UNIVERSITY OF CINCINNATI

September 18 , 20 02

I, Iyad Elias Ouaiss **,**
hereby submit this as part of the requirements for the degree of:

Doctorate of Philosophy

in:

Computer Engineering

It is entitled:

Hierarchical Memory Synthesis in Reconfigurable Computers

Approved by:

Ranga Vemuri, PhD (chair)
Harold Carter, PhD
Karen Tomko, PhD
Karen Davis, PhD
Stephan Pelikan, PhD

HIERARCHICAL MEMORY SYNTHESIS IN RECONFIGURABLE COMPUTERS

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTORATE OF PHILOSOPHY (Ph.D.)

in the Department of Electrical and Computer Engineering
and Computer Science
of the College of Engineering

2002

by

Iyad Elias Ouaiss

B.S. Computer Engineering, University of Cincinnati, 1994

Committee Chair: Dr. Ranga R. Vemuri

Abstract

A Reconfigurable Computer (RC) is a hardware platform that typically includes several programmable devices, memory devices, and possibly specialized devices such as analog-to-digital converters. Such high-performance platforms are capable of accommodating large designs while avoiding the time-to-market associated with ASIC implementations.

This work addresses the process of mapping data structures of an application onto the storage elements of RCs with hierarchical memories. In order to optimize the placement of data, several aspects of data mapping are addressed. Input specification styles and synthesis-related issues, physical resource conflicts and arbitration issues, several memory mapping techniques, and interaction between memory mapping and logic partitioning are presented and discussed.

The state-of-the-art in reconfigurable computers and their memory subsystems is reviewed and RCs are classified based on their architectures. The importance of hierarchical memories in RCs and the trend in increasing complexity is discussed.

A specification model that is well-suited for the memory mapping problem is introduced and the synthesis mechanism involved is described.

Several memory mapping techniques are presented and their applicability on existing hardware platforms is discussed. Integer Linear Programming (ILP) formulations are used and assignment techniques that cater to different RC features are developed. With this technique, small to medium sized designs are solved in a reasonable amount of time. Furthermore, these solutions are optimal.

On the other hand, with large sized designs, these ILP techniques become time consuming. Because of their slow execution speed and the complexity of the problem, a novel methodology that speeds up the execution while retaining a high mapping quality is introduced. This methodology divides the mapping process into two, global/detailed, sequential steps (ILP-based) and produces fast mappings at a relatively small quality cost.

One important issue when dealing with the memory assignment problem is resource conflicts. If the number of physical memories on the RC is limited, the assignment is forced to reuse these resources thus creating access conflicts. This problem is presented and an efficient arbitration solution that is well-suited for RC environments is proposed and implemented.

Finally, memory mapping techniques are extended to interface with logic partitioning tools. A full spatial partitioning framework is presented where memory mapping interacts with logic partitioning and optimizes the overall placement of both computations as well as data in the design.

Acknowledgments

I would like to thank my advisor Dr. Ranga Vemuri for his help and support during my graduate years. Thank you for creating a comfortable environment for me, an environment in which I felt accepted, respected, and valued.

I also wish to thank Dr. Harold Carter, Dr. Karen Davis, Dr. Karen Tomko, and Dr. Stephan Pelikan for serving on my committee and for providing me with valuable comments and feedback.

This research was supported in part by the U.S. Air Force, Wright Laboratory, WPAFB, under contract number F33615-97-C-1043. I thank Mr. Al Scarpelli, Mr. Darrell Barker, Mrs. Kerry Hill, and Dr. John Hines for their encouragement and support.

To the friends that I have met during my stay in Cincinnati, especially to Jane who touched my life in so many good ways.

In the last few critical weeks, I thank Marco for being so understanding and generous towards me, and Yara who, remotely, did everything possible to keep me relaxed and happy.

I thank my parents and my brother who were always present and supportive. My brother for always encouraging me, my mother for her unconditional love that was constantly felt from across the world, and my father for showing me what patience really means.

To my uncle, Dr. Saad Ghosn, for his support and his presence in my life during my stay in Cincinnati. Saad taught me to enjoy life by appreciating even the smallest things around me, and taught me to keep faith in people by accepting them unconditionally. Every day in my life, I try to live up to and fight for the beautiful values he has showed me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reconfigurable Computers (RCs) embody field-programmable devices that provide on-the-fly hardware programmability and enable the development of fast and efficient designs. RCs typically consist of several programmable devices, such as Field-Programmable Gate Arrays (FPGAs), and memory components connected by either a fixed or programmable interconnection network. RC platforms provide a robust alternative to ASIC implementations since they accommodate high performance implementations of complete designs at a fraction of the time-to-market cost. However, there is an imperious need for novel CAD techniques to exploit the power of RCs. Automated logic partitioning, data mapping, and high-level synthesis tools are crucial to explore more solutions in the design space in shorter time-to-market periods.

Reconfigurable Computers (RCs) offer a wealth of design gates, physical memory banks, and interconnection wires. These systems are built around programmable devices that offer high performance at low turn-around times. The recent improvements in integrated circuits technology lead to an outburst of programmable devices capable of accommodating large applications. These applications are no longer simple computational modules required to run at high speeds, but complete designs that contain computational modules as well as complex data structures. The major bottleneck in using reconfigurable hardware is the slow and tedious design process. Typically, good tool support exists for programmable devices (e.g. for an FPGA), but these tools are designed to optimize the performance of the device they target, without the global view of the reconfigurable system that might include several programming devices and memory banks.

To take full advantage of the large logic area that is available on field-programmable devices, reconfigurable computing systems were built to include one or more of these devices along with one or more memory components. This allowed a stand-alone and efficient execution

of an application on the RC platform: applications do not have to access data on slow off-board memory. The trend of getting physical storage closer to the programmable logic proved indispensable, as in the case of general-purpose computer systems. This prompted the recent production of on-chip memory banks in field-programmable devices. Current field-programmable gate arrays (FPGAs) offer a multitude of physical RAM banks in addition to the programmable logic gates. Therefore, with physical storage areas available off-board (e.g. on a host computer), off-chip (e.g. RAM structures on the RC platform), and on-chip (e.g. flip-flops and RAM banks in the FPGAs), existing reconfigurable computers include hierarchical and complex memory structures.

Since an RC system is a collection of devices (programmable gate devices and memory devices) that are interconnected in a specific way, there are two factors that are important for constructing a useful yet powerful RC platform:

- First, the number and type of components and their inter-connectivity must be selected carefully in order to *build* a useful system. For instance, too much memory and very few programmable gates might lead to a non-generic system capable of accommodating only a handful of applications. Such applications include several image processing algorithms that require a large storage space for the input, output, and temporary images being processed but a small circuit area for the algorithm. Similarly, a system containing a large design area and very little memory space is also a candidate platform for a few applications such as Boolean satisfiability and some partitioning problems. A careful balance between the number of different components is essential for creating a general-purpose RC system but is not sufficient. The inter-connectivity of the components and their connectivity to the host computer and environment are crucial. An RC coprocessor that communicates with the environment with only a few bits is probably not the ideal choice for real-time applications. Also, an RC that offers limited bandwidth between its different processing elements, restricts the type of applications that can be mapped onto it.

- Second, for any RC system to be useful, adequate CAD tools must exist to *exploit* the features of the underlying architecture. From partitioning tools and memory management tools to high-level, logic, and layout synthesis tools, it is necessary to include a global view of the RC architecture instead of just considering individual devices on the hardware. For instance, in an RC platform that contains multiple processing elements, an automated partitioning tool can divide the design into clusters that fit individually on the processing elements. Such tools allow the user to design applications without requiring board-level knowledge of the hardware. E.g. it is not important for the user to know how many processing elements exist on the board and how many wires are

running between them. Not only do such tools hide the architectural details of the hardware but they also allow the user to explore several solutions at a fraction of the time it would take if the task was performed manually.

This work addresses the second problem described above: given a specific RC coprocessor or a generic description of a family of RCs, tools are designed to exploit the capabilities of these platforms. Ideally, a CAD tool should be flexible to adapt to any architecture. For instance, a good logic partitioner should be able to target a board with four large processing elements as well as a board with six smaller processing elements. However, this is easier said than done: many features that exist on available platforms are unique and seem to require special handling. Also, for a given CAD tool, slight differences in hardware platforms sometimes lead to quite different solutions to the problem.

Nevertheless, the existence of good CAD tools can determine the success of a reconfigurable computer. Not only do they make the hardware platforms more appealing to application designers, but they also provide important clues to RC hardware designers. Consistent behavior of some CAD tools on a specific RC can point to either a flaw in the architecture or a powerful feature. For example, if partitioning tools consistently under-utilize one of the processing elements of the RC, this could be an indication that this processing element is badly placed on the hardware. On the other hand, if a board-level routing tool consistently succeeds, this could be an indication that the device inter-connectivity on the hardware is adequately designed.

Furthermore, CAD tools provide a framework in which the performance of new architectures can be assessed. How large should a RAM bank be to successfully map an application onto a specific RC? How many interconnection wires are needed between two processing elements? And what is the effect of having more RAM banks? All these issues and others can be investigated provided that the CAD tools target generic hardware architectures. By issuing the exact platform architecture along with the application to be synthesized to the CAD tools, a hardware designer can vary some parameters of the architecture and learn valuable feedback on their effect on the final mapped design.

One major issue that remains to a large extent not addressed by CAD tools is the automated task of data mapping and handling in RC platforms. Given an application that makes use of several variables and data structures, a tool is needed to automatically assign these data variables onto the physical storage banks of the RC coprocessor. Since multiple control threads are allowed to execute concurrently, the memory structures of RCs become more complex than the ones found in traditional computer systems. For a single thread of execution, simultaneous memory accesses only occur when caching or branch prediction

is required. However, in systems with multiple threads of execution, the application could benefit from complex memory structures where multiple memory accesses are taking place simultaneously. Furthermore, this problem is becoming more pronounced with the advent of *on-chip* RAMs in the newer families of FPGAs. Not only do these RAM banks provide a wide variety and quantity of storage elements, they also introduce new features such as variable configurations. The depth/width ratio of the majority of on-chip RAMs is variable: out of a set of possible configurations, the designer selects the depth/width pair that best suits the data being assigned to the bank.

The following sections provide a brief survey of RC architectures and the hierarchical structures of their memories, and introduce the memory mapping and synthesis problems. They define the concepts used in the following chapters and delimit the framework of this research.

1.1 RC Architectures

A generic adaptive reconfigurable hardware system, or a reconfigurable computer, is shown in Figure 1.1. An RC typically consists of the following components:

1. *Reconfigurable devices*: Although the capacities of a reconfigurable device, such as a Field Programmable Gate Array (FPGA) [1], has been increasing rapidly, it is still common to design large-scale RC systems using multiple reconfigurable devices such as FPGAs or special-purpose processors, on a single printed circuit board.

