decoder as well as the datapath. The NISC model only captures the datapath netlist and all combinations of possible operations are generated by the compiler. For example, if datapath has two fully connected adder and multiplier units, the NISC model only need to capture the existence of these two units. However, in an instruction based ADL, all compilations of ADD, MUL, ADD_MULL, MULL_ADD instructions with both register operands and constant operands must be described. In a more complex datapath with more possibilities, this problem becomes worse.
- Behavioral ADLs rarely support hardware generation and only target compilation or simulation. Even those who generate RTL cannot achieve good hardware quality because many of the architectural details are missing in the description. Even the structural ADLs do not capture the architecture in the level of details of the NISC model (i.e. capturing down to wires, multiplexers, and control signals). Therefore, the NISC model not only supports compilation, but also is suitable for generating good quality RTL for hardware implementation.

- All retargetable compilers target a complete processor and none of them can be used for generating dedicated IP blocks. Approaches such as TIPI [68] that have attempted to target IP blocks, have paid too much attention to hardware generation but little to compilation, which has prevented them from generating a compiler for their approach. The NISC model is a balanced abstraction that enables both efficient compilation and efficient hardware generation for any datapath complexity.
- Due to its detailed architectural information, the NISC model enables the compiler to use operation chaining, multi-cycle or pipelined operations. These features are supported in NISC by (a) properly describing the architecture structure and timing information in the model, and (b) a new scheduling algorithm explained in Chapter 4. None of the other ADLs can directly support these features.
- All previous compilation models always assume that the data forwarding connections connect the output of all functional units to all of the inputs of all functional units. This results in a huge hardware overhead on the final processor. Only one approach [8] has addressed partial data forwarding and bypass paths. However, this approach uses operation tables that must describe all permutations of forwarding paths between all operations, which can be very lengthy. The NISC model can accurately describe non-uniform data forwarding by only capturing the actual wires between functional units. The NISC compiler algorithms can then use this information to utilize bypass paths properly.

- Other modeling approaches for retargetable compilation do not consider the value of clock period in the model. They only capture timings based on number of clock cycles. In contrast, the NISC model and the corresponding algorithms also consider the length of clock period to determine whether operations can be chained or if an operation takes several cycles. This important feature the means of readjusting the schedule based on the actual timings collected form a real implementation.
- Other retargetable compilation approaches typically need one model and description for mapping operations to assembly instructions and another model for mapping assembly instructions to their binary values. In other words, one model for compiler and another model for the linker. By capturing the control connections in the datapath model, the NISC compiler can directly map the operations to the corresponding control binary.

Chapter 4. Compilation

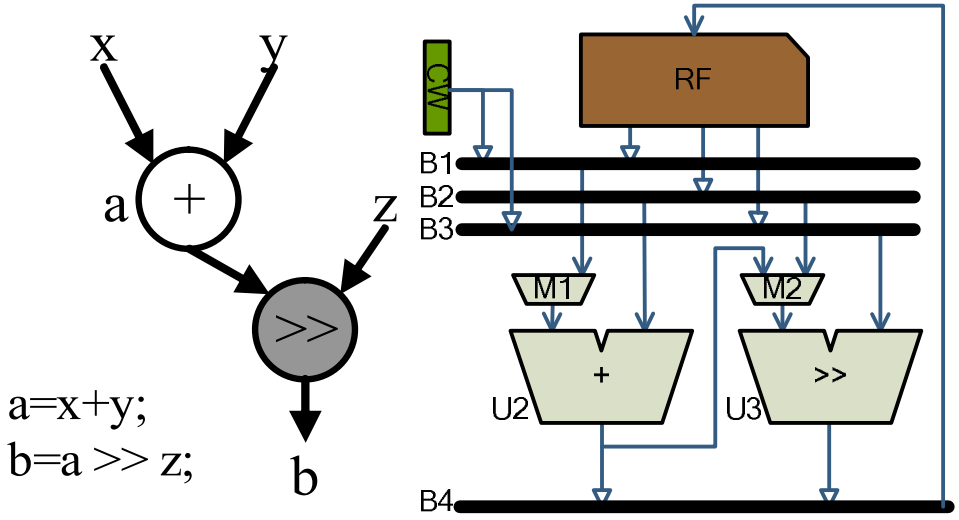
In the rest of this chapter we explain the compilation algorithm. From this point on, any reference to the information of the datapath implies that the model of the NISC architecture contains that information.

4.1 Overview of compilation algorithm

In this section we illustrate the basis of our scheduling and binding algorithm using several examples. The input of algorithm is the CDFG of an application, netlist of datapath components and the clock period of the system. The output is an FSM in which each state represents a set of Machine Actions (MAs) that execute in one clock cycle. The set of MAs are later used to generate the states of FSM and the control bits of components.

As opposed to traditional HSL, we can not schedule operations merely based on the delay of the functional units. The number of clock cycles (or states) between the schedule of an operation and its successor depends on both the binding of operations to functional units (FU) and the delay of the path between corresponding FUs. For example, suppose we want to map a DFG on a datapath as shown in Figure 4.1. Operation shift-left (>>) can read the result of operation + in two ways. If we schedule operation + on *U2* and

store the result in register file RF, then operation >> must be scheduled on *U3* in the next cycle to read the result from RF through bus *B2* and multiplexer *M2*. Operation >> can also be scheduled in the same cycle with operation + and read the result directly from *U2* through multiplexer *M2*. Therefore, selection of the path between *U2* and *U3* can directly affect the schedule. Since knowing the path delay between operations requires knowing the operation binding, the scheduling and binding must be performed simultaneously.



(a) DFG **(b) Datapath**

Figure 4.1. Different possible schedules for the DFG depending on binding

The basic idea in the algorithm is to schedule an operation and all of its predecessors together. An *output operation* in the DFG of a basic block is an operation that does not have a successor in that basic block. We start from output operations and traverse the DFG backward. Each operation is scheduled after all its successors are scheduled. The scheduling and binding of successors of an operation determine when and where the result of that operation is needed. This information can be used for: utilizing available paths between FUs efficiently, chaining operations, avoiding unnecessary register file read/writes, etc.

During schedule, we consider each operation and all of its predecessors, i.e. the sub-tree of operations behind each operation. Therefore, we need to partition the DFG of the basic block into sub-trees. To make sure the whole DFG is processed, we start from sub-tress whose root is an output operation of the basic block. The leaves are input variables, constants, or output operations from other basic blocks. If the successors of an operation belong to different sub-trees, then that operation is considered as an *internal output* and will have its own sub-tree. Such nodes are detected during scheduling. Figure 4.2 shows an example DFG that is partitioned into three sub-trees. The roots of the sub-trees (shown with shaded nodes) are the output operations. The algorithm schedules each sub-tree separately. If during scheduling of the operations of a sub-tree, the schedule of an operation fails, then that operation is considered an internal output and becomes the root of a new sub-tree. A sub-tree is available for schedule as soon as all successors of its root (output operation) are scheduled. Available sub-trees are ordered by the mobility of their root. The algorithm starts from output nodes and schedules backward toward their inputs, therefore more critical outputs tend to be generated towards the end of the basic block (similar to ALAP schedule).

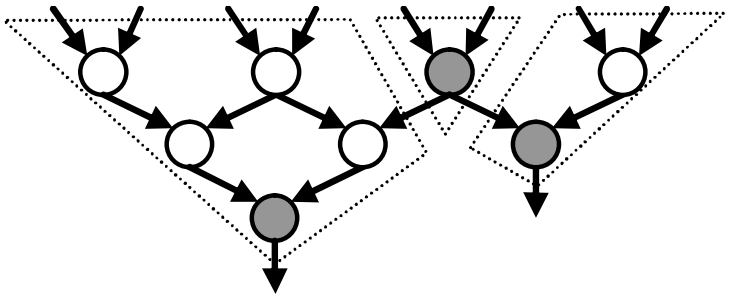
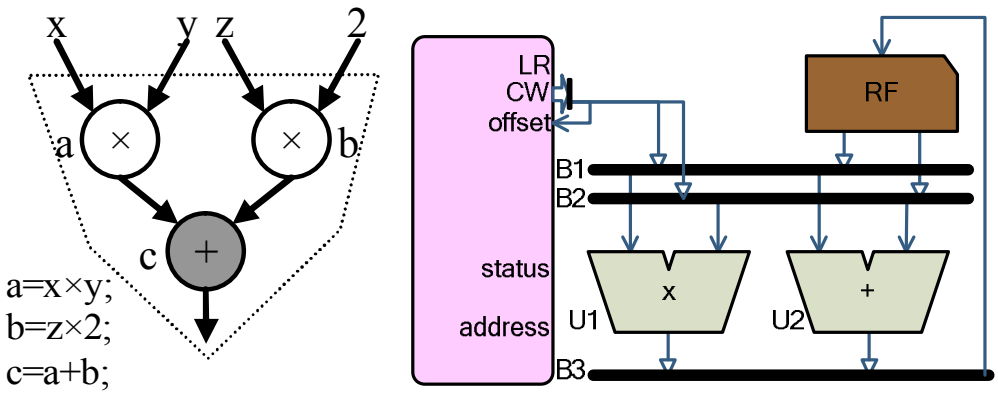


Figure 4.2. Partitioning a DFG into output sub-trees

During scheduling, different types of values may be bound to different types of storages (variable binding). For example, global variables may be bound to memory, local variables to stack or register file, and so on. A constant is bound to memory or the

constant fields in the control word (CW) register, depending on its size. A control word may have limited number of constant fields (e.g. Figure 3.2) that are generated in each cycle along with the rest of control bits. These constant fields are loaded into the CW register and then transferred to a proper location in datapath after sign extension (see Figure 3.6). The NISC compiler determines the values of constant(s) in each cycle. It also schedules proper set of MAs to transfer the value(s) to where it is needed.

In the rest of this section, through several examples, we illustrate how the DFG is partitioned into sub-trees during scheduling. We also show that how the scheduling algorithm works for simple datapaths as well as those that have feature such as multi-cycle operation, pipelined operation, heterogeneous pipelining, heterogeneous data forwarding, and operation chaining.



(a) DFG **(b) Datapath without pipelining**

Figure 4.3. Compiling on a datapath without pipelining

4.1.1 Example: Simple datapath

Consider the example DFG of Figure 4.3(a) to be mapped on the simple datapath of Figure 4.3(b). Assume that the clock period is 20 units and delays of *U1*, *U2*, and busses are 17, 7, and 1 units, respectively. We schedule the operations of basic block so that all results are available before last cycle, i.e. 0; therefore, the MAs are scheduled in negative cycle numbers. In each step of the algorithm, we try to schedule the sub-trees that can

generate their results before a given cycle *clk*. The *clk* starts from 0 and is decremented in each step until all sub-trees of a basic block are scheduled.

When scheduling an output sub-tree, first step is to know where the output is stored. In our example, assume *c* is bound to register file *RF*. We must schedule operation + so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. From the list of available functional units (FUs), we first select an FU that implements + (operation binding). Then we make sure that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1 (interconnect binding). In this example we select *U2* for + and bus *B3* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper MAs in order to transfer the value of *a* to the left input port of *U2* and value of *b* to the right input port of *U2*. Figure 4.4 shows the status of schedule after scheduling the + operation. The figure shows the set of MAs that are scheduled in each cycle to read or generated a value. At this point, *B1* and *B2* are considered the *destinations* to which values of *a* and *b* must be transferred in clock cycle -1, respectively.

clock	Scheduled MAs
-1	B1=?; B2=?; B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.4. Schedule of MAs after scheduling + operation

In order to read the values of *a* and *b*, we need to schedule them on their corresponding FU (i.e. *U1* multiplier) and then send the result to *U2*. Since there is no path between any *U1* and *U2*, then we assume that *a* and *b* are stored in register file *RF* and will try to schedule them later. Since in cycle -1, we can read from register file, then we schedule the proper *Read* MAs from *RF* and consider *a* and *b* as internal outputs.

Figure 4.5 shows the status of schedule after scheduling *Read* MAs. The sub-tree of *c* is now completely scheduled and the resource reservations can be finalized.

clock	Scheduled MAs
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.5. Schedule of MAs after scheduling *h* sub-tree

The results of sub-tress of *a* and *b* must have their result ready before cycle -1. Therefore, the corresponding MAs must be scheduled in or before clock cycle -2 and write the result in register file *RF*. We start from *a* and schedule its sub-tree first. We need to write the result to *RF* and choose an FU to perform × operation. We choose multiplier *U1* and schedule the operands from *RF* to *U1* through busses *B1* and *B2*. Figure 4.6 shows the schedule status after completing the schedule of *a*.

clock	Scheduled MAs
-2	B1=RF(x); B2=RF(y); B3=U1(B1, B2); RF(a)=B3;
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.6. Schedule of MAs after scheduling *a* sub-tree

After scheduling *a* on *U1*, there are no more multipliers left to schedule operation *b*. Therefore, we need to decrement the clock value and try again in cycle -3. In this cycle, we need multiplier *U1*, both busses *B1* and *B2*, one register file port, and the constant filed in the CW. The constant filed is read from CW and is sign extended and then passed to the proper bus. Since all these resources are available in clock cycle -3, we can successfully schedule *b*. The final schedule of the complete DFG of Figure 4.3 is shown in Figure 4.7.

clock	Scheduled MAs
-3	B1=RF(z); B2=CW; B3=U1(B1, B2); RF(b)=B3;
-2	B1=RF(x); B2=RF(y); B3=U1(B1, B2); RF(a)=B3;
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.7. Schedule of MAs after scheduling DFG of Figure 4.3

4.1.2 Example: multi-cycle operation

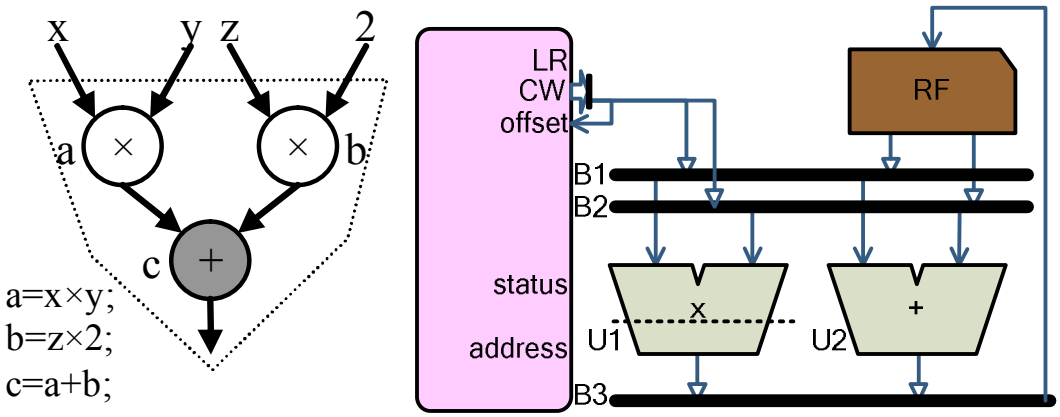
Consider the example of Figure 4.3 again but this time, let's assume that the clock period is still 10 units, and the delay of *U1*, *U2*, and busses are 17, 7, and 1 units, respectively. In this case, the delay of multiplier *U1* is longer than a single cycle. If we repeat the scheduling steps, everything remains the same as what was explained in Section 4.1.1. But this time, the schedule of *a* and *b* are two cycles. This means their inputs must remain stable for two cycles, but they can write their results at the end of second cycle. Figure 4.8 shows the schedule status after scheduling the DFG. Note that the result in Figure 4.8 is two cycles longer than that of Figure 4.7 but since the clock period is shorter, it runs overall faster (i.e. 5x10 < 3x20). This shows one of the benefits of multi-cycle operations, and our algorithm can fully utilize it.

clock	Scheduled MAs
-5	B1=RF(z);B2=CW;B3=U1(B1, B2);
-4	B1=RF(z);B2=CW;B3=U1(B1, B2); RF(b)=B3;
-3	B1=RF(x);B2=RF(y);B3=U1(B1, B2);
-2	B1=RF(x); B2=RF(y); B3=U1(B1, B2); RF(a)=B3;
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.8. Schedule of Figure 4.3(a) DFG with a multi-cycle multiplier

4.1.3 Example: pipelined operation

Consider the example DFG of Figure 4.9(a) to be mapped on the datapath of Figure 4.9(b) which has a pipelined multiplier. Assume that the clock period is 10 units and delays of *U2* and busses are 7 and 1 units, respectively. Also *U1* has two pipeline stages. This means that *U1* takes two cycles to generate its results, but the inputs can change every cycle, and respectively the output can be read in every cycle. As before, we schedule the operations of basic block so that all results are available before last cycle, i.e. 0; therefore, the MAs are scheduled in negative cycle numbers.



(a) DFG **(b) Datapath without pipelining**

Figure 4.9. Compiling on a datapath without pipelining

When scheduling an output sub-tree, first step is to know where the output is stored. In our example, assume *c* is bound to register file *RF*. We must schedule operation + so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. From the list of available functional units (FUs), we first select an FU that implements + (operation binding). Then we make sure that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1 (interconnect binding). In this example we select *U2* for + and bus *B3* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper MAs in order to transfer the value of *a* to the left input port of *U2* and value of *b* to the right input port of *U2*. Figure 4.10 shows the status of schedule after scheduling the + operation. The figure shows the set of MAs that are scheduled in each cycle to read or generated a value. At this point, *B1* and *B2* are considered the *destinations* to which values of *a* and *b* must be transferred in clock cycle -1, respectively.

clock	Scheduled MAs
-1	B1=?; B2=?; B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.10. Schedule of MAs after scheduling + operation

In order to read the values of *a* and *b*, we need to schedule them on their corresponding FU (i.e. *U1* multiplier) and then send the result to *U2*. Since there is no path between any *U1* and *U2*, we assume that *a* and *b* are stored in register file *RF* and will try to schedule them later. Since in cycle -1, we can read from register file, then we schedule the proper *Read* MAs from *RF* and consider *a* and *b* as internal outputs. Figure 4.11 shows the status of schedule after scheduling *Read* MAs. The sub-tree of *c* is not completely scheduled and the resource reservations can be finalized.

clock	Scheduled MAs
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.11. Schedule of MAs after scheduling *h* sub-tree

The results of sub-tress of *a* and *b* must have their result ready before cycle -1. Therefore, the corresponding MAs must be scheduled in or before clock cycle -2 and write the result in register file *RF*. We start from *a* and schedule its sub-tree first. We need to write the result to *RF* and choose an FU to perform × operation. We choose multiplier *U1* and schedule the operands from *RF* to *U1* through busses *B1* and *B2*.This time, since *U1* is pipelined, we schedule its output to be read in cycle -2 while its inputs are scheduled in cycle -3. Figure 4.12 shows the schedule status after completing the schedule of *a*.

clock	Scheduled MAs
-3	B1=RF(x); B2=RF(y);
-2	B3=U1(B1, B2); RF(a)=B3;
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.12. Schedule of MAs after scheduling *a* sub-tree on pipelined multiplier

After scheduling *a* on *U1* in cycle -2, there is no more multiplier left to schedule operation *b*. Therefore, we need to decrement the clock value and try again in cycle -3. In this cycle, multiplier *U1* itself is available but both of its inputs as well as busses *B1* and

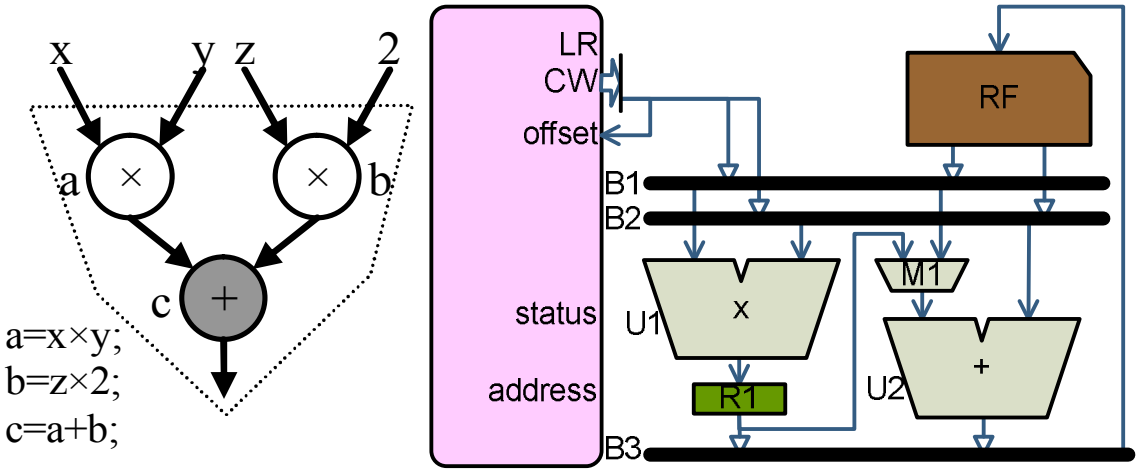
B2 are not available. However, we do need these resources until cycle -4. Therefore, we can successfully schedule operation *b* in cycle -3. The final schedule of the complete DFG is shown in Figure 4.13. Note that, because of pipelined FU, the result in Figure 4.13 is one cycle shorter (faster) than that of Figure 4.8. In all of these cases, we are using the same algorithm, i.e. walk back on DFG and schedule and bind!

clock	Scheduled MAs
-4	B1=RF(z); B2=CW;
-3	B1=RF(x); B2=RF(y); B3=U1(B1, B2); RF(b)=B3;
-2	B3=U1(B1, B2); RF(a)=B3;
-1	B1=RF(a); B2=RF(b); B3=U2(B1, B2); RF(c)=B3;
0	

Figure 4.13. Schedule of MAs after scheduling DFG of Figure 4.9

4.1.4 Example: heterogeneous pipelining and data forwarding

In this section we show an example to illustrate how our algorithm supports heterogeneous pipelining and data forwarding. This is one of the unique features of our algorithm that has no been covered by previous scheduling algorithm. Consider the example DFG of Figure 4.14(a) to be mapped on the datapath of Figure 4.14(b). Assume that the clock period is 20 units and delays of *U1*, *U2*, and busses are 17, 7, and 1 units, respectively.



(a) DFG **(b) Datapath with heterogeneous pipelining and forwarding**

Figure 4.14. Compiling in presence of heterogeneous pipelining and data forwarding

When scheduling an output sub-tree, first step is to know where the output is stored. In this example, assume *c* is bound to register file *RF*. We must schedule operation + so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. From the list of available functional units (FUs), we first select an FU that implements + (operation binding). Then we make sure that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1 (interconnect binding). In this example we select *U2* for + and bus *B3* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper MAs in order to transfer the value of *a* to the left input port of *U2* and value of *b* to the right input port of *U2*. Figure 4.15 shows the status of schedule after scheduling the + operation. The figure shows the set of MAs that are scheduled in each cycle to read or generate a value. At this point, *M1* and *B2* are considered the *destinations* to which values of *a* and *b* must be transferred in clock cycle -1, respectively.

clock	Scheduled MAs
-1	M1=?; B2=?; B3=U2(M1, B2); RF(c)=B3;
0	

Figure 4.15. Schedule of MAs after scheduling + operation

In order to read the values of *a* and *b*, we need to schedule them on their corresponding FU (i.e. *U1* multiplier) and then send the result to *U2*. We start from *a* and schedule its sub-tree first. We can utilize the available forwarding path by scheduling *a* on multiplier *U1* and passing the results to the left port of *U2* via register *R1* and multiplexer *M1*. Because of the register *R1*, the multiplication operation on *U1* is actually scheduled one cycle before operation of *U2*. We then continue to schedule the operands of *a* and since all resources are available, we can schedule the sub-tree of *a* as well.