2. *Memory banks*: These are usually based on RAMs (Random-Access Memory) that provide data storage space for computation. They also provide means of data communication between the external environment and the reconfigurable devices. Off-chip memory banks can be viewed as either being *shared* between multiple reconfigurable devices, or *local* to a single reconfigurable device. On-chip memory banks are typically local to the device in which they are contained, however they can be accessed by external logic if required.

3. *Interconnection network*: Interconnection network is a set of *dedicated* or *programmable* connections between the components of the RC. A programmable interconnection network can be configured to provide a desired set of connections, whereas a dedicated network offers fixed hardware connections.

4. *Interface hardware*: The interface consists of a variety of application-specific IO connectors such as FIFO-based queues interfacing with the system bus, and/or extension

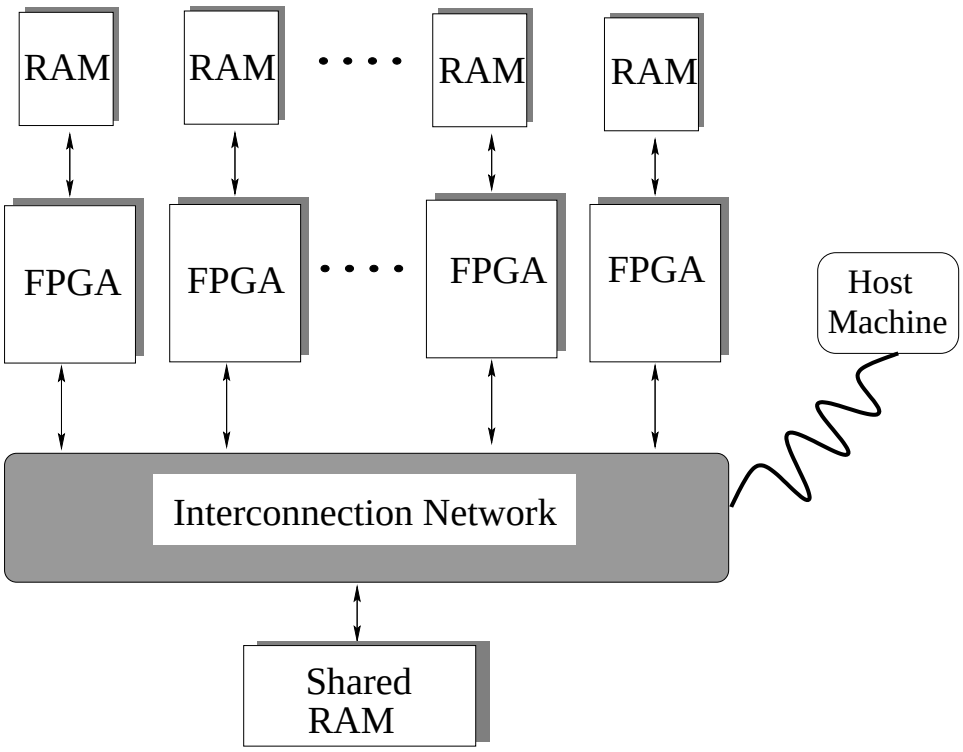


Figure 1.1: Generic Reconfigurable Computer

ports that expand the RC hardware. The interface is used for downloading input data to the RC, controlling its reconfiguration, and monitoring its execution.

Over the last decade, several RC architectures have emerged in order to meet the increasing computational demands of various application domains. Most RCs that are available currently can be classified based on the type of reconfigurable devices used:

1. *Field-Programmable Gate Array*: The FPGA [1–3] is usually a stream-reprogrammable SRAM-based device that can be reconfigured by serially loading the entire configuration bit-stream into a logical program register. For such devices, the reconfiguration time is sufficiently high (in the order of milli-seconds [1]) since the entire bit-stream needs to be reloaded. An RC architecture may be based on one or more FPGA devices. The *Xilinx Demo Board* [1] is an RC with a single-FPGA device. Examples of multi-FPGA RCs are: the *Wildforce* [4] platform, with four FPGAs each with a local RAM shown in Figure 1.2; the *GigaOps* [5] coprocessor, shown in Figure 1.3, has four Xilinx XC4000 series FPGAs three of which are available for user logic with 4MB DRAM and 256KB SRAM; and the *Garp* [6] coprocessor, shown in Figure 1.4, has a main processor that executes a MIPS instruction set and a reconfigurable array that correspond to CLBs

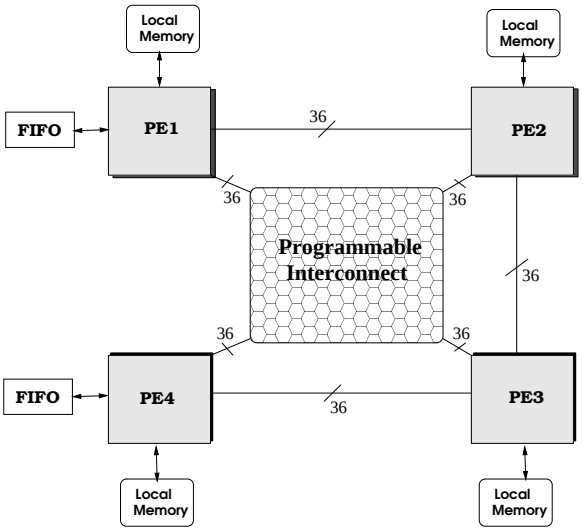


Figure 1.2: Wildforce Architecture

of the Xilinx XC4000 series FPGAs.

2. *Reconfigurable Processor Array*: The FPGA consists of an array of *fine-grained* programmable elements called CLBs (Configurable Logic Blocks) [1]. In contrast, the reconfigurable processor array is a device consisting of an array of *coarse-grained* processing elements. The Kress ALU array [7] is a coarse-grained device that consists of a set of 32-bit data paths with arithmetic logic units that are locally interconnected to each other in a mesh. Also, global data buses provide access to non-adjacent cells. Figure 1.5 depicts the architecture of the Kress array. The REMARC [8] is another example consisting of an array of 64 16-bit programmable units and a global control unit. Each programmable unit has a 32-entry instruction RAM, a 16-bit ALU, 16-entry data RAM, instruction registers, and other registers. The REMARC device, unlike an FPGA, uses instruction words to control the configuration of the processing elements as well as interconnection network. An RC architecture based on the REMARC reconfigurable device that targets multimedia applications is presented in [9] and is depicted in Figure 1.7. Another RC architecture based on a reconfigurable processor array is the RAW microprocessor [10]. The RAW processor is depicted in Figure 1.6 and it contains an array of 32-bit MIPS processors where each processing unit has 32KB SRAM of data memory and 32KB SRAM of instruction memory. Every processor is directly connected to its four adjacent neighbors.

3. *Partially Reconfigurable FPGA*: The recently introduced bit-reprogrammable FPGAs, the Xilinx 6200 series [11] followed by the Virtex series devices [12], make it possible to reconfigure selected portions of the device without having to reload the entire

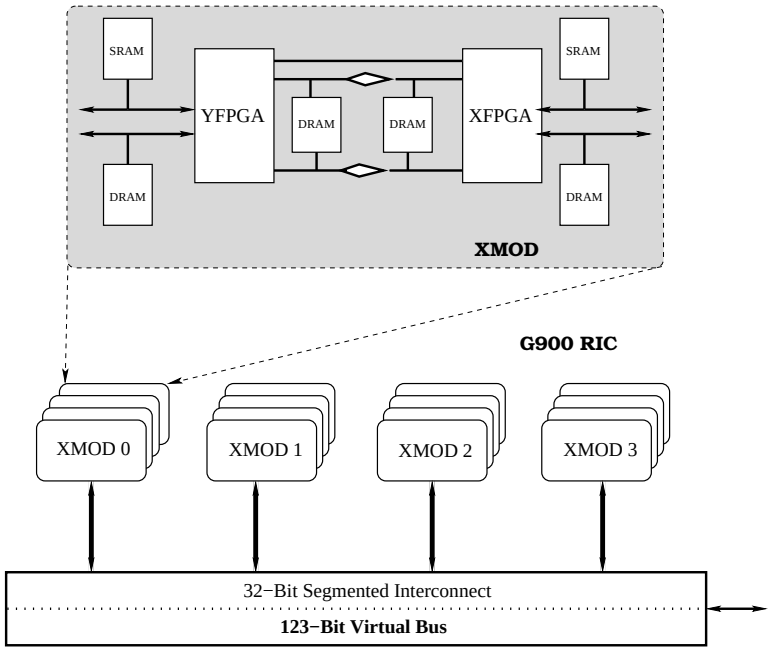


Figure 1.3: GigaOps Architecture

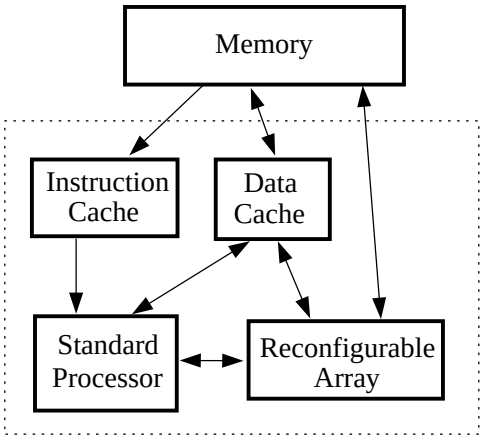


Figure 1.4: GARP Architecture

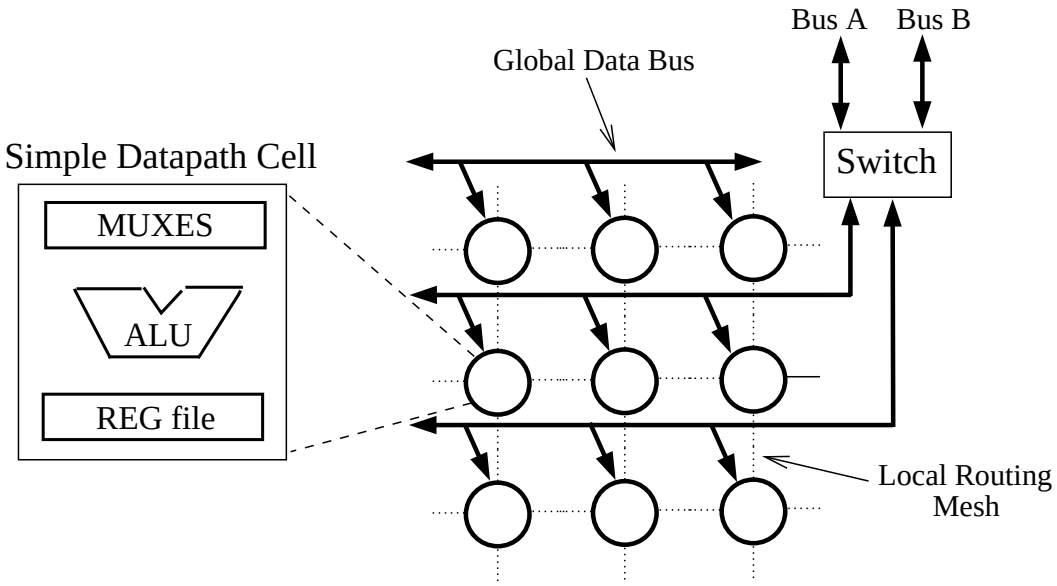


Figure 1.5: The Kress ALU Array

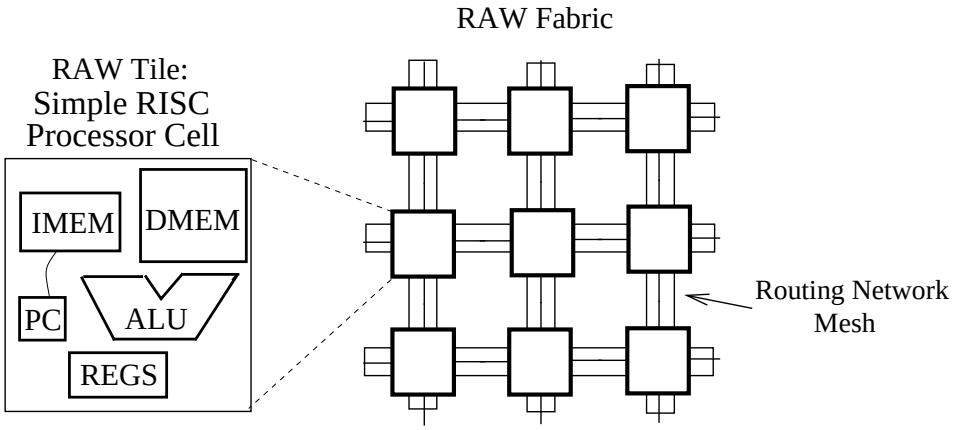


Figure 1.6: The RAW Processor

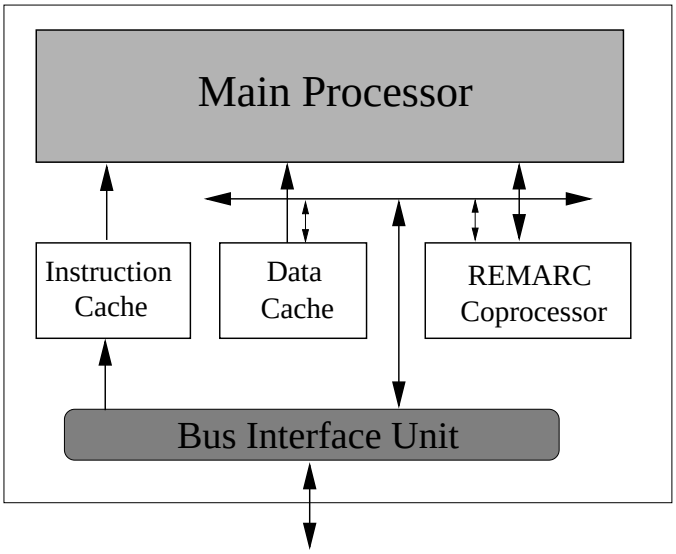


Figure 1.7: The REMARC Processor

bit-stream. In these architectures, the reconfigurable logic is fully accessible such that specific logic cells can be addressed, or *memory-mapped*, and reconfigured as desired. Bit-reprogrammable devices reduce the reconfiguration time by a factor of 1,000,000x, from milli-seconds to nano-seconds [13]. These devices are quite conducive to permit reconfiguration mid-way through an application execution. The *Firefly* [14] and *Virtual Workbench* [15] are examples of RCs based on a single partially reconfigurable FPGA. The *ACE Card* [16], shown in Figure 1.8 is based on two Xilinx XC6264 FPGAs each with a typical gate count range of 64,000 - 100,000 gates and 16KB registers. And, the *Wildstar* [17] platform, depicted in Figure 1.9, is another example of an RC based on multiple partially reconfigurable FPGAs. The Wildstar contains four partially reconfigurable Xilinx Virtex devices, however it does not support partial reconfiguration.

4. *Context-Switching FPGA*: This is an enhancement that allows complete reconfiguration of an FPGA at a rate far better than that of the standard FPGA. A context-switching FPGA device has the ability to store multiple configurations (or *context*s) and switch between them on a clock-cycle basis. Also, a new configuration can be loaded while another configuration is active (or in execution). The *Time-Multiplexed* FPGA [18], from *Xilinx Corporation* can store up to seven contexts and switch between them through an internal controller. The *Context-Switching* FPGA [19] from *Sanders - A Lockheed Martin Company*, can store four contexts and has a powerful cross-context data sharing mechanism implemented within the device. The *Dynamically Programmable Gate Arrays (DPGAs)* [20], developed at MIT, have four contexts and represent hybrid

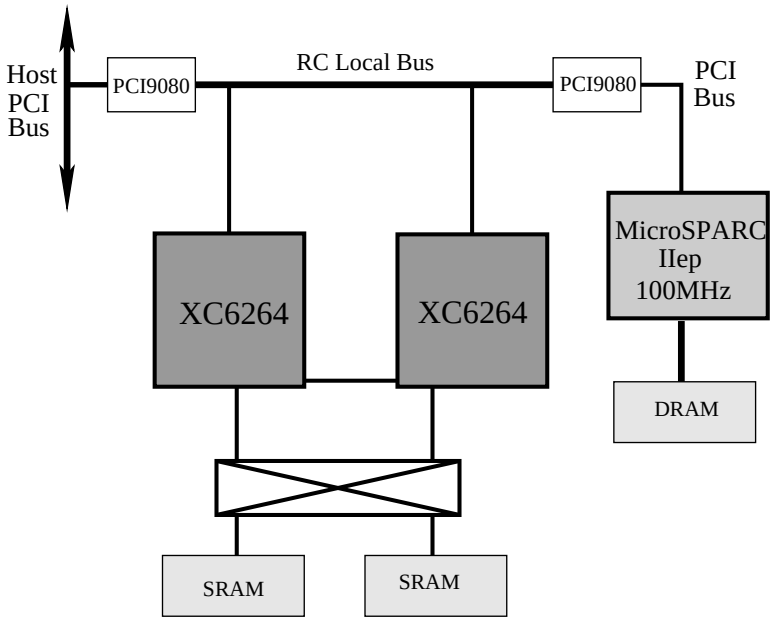


Figure 1.8: The ACE Card

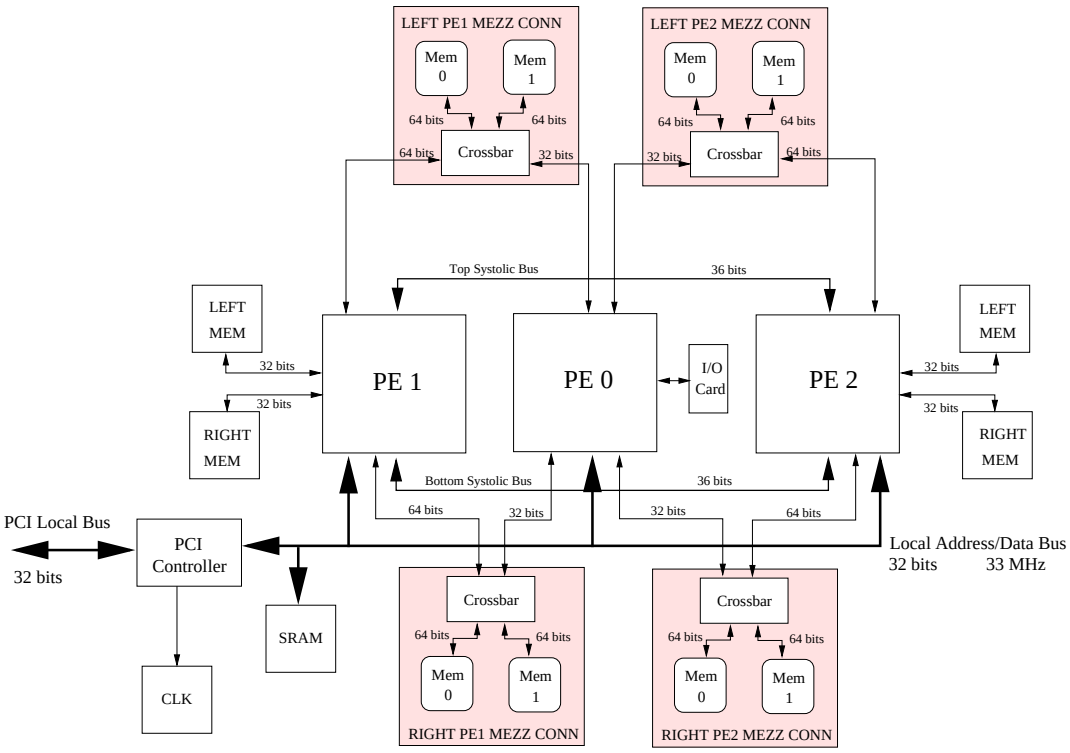


Figure 1.9: The Wildstar Card

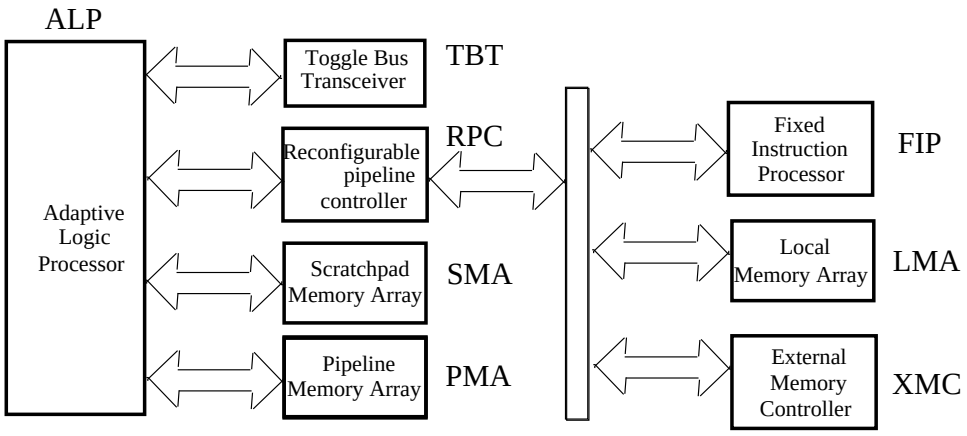


Figure 1.10: NAPA Hybrid Architecture

architectures lying between traditional FPGAs and SIMD arrays. Currently, context-switching devices are still experimental and have not yet been incorporated in RC platforms.

5. *Hybrid Processor*: This type of a hybrid architecture comprises of a main processor that is tightly coupled with a Reconfigurable Logic Array (RLA). A portion of the device area is devoted for a conventional microprocessor architecture and the remaining for a RLA. NAPA [21], depicted in Figure 1.10, is such a hybrid processor consisting of an *Adaptive Logic Processor* resource combined with a general-purpose scalar processor called the *Fixed Instruction Processor*. An RC architecture based on the NAPA processor is also presented in [21]. MorphoSys [22] is another RC architecture that has a coarse-grained reconfigurable logic array and a MIPS-like RISC processor.