Afterwards, if we try to schedule sub-tree of *b*, we will see that there is no path between multiplier *U1* and right port of *U1*. Therefore, we assume that *b* can be read from register file *RF* in clock cycle -1 and it will be scheduled later as an internal output to write its output to *RF*. Figure 4.16 shows the status of schedule at this point.

clock	Scheduled MAs
-2	B1=RF(x); B2=RF(y); B3=U1(B1, B2); R1=B3;
-1	M1=R1; B2=RF(b); B3=U2(M1, B2); RF(c)=B3;
0	

Figure 4.16. Schedule of MAs after scheduling *h* sub-tree

At this point the only sub-tree left to schedule is *b* that must have its result ready before cycle -1. Therefore, the corresponding MAs must be scheduled in or before clock cycle -2 and write the result in register file *RF*. We first choose the proper FU (i.e. *U1*) and find a path between its output and the input of *RF*. The only available path is the pipelined path that goes through register *R1* and bus *B3*. Therefore, the execution of *U1* is pushed back one more cycle from -2 to -3. Figure 4.17 shows the full schedule of DFG after scheduling all available sub-trees.

clock	Scheduled MAs
-3	B1=RF(z); B2=CW; B3=U1(B1, B2); R1=B3;
-2	B1=RF(x); B2=RF(y); B3=U1(B1, B2); R1=B3; RF(c)= R1;
-1	M1=R1; B2=RF(b); B3=U2(M1, B2); RF(c)=B3;
0	

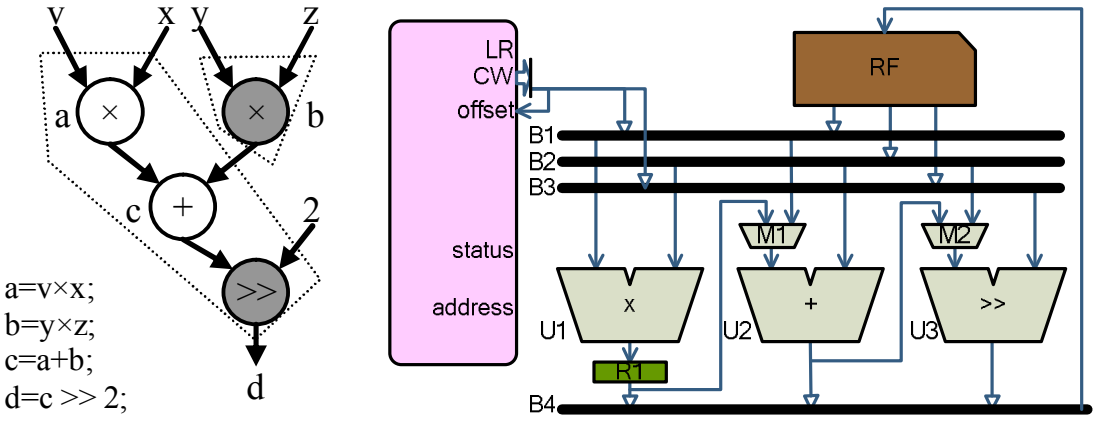
Figure 4.17. Schedule of MAs after scheduling *a* sub-tree

Note that results of Figure 4.17 and Figure 4.7 both take 3 cycles. However, since the datapath of Figure 4.14(b) is pipelined, it can potentially run at a faster clock frequency than datapath of Figure 4.3(b). The same scheduling algorithm could handle the available pipelining and forwarding although not all paths were pipelined and not all units had data forwarding.

4.1.5 Example: pipelining, forwarding, and operation chaining

Consider the example DFG of Figure 4.18(a) to be mapped on the datapath of Figure 4.18(b). Again, assume that the clock period is 20 units and delays of *U1*, *U2*, *U3*, multiplexers and busses are 17, 7, 5, 1 and 1 units, respectively. Since there are enough busses in this datapath and also the delay of busses, multiplexers, *U2,* and *U3* together is less than the clock period, then we should be able to chain + and >> operations into one cycle. This example also has heterogeneous pipelining and data forwarding. The following shows how our algorithm supports all of these features simultaneously.

As before, we schedule the operations of basic block so that all results are available before last cycle, i.e. 0; therefore, the MAs are scheduled in negative cycle numbers. In each step of the algorithm, we try to schedule the sub-trees that can generate their results before a given cycle *clk*. The *clk* starts from 0 and is decremented in each step until all sub-trees of a basic block are scheduled.



(a) DFG. **(b) datapath with forwarding and operation chaining**

Figure 4.18. Compiling in presence of forwarding and operation chaining

As before, assume *d* is bound to register file *RF*. We must schedule operation >> so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. We first select a FU that implements >> (operation binding). Then we make sure

that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1 (interconnect binding). In this example we select *U3* for >> and bus *B4* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper MAs in order to transfer the value of *c* to the left input port of *U3* and constant *2* to the right input port of *U3*. Figure 4.19 shows the status of schedule after scheduling the >> operation. The figure shows the set of MAs that are scheduled in each cycle to read or generate a value. At this point, *B3* and *M2* are considered the *destinations* to which values of *2* and *c* must be transferred in clock cycle -1, respectively.

clock	Scheduled MAs
-1	M2=?; B3=?; B4=U3(M2, B3); RF(d)=B4;
0	

Figure 4.19. Schedule of MAs after scheduling >> operation

In order to read constant *2*, we need to put the value of CW register on bus *B3*. As for variable *c*, we schedule the + operation on *U2* to perform the addition and pass the result to *U3* though multiplexer *M2*. Note that delay of reading operands of + operation and executing it on *U2*, plus the delay of reading operands of >> operation and executing it on *U3* and writing the results to *RF* is less than one clock cycle. Therefore, all of the corresponding MAs are scheduled together in clock cycle -1. The algorithm chains the operations in this way, whenever possible. The new status of scheduled MAs is shown in Figure 4.20. In the next step, we should schedule the × operations to deliver their results to the input ports of *U2*.

clock	Scheduled MAs
-1	M1=?; B2==?; M2=U2(M1, B2); B3=CW; B4=U3(M2, B3); RF(d)=B4;
0	

Figure 4.20. Schedule of MAs after scheduling + operation

The left operand (i.e. *a*) of operation *c* can be scheduled on *U1* to deliver its result through register *R1* in cycle -2 and multiplexer *M1* in cycle -1. At this point, no other multiplier is left to generate the right operand (*b*) and directly transfer it to the right input port of *U2*. Therefore, we assume that *b* is stored in the register file and try to read it from there. If the read is successful, the corresponding × operation (*b*) is considered as an internal output and will be scheduled later. Figure 4.21 shows the status of schedule at this time. The sub-tree of output *c* is now completely scheduled and the resource reservations can be finalized.

clock	Scheduled MAs
-2	B1=RF(v); B2=RF(x); R1=U1(B1, B2);
-1	M1=R1; B2=RF(b); M2=U2(M1, B2); B3=CW; B4=U3(M2, B3); RF(d)=B4;
0	

Figure 4.21. Schedule of MAs after scheduling *c* sub-tree

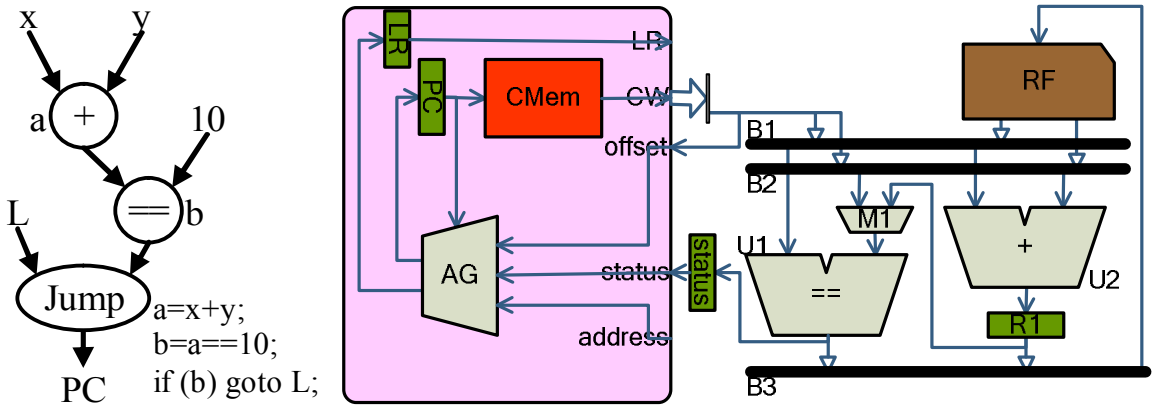
The sub-tree of internal output *b* must generate its result before cycle -1 where it is read and used by operation +. Therefore, the corresponding MAs must be scheduled in or before clock cycle -2 and write the result in register file *RF*. The path from *U1* to *RF* goes through register *R1* and hence takes more than one cycle. The second part of the path (after *R1*) is scheduled in cycle -2 and the first part (before *R1*) as well as the execution of operation × on *U1* is scheduled in cycle -3. The complete schedule is shown in Figure 4.22.

clock	Scheduled MAs
-3	B1=RF(c); B2=RF(d); R1=U1(B1, B2);
-2	B1=RF(a); B2=RF(b); R1=U1(B1, B2); B4=R1; RF(f)=B4;
-1	M1=R1; B2=RF(f); M2=U2(M1, B2); B3=CW; B4=U3(M2, B3); RF(h)=B4;
0	

Figure 4.22. Schedule of MAs after scheduling all sub-trees

4.1.6 Example: Controller pipelining

As a final example, consider the CFG of Figure 4.23(a) on the datapath of Figure 4.23 which has controller pipelining through register *status*. The internals of the controller is also shown in this figure. The CFG is in fact a conditional jump. In order to handle jump operations similar to other operations, we assume that jump writes its output to PC register and has two inputs: (a) the target address, and (b) a condition value. If we assume that the address generator (*AG*) inside the controller implements a *Jump* operation then handling (conditional) jumps in presence of controller pipelining becomes very similar to handling other operations in presence of datapath pipelining as shown bellow.



(a) CFG **(b) Datapath with controller pipelining**

Figure 4.23. Compiling CFG in presence of controller pipelining

In this example, we know that the output of *Jump* operation is bound to *PC* register inside the controller. We need now to schedule *Jump* operation so that its result can be stored in destination *PC* in cycle -1 and be available for reading in cycle 0. We first select a FU that implements *Jump* (i.e. *AG*). Then we make sure that a path exists between selected FU and destination *PC* and all elements of the path are available in cycle -1. In this case, the path consists of only a wire so it is always available. As before, now we need to schedule the input operands of the *Jump* operation to pass their values to proper ports of the *AG*, i.e. *AG.offset* and *AG.status* ports. Figure 4.24 shows the status of

schedule after scheduling the *jump* operation. At this point, *AG.offset* and *AG.status* are considered the *destinations* to which values of *L* and *b* must be transferred in clock cycle -1, respectively.

clock	Scheduled MAs
-1	AG.offset=?; AG.status=?; PC=AG(AG.offset, AG.status);
0	

Figure 4.24. Schedule of MAs after scheduling *jump* operation

The value of *L* can be directly read from the CW in cycle -1. We need to schedule the value *b* to be available at *AG.status* in cycle -1. We first find the proper FU for executing *b*, i.e. *U1*. There is a pipelined path between *U1* and *AG.status* which goes through register *status*. Therefore the execution of *b* is pushed back one cycle to cycle -2. Figure 4.25 shows the status of the schedule after scheduling the == operation. At this point we need to schedule operation *a* and constant 10 to be available in cycle -2 on the input ports of *U1*, i.e. bus *B1* and multiplexer *M1*. Note that since == operation is symmetric; the input operands can appear on either ports of *U1*.

clock	Scheduled MAs
-2	B1=?; M1=?; status=U1(B1, M1);
-1	AG.offset=CW; AG.status= status; PC=AG(AG.offset, AG.status);
0	

Figure 4.25. Schedule of MAs after scheduling == operation

Between the operands of *b*, we first try to schedule *a* since it has a deeper sub-tree. We can select FU *U2* and route the result to right input port of *U1* through register *R1* and multiplexer *M1*. Again since this path is pipelined, it will push the execution of *b* one cycle back to cycle -3. We can then schedule a read from operands of *a* in cycle -3. Once we have successfully scheduled *a*, we get back to second operand of *b*, i.e. constant 10. Since bus *B1* and constant filed of CW are both free in cycle -2, we can successfully

schedule a read from CW via bus *B1* to get the constant to the left port of *U1*. Figure 4.27 shows the complete schedule of the jump operation of Figure 4.23.

clock	Scheduled MAs
-3	B1=RF(x); B2=RF(y); R1=U2(B1, B2);
-2	B1=CW; M1=R1; status=U1(B1, M1);
-1	AG.offset=CW; AG.status=status; PC=AG(AG.offset, AG.status);
0	

Figure 4.26. Full schedule of *jump* in presence of controller pipelining

4.2 Cycle-accurate compilation algorithm

In this section, we describe our algorithm for compiling the application to a custom datapath. When compiling the CDFG of each function of a program, we must consider the structure of the controller for compiling the control-flow graph (CFG) and consider the structure of datapath for compiling the DFG. This process is described in the next two subsections. Description of the algorithm uses the following definitions:

- Each basic block has a schedule status *ss*, where *ss.MAs*(*clk*) stores the set of scheduled MAs in clock cycle *clk*, and *ss.resTable*(*clk*) stores the reservation status of resources in clock cycle *clk*, and *ss.length* shows the number of scheduled states for that block.
- For an operation *op*, *op.result* is the value generated by *op* and *op.operands* is the list of results of predecessors of *op*.
- For a functional unit *FU*, *FU.output* is the output port of *FU* and *FU.inputs* is the set of input ports of *FU*. A functional unit may implement multiple operations. For each operation, *FU.timing* represents the delay of the unit (or its stages if it is pipelined) as well as the duration of applying the control signals to the unit.
- A path *p* is the list of resources that can transfer a value from one point to another. These resources include busses, multiplexers and registers. The timing of resources of *p* is stored

in *p.timings* and is calculated based on delay of buses or multiplexers, or setup time and read delay of registers or register-files.

- A destination *dst* is a storage or an input port of a functional unit.

4.2.1 Mapping the CFG of the program

The result of NISC compiler is an FSM that can be implemented in logic or using a memory. In a memory-based implementation the state register is a program counter register (PC). Therefore, a state change in the FSM corresponds to incrementing the PC or loading it with a new value using a jump operation. While incrementing PC always takes one cycle, loading it with a new value may take more than one cycle. The result of scheduling a basic block is always a sequence of states (marked by value of *clk*). We may only need a jump at the end of a basic block, if the last state of the block is not before the first state of the next basic block. In the algorithm, we assume that the order of basic blocks is given, and that there may be jump operation at the end of some basic blocks.

Since we perform the scheduling backward, the result will be a set of states numbered from $-N$ to $+bd$. The return address of a function is loaded into PC at state 0. Constant *bd* is the branch delay of the architecture, i.e. in a basic block, after loading the target address of a jump operation into PC, *bd*–number of control words will be executed from that basic block. Value of *bd* depends on the distance between *PC* and control word register, which is fixed and unique. Usually, this delay is 0 or 1 cycle in NISC.

In procedure *ScheduleFunction* (Figure 4.27), the *blkList* contains the topologically ordered list of basic blocks where the last element of the list is the *return block*. The blocks of *blkList* are processed in reverse order, starting from return block and after scheduling each block, the results are added to the *fsm*. In the main loop of

ScheduleFunction (lines 3-8), before scheduling the body of a basic block, the jump operation at the end of block is scheduled. The same way that a + operation is mapped to an adder or ALU and writes its results to a register or register file, the jump is considered an operation that is mapped to address generator and writes its result to the PC register in cycle *clk*. In this way, we can schedule jump the same way that we schedule other operations (line 5). In order to make sure that the branch delay of the jump operation is filled by other operations in the basic block, we try to schedule the DFG of the basic block from cycle *clk*+*bd* (line 6). After scheduling each basic block, the new value of *clk* is calculated by decrementing the number of states in the block (line 7). The *ScheduleBasicBlock* and *ScheduleOpertion* functions are described in Section 4.2.2. After scheduling all functions of a program, *fsm* will contain the final FSM of the design.

```
00 ScheduleFunction(FSM fsm, ordered list of basic blocks blkList)
01   clk = 0;
02   bd = branch delay;
03   foreach (blk ∈ reverse of blkList)
04     if (blk has a jump operation)
05       ScheduleOperation(blk.jump, clk, blk.ss, PC);
06     ScheduleBasicBlock(blk, clk+bd);
07     add blk.ss states to fsm;
08     clk = clk – blk.ss.length;
```

Figure 4.27. Pseudo code of ScheduleFunction

4.2.2 Mapping the DFG of the program

The variable, operation, and interconnect bindings are performed during the schedule of each operation. We also allow pre-binding of variables and operations so that the designer or other algorithms can control the results (Chapter Chapter 5). For example, a partitioning algorithm may partition the variables and pre-bind them to two memory units.

Figure 4.28 shows the *ScheduleBasicBlock* procedure that performs the scheduling and binding for each basic block of a CDFG. In the main loop of this function (lines 3-16), the *available* output operations, i.e. sub-tree roots that can generate their results at clock cycle *clk*, are collected and sorted based on a priority function, such as operation mobility. During scheduling of each of these output operations, some internal outputs may be generated. If the schedule of the operation is successful, then the operation is removed from the set of sub-tree roots (*Roots*) and the newly generated internal outputs are added to the list in order to be processed later (lines 14-15). In each iteration of the loop, the *clk* is decremented and available output operations are collected and scheduled until all sub-trees in the block are processed.

```
00 ScheduleBasicBlock(block blk, clock lastClock)
01   Roots = {output operations in blk.DAG};
02   clk = lastClock;
03   while(Roots ≠ ∅)
04     AvailableOutputs = ∅;
05     foreach (operation op ∈ Root)
06       if (all successor of op are scheduled after clock clk)
07         AvailableOutputs = AvailableOutputs + {op};
08     Sort AvailableOutputs by OperationPriorities;
09     foreach (operation op ∈ AvailableOutputs)
10       internalOutputs=∅;
11       if (op.result is not pre-bound to a storage)
12         bind op.result
13       destination dst = storage of op.result
14       if ( ScheduleOperation(op, clock ,blk.ss, dst))
15         Roots = Roots – {op} + internalOutputs;
16     clk=clk-1;
```

Figure 4.28. The ScheduleBasicBlock procedure

The *ScheduleOperation* function (Figure 4.29) tries to schedule an operation *op* so that its result is available at *dst* at clock cycle *clk*. If *op* is not pre-bound to a specific functional unit, then the list of functional units that can execute *op* is stored in *F* and sorted by the UnitPriorities (lines 1-4). This priority function depends on the delay of the unit as well as the paths from output of the unit to the destination *dst*. After selecting a

functional unit *FU*, all paths from *FU* to *dst* are stored in *P* and sorted by a PathPriority. The timings of *FU* and a selected path *p* are calculated so that the output of *FU* is available at *dst* at clock cycle *clk* (lines 7-12). If *FU* and all of the resources on the path *p* are not reserved in the *ss.resTable* at the corresponding calculated times, then algorithm tries to schedule the operands of *op* by calling the *ScheduleOperands* function. If the schedule of operands succeeds, then selected functional unit *FU* and path *p* are reserved (operation and interconnect binding) (lines 15-19). We pass a copy of scheduling status (*copyStatus*) to function *ScheduleOperands* to make sure that original status changes only if all operands are successfully scheduled. If scheduling failed after trying all functional units, the *ScheduleOperation* function tries to bind the result of operation to a storage and schedule a read from that storage. If the read succeeds, the operation is added to the *internalOutputs* for later processing.

The *ScheduleOperands* function (Figure 4.30) schedules the operands of an operation *op* on a selected functional unit *FU* so that their values are available on corresponding input ports of *FU* at clock cycle *clk*. If an operand is a variable or a constant, then this function tries to schedule a read from the corresponding storage. Otherwise, it calls the *ScheduleOperation* function. The function succeeds only if all operands can be scheduled.

In the *ScheduleRead* function (Figure 4.31), the best available path that can transfer a value from its storage to the specified destination at clock cycle *clk* is selected and scheduled.

```
00 bool ScheduleOperation(operation op, clock clk, schedule status ss, destination dst)
01   if (op is pre-bound to a functional unit)
02     F = {functional unit to which op is pre-bound};
03   else
04     F = {functional units that implement op sorted by UnitPriorities};
05   foreach(FU ∈ F)
06     P = {paths from FU.output to dst sorted by PathPriorities};
07     foreach(p ∈ P)
08       p.timings.end = clock;
09       calculate p.timings.start;
10       if (resources of p are not reserved in ss.resTable)
11         FU.timing.end = p.timings.start;
12         calculate FU.timing.start;
13         if (FU is not reserved in ss.resTable)
14           copyStatus = ss;
15           if (ScheduleOperands(op, FU.timing.start, copyStatus, FU))
16             ss = copyStatus;
17             reserve FU and p in ss.resTable;
18             add corresponding MAs to ss.MAs;
19             return TRUE;
20   bind op.result;
21   if (ScheduleRead(op.result, clk, ss, dst))
22     internalOutputs = internalOutputs + {op};
23     return TRUE;
24   return FALSE;
```

Figure 4.29. The ScheduleOperation function

```
00 bool ScheduleOperands(operation op, clock clk, schedule status ss, functional unit FU)
01   foreach(operand o ∈ op.operands)
02     destination dst = FU.inputs corresponding to o;
03     if (o is a variable or a constant)
04       if (o is not pre-bound to a storage)
05         bind o to a storage;
06       if (! ScheduleRead(o, clk, ss, dst))
07         return FALSE;
08     else if (! ScheduleOperation(o, clk, ss, dst))
09         return FALSE;
10   return TRUE;
11
```

Figure 4.30. The ScheduleOperands function

```
00 bool ScheduleRead(value v, clock clk, schedule status ss, destination dst)
01   P = {paths from storage of v to dst sorted by PathPriorities};
02   foreach(p ∈ P)
03     p.timings.end = clk;
04     calculate p.timings.start;
05     if (resources of p not reserved in ss.resTable)
06       reserve p in ss.resTable;
07       add corresponding MAs to ss.MAs
08       return TRUE;
09   return FALSE;
```

Figure 4.31. The ScheduleRead function

4.3 Other scheduling algorithms

Because the architecture style of NISC is new, little research has been done on the mapping algorithms for NISC. However, some of techniques developed in the areas of ASIPs, high level synthesis, and retargetable compilers can be directly or indirectly related to NISC. There has been an extensive body of work on scheduling and binding algorithms in the area of high level synthesis and retargetable compilers, which we review in this section.

Force directed scheduling (FDS) [49][50] is commonly used to solve the timed constrained scheduling problem. This algorithm, distributes the execution of similar operations in different clock cycles in order to achieve high utilization of functional units while meeting the time deadline. Path-based scheduling algorithm [55] tries to minimize the number of clock cycles needed to execute the critical paths that exist in the given CDFG. To do so, the algorithm gives emphasis to conditional branching i.e. it starts by extracting all possible execution paths from the given CDFG and schedules them independently. Then the schedules of different paths are combined to generate the final schedule for the whole design. However, the path-based approach restricts the execution order of the operations before scheduling.