From a different perspective, the RCs can also be classified based on their usage:

1. *Coprocessor*: RC coprocessors are subservient to a host processor. The host processor executes the application, occasionally configuring the coprocessor to perform a special function and delegating portions of the application, that need a special function, to the coprocessor. The host maintains the set of permissible coprocessor configurations as a library of "hardware functions". If the host is a high-end workstation computer, it is possible to keep a large number of configurations on the hard disk. If the host is a small DSP-style motherboard, then it is possible to store only a small number of configurations in the RAM. This number of configurations that can be stored in the system, impacts the synthesis process. Often coprocessor applications attempt to absorb the reconfiguration overhead through parallel execution of the host processor

and the coprocessor. RCs such as the Wildforce and the Wildstar are typical examples of coprocessors that can be monitored by a host computer.

2. *Embedded Processor*: One can view the RC as an embedded processor when it is not attached to any host processor. Embedded RC architectures contain a finite number of alternative configurations stored on ROMs (Read-Only Memory) on-board. A micro-operation system, also loaded in an on-board ROM, controls the loading of these configurations. RCs available in the market today usually are connected to a host computer and are used as coprocessors; however, boards such as the Telsys ACE Card have the means of running in a stand-alone mode. Not only can the programmable devices be configured and controlled without host interaction, but these devices are accompanied by a MicroSPARC embedded processor that provides general-purpose processing. The MicroSPARC processor can also act as an on-board controller to the programmable devices.

3. *Statically/Dynamically Reconfigurable Processor*: During the course of execution of an application the RC may be reconfigured only once to act as a *statically* reconfigurable processor, or several times to act as a *dynamically* reconfigurable processor. From an application perspective, statically reconfigurable RCs offer a finite set of hardware resources that can be configured to execute the application. On the other hand, dynamically reconfigurable RCs offer an infinite set of hardware resources, only a finite number of which can be used at any time during the execution of the application. All available RC platforms can be used as either statically or dynamically reconfigurable processors.

In this section, we described the generic RC architecture model, provided references to a number of RC architectures and classified them according to the reconfigurable device type and usage. In the following sections, we will briefly describe the memory subsystems in reconfigurable computers and design automation techniques for RCs.

1.2 Hierarchical Memories in RCs

Reconfigurable computers have evolved over the past decade especially with respect to their memory subsystems. There is a clear shift in industry to reconfigurable computers that contain complex memory structures. Emulation boards evolved from having virtually no memory components in the mid eighties and late eighties (such as the Xilinx FPGA demonstration board with a single XC4000 series device [23]), to boards having simple off-chip

Year Introduced	Device	RAMs (# banks)	Size (# bits)	Configurations
1998	Xilinx Virtex (2.5 V)	8 → 32	4096	4096x1 2048x2 1024x4 512x8 256x16
1999	Xilinx Virtex E (1.8 V)	16 → 208	4096	4096x1 2048x2 1024x4 512x8 256x16
2000	Xilinx Virtex EM (1.8 V)	140 - 280	4096	4096x1 2048x2 1024x4 512x8 256x16
2000	Xilinx Virtex II (1.5 v)	4 → 168	18432	16384x1 8192x2 4096x4 2048x9 1024x18 512x36
2002	Xilinx Virtex II Pro (1.5 v)	12 → 556	18432	16384x1 8192x2 4096x4 2048x9 1024x18 512x36

Table 1.1: Evolution of Xilinx FPGA On-Chip RAMs

memories in the early nineties (such as the Protozone processor [24]), to boards having substantial amount of off-chip memories in the mid nineties and late nineties (such as the Annapolis Microsystems' Wildforce reconfigurable computer [4]), to finally boards that have substantial off-chip memories in addition to on-chip memories in the late nineties (such as the Virtex-based reconfigurable computers: Annapolis Microsystems' Wildstar [17] and the SLAAC-1V [25]). Furthermore, this trend can also be seen in the development of FPGAs with on-chip memories. For instance, the turning point for on-chip memories at Xilinx [23] occurred with the introduction of the Virtex series devices. Xilinx FPGAs did not contain explicit on-chip memories before the advent of the Virtex series. Since then, on-chip memories have been evolving at a rapid pace. Table 1.1 depicts the evolution of the on-chip memory components, the Block SelectRAM [12, 26–29], starting with the Virtex, Virtex-E, Virtex-EM, Virtex-II and ending with the Virtex-II Pro series. This evolution took place in the last five years (1998 - present) since the inception of the Virtex technology.

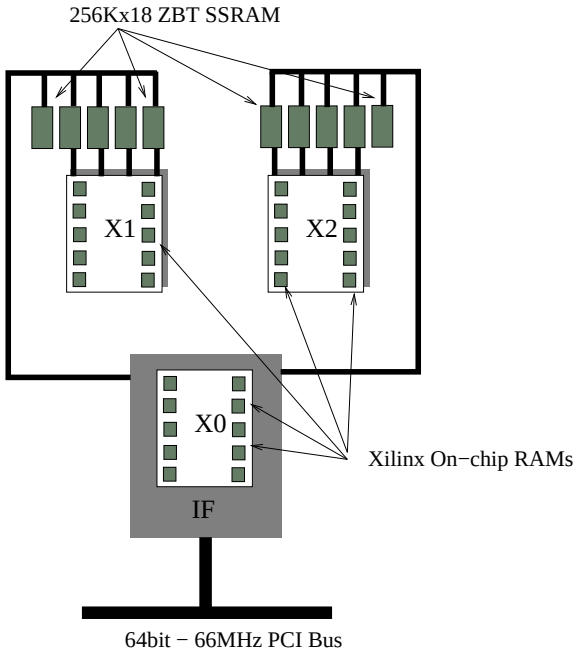


Figure 1.11: SLAAC-1V Memory Banks

Reconfigurable computers built around Virtex-style FPGAs consist of complex hierarchical memory subsystems. As can be seen from Table 1.1, an RC built around the Virtex-II Pro series FPGAs can have up to 556 on-chip RAMs per FPGA! Hence, a board consisting of four FPGAs could have in excess of two thousand on-chip memory banks in addition to some local memories and possibly shared memories among the several FPGAs. In comparison, prior to Virtex-style FPGAs, RCs were limited by the number of off-chip banks available which typically did not exceed five memory banks per FPGA. E.g. a four-FPGA board would usually have not more than twenty memory banks.

As an example, Figure 1.11 shows the SLAAC-1V board [25] that consists of three Virtex XCV1000 FPGAs (three-million equivalent logic gates) and ten 256Kx36 off-chip, local ZBT SRAMs. Each Virtex contains 32 on-chip BlockRAM banks. Thus, there are a total of 106 physical memory banks on this board, 96 of which are on-chip, dual ported, and configurable.

Figure 1.12 depicts the different memory structures that are available in reconfigurable computers. These storage elements form the hierarchical memory subsystem of an RC: they are usually classified from fastest to slowest depending on the relative proximity of the memory banks to the processing logic units. A generic memory subsystem in reconfigurable computers contain the following storage elements:

- *On-chip flip-flops:* These consist of the registers that coexist with the processing logic.

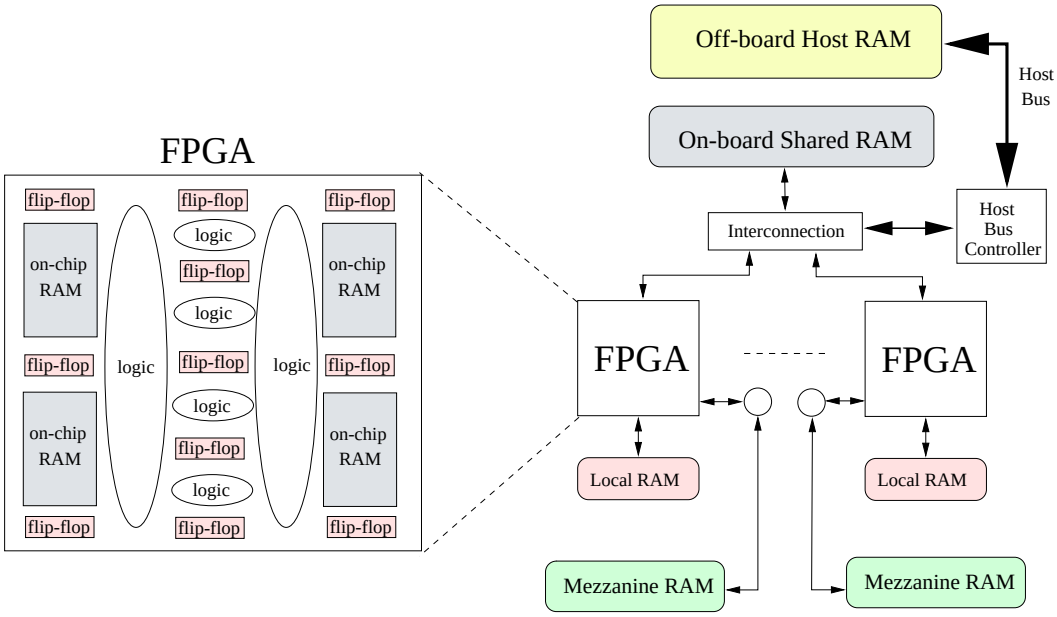


Figure 1.12: Generic Memory Subsystem

They are typically used to store small temporary variables of an application, to latch outputs of computational operations, or to break a path for shorter clock periods or more efficient pipelined designs. On-chip flip-flops provide fast access since no read/write latencies apply. They are also very abundant in a programmable device: the interconnection cost to route to and from these flops is usually negligible. They are depicted in Figure 1.12a.

- *On-chip RAMs:* These RAMs are usually limited in depth but are quite fast and flexible for configuration. They can be used as register files or as storage space for design input or output data structures. If used as design input or output storage, host access to these banks is required: a host computer attached to an RC must be able to write to and read from the on-chip RAMs without incurring a large performance penalty. Typically, access to on-chip RAMs is quite fast. These banks exhibit very short read and write latencies. Interconnection cost is low, however since the number of banks is limited on a chip, the interconnection cost is higher than in the case of on-chip flip-flops. On-chip RAMs are depicted in Figure 1.12a.

- *Off-chip/on-board local memories:* These are the most common memory banks available in RCs. They provide a large storage space for variables and data structures of the

application while not requiring complex interconnection. Since they are local memories, dedicated interconnection pins connect the banks to the corresponding processing unit. Interconnection costs are typically low, but read and write latencies could be substantial. Local memories are shown in Figure 1.12b.

- *Off-chip/mezzanine local/shared memories:* Another type of local memory comes in the form of a mezzanine card. Mezzanine cards are add-on modules that go onto on-board, off-chip connectors and provide large local storage banks for the processing units. Mezzanine local RAMs are shown in Figure 1.12b. Since these cards are easily detachable, they provide a nice feature to an RC system: mezzanine cards with different amount of memory storage can be replaced instantly. Furthermore, mezzanine memory cards can typically be replaced by cards with digital signal processing capability, analog-to-digital and digital-to-analog converters, etc.

- *Off-chip/on-board shared memories:* These banks provide large memory storage and ease the sharing of data across several processing units. Shared memories typically have multiple access ports to allow parallel access to the bank. Read and write latencies can be substantial and interconnection cost is usually high especially if no dedicated pins connect the processing units to the bank. A shared memory is shown in Figure 1.12b.

- *Off-chip/off-board shared memories:* These provide a virtually immense storage space at a high performance cost. These banks, or host memories, can hold data that need to be processed by both the RC and its host computer. The access to these banks is severely limited by the host bus, typically a PCI bus. The interconnection cost becomes very high and the RC must compete with all other devices accessing the host bus. An example of a host memory is shown in Figure 1.12b.

1.3 Memory Mapping and Synthesis

1.3.1 Reconfigurable Computer Applications

The application domain suitable for RCs usually covers problems that are computationally intensive and exhibit parallelism. These include digital signal processing algorithms [30,31], image processing [32], database search algorithms [32], multimedia algorithms [9], genetic algorithms [33], and neural network applications [34].

Designs have been primarily hand-crafted for various RC architectures and the results published. These examples include *Genetic Partitioning* (GP) [35], *Boolean Satisfiability*

(BS) [36], *Neural Networks* (NN) [37], *Image Correlation* (IC) [38] and *Mean Filter* (MF) [39]. An important feature is that all these examples have inherent data parallelism at different levels of abstraction. Most of the examples use little inter-device communication. Similarly, the usage of the RC memory is very little, with data mostly being hard-wired or self-generated. Speed-up in the examples using *Xilinx 4000* devices, required multiple FPGA devices without using host-to-device communication and using only self-generated data. Speed-up in the examples using *Xilinx 6200* devices, required the data to be distributed within each device using the feature of direct addressability of logic cells. Also, these design examples were carefully synthesized into parallel and pipelined structures.

Goldstein and Schmit [40] have compiled a similar variety of design examples that demonstrate a 10x speed-up when using an RC over a conventional microprocessor.

1.3.2 Memory Issues in RCs

As seen in the previous section, signal processing applications are good candidates for RC implementations since they are typically computationally intensive and can exhibit a good amount of parallelism. Furthermore, these applications usually involve a great amount of data and the efficient use of memory banks on the RCs becomes crucial. The objective of *memory mapping and synthesis* is two-fold:

- Map the data structures of the application to memory banks of the RC platform. The output of this phase is a mapping of each data structure onto exact locations in the memory banks of the RC. This step is often called *logical-to-physical memory mapping*.
- Once the mapping is complete, introduce address generation logic as well as arbitration to ensure proper memory accesses. This involves altering the computational tasks that access data structures in order to offset the addresses of the memory accesses. It also involves introducing a conflict resolution scheme, an arbitration mechanism, between computational tasks that are accessing data structures mapped onto the same port of a memory bank. This second step is referred to as *memory synthesis*.

Given the complexity of the memory substructures of RCs, as described in Section 1.2, and the number of data structures that an application can contain, the problem of efficiently mapping the data onto the RC becomes important. The problem size is large and a manual memory synthesis approach is almost impossible. Furthermore, the exploration of good mapping solutions is required since data accesses of the design can largely affect the performance of the application.

The next chapter formally introduces the memory mapping problem and reviews available CAD techniques.

1.4 Outline of the Thesis

This work introduces the components involved in the design for reconfigurable computers and addresses the missing links in the memory mapping and synthesis process: Chapter 2 defines the problem of memory mapping and reviews available work. It also formally introduces the problem with respect to Integer Linear Programming (ILP) and refers to Appendix A for a description summary of all variables and parameters defined in the thesis. Chapter 3 exposes the specification styles used in the design for reconfigurable computers.

Chapters 4, 5, and 6 address the problem of mapping design variables onto storage banks of RC systems: Chapter 4 introduces the memory mapping in an ILP framework and provides a mapping solution for RAM banks with single configurations. At the price of added complexity and lower execution speed, Chapter 5 extends the formulation to RAM banks with multiple configurations. To simplify the problem in the latter case, Chapter 6 divides the mapping into two steps that significantly reduce the complexity of the formulation while retaining the quality of the assignments.

Chapter 7 presents the general partitioning methodology that includes both the mapping of data structures to memory banks and the assignment of computational tasks to processing elements of the RC system. In addition, it provides a foundation for the spatial partitioning ILP formulation by improving the formulations presented in the earlier chapters. Chapter 8 presents ILP formulations and solutions to the three spatial partitioning views defined in Chapter 7.

Chapter 9 discusses the problem of resource conflicts, a common problem in the design for RCs, and describes an arbitration solution. Chapter 10 discusses memory synthesis issues for the specification of a design. Finally, Chapter 11 concludes the discussion by listing the contribution of this work and presenting possible extensions to this work.

Chapter 2

Memory Mapping

2.1 Introduction

One step in the synthesis for FPGA-based reconfigurable computers involves mapping the design data structures onto the physical memory banks available in the hardware. The advent of Xilinx Virtex-style FPGAs and of hierarchical memory schemes on reconfigurable boards introduced an added complexity to this mapping. The new RC boards offer a wealth of memory banks, many of them on-chip (such as the BlockRAMs available in the Virtex architecture), and many of them offering variable number of ports and several depth/width configurations. Along with the external RAMs, a hierarchy of memories with varying access performances are available in a reconfigurable computer. It becomes critical to perform a good mapping to achieve optimal design performance.

When designing a circuit in digital hardware (ASIC or FPGA), the task usually consists of mapping operations of the application onto the hardware logic and mapping the data structures onto physical memory banks of the hardware.

A synthesis process typically takes the design from a level of abstraction to a lower one getting the design closer to a hardware implementation. While doing so, it is important to intelligently assign data structures to physical memory banks in order to minimize the overall cost of memory accesses. This is especially the case in today's data-intensive applications that require a very high-speed data access and large storage area. In image processing and speech recognition applications, for instance, the memory component of an ASIC implementation could contribute to more than 75% of the entire design area! Such a high area has a direct effect on memory performance.

Memory mapping has been researched in the case of ASICs where the mapping consists of

selecting memory components from a library and/or selecting where the memory components are placed and the way in which they are connected to the hardware logic. This view assumes that the hardware is custom-built for the application: starting with an application, the synthesis process selects the best physical memory components and interconnects them to the logic hardware in the most efficient manner. In that case, the physical constraints consist of area requirements, number of available pins, etc., and the selection of memory components is limited by the richness of the module library.

In reconfigurable computing, the problem is different: starting with a fixed hardware platform where only logic and few interconnection switches are programmable, the synthesis process tries to map, in the best possible way, the application onto the RC. Thus, the number of memory banks, the type of memory banks, and, to a certain extent, the way they are connected to the processing units, are usually fixed. Before the introduction of Xilinx Virtex-style FPGAs, the mapping was limited to the physical banks that are external to the processing units; and since the number of external physical banks was typically low, designers would perform the mapping manually or use a very simple assignment algorithm to do so.

However, in the case where a large number of physical memory banks is available, a manual assignment is very difficult. Furthermore, the Virtex-style FPGAs introduced three new complexities to the mapping: First, there are potentially a large number of on-chip memory banks (*BlockRAMs* for Xilinx Virtex devices [12], *Embedded Array Blocks* for Altera FLEX 10K devices [41], and *Embedded System Blocks* for Altera APEX E devices [42]). In the Xilinx Virtex-II Pro FPGAs, this number ranges from 12 BlockRAMs for the XC2VP2 device to 556 BlockRAMs for the XC2VP125 device! Second, the number of memory ports for each on-chip memory bank could be greater than one; two ports for every Xilinx Virtex BlockRAM, each port accessing the same physical space. And third, the depth/width ratio of each memory bank could be variable; for each Virtex BlockRAM, the five configurations are shown in Table 2.1. Taking into account these features, and adding the external memory banks to the on-chip banks, the problem of logical-to-physical memory mapping becomes quite complicated.

Little work has been done to automatically assign data structures to complex RC systems; furthermore, features of on-chip Virtex-style FPGAs have not yet been incorporated in the synthesis process. This chapter exposes the different features of a complex, hierarchical hardware memory structure and presents a formal integer-linear programming notation as an approach to memory mapping.

Configuration Number	Depth (# words)	Word Size (# bits/word)
1	4096	1
2	2048	2
3	1024	4
4	512	8
5	256	16

Table 2.1: Xilinx BlockRAM Configurations

2.2 Previous Work

There exist several studies on utilizing memory modules in data path synthesis. As mentioned in Section 2.1, the majority of memory mapping studies focus on the ASIC implementation. The tools pick a set of physical memory modules from a library of available banks and select the interconnection structure to connect the processing units to the memory banks. Very few studies target reconfigurable boards where memory banks and interconnection structure are fixed *before* synthesizing the application.

Memory access optimizations are targeted in [43], where the goal is to optimize memory accesses in pipelined designs. Synthesis transformations take advantage of on-chip RAMs and intelligent schedules are produced to maximize parallel accesses. Therefore, memory mapping takes place *during* synthesis and does not address hierarchical memory banks.

Several studies were conducted in the realm of high-level synthesis where cliques of the design variables are partitioned to form data segments. Some researchers performed this task without taking into consideration the interconnection structure of the hardware [44], while others consider the cost of interconnection during variable grouping for multi-port memory banks [45].

An integer linear programming model is used in [46, 47] to group registers and form multi-port memory modules. Instead of dealing with fixed hardware configurations, these studies construct the hardware based on minimizing the needs of memory banks and minimizing the cost of interconnection.

In the above references, interconnection cost is one of the most important constraints, and the research aimed at minimizing this cost as well as minimizing the number of physical memory banks . With on-chip memory and fixed external memory banks and interconnection structures, the problem becomes different. On-chip memory does not require off-chip communication thus reducing external interconnections. Furthermore, given a fixed memory

structure on an RC board, minimizing the number of required memory banks might not be the best mapping solution; as long as the mapper does not exceed the physical storage area, it should be allowed to use as many banks as it deems fit.

It is worth mentioning that, contrary to the older works mentioned above, FPGA on-chip memory banks are targeted in [48, 49]. However, the banks are used for implementing logic. These studies address technology mapping of logic onto on-chip memories, whereas the storage capability of the banks is not considered.

In [50], memory mapping for FPGAs with on-chip memories is addressed; however, only single-ported memory banks are assumed. The same technique was improved in [51] so that the mapping caters to recent FPGAs containing dual-ported on-chip banks. In both works, the focus is on hardware containing a single type of memory banks (either single or dual ported) and does not simultaneously consider off-chip memories that exhibit different performance numbers.

Given data structures and access constraints to these structures, [52] finds a legal packing of the logical segments into the physical segments while minimizing the area. Again, since the storage area in the RC framework is fixed, it might not be beneficial to minimize the area. On the other hand, instead of meeting access constraints to the data structures, our work targets on minimizing all memory accesses. The work can be extended in the event hard access constraints must be met on some data structures.