List-based scheduling techniques [19] are used to solve resource constrained scheduling problem in which the number of resources of different types are limited. List scheduling processes each state sequentially. At each state, it tries to choose the best operation from the list of candidate operations, subject to resource constraints. List scheduling uses a ready-list, which keeps all nodes that their predecessors are already scheduled. The ready-list is always sorted with respect to a priority function. The priority

function always resolves the resource contention among operations, i.e. operations with lower priority will be deferred to the next or later states. The quality of the results produced by a list-based scheduler depends predominantly on its priority function.

Mobility of the operation, i.e. the difference between ASAP (as soon as possible) and ALAP (as late as possible) times, is commonly used as the priority function in many HLS systems. Different priority functions and heuristics have been proposed to improve the quality of list scheduling. The proposed list scheduling algorithms in [65] and [9] uses mobility as the primary priority functions. To break the tie among a set of available operations with similar mobility, they assign higher priority to those operations that contribute to the same output. Before scheduling begins, they analyze the outputs of operations in the DFG by constructing a set of trees (cones) that start from output nodes as roots. However, they use a conventional scheduler that starts from inputs and proceeds forward, and the output trees are only used to break the tie during schedule. A similar approach is used in [21] and [31] for scheduling on VLIW architectures. Output trees in DFG are also used for instruction selection using the maximal-munch algorithm. Processing the DFG backward, from outputs towards inputs, has proven to be very fruitful. However, this idea has been mainly used in priority functions but not the scheduling algorithm itself.

All of these scheduling algorithms only schedule operations and assume that binding is done later. While theoretically it is possible to support multi-cycle and pipelined operations in HLS scheduling algorithms, none can consider operation chaining as accurate as our algorithm because the information of wire between functional units will not available in traditional HLS until after resource and variable binding. Lack of access

to wire information also prevents traditional HLS from supporting pipelining and data forwarding. Also, all HLS approaches generate normal FSMs and do not support pipelined controller the way our algorithm does.

On the other hand, VLIW and other similar compilation algorithms do not support partial forwarding and pipelining. Only one scheduling approach [8] has considered partial data forwarding. However, this approach can only handle direct forwarding paths, while ours can support pipelined paths as well. Also, their approach requires the behavior of every instruction to be defined in terms of all possible forwarding paths, while in our approach; we only describe the structure of architecture and then extract and analyze the possibilities during scheduling. Finally, supporting multi-cycle, pipelined, or chained operations is outside of the scope of standard compilation algorithms because they do not consider the low-level structural details of the architecture.

Chapter 5. Low-level programming in NISC using C

Languages such as C are generic enough to cover majority of the application needs, but sometimes in applications, the underlying hardware must be controlled directly through special registers or instructions. In instruction-based processors, programmers use assembly code to perform tasks such as peripheral IO operations, configuring the interrupt unit, or use resources with custom functionalities that cannot be expressed in C. Since in NISC, the architecture has no predefined instruction-set, it does not have any assembly code either. This is specially limiting when an application requires interrupt or needs to communicate with other cores in a system. In statically-scheduled architectures, use of microcode for low-level programming requires that the programmer also provide an accurate cycle-by-cycle schedule of the microcodes. This makes direct use of microcodes (a) tedious and error prone, and (b) impractical in C language. This issue has not been addressed in the past and all architectures that use microcode for programming assume that if low-level programming is needed, it will be done manually. Approached such as TIPI do not even have a compiler and assume all programming is done manually.

5.1 *Motivating example*

Consider Figure 5.1 that shows a sample code for finding the maximum of four numbers as well as the corresponding CDFG when compiled on the shown datapath. It is clear from the program and the CDFG that: (1) there are too many states in the CDFG and therefore the code will take several clock cycles to run on the datapath, and (2) the code does not have any parallelism that can be utilized for speeding up the computation.

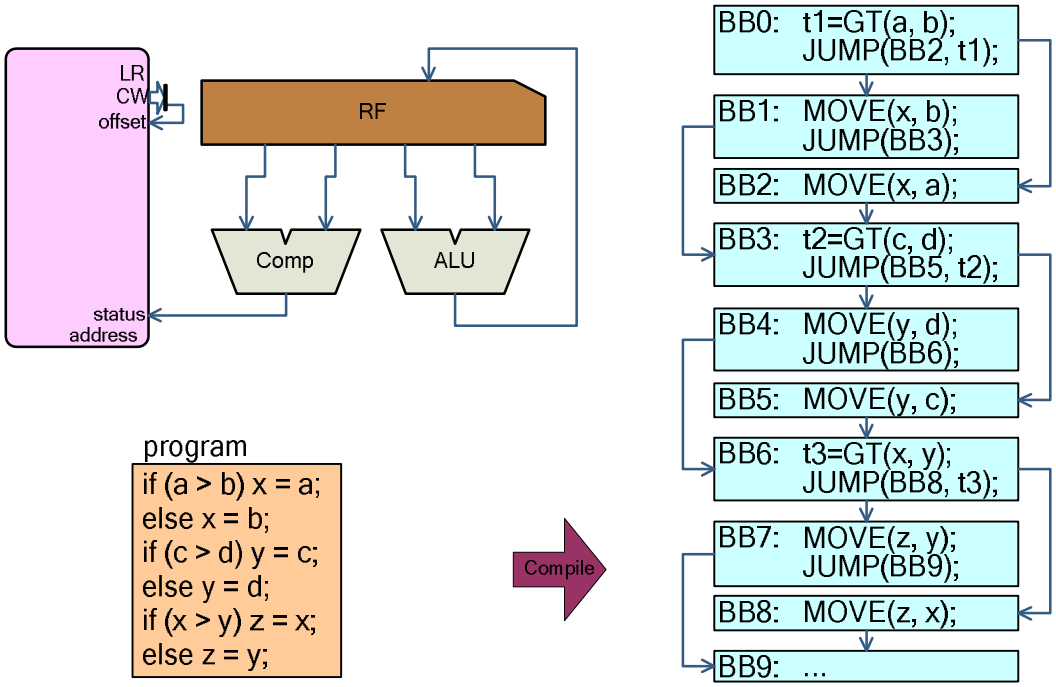


Figure 5.1. Normal code for finding maximum of four numbers

If we look at the code for finding maximum of four numbers in Figure 5.1 we can see that there are three similar conditional clauses that find the maximum of two numbers. We can execute this code much more efficiently if we construct a datapath that has a custom functional unit (called *Max*) for finding maximum of two numbers, as shown in Figure 5.2.

Using the *Max* FU, we can find the maximum of four numbers in just two clock cycles as shown in Figure 5.3. But the question is what should we write in the C code for

the NISC compiler to generate such a schedule? Since NISC does not have any predefined instruction-set, it also does not have any assembly. Therefore, we can not use the assembly to directly program and use the underlying *Max* FU in the datapath of Figure 5.2. Furthermore, any mechanism that provides low-level access in C, should also enable the NISC compiler to freely schedule the custom functions along with other operations if possible.

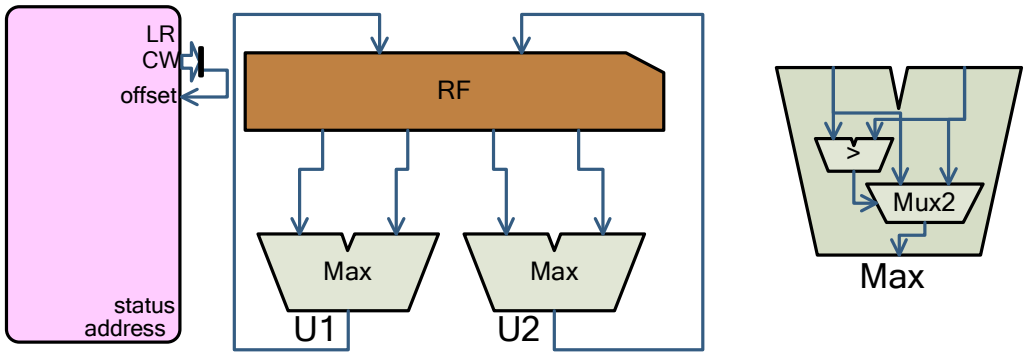


Figure 5.2. Datapath with custom function unit for finding maximum of two numbers

Clock	MAs
0	RF(x)=Max(R(a), RF(b)); RF(y)=Max(RF(c), RF(d));
1	RF(z)=Max(RF(x), RF(y));

Figure 5.3. Finding maximum of four numbers using a custom FU in Figure 5.2

5.2 Providing low-level programming in NISC

To support low-level programming in NISC, we introduce the concept of *pre-bound* functions and variables in the NISC compiler. These functions and variables have common C syntax but instead of implementing them in the normal way, the compiler maps them to specific hardware resources. During code generation, the compiler generates proper control bits to access their corresponding hardware resources.

To better understand the difference between pre-bound functions and normal functions, let's first see how a normal function is executed. Figure 5.4(a) shows the C code of a function `f()` that calls another function `Max()`. During call, two parameters

are passed to `Max` and then one result is returned. Figure 5.4(b) shows the corresponding CDFG of this code, while Figure 5.4(c) shows its stack behavior. As the CDFG shows, in the caller function, first the stack must be extended for the return value and the parameters, then the parameters are pushed on the stack, and finally the execution flow jumps to the beginning of the callee, i.e. *Mux_prolog* block. In the prolog of the callee, the return address and the FP register values are pushed on the stack and the stack is extended by incrementing the value of SP. After executing the body of the function and writing the return value, the SP and FP register values are restored and the execution flow jumps back to the caller. This example shows how normal function calls are implemented. It also shows the relatively high overhead of functions calls.

A pre-bound function is directly mapped to a hardware resource and is treated the same as other operations. Therefore, if we assume that the `Max` function in Figure 5.4(a) is a pre-bound function, then we do not need to specify the body of this function and the CDFG of the program becomes a lot simpler as shown in Figure 5.5(b). In this case, the *Max* "operation" is treated exactly the same way that the *ADD* or *MUL* operations are handled. In this way, we can not only directly use a component in the datapath, but also will have a more efficient execution since the CDFG is simpler and the *Max* operation can be scheduled in parallel with the rest.

Figure 5.6 shows the C code and the corresponding CDFG for finding the maximum of four numbers using the *Max* pre-bound function. Compiling this code on the architecture of Figure 5.2 will automatically result in the schedule of Figure 5.3.

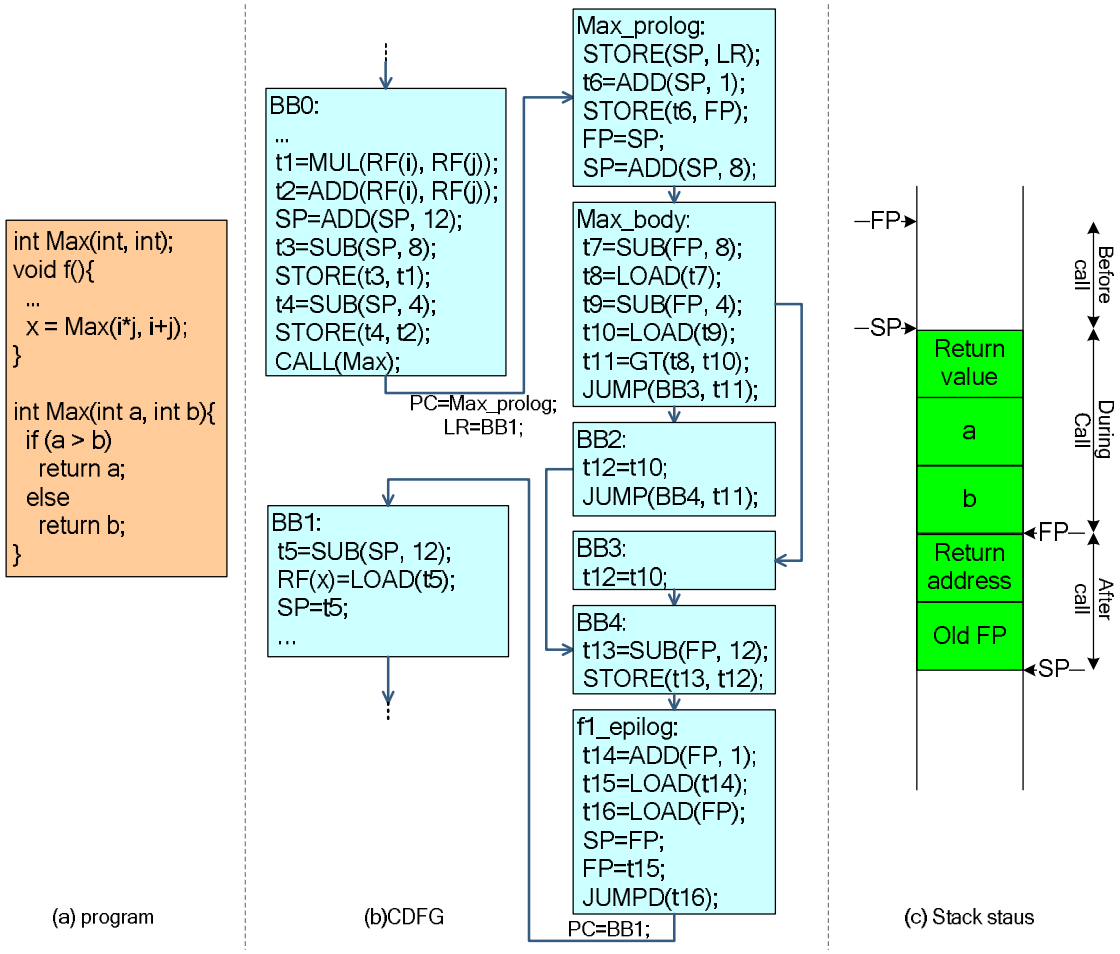


Figure 5.4. Execution of normal function calls

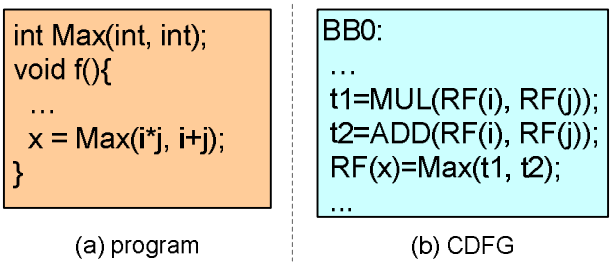


Figure 5.5. Executing a pre-bound function

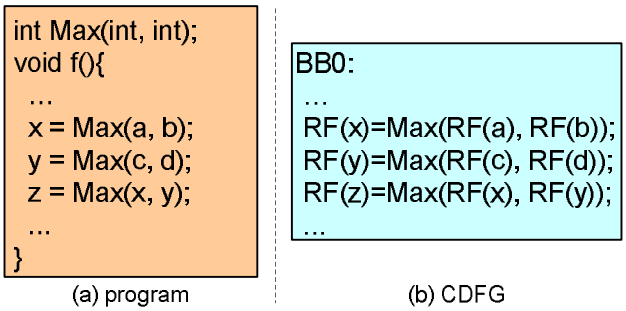


Figure 5.6. Finding maximum of four numbers using *Max* pre-bound function

Note that pre-bound functions are different from intrinsic functions, commonly used in the compilers. Pre-bound functions affect the functionality of the application but have no implementation and are treated similar to other operations. Therefore, they can be scheduled in parallel with other operations and with each other. On the other hand, the intrinsic functions are implemented in the same way as other normal functions, i.e. inlined or jumped to. But since the compiler has a built-in knowledge of how the intrinsic functions behave, it can optimize them more than normal code. Also, some intrinsic functions only provide hints to the compiler (e.g. for optimizations) but have no implementation or have no effect on the program.

5.3 Pre-bound functions in GNR and C

As we mentioned in Section Chapter 3, the NISC architecture is described in GNR format. In the model of the architecture, we describe pre-bound functions for functional units the same way that their operations are defined. The description also maps the function output and parameters to the ports of the component and specifies the timing and corresponding control bit values. We also specify whether the scheduler can freely move the function and schedule it with other operations, or it should preserve the order of the function with respect to operations that appear before and after it in the code. For example, the *Max* pre-bound function, discussed in Section 5.2, can be freely moved and scheduled with other operations in the program because it does not store any internal state or change the state of the architecture. As another example, assume we have a *Push* and a *Pop* pre-bound function that are mapped to a hardware queue. The hardware implementation of these functions changes the state of the architecture, i.e. add a value in

the queue or remove one from it. Therefore, the execution order of these functions in the code must be preserved by the compiler otherwise the result would be wrong.

To support pre-bound functions and variables, we added a new tool, *PreboundCGenerator*, to the flow of Figure 2.4. The new flow is shown in Figure 5.7. Before compiling the application on the given datapath, the *PreboundCGenerator* tool processes the architecture description and generates a C header (.h) and source (.c) file that contains the declarations of the pre-bound variables and functions. For every register in the datapath (including registers in the register-file) a variable is declared in the generated source file, the function descriptions of the functional units are also translated to proper C function declarations. The tool also provides this information to the NISC compiler so that it knows which functions and variables are pre-bound to what hardware components. The generated source files are included in the application and the programmer can use them the same way they are normally used in C. During compilation, instead of binding variables to global memory, or stack, they are bound to their corresponding registers. Similarly, instead of implementing calls to pre-bound functions with *jump* operations, these calls are treated the same way that for example an *add* or *multiply* operation is treated.

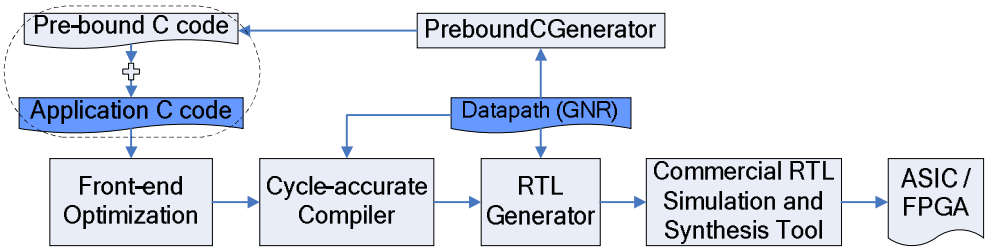


Figure 5.7. NISC tool flow with pre-binding

Figure 5.8 shows the GNR of the *Max* unit shown in Figure 5.2. This description defines *Max* as a hierarchical module and assigns compiler aspect to it. As we explained

in Section 3.3, when the compiler loads this component, it uses the information in the compiler aspect and will not process the internal netlist of the module. The internal netlist specifies the actual implementation of this unit based on the GNR description of other components in the library. This information is used by the RTL generator tool. Instead of the netlist, we could have a Verilog aspect that directly describes the implementation of the component in Verilog HDL. In the compiler aspect of this component, a *Function* (rather than an *Operation*) is defined that maps the function name *Max* to the input and output ports of the module. The description also specifies that this function has no state dependency and hence the compiler can safely move it and schedule it in parallel with the rest of the code.

After running the *PreboundCGenerator* (Figure 5.7) on the GNR of the description of Figure 5.2, it will generate the proper .h and .c files that can be include in the application project for using the pre-bound functions. Figure 5.9(a) shows the header file (.h) that includes the declaration of the *Max* pre-bound function. Note that in this file, in addition to the *Max* function, there are two other function declarations that are the same as *Max* but include the instance name of the unit in the datapath. These functions are bound to specific instance of the unit. If the *Max* function is used in the program, then the compiler can choose either *U1* or *U2* units for scheduling the *Max* pre-bound function. Otherwise, if any of the `__$U1_Max` or `__$U2_Max` functions are used then the compiler will schedule the function on the corresponding component instance. Figure 5.9(b) shows the source file (.c) that includes the definition of the pre-bound functions. These function definitions are only used to allow the standard C front end of the NISC compiler correctly compile the whole program. However, the bodies of these functions

are not used by the NISC compiler and will be ignored. The *PreboundCGenerator* can also choose to generate a valid body for the pre-bound functions that describe their actual behavior and can be used for debugging or simulation of the C code.

```
<Module type="Max">
  <Ports>
    <InPort n="i0" bitWidth="32" />
    <InPort n="i1" bitWidth="32" />
    <OutPort n="o" bitWidth="32" />
  </Ports>
  <Netlist>
    <Components>
      <Instance n="c" type="GreaterThan" lib="Lib" />
      <Instance n="m" type="Mux2" lib="Lib" />
    </Components>
    <Connections>
      <Conn src="" srcPort="i0" dest="c" destPort="i0" />
      <Conn src="" srcPort="i1" dest="c" destPort="i1" />
      <Conn src="c" srcPort="o" dest="m" destPort="sel" />
      <Conn src="" srcPort="i0" dest="m" destPort="i0" />
      <Conn src="" srcPort="i1" dest="m" destPort="i1" />
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Functions>
      <Function n="Max" stateDependency="none" delay="0">
        <Output port="o"><Type n="int" /></Output>
        <Input port="i0"><Type n="int" /></Input>
        <Input port="i1"><Type n="int" /></Input>
      </Function>
    </Functions>
  </Annot_compiler>
</Module>
```

Figure 5.8. GNR of description of *Max* unit shown in Figure 5.2

```
void Max(int, int);
void __$U1_Max(int, int);
void __$U2_Max(int, int);
```

(a) .h file

```
void Max(int i0, int i1) {return 0;}
void __$U1_Max(int i0, int i1) {return 0;}
void __$U2_Max(int i0, int i1) {return 0;}
```

(b) .c file

Figure 5.9. Generated pre-bound C codes

5.4 Benefits of pre-bound functions and variable

While providing similar capabilities, our pre-binding approach is more flexible than using assembly in instruction based processors. The pre-bound constructs have C syntax and can be merged with the rest of the application much easier than assembly code. Also, the programmer does not need to worry about the scheduling of these constructs.

ASIP approaches such as LISA [48] capture the description of custom instructions in ADL and then generate assembler. Therefore, to access low level resources the program should use assembly of custom instructions. In Tensilica [70], the base processor is extended by adding custom instructions. Each custom instruction is defined in their TIE proprietary language and must be explicitly used in the C code via special function calls. This idea is similar to our pre-bound functions. However, in contrast to our approach, Tensilica's custom instructions (and their corresponding functions) have only one possible implementation. They cannot represent several similar resources, or be scheduled (and potentially moved) the way pre-bound functions are handled in NISC compiler.

All other statically-scheduled architectures (i.e. microcoded or VLIW) require the programmer to explicitly provide the schedule of the microcode or assembly. In NISC, the compiler schedules the calls to pre-bound functions and accesses to pre-bound variables. Therefore the programmer uses these functions and variables exactly the same way that normal C functions and variables are used.

In NISC, the main goal is to develop the application in an architecture independent high-level language (e.g. C) so that it can be mapped on different custom architectures. Another benefit of our pre-binding approach is that a C code using pre-bound functions or variables can execute on any architecture as long as that architecture contains the corresponding hardware resources. In this way, the backward compatibility can be maintained at source code level without imposing as tight constraints as backward binary compatibility requires.

Chapter 6. Interrupt handing

In traditional microcoded processors, the microcode or nanocode was used inside the processor to implement the instructions of the instruction-set. In other words, the instructions, rather than the microcode, would define the processor's external behavior seen by the programs. The instruction abstraction (a) enables backward binary compatibility, (b) simplifies low-level programming through assembly, and (c) defines fine-grained intervals where interrupts could be handled by the processor. In contrast, in NISC nanocode and in MIMOLA [63], TIPI [68], PICO [36][61], and ARM OptimoDE [45], [35] microcode is used instead of instructions to execute the program. In these techniques all of the aforementioned benefits of instruction abstraction are lost. In embedded and custom processors, backward binary compatibility is not as important as it is in the general-purpose processors. However, interrupt and assembly programming are necessary features. For example, developing different communication protocols rely on interrupts and low-level access to the hardware.