As classified in [53], the problem of *logical-to-physical memory mapping* [50] can be divided into two steps: Firstly, translating the storage requirements onto *logical memories*; i.e. forming the data structures needed by the design. Secondly, mapping the logical memories onto the *physical memories* of the hardware; i.e. assigning the data structures to memory banks. In [53], the mapping of logical memories is onto physical memories chosen from a library; however, in this chapter, the mapping is onto a fixed architecture dictated by the RC board.

An analysis of several memory mapping studies is presented in [53]. The authors compare the techniques based on the number and the type of logical memories and physical banks considered simultaneously. In addition, the book by Catthoor et al. [54] and the book by Panda, Dutt, and Nicolau [55] provide an excellent source for topics in memory system optimization, exploration, and management.

2.3 Problem Definition

Several features exist for different types of RC boards. This section generalizes the approach of memory mapping by targeting a flexible hierarchical memory structure.

A memory mapper must take as an input both the architecture of the target RC board as well as a description of the design to be synthesized and mapped onto the board. For this chapter, it is assumed that the RC board contains only one processing unit. As part of future enhancement, this work will be extended to multi-processing units where logic placement and pin constraints during routing will be addressed.

2.3.1 Architecture Description

The RC board architecture is described by a collection of *memory types*. There could be several instances of each memory type, but all instances share the same storage and access speed specifications, and share the same proximity and ease of access from the processing unit.

For each memory type, a *number of instances* tells the mapper how many instances of each type exist on the board. The *number of ports* of a type is one if the memory is a single-ported memory, two if dual-ported, etc. As shown in Figure 2.1, the depth/width ratio of a memory could be variable; the *number of configurations* for each type is the number of possible settings of each port of that type.

The *number of words* (depth) and the *number of bits per word* (width) of a type are unique numbers if only one configuration exists. Otherwise, these are equal-length lists of numbers describing the possible configurations of the depth/width ratio. Entry i of the depth list together with entry i of the width list correspond to configuration i. It is assumed that the capacity of each configuration is a constant; i.e.

$$\forall_{i \in configurations}$$

$$depth(i) * width(i) = capacity(memory) \tag{2.1}$$

The access latency for each type of memory is variable; the *read latency* is the number of clock cycles required after performing a read and before getting valid data out of the memory bank. Similarly, the *write latency* is the number of clock cycles required after performing a write operation and before the data is correctly stored in the memory bank.

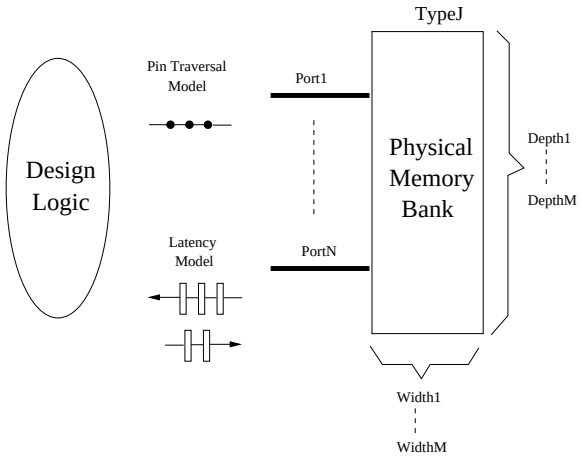


Figure 2.1: Generic Memory Bank

Finally, with respect to the physical location of the memory bank, the *number of pins traversed* depicts the proximity of the physical memory bank to the processing unit. If a bank is on-chip, zero pins are traversed. If an off-chip bank is directly connected to the memory, two pins are traversed. If an indirect connection exists between the processing unit and the external memory bank, then additional pins are traversed. In general, the aim is to map data structures to physical banks that are as close as possible to the processing unit; the further away they are, the larger the impact on the overall memory access performance.

A generic physical bank is shown in Figure 2.1. The latency model captures the number of read and write clock delays and the pin traversal model captures the number of pins traversed between the processing unit and the memory bank. In the Figure, bank type J is an N-ported memory, has M different configurations: depth*1*/width*1*, depth*2*/width*2*, ..., depth$M$/width$M$.

2.3.2 Task graph Description

On the design side, a description of the data structures is required. Since this work focuses on placing the data structures on the physical banks, it is assumed that the structures are already formed.

For each data segment in the design, the *number of words* (depth) in the segment and the *number of bits per word* (width) are required. A footprint analysis of the memory accesses could tremendously help in guiding the mapping process: e.g. data segments that are extensively accessed should be assigned to faster and closer physical banks.

2.3.3 Conflict Description

During synthesis of a design, scheduling determines the *life times* [56,57] of the variables and data structures. This life cycle analysis could further improve the memory mapping since segments that can overlap could be placed in the same storage area, thus decreasing the total storage requirement. For this purpose, the mapper needs to know which data segments life cycles overlap. A set of *conflicting pairs* captures this requirement; pair (L1, L2) means that data segment L1 cannot share storage space with segment L2.

Note that on one extreme, if no conflicting pairs are given, all segments could be ideally mapped to the same physical bank. On the other end of the spectrum, if all conflicting pairs exist, no overlapping of storage space can take place.

2.4 ILP Formulation

A 0-1 Integer Linear Programming (ILP) model is presented in this work. This approach yields an optimal solution to the mapping problem, however, it typically takes a very long time to execute and might not finish execution within reasonable time for larger problems.

The advantage of using this approach is that it exposes the constraints and objectives of the problem and pinpoints areas that introduce added complexities to the model. The ILP method is used here as a backbone solution to the problem where an interaction with heuristical approaches would result with fast, low-cost solutions.

It should be noted that linearization techniques are used at times when non-linear constraints arise. A survey of linearization approaches can be found in [58].

For the formulation presented in this section, we assume the following notation. Note that Appendix A provides a description summary of all variables and parameters defined in the thesis. There are M data structures:

$$DS = \{DS_1, DS_2, ..., DS_M\}$$

to be mapped onto N different types of physical memory banks:

$$PB = \{PB_1, PB_2, ..., PB_N\}$$

Note that N is not the number of available banks on the RC system. There could be multiple instances of each type of memory bank.

For each logical data structure d, we have:

$$\begin{cases} D_d & \text{Number of words in segment } d. \\ W_d & \text{Number of bits per word in segment } d. \\ RA_d & \text{Total number of estimated reads from segment } d. \\ WA_d & \text{Total number of estimated writes to segment } d. \end{cases}$$

For each type of physical memory bank t, we have:

$$\begin{cases} I_t & \text{Number of banks of type } t. \\ P_t & \text{Number of ports in a bank of type } t. \\ C_t & \text{Number of depth/width configurations in a bank of type } t. \\ D_t & \text{Array of number of words in a bank of type } t. \\ W_t & \text{Array of number of bits per word in a bank of type } t. \\ RL_t & \text{Read latency in number of clock cycles.} \\ WL_t & \text{Write latency in number of clock cycles.} \\ T_t & \text{Number of pins traversed from the processing unit to a bank of type } t. \end{cases}$$

where the depth/width ratio variables are:

$D_t = \{d_1, d_2, ..., d_{C_t}\}$ and: $W_t = \{w_1, w_2, ..., w_{C_t}\}$

Finally, there are Q conflict pairs in the design where each associates two logical structures: (DS_x, DS_y) where:

$x \neq y$.

The following are some simplifying assumptions made in this work; they can be easily alleviated by adding new constraints and/or new modeling variables to the formulation:

- All ports of the physical banks are assumed to be read/write ports. Overlooking capacity constraints, any data structure could be mapped to any port of a physical bank.
- If a multi-ported physical bank has several depth/word configurations, it is assumed that each port can have its own configuration setting. This is in agreement with the Virtex BlockRAM architecture. Note that forcing all ports to have the same configuration setting, simplifies the ILP formulation.

- A data structure that is too deep or too wide to fit on any single physical bank will be mapped onto more than one bank. The mapper should select banks having the same type in order to preserve similar access latency: an access to any word in a data structure should have the same latency irrespective of the location of the word within the structure.

Finally, the remaining notations pertain to the 0-1 variables used in the model. Z_{dt} associates a data structure to a memory type:

$$Z_{dt} = \begin{cases} 1 & \text{if data structure } d \text{ is assigned to some} \\ & \text{instance(s) of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

Z_{dt} is used to force an over-sized data structure to be split across banks of the *same type*. Similarly, X_{dtip} associates a data structure to a specific physical bank and to a specific port of that physical bank instance:

$$X_{dtip} = \begin{cases} 1 & \text{if part or all of data structure } d \text{ is assigned} \\ & \text{a to port } p \text{ of instance } i \text{ of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

And, only for multi-configuration physical banks (i.e. $C_t > 1$), Y_{tipc} sets a specific configuration to a port of a memory bank:

$$Y_{tipc} = \begin{cases} 1 & \text{if configuration } c \text{ is selected for port } p \\ & \text{of instance } i \text{ of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

2.5 ILP Solutions

This work proposes two general ILP methodologies for solving the memory mapping problem:

1. *Flat approach:* This approach implements memory mapping techniques that perform a complete memory assignment in a single step. The inputs to this tool is the non-mapped data structures and the hardware architecture and the output clearly maps each data structure to exact locations on the RC platform. Since ILP techniques search for optimal solutions, the formulation becomes quite lengthy and the solution time explodes for large problems.

2. *Two-layer approach:* Since the flat-approach formulation becomes quite lengthy and the solution time explodes for large problems, the two-layer approach divides the logical-to-physical memory mapping process into two steps: first, *global memory mapping* assigns each data structure in the application to one *type* of physical memory banks. Second, *detailed memory mapping* performs the lower-level assignment; it restructures the data segments and assigns them to specific banks of the type that was dictated by global mapping. Since all banks of the same type share the same performance and architecture specifications, detailed memory mapping does not affect the overall memory mapping cost. The optimization goal is thus sought after during global memory mapping, and the complexity of the global mapping is reduced by avoiding detailed mapping. By reducing the size of the problem, an ILP formulation becomes simpler and the solution is obtained faster. As a result, designs with several hundred data structures and memory banks are efficiently handled.

2.6 Conclusion

This chapter establishes the foundation for the next few chapters. The problem of memory mapping is defined and existing literature is reviewed. Since this work focuses on integer linear programming solutions to the memory mapping problem, ILP notations are defined and the different parameters are introduced.

In the ILP approaches followed in the next chapters, the same notation and assumptions are made about the data structures of an application as well as the hardware characteristics of the RC platform. This chapter described these notations and assumptions while attempting to keep the formulation as general and as flexible for expansion as possible.

Chapter 3

Application Specification for RCs

3.1 Introduction

This chapter describes the specification of an application in the RC framework[1]. The specification language used to model an application is very important since it molds the application in a way such that synthesis tools can process it. Furthermore, a good modeling language exposes many characteristics of the application: parallel threads of execution can be neatly presented, data structures readily formed, etc. In memory synthesis, the requirements of an efficient modeling language include being able to introduce address generation logic without intrusive modifications. Once data structures of the application are mapped to memory banks of the hardware, the process of adding address generation logic and arbitration mechanisms should be straightforward. Also, for memory synthesis, a succinct representation of the variables and data structures of an application is essential. In order to map data structures of the application onto banks of the RC, data structures should be extracted from the application and presented in a form suitable for synthesis.