All approaches that use nanocode or microcode for programming are statically-scheduled architecture. In statically-scheduled pipelined architectures, different stages of

execution of an operation (e.g. read, execute, write-back) are implemented with several micro-operations. The overlapping execution stages of different operations are combined in micro-instructions which determine the control-word (CW) for each clock cycle. As a result, execution of micro-instructions cannot be arbitrarily interrupted; otherwise, the interrupt routine may need to store/restore datapath registers in addition to the registers of the register-file. A safe and efficient interrupt mechanism is needed in statically-scheduled pipelined architectures before they can be used in embedded systems. PICO and OptimoDE are designed as co-processors only and hence interrupts are assumed to be processed by the main processor. MIMOLA and TIPI have not considered interrupt problem at all.

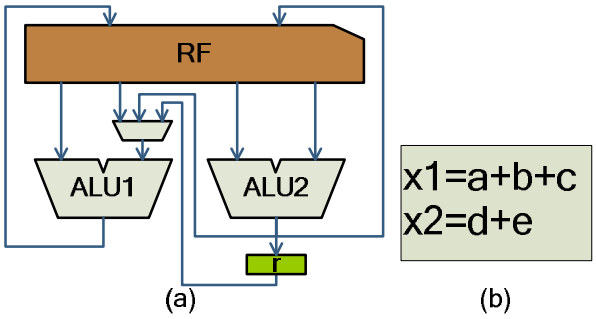


Figure 6.1. (a) Sample datapath, (b) sample code

6.1 Challenge of interrupt support in NISC

While other microcode based approached focus only on single-cycle operations, in NISC operation chaining (sub-cycle operations) and multi-cycle operations are also supported. Consider the datapath of Figure 6.1(a) that is used to compile the set of expressions shown in Figure 6.1 (b). Depending on the clock frequency of the system and the delay of the components, the NISC cycle-accurate compiler can choose to chain two operations in one cycle or execute one operation over multiple cycles. Assume that clock period of the system is *T*, delay of *ALU1* is *d1*, and delay of *ALU2* is *d2*. Also assume that

ALU2 is slower but consumes less power (*d1* < *d2*). Depending on the values of *T*, *d1*, and *d2* three cases are possible:

- If *d1* < *T* and *d2* < *T* but *T* < *d1+d2*, then each operation must be scheduled in one cycle and intermediate data must be stored in the register-file or datapath register *r* (Figure 6.2(a)).
- If *d1+d2* ≤ *T*, then two operations can be chained in one cycle and register-file is accessed only once for writing back the final results (Figure 6.2(b)).
- If *d1* < *T* < *d2*, then the faster *ALU1* can be used to execute two operations in two consecutive cycles while the slower *ALU2* executes the other operation in two cycles (Figure 6.2(c)).

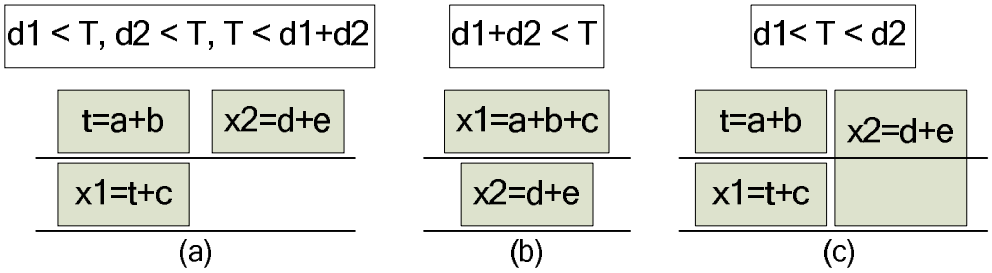


Figure 6.2. (a) single-cycle, (b) chained, (c) multi-cycle operations

As this example illustrates, in NISC the datapath can be utilized very efficiently because the compiler has complete control over it. While instruction-set based compilers are mainly concerned with performance, the NISC compiler can also consider other design parameters such as timing and power consumption of individual datapath components. However, as mentioned before, this architectural style introduces new challenges for supporting interrupts.

6.2 Adding interrupt handling to NISC

In traditional processors, the interrupt is checked between every two instructions. The execution flow can be interrupted between instructions because all instructions store

their result back to the register-file. Therefore, the interrupt routine may only need to store/restore the value of registers in the register-file in its prolog/epilog.

In NISC, the intermediate results of operation may be stored in the internal registers of the datapath. Furthermore, an operation may take more than one cycle (e.g. (Figure 6.2(c)) and hence span across multiple CWs. Therefore, in NISC the execution flow cannot be interrupted between any two arbitrary CWs. Detecting the dependencies between CWs at run time is very difficult (if not impossible). Also, in addition to the registers of the register-file, an interrupt routine may need to store/restore the intermediate registers of the datapath as well.

To address this problem, we need to find an easily identifiable location in the program where execution flow can be safely interrupted. The boundary of basic blocks is a good candidate for this purpose. A basic block is a sequence of operations that always execute together. The execution sequence of basic blocks of the program is data or control flow dependent. Consequently, every basic block *must* read its inputs from memory or register-file and *must* write its outputs back to memory or register-file. In other words, since execution of operations of a basic block cannot depend on the intermediate datapath values of other basic blocks, the interrupt can be safely serviced at the end of basic blocks. In fact, one of the goals of NISC is to execute each basic block *as if* it was executed with one custom instruction. Based on this observation, the controller of NISC checks for interrupts only when bits corresponding to jump operations are set, i.e. at the end of basic blocks. After a jump operation, the execution flow goes to the target of the jump or an interrupt routine. In presence of an interrupt, the target of the original jump is passed to the interrupt routine as its return address. Note that this scheme

also simplifies the implementation of atomic functionalities because the programmer can now count on atomic execution of basic blocks.

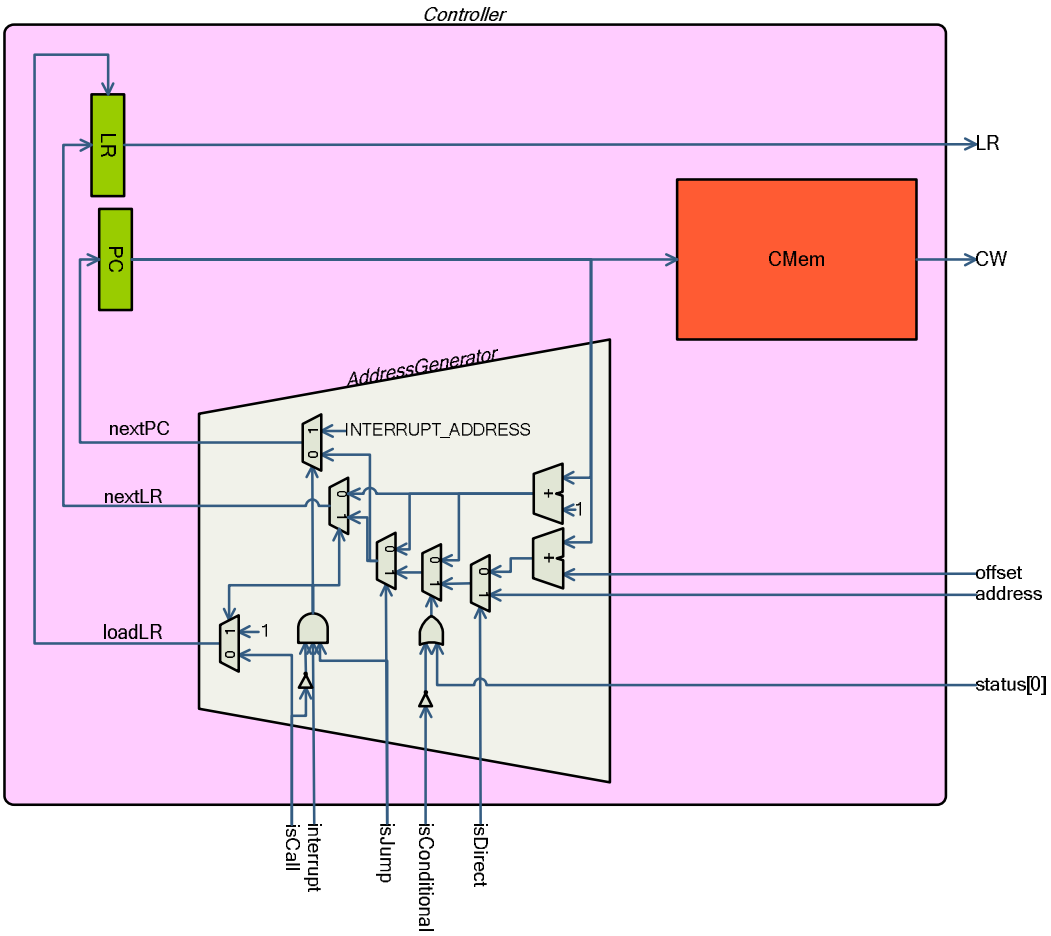


Figure 6.3. Updated controller for supporting interrupt

Figure 6.3 shows the updated controller of Figure 2.2 for supporting interrupt. In this new controller, an *interrupt* bit indicates whether there is a pending interrupt. The AND gate in the *AddressGenerator* causes the execution flow to jump to the interrupt service routine (ISR) only if (a) there is a pending interrupt, (b) end of basic block has reached, i.e. a jump operation is being executed, and (c) the jump is not because a *call* operation.

To better understand how the interrupts are handled in NISC, we first need to understand how a function call is executed. Figure 6.4 shows the CDFG and execution

flow when a function is called. In the caller, first the parameters of the callee function are pushed on the stack and then execution flow jumps to the beginning of the callee function (e.g. basic block *BB0* in Figure 6.4). A call operation is the same as a *jump*, but it also loads the next PC value in the LR register before jumping to the target address. In this way the LR register holds the return address immediately after a *call* operation. Every function has at least three basic blocks: (i) a prolog block that saves the return address (i.e. the LR register value) on the stack as well as the value of registers that will change in the function, (ii) a block as the starting point of the main body of the function, and (iii) an epilog block that restores the value of modified registers and jumps back to the return address. After the call, the caller function (basic block *BB1*) pops back the parameters and return value from the stack before it continues its execution.

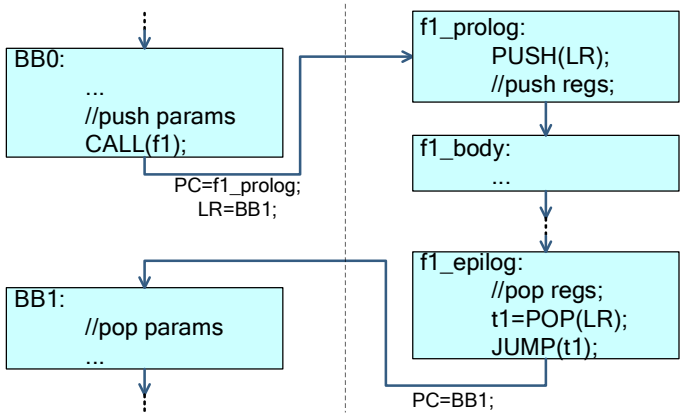


Figure 6.4. CDFG of a typical function call

The interrupt is very similar to a function call, but rather than using a call operation in the program, the jump to the ISR is initiated in the controller when the interrupt signal becomes '1'. Figure 6.5 shows how ISR is executed at the end of a basic block when interrupt signal is enabled. The ISR has no parameters and does not return any value, therefore, there is no need to parameter push/pop in the main execution flow. The modified controller structure in Figure 6.3 also guarantees that the target of the jump

address is stored in the LR before going to ISR. After ISR is finished, it returns to the block where the original jump was supposed to go to.

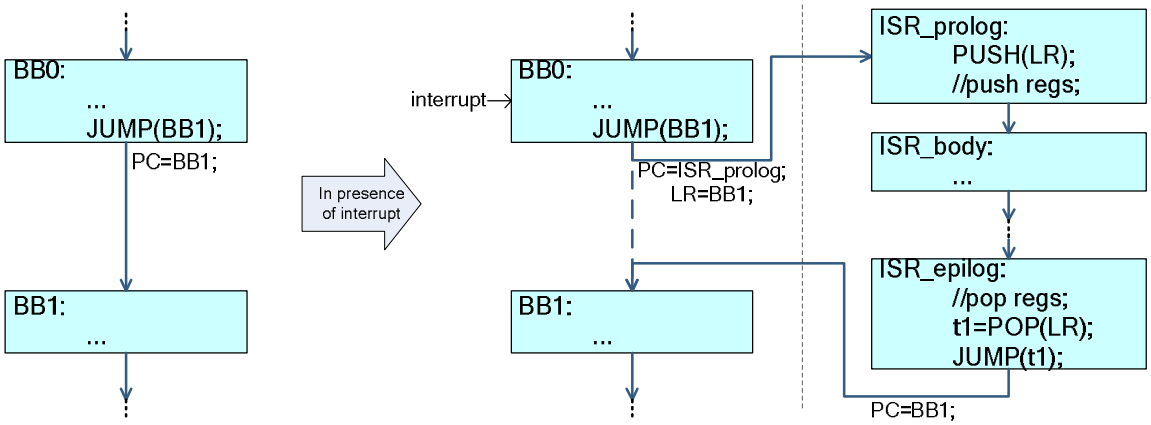


Figure 6.5. Interrupt exaction after a *jump* operation

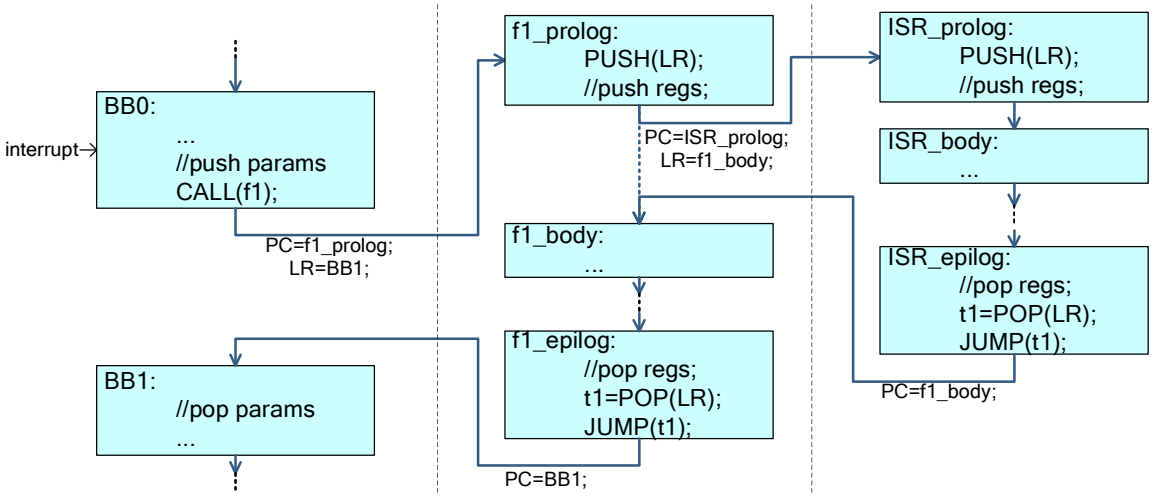


Figure 6.6. Interrupt execution after a *call* operation

Since both interrupt service routine and the normal function call depend on the value LR register, we should make sure that these two mechanisms do not interfere with each other. To simplify the implementation, if interrupt is enabled right before a function call, we first execute the call and then process interrupt. In this way, the prolog of the function stores its return address and will no longer need the value of LR. Then the interrupt mechanism in the controller can safely overwrite the value of LR. After interrupt handler is finished, the execution flow returns back to the called function. This mechanism is

illustrated in Figure 6.6. This simplification minimizes the required changes to the controller for implementing interrupts. Additionally it enables (a) use of normal function calls in the ISR, and (b) support of nested interrupts.

Usually a processing element must support multiple interrupts. The interrupt signals are not synchronized with the clock and may come at any time. Furthermore, an interrupt signal may be deactivated before it is processed. Therefore, we need a hardware mechanism to catch all interrupts when they come and also can handle multiple interrupts. One option is to add such mechanism to the controller; but this jeopardizes the NISC philosophy. In NISC we want to be able to customize and remove as much unused resources as possible. Since the designer can freely customize the datapath, we put the interrupt catch mechanism in an interrupt unit (IU) in the datapath and connect it to the interrupt port of the controller. If no interrupt support is needed in the design, the IU is removed from datapath and the interrupt port of controller is connected to "0" (grounded). When we connect the interrupt port to "0", the AND gate and the multiplexers who are controlled with this AND gate will be removed during logic optimization. In this way, the controller of Figure 6.3 becomes exactly the same as the original controller of Figure 2.2. With this approach, the interrupt handling hardware is added to the design only if the interrupt support is required. In the next section, the details of the IU and its usage are explained.

6.3 The interrupt unit (IU)

Figure 6.7 shows the internal implementation of Interrupt Unit (IU). Each interrupt connected to the asynchronous-set port of a flip-flop which latches that interrupt. The flip-flop also has a synchronous-reset that allows the programmer to clear the flip-flop by

provide an interrupt number on the port *i* and setting the *clearInterrupt* control port. There is also a *mask* register that has one bit for every interrupt and if the bit is 0 then the corresponding interrupt is disabled. Therefore, writing an integer 0 in the mask disables all interrupts. A priority encoder determines the interrupt number of the highest priority activated interrupt. Finally, an OR gate generates a notification signal for the controller indicating that at least one interrupt is available for processing. The input (*i*) and output (*o*) ports of the UI are connected to the datapath.

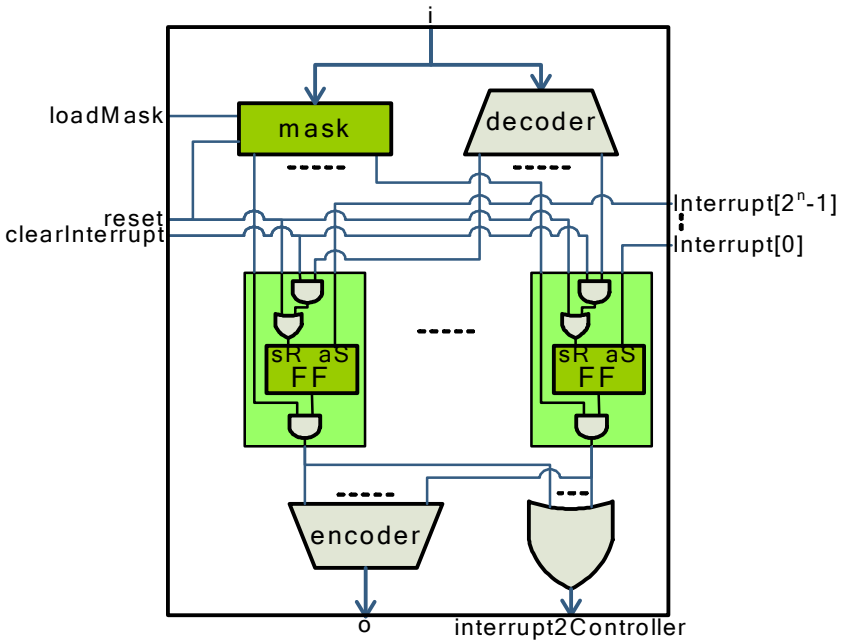


Figure 6.7. The structure of Interrupt Unit

Figure 6.8 shows the GNR code of the IU that has thee pre-bound functions, i.e. *setMask*, *clearInterrupt*, and *interruptNumber*. The component has a set of input, output and control ports. Function descriptions specify the mapping between their inputs/output and the input/output ports of the component. The description also determines the control values that must be assigned to corresponding control ports for execution of the function. The functions in this example indicate *stateDependency*="*all*". This means that the

compiler must preserve the order of operations before and after these functions during scheduling. These pre-bound functions provide all the means for directly controlling the IU from C code.

```
<FU type="InterruptUnit">
  <Params>
    <Param n="BIT_WIDTH" />
    <Param n="INTERRUPT_COUNT" />
    <Param n="DELAY" val="0"/>
  </Params>
  <Ports>
    <Clock n="clk" bitWidth="1" />
    <InPort n="reset" bitWidth="1" />
    <CtrlPort n="clearInterrupt" default="0" bitWidth="1" />
    <CtrlPort n="loadMask" default="0" bitWidth="1" />
    <OutPort n="interrupt2C_ntroller" bitWidth="1" />
    <InPort n="interrupts" bitWidth="{@INTERRUPT_COUNT}" />
    <InPort n="i" bitWidth="{@BIT_WIDTH}" />
    <OutPort n="o" bitWidth="{@BIT_WIDTH}" />
  </Ports>
  <Annot_verilog><!--determines the implementation info. --></Annot_verilog>
  <Annot_compiler>
    <Functions>
      <Function n="setMask" delay="{@DELAY}" stateDependency="all">
        <Input port="i"><Type n="unsigned char" /></Input>
        <Ctrl port="loadMask" val="1" />
      </Function>
      <Function n="clearInterrupt" delay="{@DELAY}" stateDependency="all">
        <Input port="i"><Type n="unsigned char" /></Input>
        <Ctrl port="clearInterrupt" val="1" />
      </Function>
      <Function n="interruptNumber" delay="{@DELAY}" stateDependency="all">
        <Output port="o"><Type n="unsigned char" /></Output>
      </Function>
    </Functions>
  </Annot_compiler>
</FU>
```

Figure 6.8. The GNR code for an Interrupt Unit (IU)

As we explained in Section 5.3, the *PreboundCGenerator* tool generates a head code that includes the declaration of the pre-bound functions in the GNR. Since the pre-bound functions of the IU all have state dependency, they can be used only for specific component instance. In NISC, we define a main interrupt handler routine that is called for all interrupts. The typical C code of this function is shown in Figure 6.9. in the *interruptHandlerMain* function, the pre-bound functions are used to control the IU. In this

function first the current interrupt number is read, and then all interrupts are disabled by setting the mask to 0. After handling an interrupt, its corresponding latch in the IU is cleared and all interrupts are enabled again by setting all bits of the mask register to 1.

```
void interruptHandlerMain()
{
 int iNum = __$IU_interruptNumber();
 __$IU_setMask(0);
 //handling the interrupt
 switch(iNum) {
  case 0: /*handling interrupt 0*/ break;
  case 1: /*handling interrupt 1*/ break;
    ...
 }
 __$IU_clearInterrupt(iNum);
 __$IU_setMask(-1);
}
```

Figure 6.9. Sample C code for using pre-bound functions of IU

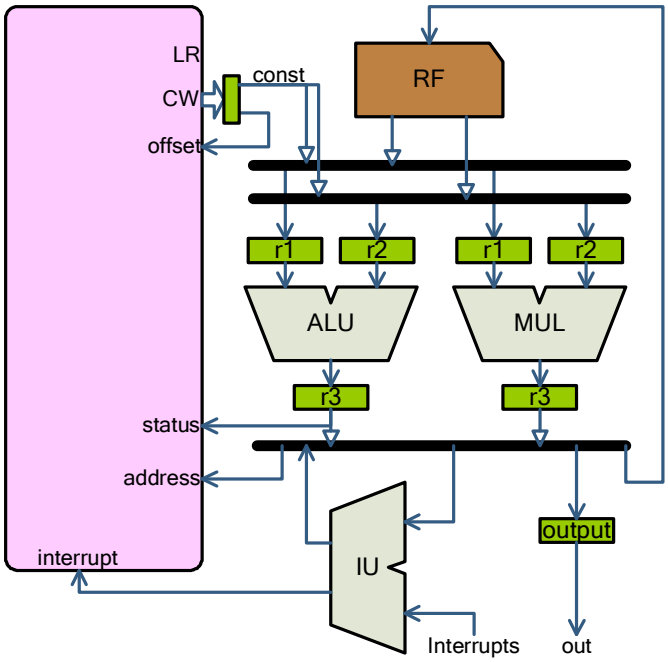


Figure 6.10. Sample datapath for pre-binding

Figure 6.10 shows an example of using the IU in a datapath. In any case, each bit of the *interrupt* port of IU is connected to the interrupt signals and its *interrupt2Controller* port is connected to the *interrupt* port of the controller. The input / output ports of IU (i.e. ports *i* and *o*) are connected to the datapath. The control ports of the IU, i.e. *loadMask*

and *clearInterrupt* are connected to control word. Whenever a pre-bound function is used, the compiler determines the correct values of the control signals and schedules them in proper clock cycle.