For the memory assignment problem described in the thesis, a hierarchical representation of an application can capture the implicit parallelism of computational tasks in the design while hiding the flow of each task's control thread. Furthermore, this hierarchical representation successfully captures the relationship of computational tasks with the variables of the design.

The hierarchical representation presented in this chapter consists of two specification styles (an illustrative example is shown in Figure 3.1):

- *Fine-grain representation:* First, each computational task is represented in the Behav-

[1]This work was done in collaboration with the SPARCS [59] research team.

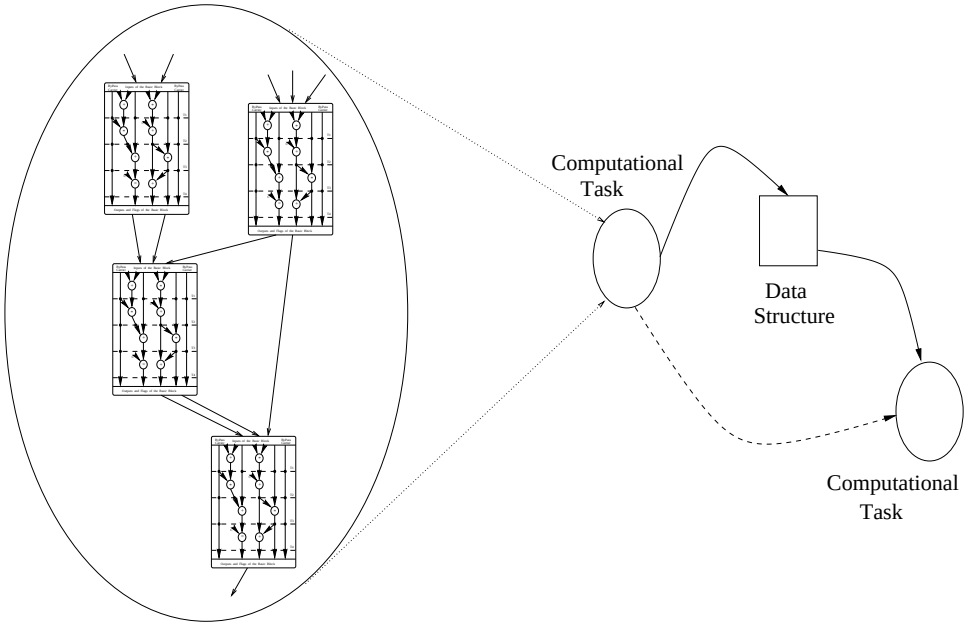


Figure 3.1: Hierarchical Modeling

ior Blocks Input Format (BBIF). This model, illustrated in Figure 3.1a is a hierarchical CDFG representation that captures the behavior of computations. Section 3.2 further describes the BBIF format and Appendix B provides further formal definitions.

- *Coarse-grain representation:* Second, the overall flow graph is represented in the Unified Specification Model (USM) format. The USM, illustrated in Figure 3.1b captures the concurrency and coordination model of a design in a style compatible with hardware description languages such as VHDL. Section 3.3 defines the USM format.

3.2 BBIF Specification

A computational model is needed to capture the behavior of the fine-grain level of parallelism in a design. The Behavior Blocks Input Format (BBIF) [60] is a hierarchical CDFG representation with features well-suited for High-Level Synthesis. A BBIF model represents a behavioral *task* with a single thread of control. The BBIF is organized as a list of behavior blocks, where the data flow and computations are captured within each behavior block, while the control flow is captured at the inter-block level. The task interacts with the environment through design input and output ports that are visible across all behavior blocks. The control flow starts at the first behavior block and transfers from one block to another through

the branch construct provided at the end of each block. The `branch` statement specifies either an unconditional transfer to a single successor block or a conditional transfer to *one* of the series of successor blocks.

In conjunction with the USM specification presented in the next section, the BBIF specification provides a modeling environment in which inter-task as well as intra-task parallelism can be efficiently expressed.

The USM specification, presented in the following section, plays an important role since the memory mapping problem is concerned with the inter-task interface and protocol. However, as part of future work, the BBIF specification will be an instrumental tool in performing the task of memory synthesis. Hence, further BBIF definitions and notations are provided in Appendix B. A formal and more detailed description of the BBIF model is provided in [60].

3.3 USM Specification

3.3.1 Introduction

This section proposes a Unified Specification Model (USM) of concurrency and coordination compatible with VHDL. The specification model embodies a uniform treatment of computation, communication channels, and memories, facilitating its use across a variety of synthesis applications. We discuss synthesis semantics of the USM representation and the advantages of the USM synchronization model in comparison to similar VHDL motivated representations.

VHDL has been used for behavior level specifications for a variety of high-level synthesis tools, hardware-software co-synthesis systems, and adaptive system synthesis environments. VHDL provides a rich set of high-level constructs to permit succinct specification of concurrent and coordinating processes. A variety of intermediate representations [57, 61–65] have been proposed to capture various specification elements in VHDL in a form suitable for further processing during synthesis. These include data-flow graphs (DFG), mixed control-data flow graphs (CDFG), timed decision tables (TDT), and various flavors of graph-based and table-based formalisms sometimes augmented with global flow information such as module call graphs (MCG). Although many of these representations share common features, they also have application-specific features that inhibit their use across various types of synthesis systems.

This section presents an overview of the Unified Specification Model (USM), and provides

insight on compatibility with VHDL.

The USM representation can be used for: 1) high-level VLSI synthesis [66] where the goal is to synthesize a CMOS ASIC; 2) hardware-software co-synthesis [67] where the target architecture contains a general-purpose processor to implement software tasks and a coprocessor to implement the hardware tasks; and 3) adaptive system synthesis [59] where the target architecture is a dynamically reconfigurable multi-FPGA board with both local memories for each FPGA and a shared memory, and a crossbar type communication fabric.

In the next sections, we present a detailed description of the Unified Specification Model and its semantics.

3.3.2 Unified Specification Model

The Unified Specification Model is a hierarchical representation for specifying the behavior of a design. The designer can also specify the behavior of the environment in which the design is to execute. A USM example is shown in Figure 3.2. At the highest level of the USM are two types of objects called tasks and memory segments. Tasks in the USM represent elements of computation and memory segments represent elements of data storage. Tasks are classified into design tasks and environment tasks. Environment tasks are written primarily to specify the I/O model between the design tasks and the environment. Hence, design tasks are those that are synthesized and environment tasks are only used to extract information about the I/O interface and protocol.

All tasks in the USM are simultaneously executing so as to model concurrency. USM objects (tasks and memory segments) can be connected through edges that are either channels or dependencies. USM allows the specification of dependencies among tasks in order to represent coordination. Channels are used to represent inter-task and task-to-memory communications.

The following paragraphs present the semantics of the memory segments, communication channels, dependencies, and finally present a detailed description of how the computation within a task is specified in the USM.

Memory segments

A memory segment is an element of storage whose size and word length are defined by the user. A memory segment can be declared as being local to a task (M3 in Figure 3.2), or shared between multiple tasks (M4 in Figure 3.2). Both environment and design tasks have

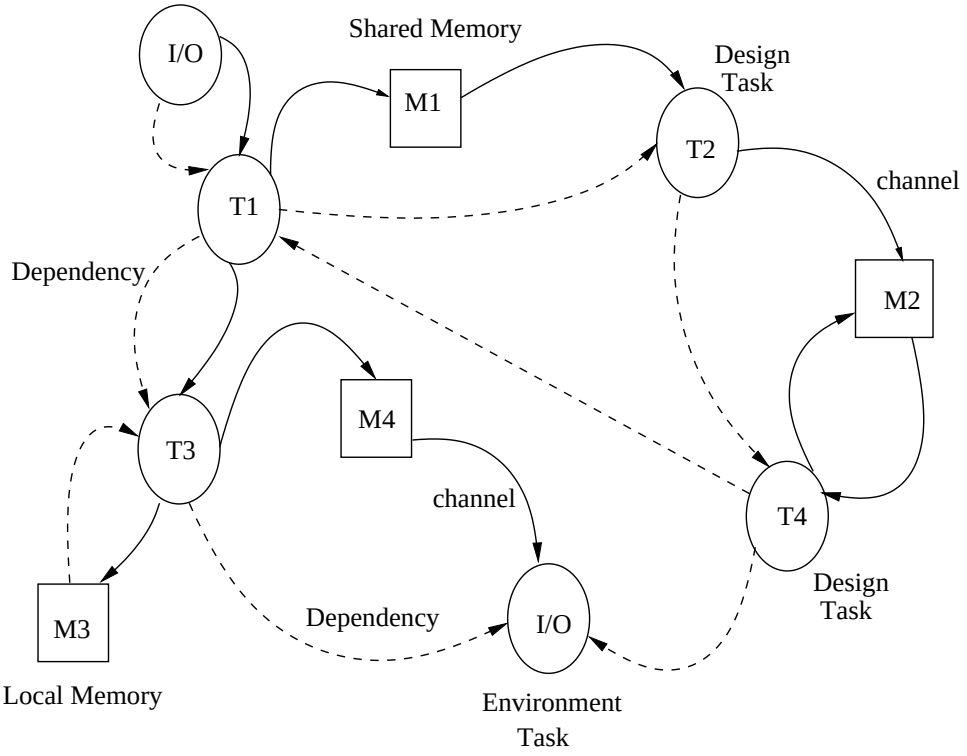


Figure 3.2: USM Task Graph Example

access to memory segments.

When a memory segment is used by more than one task as a means of transferring data from one task to another (i.e. one task is writing to and another is reading from the memory segment), the designer is responsible for synchronizing the tasks using dependencies. However, if multiple tasks are accessing different areas of the same segment, or if the tasks are only reading from the segment, no synchronization is required. It is assumed that the synthesis process would introduce memory arbitration between the tasks whenever needed (refer to Chapter 9). This way, the designer need not worry about resolving conflicts between memory access operations. This keeps the design architecture-independent: the synthesis process can map the memory segment to a physical memory that has either one port or multiple ports. Also, the synthesis process can map multiple memory segments to the same physical memory. Memory segments are easily implemented in VHDL as local or shared variable arrays.

Communication channels

Channels provide the means of communication between tasks (environment as well as design tasks), and between tasks and memory segments. Depending on the bitwidth required between source and destination, the designer must fix the size of the channel. Between each pair of communicating objects, one or more channels could be used. However, the design should not share the same channel across different pairs of objects. Again, if the synthesis process decides to share channels due to resource constraints, then it will automatically identify those channels and provide arbitration. This simplifies the task of the designer since manual introduction of arbitration mechanism for shared channels is not required.

A communication channel used by the designer is unidirectional. However, since synthesis tools might introduce channel sharing, a physical channel might be bi-directional.