6.4 Analysis of NISC interrupt handling approach

So far we showed how NISC handles interrupts in between basic blocks of the program. The only concern is that servicing the interrupt only between basic blocks may increase the overall interrupt service delay if the basic blocks are very large. There are two contributing factors to the interrupt service delay: (1) interrupt latency, i.e. the time between when the interrupt is activated and when the execution flow is transferred to the interrupt service routine (ISR); and (2) the delay of ISR itself, i.e. the time it takes to execute the code in the ISR.

In our proposed approach, the size of basic blocks in the running application can affect the interrupt latency. To examine this effect, we ran a series of embedded benchmarks on a generic architecture (GN) shown in Figure 6.11. The benchmarks include *qsort*, *dijkstra*, *sha*, *adpcm.coder*, *adpcm.decoder* and *crc32* from MiBench (the free version of EEMBC embedded benchmarks available at [40]), and a fixed-point Mp3 decoder (more than 10,000 lines of C code available at [42]). We generated the RTL Verilog code of the design and used Xilinx ISE 8.1 toolset for simulation and synthesis of the results. We synthesized the GN (Figure 6.11) on a Xilinx Virtex4 (90-nm) FPGA package and achieved a clock frequency of 80 MHz. The Xilinx toolset also provides a soft-core 32-bit RISC processor (MicroBlaze) that is already optimized Xilinx technology. On a Vertix4 FPGA package, MicroBlaze runs at 105 MHz. MicroBlaze core comes with specific fine-grained timing constraints that direct the synthesis tool to

achieve the highest possible clock frequency. For synthesizing GN we only used a general clock constraint and we expect that the clock frequency of GN can be further improved by using more specific constraints. In any case, the achieved 80 MHz clock frequency for GN seems to be reasonable enough to be used in our calculations.

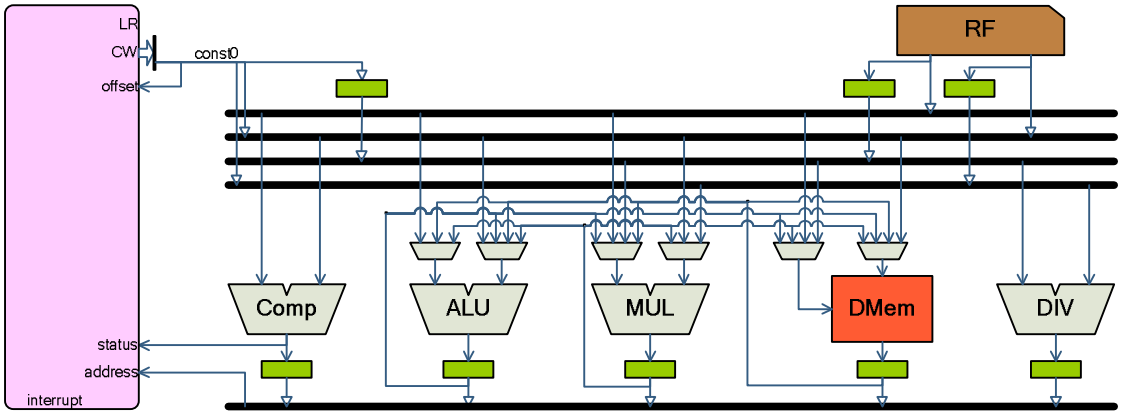


Figure 6.11. A generic NISC Architecture (GN) used for analyzing size of basic blocks

Figure 6.12 shows the distribution of number of basic blocks that take less than 100 clock cycles to execute. The first column in this figure shows the number of basic blocks that take 0 to 9 cycles to execute; the second column shows the number of basic blocks that take 10 to 19 cycles, and so on. It is clear that in these benchmarks, the majority of basic blocks take between 10 to 30 cycles. In other words, if we service interrupts in between basic blocks, most of the time the interrupt latency will be less than 0.5 µ sec (=50 cycles / 80 MHz).

Figure 6.13 shows the distribution of number of basic blocks that are longer than 100 cycles. Overall, there are 13 basic blocks in all of the benchmarks that are longer than 100 cycles. In general, although large basic blocks are rare in applications, in cases where interrupt delay is critical, the compiler can break large basic blocks into a sequence of smaller blocks whose size is determined by the frequency of the interrupts or the upper

bound of their delay. Note that large basic blocks are typically the result of techniques that improve the operation-level parallelism of the code. The compiler can break large blocks into smaller ones after or during operation scheduling without negatively affecting the utilization of parallelism. Compiler can also enable interrupt handling after fall-through basic blocks (not ending with a jump) by adding a jump to the next block.

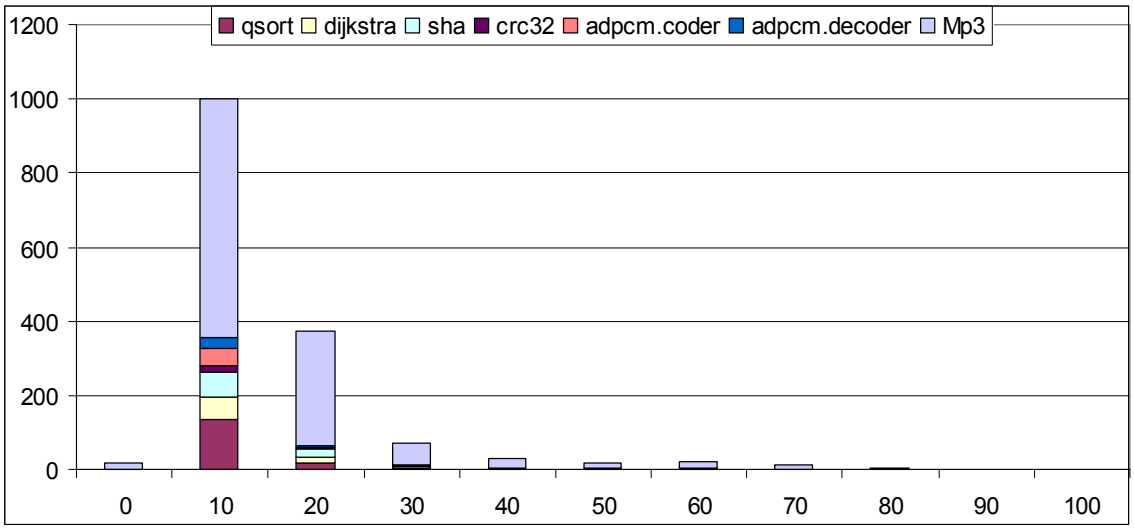


Figure 6.12. Distribution of basic blocks shorter than 100 cycles

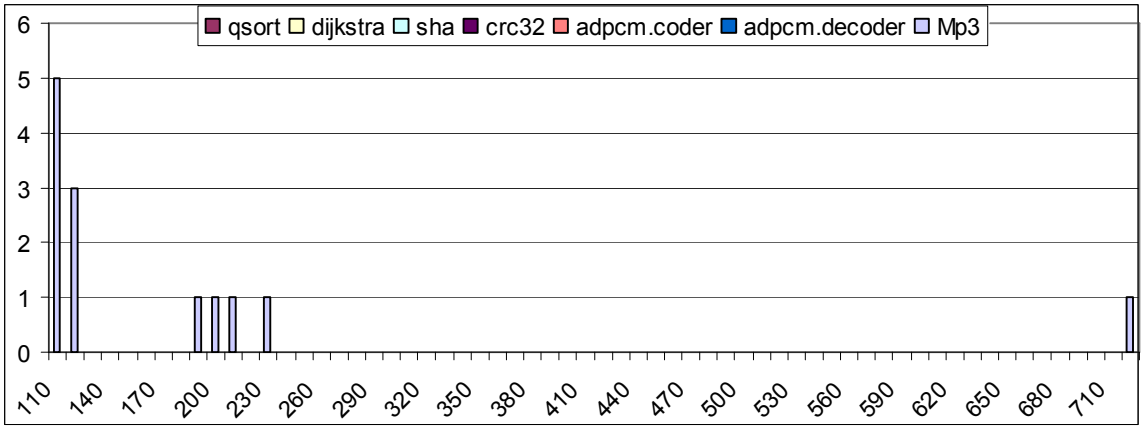


Figure 6.13. Distribution of basic blocks longer than 100 cycles

A more important factor in servicing interrupts is the ISR execution delay. We ran the aforementioned benchmarks on both MicroBlaze and GN to compare their performance. On average, the benchmarks ran 5 times faster on GN than MicroBlaze. We

believe the performance of a typical ISR routine benefits similarly from execution on GN. Additionally, in NISC, we can customize the architecture to further improve the performance of particular piece of code, including an ISR.

The above experiments show that by processing interrupts in between basic blocks, NISC and other statically-scheduled architectures can handle interrupts almost as efficiently as their RISC counterparts.

Chapter 7. Communication case studies

In previous chapters we addressed different issues for designing a single IP block using NISC Technology. However, rarely a single component is enough for a real life application. Therefore, we have to make sure NISC based IPs can communicate with the rest of the system. In this chapter, we show that solving the main three problems; i.e. compilation, low-level programming, and interrupt support; is *necessary* and *enough* for supporting *any* communication protocol, once we know how to model cycle-accurate behaviors in NISC. This is an important benefit of NISC approach (and this thesis) because we do not need to change the controller, the compiler, or even the C language to support any communication scheme. In the rest of this chapter, we explain how to model cycle-accurate behaviors in NISC. We show how any communication protocol can be added to a NISC component and illustrate the approach on different typical communication schemes.

As size of transistors shrink, delay and power of interconnects become more dominant. Therefore, communication among components in a system becomes more costly. To reduce communication cost, designers must explore different communication

architectures and protocols to find the best combination for a particular application. In such design methodology, the processing elements (IPs) must be flexible enough to adapt to different communication paradigms (i.e. architecture and protocol). Such flexibility is also the key feature for enabling IP reuse across different platforms. Therefore, it is important that we do not fix the communication protocol of NISC components and be able to support any possible communication scheme.

Communication protocols are often defined by an accurate timing diagram of events on a set of involved signals. Both sender and receiver must comply with the required events and their timing in order to successfully communicate. For IPs described at Register-Transfer-Level (RTL), the timing of internal behavior of the IP is completely known for the designer. Therefore, the communication protocol can be combined with the description of the IP.

However, the problem becomes more challenging in design methodologies that use un-timed high-level languages (e.g. C) to generate the RTL description automatically. In such methodologies, the IP behavior is un-timed and hence, it is difficult to combine it with the timed behavior of a communication interface.

In the processor domain, this problem is solved by adding a Communication Interface (CI) unit to the processor architecture, and programming it in low-level assembly language using IO instructions. One problem with this approach is that the IO instructions (and hence the communication protocol) is tightly integrated in the processor's instruction decoder, controller, and its pipeline. Therefore, changing the communication protocol requires complex changes in the processor. On the other hand, this approach is not directly applicable to C-to-RTL methodologies because they do not

have any instruction-set or assembly language. To address this issue, some synthesis tools [39][61] limit the interfaces to simple queues and registers, and require special variable-naming convention in the C code. Other synthesis tools [23], seek the solution in going beyond the C language and using SystemC for describing the interfaces. This requires relatively significant modification in the code. Furthermore, synthesizing SystemC is more complex and is possible for only a very limited subset. Enforcing the synthesizable subset is very challenging as well. Ideally, the goal is to develop IPs in a high-level language and easily connect them to any communication protocol. In this chapter, we show how NISC Technology can be used to achieve this goal without requiring any extension to the C language and merely relying on the solutions presented in the previous chapters.

7.1 Adding a communication protocol to NISC

On-chip communication protocols differ from each other based on the underlying network topology and the way they handle *synchronization*, *arbitration* and *data transfer*. *Synchronization* is referred to the mechanism used by producer to notify the consumer about data being ready. Synchronization is usually carried out through interrupt or polling flags. *Arbitration* means how to resolve conflicting requests for shared communication resources. Prioritizing the requests or time-multiplexing them are two well-known arbitration mechanisms. There are different ways for transferring data: commonly used methods include memory-mapped, DMA and direct transfer.

To add any communication protocol to a NISC component, we partition it into two parts: (a) a synchronous part that *must* be timed, and (b) an asynchronous part that *can be* un-timed. We describe the timed part of the protocol in RTL and model the un-timed part

in C. As Figure 7.1 illustrates, the un-timed part of the protocol uses pre-bound functions to reference (i.e. control or poll) the timed behavior; and the timed part of the protocol uses interrupt to notify the un-timed behavior about and event.

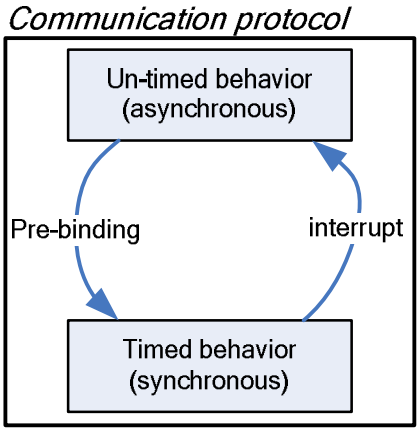


Figure 7.1. dividing a protocol to un-timed and timed behaviors

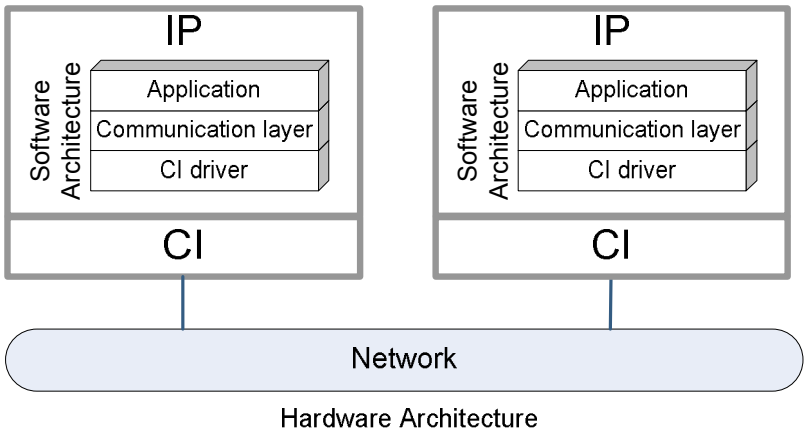


Figure 7.2. Software and hardware architecture of an IP

Figure 7.2 shows the software and hardware architecture of two IPs plugged into a network. The software part includes the behavior of the application, *communication layers* and the CI driver. In general, *communication layers* are used to properly assemble or disassemble packets of data. Depending on the complexity of the communication protocol, size and complexity of the communication layer varies.

Both software and hardware must follow a particular protocol before an IP can be plugged to a new network. On the hardware side the CI unit is replaced with the new one, while on the software side the communication layers and CI driver must be updated. If communication layers are properly used to separate the application from the driver, then the application remains intact. Otherwise, the application must also be modified.

7.2 Case studies: communication interfaces for NISC

To add a communication interface to a NISC, we first need to identify the synchronous and asynchronous parts of a given protocol. Then, we implement the synchronous part in HDL languages, and the asynchronous part in software. The software and hardware parts are ultimately integrated using pre-bound functions. This section shows the concept using three examples: In the first example, a simple point-to-point single-word interface is designed. In the second and third examples interface of shared queue and bus are designed, respectively. In the first two examples, the protocol has not specific synchronous part and therefore, there is not need to specify anything in HDL and we can just use the pre-bound functions to control the underlying hardware. In the last example, the protocol involves cycle-by-cycle behavior as well. This behavior is modeled in standard HDL (in our case Verilog). The software uses pre-bound functions to control the underlying hardware; and the hardware uses interrupt to notify the software about its status.

7.2.1 Point-to-point single-word interface

The simplest way of communicating between two IP is through a register. To communicate correctly, the producer and consumer must synchronize with each other when a data is placed in the shared register or when it is consumed. To do so, one well-

known way is to use *Ready* and *Ack* signals: the producer places the data into the register and notifies the consumer by raising the *Ready* signal. The consumer reads the data and raises the *Ack* signal. Once the producer receives the acknowledgement, it lowers the *Ready* signal. Next, the consumer lowers the *Ack* signal to indicate the end of communication. If more than one word must be transferred, the same sequence of signals is repeated for each word. This protocol is called double-handshake because producer and consumer synchronize with each other once before and once after transferring data. Since, this protocol is asynchronous, producer and consumer may operate at two different clock frequencies. Such asynchronous protocol can be used for communication between voltage and frequency islands [5] designed to reduce power consumption. Since this protocol is asynchronous, it can be completely described in software.

Figure 7.3 shows the hardware block diagram of the send and receive communication interfaces. The send CI has two registers that store the values of *Data* and *Ready* signals. The input signals of these register are connected to the IP. The receive CI has only one register that stores the value of the *Ack* signal.

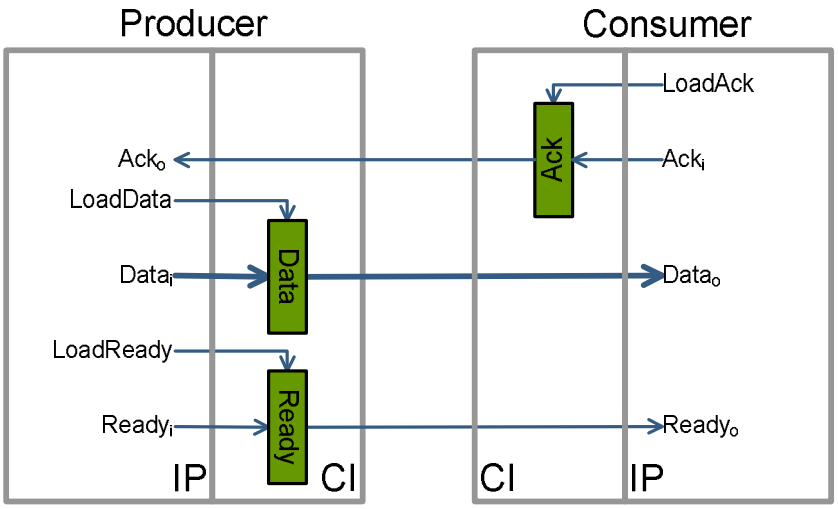


Figure 7.3. Block diagram of point-to-point single-word CIs (send and receive)

Figure 7.4 and Figure 7.5 show the GNR of the CI components in the producer and consumer components, respectively. These modules must be instantiated in the corresponding component and properly connected to the internal datapath and external ports of the architecture. Each CI provides a set of pre-bound functions for accessing the *Data*, *Ready*, and *Ack* values. Note that the *write* functions, i.e. *writeData*, *writeReady*, and *writeAck* write a value to a register, therefore they are defined as pipelined operations with 1 stage. This way the compiler correctly schedules these pre-bound functions.

```
<Module type="SingleWordProducerCI">
  <Ports>
    <Clock n="clk" bitWidth="1" />
    <InPort n="iAck" bitWidth="1" />
    <OutPort n="oAck" bitWidth="1" />
    <CtrlPort n="LoadReady" bitWidth="1" default="0" />
    <InPort n="iReady" bitWidth="1" />
    <OutPort n="oReady" bitWidth="1" />
    <CtrlPort n="LoadData" bitWidth="1" default="0" />
    <InPort n="iData" bitWidth="32" />
    <OutPort n="oData" bitWidth="32" />
  </Ports>
  <Netlist>
    <Components>
      <Instance n="Data" type="Register" lib="MainLib"/>
      <Instance n="Ready" type="Register" lib="MainLib"/>
    </Components>
    <Connections>
      <Conn src="" srcPort="iAck" dest=" " destPort="oAck" />
      <Conn src="" srcPort="LoadReady" dest="Ready" destPort="load" />
      <Conn src="" srcPort="iReady" dest="Ready" destPort="i" />
      <Conn src="Ready" srcPort="o" dest="" destPort="oReady" />
      <Conn src="" srcPort="LoadData" dest="Data" destPort="load" />
      <Conn src="" srcPort="iData" dest="Data" destPort="i" />
      <Conn src="Data" srcPort="o" dest="" destPort="oData" />
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Functions>
      <Function n="readAck" stateDependency="none" delay="0">
        <Output port="oAck"><Type n="int" /></Input>
      </Function>
      <Function n="writeReady" stateDependency="all" stages="1" delay="0">
        <Input port="iReady"><Type n="int" /></Input>
        <Ctrl port="LoadReady" val="1" />
      </Function>
      <Function n="writeData" stateDependency="all" stages="1" delay="0">
        <Input port="iData"><Type n="int" /></Input>
        <Ctrl port="LoadData" val="1" />
      </Function>
    </Functions>
  </Annot_compiler>
</Module>
```

Figure 7.4. GNR of single word point-to-point CI for producer component

Assuming that the instance name for CI component in the producer and consumer datapaths is *ci*, the *PreboundCGenerator* tool generates `__$ci_readAck`, `__$ci_writeReady`, and `__$ci_writeData` pre-bound functions for the producer side; and `__$ci_writeAck`, `__$ci_readReady`, and `__$ci_readData` for the consumer side.

```
<Module type="SingleWordConsumerCI">
  <Ports>
    <Clock n="clk" bitWidth="1" />
    <CtrlPort n="LoadAck" bitWidth="1" default="0" />
    <InPort n="iAck" bitWidth="1" />
    <OutPort n="oAck" bitWidth="1" />
    <InPort n="iReady" bitWidth="1" />
    <OutPort n="oReady" bitWidth="1" />
    <InPort n="iData" bitWidth="32" />
    <OutPort n="oData" bitWidth="32" />
  </Ports>
  <Netlist>
    <Components>
      <Instance n="Ack" type="Register" lib="MainLib"/>
    </Components>
    <Connections>
      <Conn src="" srcPort="LoadAck" dest="Ack" destPort="load" />
      <Conn src="" srcPort="iAck" dest="Ack" destPort="i" />
      <Conn src="Ack" srcPort="o" dest="" destPort="oAck" />
      <Conn src="" srcPort="iReady" dest="" destPort="oReady" />
      <Conn src="" srcPort="iData" dest="" destPort="oData" />
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Functions>
      <Function n="writeAck" stateDependency="all" stages="1" delay="0>
        <Input port="iAck"><Type n="int" /></Input>
        <Ctrl port="LoadAck" val="1" />
      </Function>
      <Function n="readReady" stateDependency="none" delay="0">
        <Output port="oReady"><Type n="int" /></Input>
      </Function>
      <Function n="readData" stateDependency="none" delay="0">
        <Output port="oData"><Type n="int" /></Input>
      </Function>
    </Functions>
  </Annot_compiler>
</Module>
```

Figure 7.5. GNR of single word point-to-point CI for consumer component

Figure 7.6(a) and (b) show driver codes of the send and receive CIs in the producer and consumer IPs respectively. In the send driver, line (3) loads the data into *Data*

register through pre-bound function `__$ci_writeData()`. Then, it writes value "1" into register *Ready* through pre-bound function `__$ci_writeReady(1)`. Next, in line (5), it waits until *Ack* signal become "1". In line (6), signal *Ready* is lowered by writing value "0" into the *Ready* register. Similarly, the receive driver (shown in Figure 7.6(b)) follows the protocol via calling pre-bound functions `__$ci_writeAck()`, `__$ci_readReady()`, and `__$ci_readData()`.The NISC cycle-accurate compiler uses the CI information captured in GNR to directly control the underlying hardware via C description. In this way, the C description of IP can be easily combined with the C description of the communication protocol to generate the correct RTL.

```
1  void send(int data)
2  {
3    __$ci_writeData(data);
4    __$ci_writeReady(1);
5    while(__$ci_readAck()==0);
6    __$ci_writeReady(0);
7    while(__$ci_readAck()==1);
8  }
```

(a)

```
1  int receive()
2  {
3    while(__$ci_readReady()==0);
4    int data = __$ci_readData();
5    __$ci_writeAck(1);
6    while(__$ci_readReady()==1);
7    __$ci_writeAck(0);
8  }
```

(b)

Figure 7.6. (a) send, (b) receive driver code for point-to-point single-word CIs

The protocol of this section has a high latency due to the need for handshaking for each word. Additionally, if one of the parties is significantly slower, the other one will also be slowed down during the transaction. To address this issue often a queue is used to buffer the data, as shown in the next section.

7.2.2 Shared queue interface

Figure 7.7 shows the block diagram of two IPs communicating through a shared queue. The queue has *Push*, *Pop*, $Data_{in}$, $Data_{out}$, and *IsEmpty* ports that are connected to send and receive CIs. When *Push* signal is "1", one word is pushed into the queue, and when *Pop* signal is "1", one word is popped from the queue. The *IsEmpty* signal becomes

"1" when the queue is empty. The queue may have two clocks in order to push and pop at two different clock frequencies. In this protocol, the producer pushes the data into the queue and the consumer pops it from the queue. The parties can use queue status (*IsEmpty* signal) for synchronization between them. Since no additional signals are necessary for synchronization, the send and receive CIs (shown in Figure 7.7) do not contain any registers. The CI components are in fact place holders in the GNR description of each IP that provide the description of pre-bound functions and act as a proxy for controlling the queue. In general, a proxy component can enable the NISC compiler to control a component that is outside of the datapath of a NISC component.

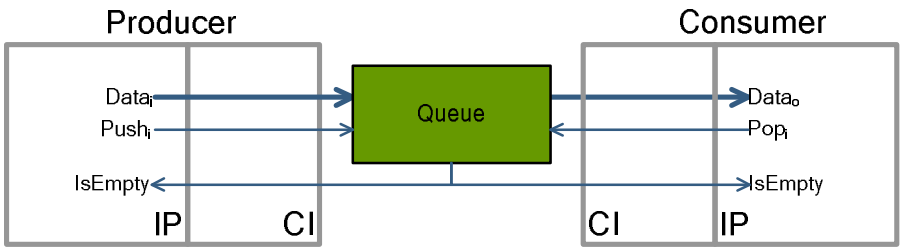


Figure 7.7. Block diagram of point-to-point queue-based CIs (send and receive)

Figure 7.8 and Figure 7.9 show the GNR description the queue CIs for producer and consumer, respectively. Each component provides pre-bound functions for push/pop or reading the status of the queue. Note that since push and pop are clocked operations, the corresponding pre-bound functions are defined as pipelined with one stage. Figure 7.10(a) and (b) show the send and receive drivers. The send driver first waits until the queue is empty of any previous data (line 3). Then, it consecutively pushes N words into the queue. On the receiver side, the driver receives N words in a loop (lines 3-6) by popping from the queue. If the sender has slower clock frequency than the receiver, then the queue may become empty in the middle of a transaction. In such case, the receiver must wait (line 4 of Figure 7.10(b)) until the sender pushes more data into the queue.

```
<Module type="QueuePrcducerCI">
  <Ports>
    <Clock n="clk" bitWidth="1" />
    <ControlPort n="iPush" bitWidth="1" />
    <OutPort n="oPush" bitWidth="1" />
    <InPort n="iIsEmpty" bitWidth="1" />
    <OutPort n="oIsEmpty" bitWidth="1" />
    <InPort n="iData" bitWidth="32" />
    <OutPort n="oData" bitWidth="32" />
  </Ports>
  <Netlist>
    <Components></Components>
    <Connections>
      <Conn src="" srcPort="iData" dest="" destPort="oData" />
      <Conn src="" srcPort="iPush" dest="" destPort="oPush" />
      <Conn src="" srcPort="iIsEmpty" dest="" destPort="oIsEmpty" />
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Functions>
      <Function n="push" stateDependency="all" stages="1" delay="0">
        <Input port="iData"><Type n="int" /></Input>
        <Ctrl port="iPush" val="1" />
      </Function>
      <Function n="isEmpty" stateDependency="none" delay="0">
        <Output port="oIsEmpty"><Type n="int" /></Input>
      </Function>
    </Functions>
  </Annot_compiler>
</Module>
```

Figure 7.8. GNR of point-to-point queue-based CI for producer component

```
<Module type="QueueConsumerCI">
  <Ports>
    <Clock n="clk" bitWidth="1" />
    <ControlPort n="iPop" bitWidth="1" />
    <OutPort n="oPop" bitWidth="1" />
    <InPort n="iIsEmpty" bitWidth="1" />
    <OutPort n="oIsEmpty" bitWidth="1" />
    <InPort n="iData" bitWidth="32" />
    <OutPort n="oData" bitWidth="32" />
  </Ports>
  <Netlist>
    <Components></Components>
    <Connections>
      <Conn src="" srcPort="iData" dest="" destPort="oData" />
      <Conn src="" srcPort="iPop" dest="" destPort="oPop" />
      <Conn src="" srcPort="iIsEmpty" dest="" destPort="oIsEmpty" />
    </Connections>
  </Netlist>
  <Annot_compiler>
    <Functions>
      <Function n="pop" stateDependency="all" stages="1" delay="0>
        <Output port="iData"><Type n="int" /></Output>
        <Ctrl port="iPop" val="1" />
      </Function>
      <Function n="isEmpty" stateDependency="none" delay="0">
        <Output port="oIsEmpty"><Type n="int" /></Input>
      </Function>
    </Functions>
  </Annot_compiler>
</Module>
```

Figure 7.9. GNR of point-to-point queue-based CI for consumer component

```
1  void send(int N, int* data)
2  {
3    while(__$ci_isEmpty()==1);
4    for(i=0;i<N; i++)
5      __$ci_push(data[i]);
6  }
```

(a)

```
1  void receive(int N, int* data)
2  {
3    for(i=0;i<N; i++)
4      while(__$ci_isEmpty()==1);
5      data[i++] = __$ci_top();
6      __$ci_pop();
7  }
```

(b)

Figure 7.10. (a) send, (b) receive driver code for point-to-point queue-based CIs

The queue protocol is considered a non-blocking protocol because producer can resume its computation without waiting for the consumer to receive the data. This improves the performance of the producer IP. In general, point-to-point communications are used when each IP communicates with one or a few other IPs. To communicate with more IPs, shared buses may be used.

7.2.3 Double-Handshake bus interface

Suppose that we have a double-handshake shared bus that allows transmission of variable size packets in a non-blocking message-passing fashion. Figure 7.11 shows the block diagram of two IPs communicating through such shared bus. On the producer side, the CI stores the data in a queue and then requests the bus from arbiter. After getting bus grant, the CI places the consumer address on the *AddrBus* and raises the *Ready* signal. Once the *Ack* signal becomes "1", the producer CI puts one word per clock cycle on the *DataBus*. After sending the entire data, the CI lowers the *Ready* signal and releases the bus for the next communication. On the consumer side, the CI stores the data in a queue and interrupts the consumer IP. The timing diagram of this protocol is shown in Figure 7.12. The arrows show the sequence of the events.

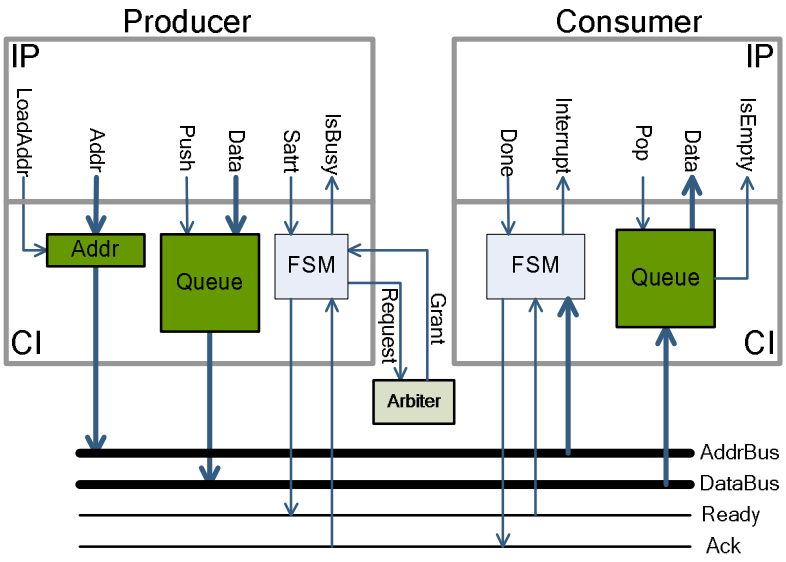


Figure 7.11. Block diagram of shared-bus CIs (send and receive)

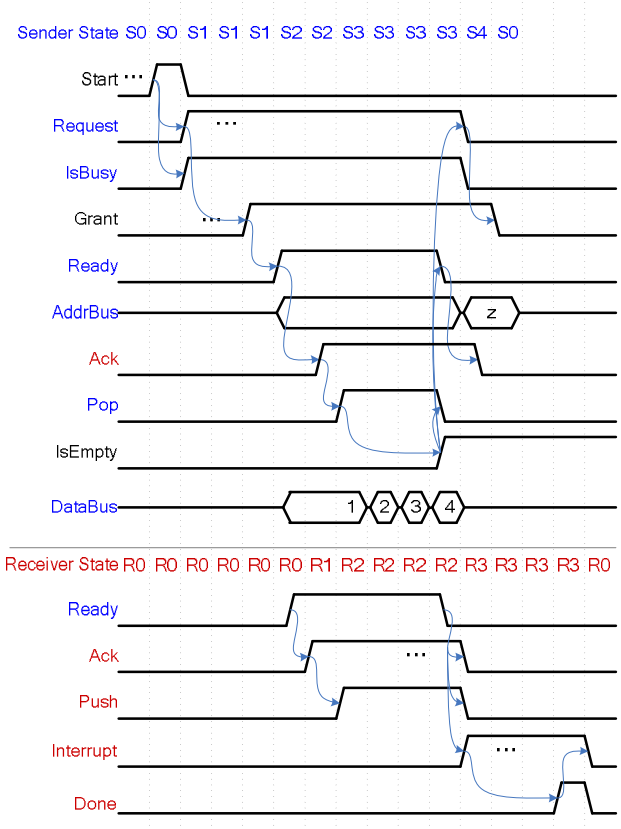


Figure 7.12. Timing diagram of the example bus protocol

This protocol is considered synchronous because it must transfers one word per cycle without any handshaking for each individual word. Implementing synchronous protocols are not possible in software, because software cannot guarantee the required timing.

Therefore, the protocol is partitioned into an asynchronous part handled by software and a synchronous part managed by Finite State Machines (FSM) inside CIs. Figure 7.13 shows the send and receive state machines. The send and receive FSMs are very small with five and four states, respectively. The FSM inside each CI will be implemented in the RTL description of the CI component. The asynchronous part of the protocol in software and its synchronous part in hardware are linked to each other via the pre-bound functions in GNR. The CI on the producer side provides four pre-bound functions that control the signals between the datapath and CI (see Producer IP in Figure 7.11). The pre-bound functions include: `__$ci_isBusy()`,`__$ci_start()`, `__$ci_push()`, and `__$ci_setAddr()`. The CI on the consumer side provides four pre-bound functions that control the signals between the datapath and CI (see Producer IP in Figure 7.11). The pre-bound functions include: `__$ci_isEmpty()`, `__$ci_pop()`, `__$ci_top()`, and `__$ci_done()`. The interrupt signal of this CI is connected to controller in the datapath of consumer IP.

Figure 7.14 shows the drivers of this protocol. The send driver (Figure 7.14(a)) waits until the CI is done sending any previous message (line 4). Then, it loads the *Addr* register with the receiver address. Next, it pushes N words into the queue (lines 6-7) and issues the *Start* command (line 8). On the consumer side, first the CI receives all data in every cycle and then notifies the IP via interrupt. Therefore the, the receiver driver is written as an interrupt routine (Figure 7.14(b)). For each interrupt, the driver pops the data from the queue and issues the *Done* command. The mapping between the pre-bound functions and ports of each CI are shown in Figure 7.15.

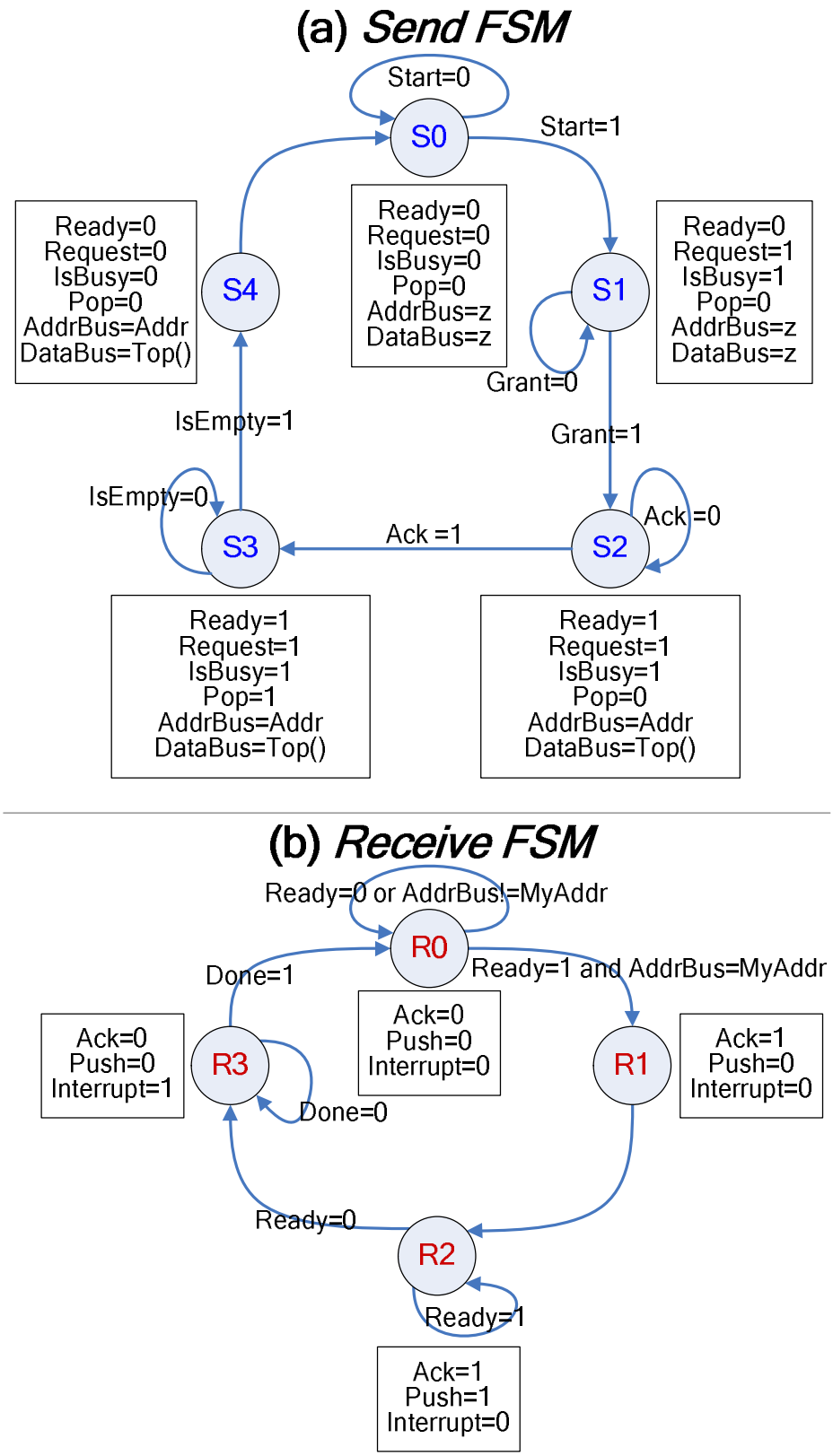


Figure 7.13. FSMs inside (a) send and (b) receive CIs

```
1  void send(int N, int* data, int recAddr)
2  {
3    while(__$ci_isBusy()==1);
4    __$ci_setAddr(recAddr);
5    for(int i=0; i<N;i++)
6      __$ci_push(data[i]);
7    __$ci_start();
8  }
9
```

(a)

```
1  interrupt Receive (){
2   disable interrupt
3   buffer[0] = top();
4   int i=1;
5   while (__$ci_isEmpty()==0)
6     __$ci_pop();
7     buffer[i++] = __$ci_top();
8   __$ci_done();
9   enable interrupt
    }
```

(b)

Figure 7.14. (a) send, (b) receive driver code for shared-bus CIs

function	input	ctrl	output
__$ci_isBusy	–	–	IsBusy
__$ci_setAddr	Addr	LoadAddr	–
__$ci_push	Data	Push	–
__$ci_start	–	Start	–

(a)

function	input	ctrl	output
__$ci_top	–	–	Data
__$ci_pop	–	Pop	–
__$ci_isEmpty	–	–	IsEmpty
__$ci_done	–	Done	–

(b)

Figure 7.15. Pre-bound functions of (a) send, (b) receive CIs for shared bus

This example shows how synchronous parts of a protocol are implemented in hardware, while its asynchronous parts are written in software. The software and hardware parts are ultimately integrated with each other through pre-bound functions. This approach can be generalized to implement other types of communication interfaces as well. These examples also show that without adding low-level programming (Chapter Chapter 5) and interrupt (Chapter Chapter 6) NISC could not practically implement any communication protocol and be integrated in a larger system. In other words, the compilation algorithm, the pre-bound functions and variables (low-level programming), and interrupt support are the necessary and sufficient features for developing an IP using NISC technology and integrating them into systems.

Chapter 8. Experiments

We have developed a complete NISC compiler and have integrated it with the rest of NISC toolset. The whole toolset is available for download from NISC Website [47]. We have also developed an online version of the toolset that does not need local installation and can be run via web. The experiments presented in this chapter can be run on the online version of the toolset as well.

In this chapter, we present four sets of experiments. To show the generality and efficiency of the compilation algorithm, in Section 8.1, we show the compilation and simulation results of several benchmarks on several different architectures. In Section 8.2, we compare the performance and code size of several benchmarks on a RISC processor versus a general-purpose NISC architecture. Then in Section 8.3, we show a customization example in NISC technology by selecting one of the benchmarks, namely the 2D DCT, and designing different custom datapaths to significantly improve its performance and energy consumption. Finally in Section 8.4 we show examples of communicating NISC components.

8.1 Compiling on different architectures

To evaluate the efficiency of the proposed compilation algorithm for NISC architectures, we compiled and ran a set of benchmarks on the set of generic NISC architectures. These architectures include: GN0, which has no pipelining and data forwarding (Figure 8.1), GN1, which has pipelining but no data forwarding (Figure 8.2), GN2, which has both pipelining and data forwarding (Figure 8.3), and GN3, with a non uniform structure (Figure 8.4). In all of these architectures, we used a clocked data memory which had pipelined operations. Also, note that the controller pipeline (the path from CW port of controller back to its status port) is different in each architecture. Hence the compiler must also detect the branch delays to generate correct FSMs. The benchmarks include *dijkstra*, *sha*, *adpcm.coder*, *adpcm.decoder*, *qsort* and *crc32* from MiBench (the free version of EEMBC embedded benchmarks available at [40]), and a fixed-point Mp3 decoder (more than 10,000 lines of C code available at [42]).

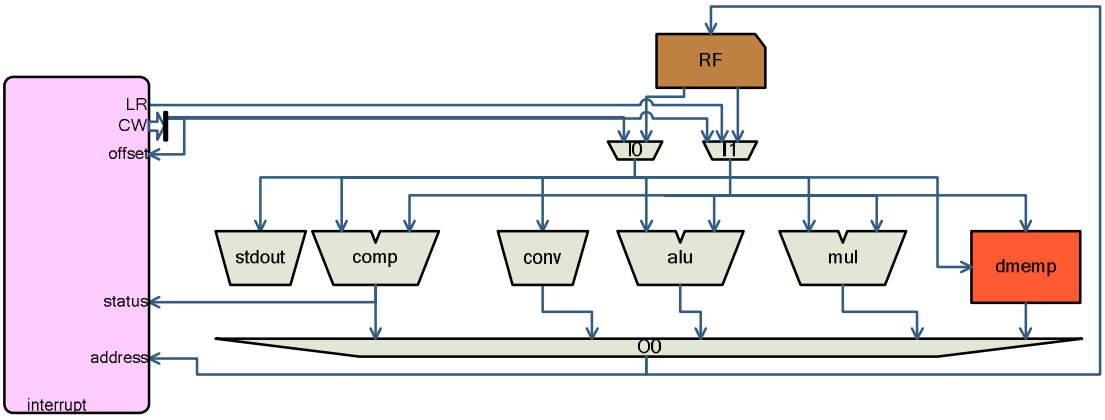


Figure 8.1. GN0 with no pipelining or data forwarding

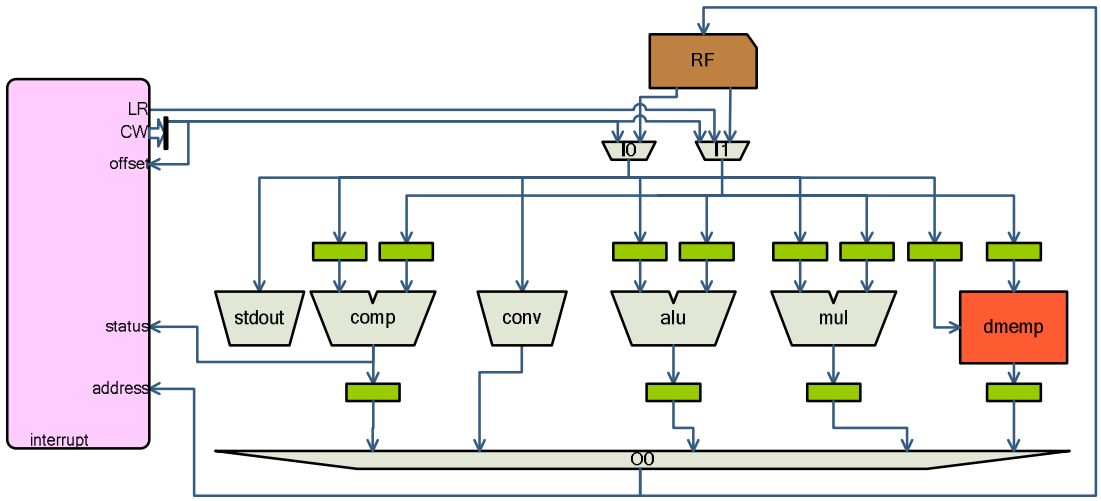


Figure 8.2. GN1 with pipelining but no data forwarding

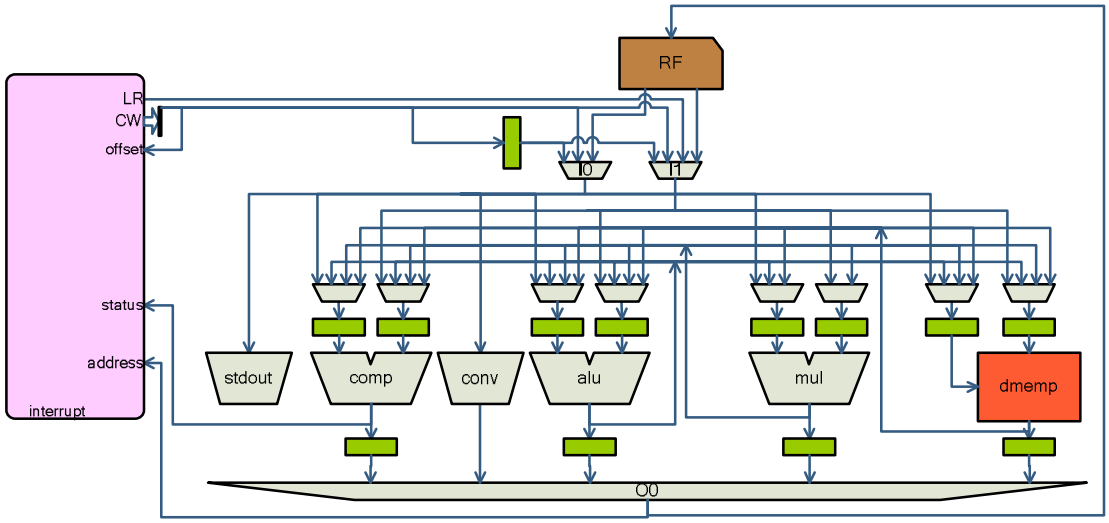


Figure 8.3. GN2 with pipelining and data forwarding

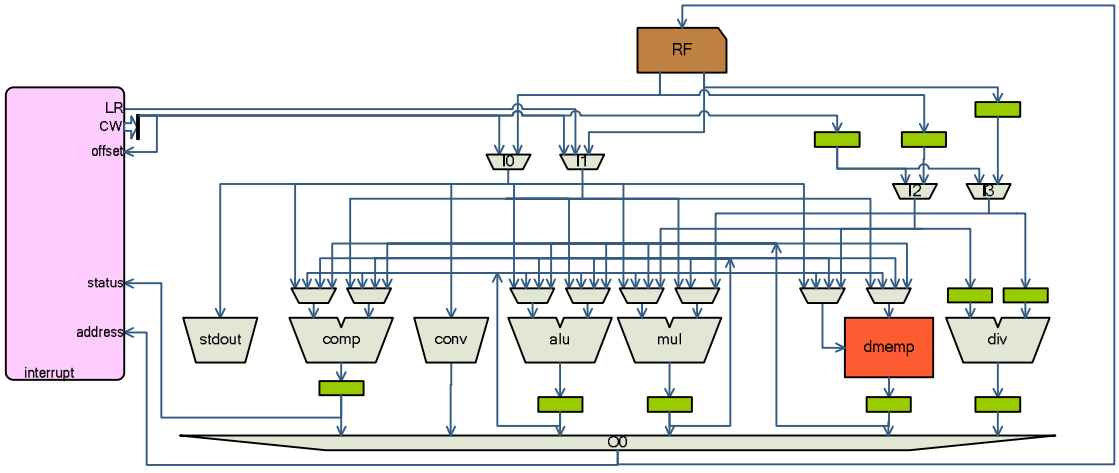


Figure 8.4. GN3 with non uniform structure

Table 8.1. Execution and compilation time for various architectures

	#cycles				compilation time (s)			
Benchmark	**GN0**	**GN1**	**GN2**	**GN3**	**GN0**	**GN1**	**GN2**	**GN3**
dijkstra	-	-	-	49629	-	-	-	0.11
sha	68874	77622	60522	58910	0.20	0.25	0.41	0.23
adpcm.coder	-	-	-	37890	-	-	-	0.08
adpcm.decoder	109518	173830	146214	118350	0.03	0.05	0.06	0.05
qsort	-	-	-	138496	-	-	-	0.19
CRC32	19083	26109	18081	15079	0.02	0.02	0.01	0.02
FpMp3	-	-	-	759296	-	-	-	19.20

We compiled every benchmark on all possible architectures and generated the RTL Verilog codes. We simulated these codes to get both the accurate execution cycle counts, and to verify that the simulation outputs exactly match with that of benchmarks running on a host PC. Table 8.1 shows the execution and compilation times for each benchmark. Some benchmarks that needed a divider could be only compiled on the GN3. These experiments showed that the compilation algorithm works correctly and can properly utilize different datapath structures. Also, the algorithm is fast enough to be used for practical settings.

8.2 Compilation on a general-purpose NISC

The main goal of NISC Technology is to enable efficient customizations, which requires the NISC cycle-accurate compiler support and handle different types of customized architectures. To get a sense of how efficient the compiler can utilize a datapath, we compiled and ran the benchmarks on the generic NISC processor shown in Figure 8.5, and compared the results with a RISC processor with similar complexity. We simulated and synthesized all results to compare their quality. We used Xilinx ISE 8.1 toolset for simulation and synthesis of the results. As a base for comparison, we chose the 32-bit Xilinx MicroBlaze soft-core processor. We chose this RISC processor because it is a commercial core that is already optimized for Xilinx technology and hence is a good

base for comparing our results to its clock frequency, area, and performance. It also comes with a complete toolset for compiling programs and cycle-accurate simulation of the results. This processor is included in the Xilinx tools as an encrypted soft-core. For each benchmark, we validated the output generated by Verilog simulation with the corresponding output generated from running that benchmark on a PC desktop.

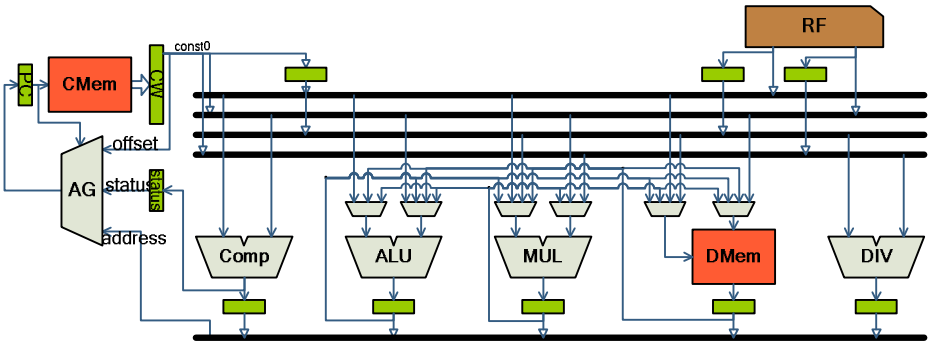


Figure 8.5. A generic NISC architecture (GN)

We synthesized MicroBlaze (MB) and a generic NISC (GN) on a Xilinx Virtex4 (90-nm) FPGA package. The bit-width of control words in GN is 101 bits as opposed to 32-bit instructions in MicroBlaze. Table 8.2 shows the area and clock frequency of these processors. The clock frequency of MicroBlaze is 105MHz, as opposed to 80MHz for NISC. In NISC, since the divider unit is pipelined and multiplier is not pipelined, the critical path goes through the multiplier. In MicroBlaze, a pipelined multiplier is used which improves the clock frequency. The third column of the table shows the area of the processors in terms of number of gates. MicroBlaze has more gates than GN, which we believe is because of the additional logic for decoding the instructions and controlling the pipeline. MicroBlaze core comes with specific fine-grained timing constraints that direct the synthesis tool to achieve the highest possible clock frequency. For synthesizing GN we only used a general clock constraint and we expect that the clock frequency of NISC can be further improved by using more specific constraints.

Table 8.2. Area and clock frequency of MicroBlaze and GN

Processors	Clock freq.(MHz)	Area (gates)
MicroBlaze	105	39574
GN	80	35317

Table 8.3. Comparing MicroBlaze with GN

	MicroBlaze		GN		GN vs. MicroBlaze	
Benchmark	#cycles	code size (bits)	#cycles	code size (bits)	speedup	code-size ratio
dijkstra	25,927,532	1,928	10,631,310	3,082	1.86	1.60
sha	183,030,479	3,156	18,371,827	4,901	7.59	1.55
adpcm.coder	256,748,693	1,364	84,251,684	2,671	2.32	1.96
adpcm.decoder	322,766,405	1,956	66,504,319	2,068	3.70	1.06
CRC32	209,436,647	1,364	26,008,604	1,100	6.14	0.81
FpMp3	8,861,336	44,620	927,307	70,426	7.28	1.58
Average	**167,795,182**	**9,065**	**34,449,175**	**14,041**	**4.81**	**1.43**

Table 8.3 compares MicroBlaze and GN in terms of number of cycles and code size. For this experiment, we set the MicroBlaze compiler optimizations to the maximum to achieve the maximum performance. MiBench provides a *small* and a *large* input for the benchmarks as well. For simulating the Mp3 decoder, we used the *scope1.mp3* (44.1KHz, 96kbit/s, stereo) available at [24]. The reported cycle numbers in Table 8.3 are for simulating the *small* inputs of MiBench benchmarks and processing 1 frame of the Mp3 file. The second column of Table 8.3 shows number of cycles that it takes for MicroBlaze to run each benchmark. The third column shows the size of instruction section (.text) of the .elf file generated by the compiler. Similarly, the fourth and fifth columns show the number of cycles and code size for GN. The sixth column shows the execution-delay ratio of MicroBlaze vs. GN. The execution delay is calculated by diving number of cycles by clock frequency of the processor. Although GN runs at a lower clock frequency, it runs the benchmarks, on average, 4.8 times faster than MicroBlaze. The last column shows the ratio between code size of GN and MicroBlaze. On average, GN consumes 1.43 times more memory compared to MicroBlaze.

These experiments illustrate that even in a general NISC without any application specific customizations; we can achieve better performance since the compiler can utilize the datapath more efficiently. The results are generated using some code compression techniques [10] that increate the controller pipelining depth and hence the branch delay slot. The compression techniques are applied during controller synthesis and are not part of this thesis. But the results also show that the compiler can correctly support controller pipelining among other features.

8.3 Custom datapath design for DCT

In this section, we include an illustrative example that shows the customization capability of NISC technology. This example explores the design space for different quality metrics such as performance, area, and energy consumption. The step by step details of this customization experiment can be found in [11]. We only include a summary of this experiment here for a quick reference and to show that our NISC cycle-accurate compiler can handle very irregular custom datapaths as well.

The goal is to design a custom pipelined datapaths for DCT algorithm to further improve the performance and power consumption of the design. The definition of Discrete Cosine Transform (DCT) [43] for a 2-D $N \times N$ matrix of pixels is as follows:

$$F[u,v] = \frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f[m,n] \cos \frac{(2m+1)u\pi}{2N} \cos \frac{(2n+1)v\pi}{2N}$$

Where u, v are discrete frequency variables ($0 \le u, v \le N-1$), $f[i,j]$ gray level of pixel at position (i, j), and $F[u,v]$ coefficients of point (u, v) in spatial frequency. Assuming $N=8$, matrix C is defined as follows:

$$C[u][n] = \frac{1}{8} \cos \frac{(2n+1)u\pi}{16}$$

Based on matrix C, an integer matrix $C1$ is defined as follows: $C1 = \text{round}(\textit{factor} \times C)$. The $C1$ matrix is used in calculation of DCT and IDCT: $F = C1 \times f \times C2$, where, $C2 = C1^{T}$. As a result, DCT can be calculated using two consecutive matrix multiplications. Figure 8.6(a) shows the C code of multiplying two given matrix A and B using three nested loops. As a base for comparison and start point for customizations, a NISC-style implementation of a MIPS M4K datapath [41] (called NMIPS) is chosen. The bus-width of the datapath is 16-bit for a 16-bit DCT precision, and the datapath does not have any integer divider or floating point unit. The clock frequency of 78.3MHz was achieved after synthesis and Placement-and-Routing (PAR). Two synthesis optimizations of retiming and buffer-to-multiplexer conversions are applied to improve the performance.

In general, customization of a design involves both software and hardware transformations. To increase the parallelism in code, the inner-most loop of the matrix multiplication code is unrolled, the two outer loops are merged, and some of the costly operations such as addition and multiplication are converted to OR and AND. In DCT, the operation conversions are possible because of the special values of the constants and variables. The transformed code is shown in Figure 8.6(b). In the next step, a custom architecture is designed for the transformed DCT code. This architecture is called CDCT1 and is shown in Figure 8.7(a). Several customizations are applied to this initial custom architecture to improve the performance, area, and energy consumption. These customizations include reducing bit width of components, removing underutilized resources, and repeatedly adding registers to break the critical path delay. After each customizations, the modified C code of Figure 8.6(b) is compiled on the refined architecture. In each step, the results are synthesized and analyzed to figure out what part

of datapath can be further customized for more improvements. After seven iterations, the final architecture is called CDCT7 as shown in Figure 8.7(b). in this last architecture, the multiplier is considered to be a multi-cycle component because in the target FPGA, this multiplier is mapped to an ASIC unit that cannot be pipelined or optimized.

```
for(int i=0; i<8; i++)
   for(int j=0; j<8; j++){
      sum=0;
      for(int k=0; k<8; k++)
         sum= sum+A[i][k]×B[k][j];
      C[i][j]= sum;
   }
```

(a)

```
ij=0;
do {
  i8 = ij & 0xF8;
  j = ij & 0x7;
  aL= *(A+(i8|0) ); bL= *(B + (0|j) );  sum =  aL × bL;
  aL= *(A+(i8|1) ); bL= *(B + (8|j) );  sum+= aL × bL;
  aL= *(A+(i8|2) ); bL= *(B + (16|j) ); sum+= aL × bL;
  aL= *(A+(i8|3) ); bL= *(B + (24|j) ); sum+= aL × bL;
  aL= *(A+(i8|4) ); bL= *(B + (32|j) ); sum+= aL × bL;
  aL= *(A+(i8|5) ); bL= *(B + (40|j) ); sum+= aL × bL;
  aL= *(A+(i8|6) ); bL= *(B + (48|j) ); sum+= aL × bL;
  aL= *(A+(i8|7) ); bL= *(B + (56|j) );
  *(C + ij) = sum + (aL × bL);
} while(++ij!=64);
```

(b)

Figure 8.6. (a) Original and (b) Transformed matrix multiplication

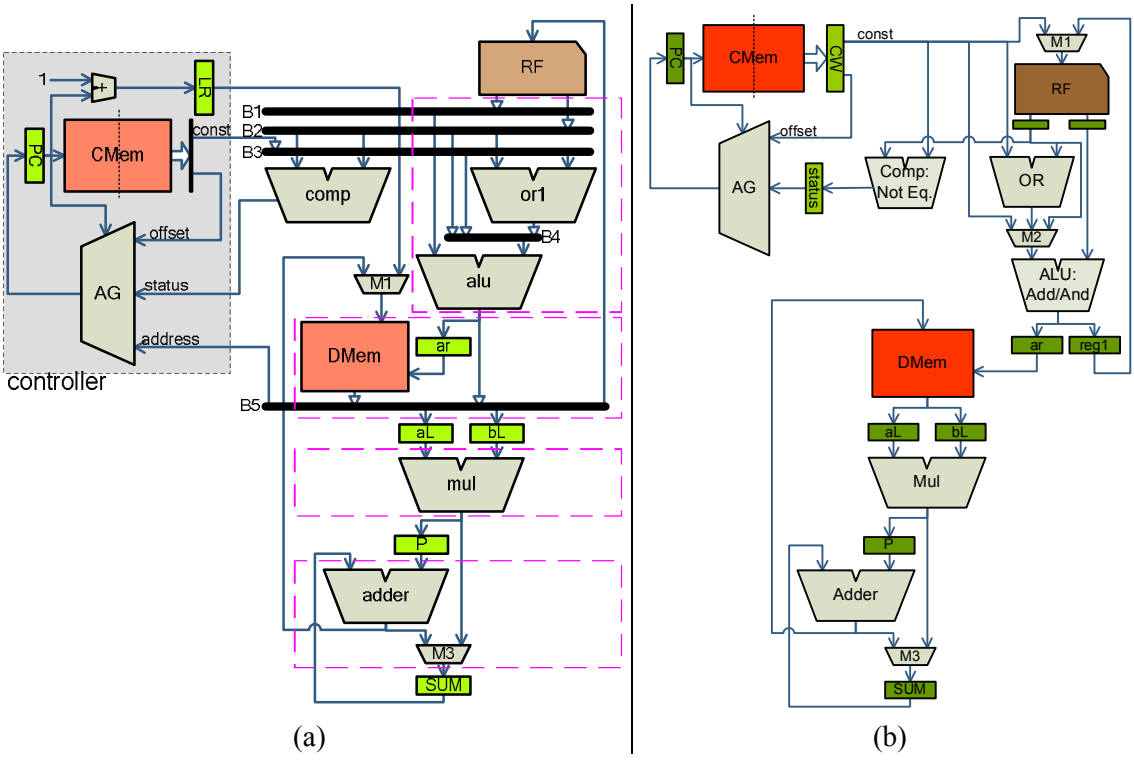


Figure 8.7. Block diagram of (a) CDCT1, (b) CDCT7

Table 8.4 compares the performance, power, energy, and area of the all NISC implementations. The third column shows the maximum clock frequency after Placement and Routing. The fourth column shows the total execution time of the DCT algorithm calculated based on number of cycles and the clock frequency. Note that although in some cases (such as CDCT4 and CDCT5) the number of cycles increases, the clock frequency improvement compensates for that. As a result, the total execution delay maintains a decreasing trend. Column fifth shows the average power consumption of the NISC architectures while running the DCT algorithm. All the designs are stimulated with the same data values. The power consumption of each design is computed using the Xilinx XPower tool and the signal activities collected from Post-Placement and Routing simulation. Column sixth shows the total energy consumption calculated by multiplying power and execution time.

Table 8.4. Performance, power, energy, and area of the DCT implementations

	# Cycles	Clock freq	DCT exec. time(us)	Power (mW)	Energy (uJ)	Normalized area
NMIPS	10772	78.3	137.57	177.33	24.40	1.00
CDCT1	3080	85.7	35.94	120.52	4.33	0.81
CDCT2	2952	90.0	32.80	111.27	3.65	0.71
CDCT3	2952	114.4	25.80	82.82	2.14	0.40
CDCT4	3080	147.0	20.95	125.00	2.62	0.46
CDCT5	3208	169.5	18.93	106.00	2.01	0.43
CDCT6	3208	171.5	18.71	104.00	1.95	0.34
CDCT7	3460	250.0	13.84	137.00	1.90	0.35

Figure 8.8 shows the performance, power, energy and area of the designs normalized against NMIPS. The total execution delay of DCT algorithm has a decreasing trend, while the power consumption decreases up to CDCT3 and then increases. The energy consumption significantly drops at CDCT1, because of the reduction in number of cycles and power consumption. From CDCT1 to CDCT7, the energy decreases gradually in a

slow paste. As shown in the figure, CDCT7 is the best design in terms of delay and energy consumption, while CDCT3 is the best in terms of power, and CDCT6 is the best in terms of area. As a result, CDCT3, CDCT6, and CDCT7 are considered the pareto-optimal solutions. Note that minimum energy and minimum power are achieved by two different designs: CDCT7 and CDCT3, respectively. Compared to NMIPS, CDCT7 runs 10 times faster, consumes 1.3 times less power and 12.8 times less energy. Also, it occupies 2.9 times less area than NMIPS.

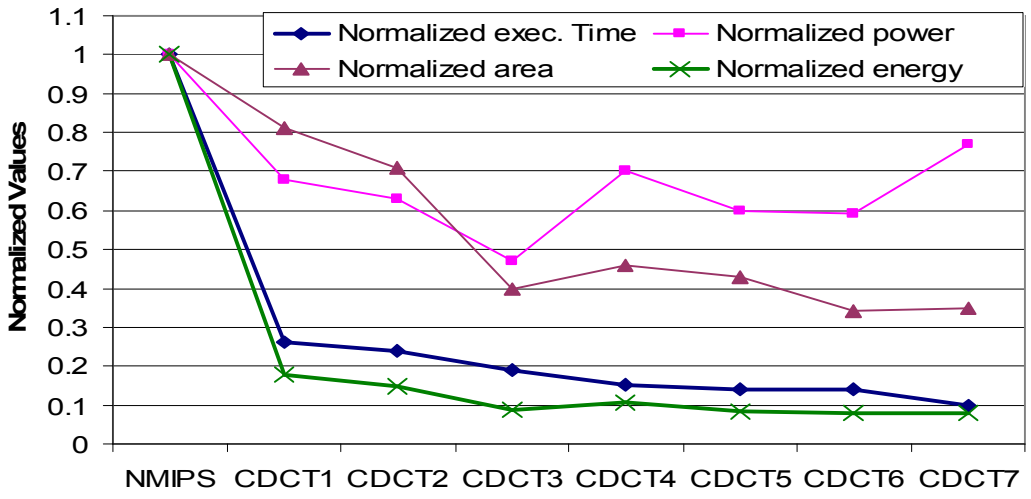


Figure 8.8 Comparing different DCT implementations

In summary, designing a custom datapath for a given application by properly connecting functional units and pipeline registers is the key to reducing number of cycles and energy consumption. Also, eliminating the unused logic and interconnects, adjusting the bus-width of the datapath to the application requirement, signal gating, and clock gating are the key to reducing power consumption. The NISC compiler makes these customizations very easy to apply. It allows the designer to modify the component netlist of the datapath and then uses the proposed compilation algorithm to automatically map the application on the given datapath.

8.4 Communicating NISC components

In this section, we describe the implementation results of two multi-NISC systems for a fixed-point Mp3 benchmark downloaded from [42]. These NISCs communicate via the shared bus protocol that we described in Section 7.2.3. In general, an Mp3 audio file contains several frames. For a stereo file, each frame has two channels (i.e. left and right channels). In the Mp3 decoder, the frames go through three main phases, namely, *decode_frame*, *synthesis_frame* and *output_pcm*. Profiling the Mp3 decoder on the generic NISC architecture of Figure 8.5 showed that 63% of execution time is spent in *decode_frame*, 25% in *synthesis_frame*, and 11% in the *output_pcm*. We realized that there are two approaches to parallelize the Mp3 application: (a) processing each channel separately, or (b) pipelining the phases. However, the Mp3 decoder was originally targeted for desktop PCs and separating the channels completely requires rewriting most of the code. Alternatively, we decided to separate the *synthesis_frame* phase for each channel because it required minimum code modifications. Such partitioning can reduce the execution time of *synthesis_frame* to half and hence can at most improve the performance by 12.5%. As for the second system, we pipelined the application into two stages where the first pipeline stage implements *decode_frame* phase and the second stage implements *synthesis_frame* and *output_pcm* phases. In this approach, processing delay of one frame is expected to increase due to the communication overhead. However, since the *decode_frame* of one frame is overlapped with the *synthesis_frame* and *output_pcm* of another frame, the overall performance can be improved by up to 36% (=min(63, 25+11)).

Table 8.5. Area and clock frequency of MicroBlaze and GN

Processors	Clock freq.(MHz)	Area (gates)	#cycles for 1 frame	speedup
MicroBlaze	105	39574	8,861,336	1
GN	80	35632	897,452	7.28
multi-core GN	80	73046	-	-

We implemented the Mp3 decoder on a MicroBlaze, a single GN, and two multi-core configuration of GN. Table 8.5 shows the clock speed and area of each architecture as well as their performance for decoding one frame of audio. For simulating the Mp3 decoder, we used the *scope1.mp3* (44.1KHz, 96kbit/s, stereo) available at [24].

Table 8.6. Throughput of three Mp3 implementations

Systems	#cycles for 1 frame	speedup for 1 frame(%)	#cycles for 25 frames	speedup for 25 frames (%)	frames/sec
SingleCore	897,452	0.00	22,800,961	0.00	88
Coprocessor	803,357	10.48	20,205,994	11.38	99
Pipelined	917,204	-2.20	16,433,655	27.93	122

Table 8.6 shows the results of implementing the Mp3 decoder in three configurations. The second and fourth columns show number of cycles for processing one frame, and 25 frames in each configuration and the third and fifth columns show the respective speedups. Figure 8.9 shows the block diagram of the three implementation configurations. Figure 8.9(a) shows the *SingleCore* configuration in which the entire Mp3 decoder runs on one GN. Figure 8.9(b) shows the *Coprocessor* configuration in which the Mp3 decoder runs on two GAs. In this case, one of GN acts as a coprocessor for the main GN and runs the *synthesis_frame* phase for left channel while the main GN runs the same phase for the right channel. The main GN also runs the other phases for both channels. The total performance improvement in this case is 10.48% which is close to the expected 12.5%. For each channel, the main GN sends 1152 words to the coprocessor GN and then receives 1152 words from it. The communication overhead is responsible for the 2% performance loss from the expected upper bound, i.e. 12.5%. Figure 8.9(c) shows the

Pipelined configuration, where one GN runs the *decode_frame* of both channels and sends 2×1152 words to the second GN to perform *synthesis_frame* and *output_pcm*. In this configuration, the processing time for a single frame is increased by 2% but the overall throughput of the system is increased by 28%. Similarly, the communication overhead is responsible for the 8% performance loss from the expected upper bound, i.e. 36%. The communication overhead in the *Pipelined* configuration has increased because of the extra synchronization which was not necessary in *Coprocessor-Sys* configuration.

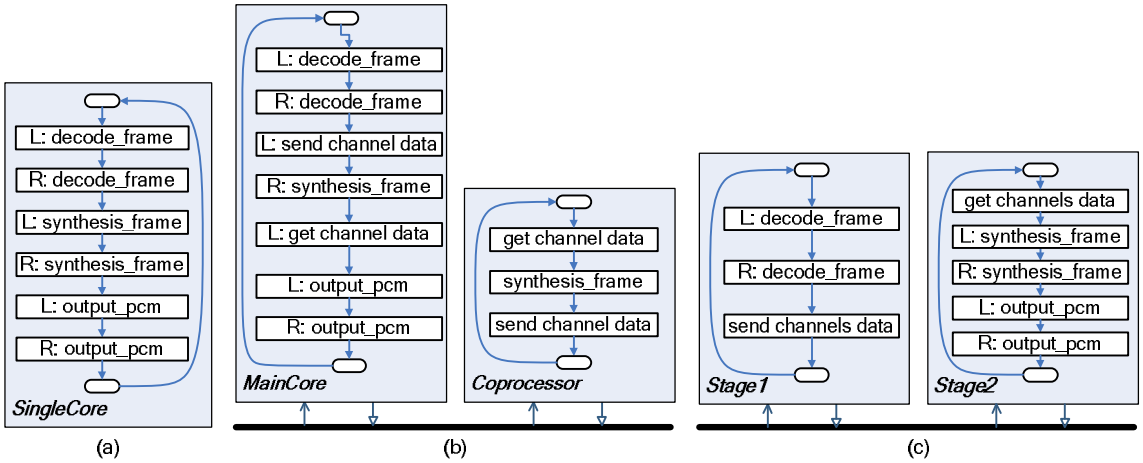


Figure 8.9. Implementing Mp3 (a) single core, (b) with coprocessor, and (c) pipelined

According to the Mp3 standard, at least 38 frames must be played per second. MicroBlaze can only run 12 frames per second. The last column of Table 8.6 shows the throughput of these three NISC based configurations. Clearly, this throughput is much more than what the standard required. To save power, the *SingleCore* and *Coprocessor* configuration can run with half their clock frequency. The clock frequency of the *Pipelined* system can be reduced by two thirds while still meeting the throughput constraints of the standard.

Chapter 9. Conclusion and future work

In this thesis, we introduced design flow based on No-Instruction-Set-Computer (NISC) Technology. In NISC datapath and controller are generated separately. Based on the application behavior, the datapath is generated or selected from a database. Then our cycle-accurate compiler directly maps a given application on a given datapath. The contributions of this thesis can be summarized as follows:

1. We explained the NISC design flow and explained that it is a better alternative to HLS and ASIP for developing custom processing elements. In HLS, designer has little control on generated results and changes in the input description (e.g. C) cannot be directly correlated to changes in the output RTL. HLS techniques suffer from bad output quality and limited application size/complexity support. This is mainly because supported target datapaths are very limited and connectivity constraints are not considered by HLS techniques. On the other hand, in ASIP, finding and implementing custom instructions is very complex. Besides, ASIP is not suitable for dedicate custom blocks because they always extend a base processor, which imposes a minimum complexity overhead on the results. In

NISC, the full datapath (resources and their connectivity) is used for compilation. Furthermore, it imposes almost no minimum requirement on the complexity of the architecture. In this way NISC can help the designer to achieve a balance between designer productivity and design quality.

2. We presented the details of a NISC architecture and its execution style. The core philosophy in designing the architecture was that "we should be able to remove or customize anything that is not used by the application from the architecture without requiring any changes to the toolset". Since the compiler depends on the internals of the controller, therefore the controller must be kept as simple as flexible as possible. The features of the architecture are modeled mainly in the datapath and the compiler generates tightly scheduled control bits to control the datapath resources in every cycle (hence the name cycle-accurate compiler). We presented a modeling approach for capturing and describing the architecture. In addition to structural details, this model uses the notion of machine actions (MA) to capture timing and to map high level operations such as storage read/write, data transfer, or operation execution to low-level hardware resources. We use four types of MAs for modeling the architecture: *Read*, *Write*, *Transfer*, and *Execute*. The compiler schedules the MAs in different clock cycles to construct an FSM and generate the stream of control words. Compared to previous modeling approaches, our model is very conscise and can simultaneously support both efficient compilation and efficient RTL generation. While other models could be used only for processors, our model can be used for small dedicated IPs as well. Additionally, in our model we can consider the actual clock period and

component timing in order to accurately support operation chaining and multi-cycling. Finally, previous models and approaches did not support controller pipelining and partial data forwarding.

3. We presented a compilation algorithm for mapping the CDFG of the program on a given datapath model. We showed that in NISC operation scheduling and resource binding must be done simultaneously. This is because the connectivity of datapath components is predetermined before compilation, and hence during scheduling we need to the binding of operations in order to know their delay and starting time of their consumer operations. We presented a compilation algorithm which is different from HLS techniques because it assumes that the datapath is given and is fixed during scheduling and binding. It performs the scheduling and binding simultaneously while processing the CDFG backward. It is also different from conventional instruction-set based compiler techniques because it directly maps the program on a given datapath without using any high-level instruction abstraction. Consequently, it must deal with all structural details of the architecture and solve more complex problems. This algorithm supports pipelined and multi-cycle operations, operation chaining, datapath/controller pipelining, and none uniform data forwarding. In previous approach operation pipelining, chaining, and multi-cycling could not be supported as efficiently as in our approaches because they did not have access to the detailed structural details of the architecture. Furthermore, previous approaches did not support controller pipelining and non-uniform data forwarding.

4. Since NISC has no predefined instruction-set, it also does not have any assembly. However, to use it in practical situations it must provide a mechanism for low-level programming. We solved this problem by adding *pre-bound functions* and *variables* to the NISC compiler enabling low-level programming in C. These functions and variables are mapped directly to the hardware resources by the compiler. The pre-bound functions do not have body, are treated as operations, and can be similarly scheduled. This is different from intrinsic functions in compilers which either have a body, or are only used as compiler directives without any effect on the execution of the program. Pre-binding is much more flexible and productive than using assembly in the program. The pre-bound functions and variables have C syntax and can be easily mixed with the rest of the application. Also, they are automatically scheduled on one or more resources by the compiler and without user intervention. In contrast, assembly codes cannot be mixed with the rest of the program as easily and in the case of statically-scheduled architectures (i.e. microcoded and VLIW), they require the programmer to explicitly provide the schedule of the operations as well.
5. NISC has no predefined instruction-set and instead executes the program using very tightly coupled statically-scheduled control words generated by the compiler. Therefore a NISC component, especially when pipelined, cannot be arbitrarily interrupted. We showed that the interrupts can be safely serviced between basic blocks in NISC. Our solution included a minimal modification int the contoroller as well as adding an interrupt unit to the datapath. When interrupt support is not needed, the interrupt unit is removed from datapath and the extra logic in the

controller is optimized away during logic optimization. Therefore, our solution does not impose any extra overhead on NISC components not requiring interrupt.

6. We showed compiler algorithm, low-level programming, and interrupt support are the necessary and enough features for handling any behavior by NISC. To do so, we divide the behavior into a timed behavior that must be implemented by an HDL and an un-timed behavior that can be implemented in software (e.g. C). The un-timed behavior (software) accesses the timed behavior (hardware) via pre-binding; and the timed behavior notifies the un-timed behavior about its status via interrupt. The compilation algorithm combines all of these into cycle-accurate RTL for final implementation. To demonstrate this concept, we illustrated how different communication protocols can be added to NISC components. Communication protocols typically include a cycle-accurate part which cannot be described at behavioral level. Behavioral IP descriptions are preferred for increasing designer's productivity. However, IPs must be easily combined with different communication protocols in order to facilitate IP reuse as well as communication exploration. Since behavioral IPs are un-timed, it is impossible to combine them with the timed description of a communication protocol. The proposed approaches in the past either limit the supported types of communication interfaces, or require significant language extensions. In this thesis, we showed that by dividing the model of a communication protocol into a synchronous part that must be timed and an asynchronous part that can be un-timed, we can easily model and combine both IPs and communication interfaces in NISC without limiting the interface or relying on language extensions.

9.1 Future work

The NISC Technology can provide a much better way of synthesis from high level description to RTL. What were presented in this thesis were the basic necessities for developing a NISC compiler. The compilation techniques and the NISC toolset can be further improved and extended in the following directions:

- The priorities in the proposed heuristic scheduling algorithm mainly focus on performance. An interesting extension is including other parameters such as power consumption or thermal behavior in the algorithm.
- The scheduling algorithm can be further improved to support multiple register files or memories. These features are necessary for supporting clustered datapaths, which can be implemented more efficiently. Alternatively instead of modifying the scheduling algorithm, an external tool can utilize the pre-binding feature of the compiler and directly control the partitioning of variables into different storages.
- The NISC compiler can equally benefit from compiler optimization techniques that are developed for VLIW processors. Adding such techniques to the compiler as well as developing new NISC specific datapath aware code transformations can drastically improve the final performance of the design.
- NISC potentially provides better parallelism per bit width comparing to a VLIW machine with a similar datapath. It is interesting to compare a VLIW RTL implementation to that of a NISC with a similar datapath. Since all VLIW compiler optimizations are also applicable to NISC, we expect that with a

comparable performance, the NISC implementation be more efficient than its VLIW counterpart.

- In this thesis we did not explore datapath generation or customizations. Although it can be done manually, but having automatic datapath generation and refinement can further improve the designer's productivity.
- An interesting research extension is to develop formal techniques that can determine whether a given program is compilable on a given datapath. Such techniques can provide an excellent guideline for improving the compiler.
- Embedded applications typically include a lot of operations with constant operands. These constants can be effectively utilized for better customization. In its current state, the NISC compiler assumes that the control word has one or more constant fields with the same bit width. A more flexible technique can enable more datapath customization.
- Many of the techniques developed for ASIP can be applied to NISC as well. the algorithms that search for custom instructions, can be used to (a) add custom functional units to a NISC datapath, and (b) modify the code and replace the corresponding functionalities with proper pre-bound function calls.

Bibliography

[1] A. Agrawala, T. Rauscher, "*Foundations of Microprogramming: Architecture, Software, and Applications*", Academic Press, 1976.

[2] A. Fauth, J. Van Praet, M. Freericks, "Describing instruction set processors using nML", in Proceedings of *Design Automation and Test in Europe (DATE)*, 1995.

[3] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability", in proceedings of *Design Automation and Test in Europe (DATE)*, pages 485-490, 1999.

[4] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, A.Wieferink, H. Meyr, "A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language", *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, 2001.

[5] A. Iyer, D. Marculescu, "Power efficiency of multiple clock, multiple voltage cores", in *IEEE/ACM Intl. Conference on Computer-Aided Design (ICCAD)*, 2002.

[6] A. Orailoglu, D. Gajski, "Flow graph representation", in *Design Automation Conference (DAC)*, 1986.

[7] A. Sharma, K. Compton, C. Ebeling, and S. Hauck, "Exploration of pipelined fpga interconnect structures," in *IEEE International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004.

[8] A. Shrivastava, E. Earlie, N. Dutt, A. Nicolau, "Operation tables for scheduling in the presence of incomplete bypassing", in *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 194–199, 2004.

[9] A.M. Sllame, V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis", in *Euromicro Symposium on Digital System Design*, 2002.

[10] B. Gorjiara, D. Gajski, "FPGA-friendly Code Compression for Horizontal Microcoded Custom IPs", in *international Symposium on Field-Programmable Gate Arrays (FPGA)*, 2007.

[11] B. Gorjiara, D. Gajski, "Custom Processor Design Using NISC: A Case-Study on DCT algorithm", in *IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA)*, 2005.

[12] B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs", in *International Conference on Computer Design (ICCD)*, 2006.

[13] B. Landwehr, P. Marwedel, R. Dömer, "OSCAR:Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming", in proceedings of *Design Automation and Test in Europe (DATE)*, 1994.

[14] B. Pangrle, D. Gajski, "State Synthesis and Connectivity Binding for Microarchitecture Compilation," in *International Conference on Computer-Aided Design (ICCAD)*, pages 210-213, 1986.

[15] C. Hwang, Y. Hsu, Y. Lin, "Optimum and Heuristic Data Path Scheduling Under Resource Constraints," in *ACM/IEEE Design Automation Conference (DAC)*, pages 65-70, 1990.

[16] C. Lin and H. Zhou, "Retiming for wire pipelining in system-on-chip," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, 2003.

[17] C.-Y. Yeh, M. Marek-Sadowska, "Delay budgeting in sequential circuit with application on fpga placement," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 202–207, 2003.

[18] D. Gajski, "NISC: The Ultimate Reconfigurable Component," Center for Embedded Computer Systems (CECS), UC Irvine, Technical report TR03-28, October 1, 2003.

[19] D. Gajski, N. Dutt, A. Wu, S. Lin, "*High-Level Synthesis Introduction to Chip and System Design*", Kluwer Academic Publishers, The Netherlands, 1994.

[20] D. Kim, J. Jung, S. Lee, J. Jeon, K. Choi, "Behavior-to-placed RTL synthesis with performance-driven placement", in *International Conference Computer Aided Design (ICCAD)*, 2001.

[21] E. Ozer, S. Banerjia, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", in *IEEE/ACM International Symposium on Microarchitecture (MICRO-31)*, 1998.

[22] F. Brewer, B. Pangrle, A. Seawright, "Interconnection synthesis with geometric constraints", in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp 158-165, 1990.

[23] Forte Cynthesizer: http://www.forteds.com/

[24] Fraunhofer-Gesellschaft website: ftp://ftp.fhg.de/pub/layer3/mp3-bitstreams.tgz

[25] H. Akaboshi, "A Study on Design Support for Computer Architecture Design", *Doctoral Thesis, Depart. of Information Systems, Kyushu Univ.*, Japan, January 1996.

[26] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. V. Meerbergen, S. Note, J. Huisken, "Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms", *Proceedings of the IEEE*, 78(2), pp. 319-335, February, 1990

[27] H. De Man, J. Rabaey, J. Vanhoof, G. Coossens, P. Six, L. Claesen, "CATHEDRAL-II Computer-aided synthesis of digital processing systems", *Computer-Aided Engineering Journal*, 1988.

[28] J. Cong, Y. Fan, Z. Zhang, "Architecture-level synthesis for automatic interconnect pipelining," in *IEEE Design Automation Conference (DAC)*, pp. 602–607, 2004.

[29] J. Cong, Y. Fan, G. Han, X. Yang, Z. Zhang, "Architectural synthesis integrated with global placement for multi-cycle communication," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 536–543, 2003.

[30] J. Hennessy, D. Patterson, "*Computer Architecture: A Quantitative Approach*", Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[31] J. R. Ellis, "*Bulldog: A compiler for VLIW architectures*", Cambridge, MA: The MIT Press, 1986.

[32] J. Van Praet, D. Lanneer, G. Goossens, W. Geurts, H. De Man, "A Graph Based Processor Model for Retargetable Code Generation", in proceedings of *Design Automation and Test in Europe (DATE)*, 1996.

[33] J. Zhu, D. Gajski, "Soft scheduling in high level synthesis", in *Design Automation Conference (DAC)*, 1999.

[34] L. Singhal, E. Bozorgzadeh, "Fast Timing Closure through Interconnect Criticality Driven Delay Relaxation", in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2005.

[35] M. Byatt, "Data plane processing with configurable architectures", ARM white paper, 2003.

[36] M. Sivaraman, S. Aditya, "Cycle-time aware architecture synthesis of custom hardware accelerators", in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2002.

[37] M. Xu, F. J. Kurdahi, "Layout-driven high level synthesis for FPGA based architectures", in *Design Automation and Test in Europe (DATE)*, pp. 446-450, 1998.

[38] M.K. Jain, M. Balakrishnan, A. Kumar, "ASIP Design Methodologies: Survey and Issues", In *Proceedings of the Fourteenth International Conference on VLSI Design*, 2001.

[39] MentorGraphics Catapult C: http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm

[40] MiBench benchmark: http://www.eecs.umich.edu/mibench/

[41] MIPS32® M4K™ Core, http://www.mips.com

[42] MPEG Audio Decoder: http://www.underbit.com/products/mad/

[43] N. Ahmed, T. Natarajan, K.R. Rao, "Discrete Cosine Transform", in *IEEE Trans. On Computers*, vol. C- 23, 1974.

[44] N. Berry, B.M. Pangrle, "SCHALLOC: an algorithm for simultaneous scheduling & connectivity binding in a datapath synthesis system", in *Design Automation Conference (DAC)*, 1990.

[45] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, K. Van Nieuwenhove, "OptimoDE: Programmable Accelerator Engines Through Retargetable Customization", Hot Chips, 2004.

[46] N. Park, A. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," in *IEEE Transactions on Computer Aided Design*, pp. 356-370, 1988.

[47] NISC Technology website http://www.cecs.uci.edu/~nisc/.

[48] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, A. Nohl, "RTL Processor Synthesis for Architecture Exploration and Implementation", in *Design Automation and Test in Europe (DATE)*, 2004.

[49] P. G. Paulin, J. Knight, "Algorithms for High-Level Synthesis", in *IEEE Design & Test of Computers*, Vol. 6, No. 6, pp. 18-31, Dec. 1989.

[50] P. G. Paulin, J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 6, pp. 661-679, 1989.

[51] P. Marwdedel, "The MIMOLA Design System: Tools for the Design of Digital Processors", in *Design Automation Conference (DAC)*, 1984.

[52] P. Mishra, N. Dutt, "Architecture Description Languages for Programmable Embedded Systems", in *IEE Proc. on Computers and Digital Techniques (CDT)*, Special issue on Embedded Microelectronic Systems: Status and Trends, vol. 152, no 3, 2005.

[53] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven Exploration of Pipelined Embedded Processors", in *International Conference on VLSI Design*, 2004.

[54] P. Paulin, J. Knight, “Scheduling and binding algorithms for high-level synthesis”, In *ACM/IEEE Design Automation Conference (DAC)*, 1989.

[55] R. Camposano, “Path-Based Scheduling for Synthesis”, in *IEEE Transactions on Computer-Aided Design*, Vol. 10, No. 1, pp. 85-93, Jan. 1991.

[56] R. Camposano, “From Behavior to Structure: High-levelSynthesis”, *IEEE Design & Test of Computers*, 1990.

[57] R. E. Bryant, “Graph-based Algorithms for Boolean Function Manipulation”, in *IEEE Trans. on Computers*, 24.3, pp. 293-318, 1992.

[58] R. Leupers, P. Marwedel, “Retargetable Code Generation based on Structural Processor Descriptions”, in *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.

[59] R. Leupers, P. Marwedel, “Retargetable Generation of Code Selectors from HDL Processor Models”, in *Design Automation and Test in Europe (DATE)*, 1997.

[60] R. Nair, C. L. Berman, P. S. Hauge, E. J. Yoffa, “Generation of performance constraints for layout”, in *IEEE Trans. Computer-Aided Design*, vol. 8, no. 8, pp. 860–874, Aug. 1989.

[61] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist and M. Sivaraman, “PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators”, in *Journal of VLSI Signal Processing*, 31(2):127-142, 2002.

[62] Robert H. Sperry, David C. Farden, "A Microprogrammed Signal Processor", in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (ICASSP), Vol. 6, pp. 579- 582, 1981.

[63] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer, “The MIMOLA Language - Version 4.1. Technical Report”, Computer Science Dpt., University of Dortmund, Sept. 1994.

[64] S. Ghiasi, E. Bozorgzadeh, S. Choudhary, M. Sarrafzadeh, “Unified theory of timing budget management”, in Proc. *IEEE International Conference on Computer-Aided Design (ICCAD)*, 2004.

[65] S. Govindarajan, R. Vemuri, "Cone-Based Clustering Heuristic for List-Scheduling Algorithms", in *Proceedings of European Design & Test Conference (ED&TC)*, 1997.

[66] S. Habib, "*Microprogramming and Firmware Engineering Methods*", John Wiley & Sons, Inc., 1988.

[67] S. Heath, "*Embedded Systems Design*", Oxford: Newnes, 2003.

[68] S. Weber and K. Keutzer, "Using Minimal Minterms to Represent Programmability", in *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2005.

[69] S. Y. Ohm, F. J. Kurdahi, N. Dutt, M. Xu, "A comprehensive estimation technique for high-level synthesis", in *International Symposium on Systems Synthesis*, 1995.

[70] Tensilica Inc. http://www.tenisilica.com

[71] W. Qin, S. Malik, "Architecture Description Languages for Retargetable Compilation", in *The Compiler Design Handbook: Optimizations & Machine Code Generation*. Y. N. Srikant and Priti Shankar, CRC Press, 2002.

[72] W.E. Dougherty, D.E. Thomas, "Unifying behavioral synthesis and physical design", in *IEEE/ACM Design Automation Conference (DAC)*, 2000.

[73] XML Schema: http://www.w3.org/XML/Schema/

[74] XML: http://www.w3.org/XML/