In VHDL, when implementing USM channels, signals with the appropriate bitwidths can be used.

Dependencies

Dependencies are used as means of providing explicit synchronization for data and control flow between tasks. A dependency is a directed control line from a source task to a destination task. The semantics of a dependency edge implies that a destination task waits until its corresponding source tasks initiate its execution. Source and destination tasks can be either environment or design tasks. There can be multiple destination tasks dependent on a source task through a single dependency edge. However, a task may be dependent on several tasks through separate dependency edges. The flow of task execution is captured by the dependencies in the task graph. More accurately, the role of a dependency edge is two-fold: it provides synchronization between one-time executing tasks, and provides synchronization mechanism between tasks involved in loops. The following two paragraphs present the semantics of these dependencies.

Synchronization using dependencies: Dependency edges provide a way of synchronizing task execution. This is equivalent to synchronizing data transfer from one task to the other. A 1-bit control flag is associated with each dependency edge. This synchronization mechanism allows tasks to start execution irrespective of the status of other tasks executing in parallel.

A dependency edge used to order execution of tasks is shown in Figure 3.3 and a dependency edge used to synchronize data transfer is shown in Figure 3.4. In the example shown in

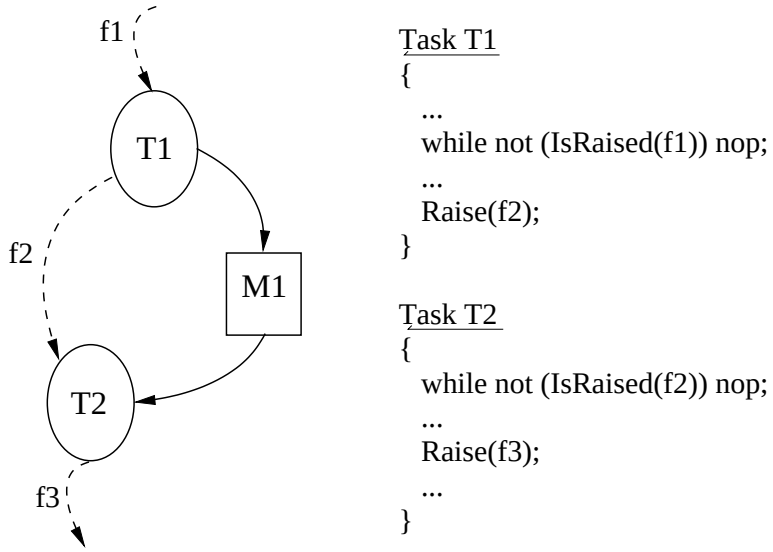


Figure 3.3: Synchronization of Task Execution

Figure 3.3, task T2 waits for the completion of task T1 before it begins execution. Whereas in Figure 3.4, task T2 waits for task T1 to write a value into the channel c12. In these figures, notice that the constructs Raise()/IsRaised() are used to set/check the value of the control flag. In order to synchronize two tasks, the source task invokes the Raised() construct on a flag, whereas the destination task waits in a loop invoking the IsRaised() construct on the same flag.

This synchronization mechanism ensures that the destination task waits until the source task triggers its execution or that the data is ready for consumption. Note that this type of dependency edge also allows the conditional execution of tasks. For example in Figure 3.5, the source task T1 raises either flag f12 or f13 based on a condition. Therefore, tasks T2 and T3 are conditionally dependent on T1.

Conditional dependencies imply that a destination task need not finish execution but may wait indefinitely. Hence, the execution semantics of the USM is defined as follows: The execution cycle for a collection of tasks is defined to finish when all the tasks are indefinitely waiting. The model assumes that there is an indefinite wait at the end of each task.

In VHDL process synchronization, all tasks have to arrive at a wait state before waiting tasks can be triggered. However, in USM, the triggering mechanism is not based on the wait command, instead, each pair of tasks has its own busy-wait synchronization procedure. This allows multiple processes to run concurrently without the need for global synchronization. On the other hand, this busy-wait procedure can be easily implemented in VHDL. For simulation purposes, a busy-wait can be replaced by a simple VHDL wait without loss of

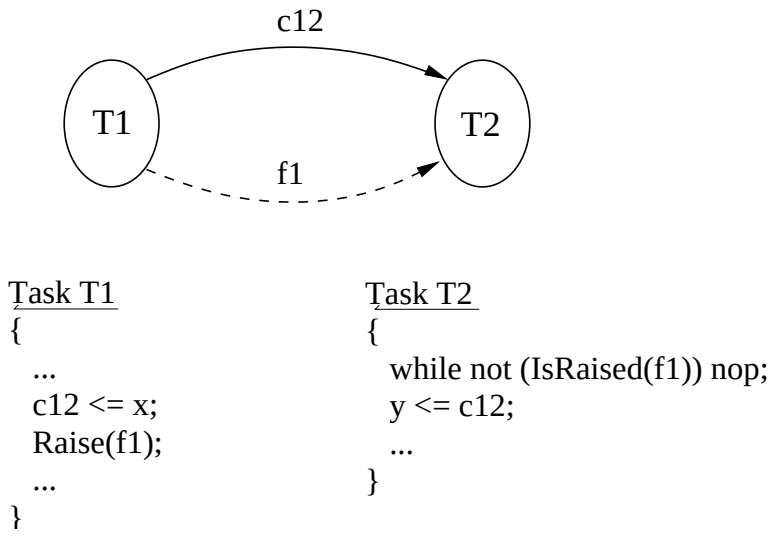


Figure 3.4: Synchronization of Data Transfer

functionality.

Loops using dependencies: Dependency edges are also used to implement loops in a task graph. Since tasks involved in loops might execute more than once, a mechanism is needed to ensure proper inter-task synchronization.

The (Raised(), IsRaised()) mechanism explained in Section 2.3.1 is not adequate to represent a loop dependency. This is because there is no control on the number of times a destination task might execute: Once the flag it is waiting on is raised, a destination task might start executing its first iteration and then incorrectly proceed to a subsequent iteration since the flag might still be raised. A solution to this problem is to provide a dependency based on a flag that is toggled each time the source needs to trigger.

Hence, to synchronize two tasks involved in a loop, a toggling dependency edge is used. Each destination task should not only wait on the value of the flag but also on the event on the flag. Thus, it is imperative for the destination task to keep track of the old value of the control flag. This is advantageous since only a single bit value has to be passed between the source and the destination tasks.

As a result, instead of using two dependency edges (with the Raise/IsRaised mechanism) to ensure proper execution of loops, a toggling dependency edge solves the problem by introducing only one flag (instead of two) and a local storage bit in the destination task.

Figure 3.6 shows an example of a loop involving three tasks: T1, T2, and T3. The control

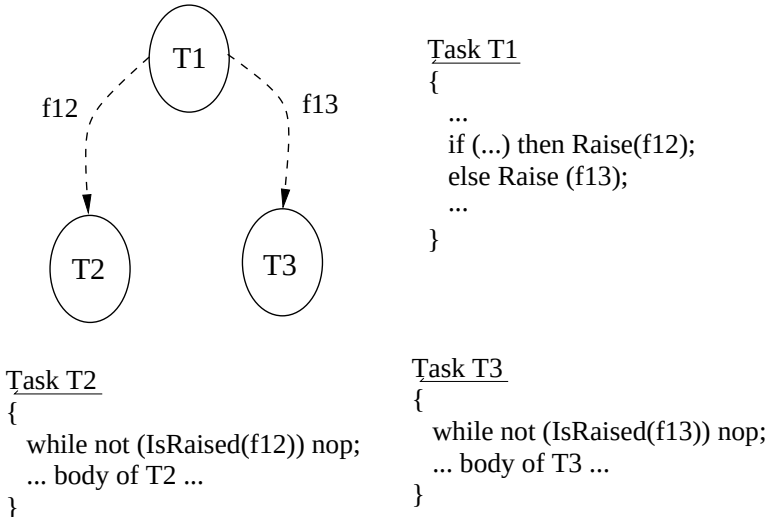


Figure 3.5: Conditional Task Execution

flows into T1 through the dependency f1. T1 triggers T2 for a number of times specified by the for loop inside T1, after which it raises flag f4 and stops. Task T2 is in turn dependent on T1 and will only be triggered by flag f12 originating in T1. Similarly, T3 is dependent on T2 through flag f23. Finally, T3 triggers the next iteration of T1 through flag f31.

Clearly, for a dependency edge involved in a loop, one 1-bit control flag is still needed but an additional 1-bit storage in each destination task is required. Note that, initially, the value of the control flag and all corresponding 1-bit storage flags should be the same (either 0 or 1).

Tasks

A Task (shown in Figure 3.7) consists of a set of inputs and outputs which can be flags, shared memories and channels, some of which representing design I/O, and a set of local storage elements that are variables/constants, local memories, or flags. Flags, as mentioned before, are special storage elements used to represent inter-task dependencies.

Computations within a task are represented as a Control Data Flow Graph (CDFG). The CDFG is a directed acyclic graph where the nodes represent operations and the edges represent data/control flow. A node in the CDFG can be represented by a 4-tuple given as:

$$< name, input\_set, output\_set, data\_dependency\_set >$$

The name denotes the id of the operation, the input_set is the set of all storage elements input to the node, the output_set is the set of all storage elements output from the node,

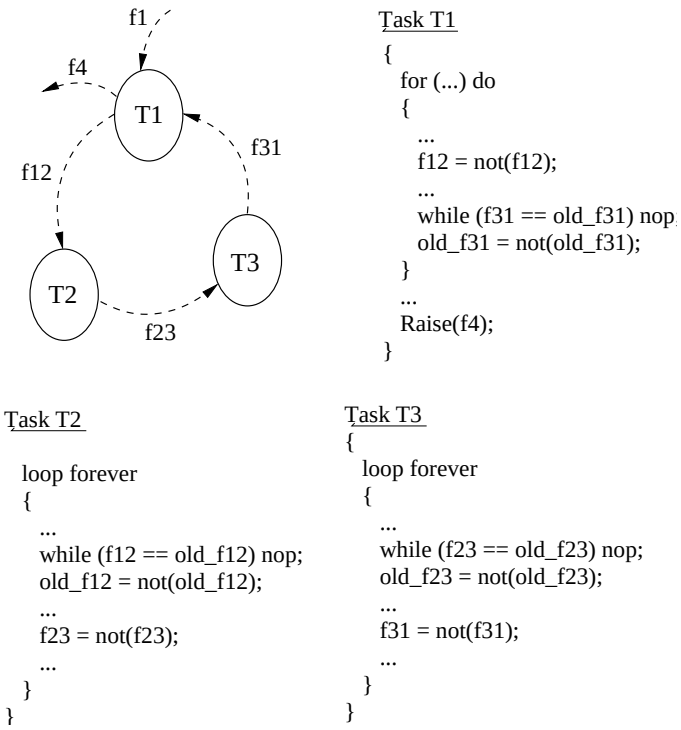


Figure 3.6: Loop Dependencies

and the data_dependency_set consists of the set of all source nodes that a node depends on.

The entire CDFG can be built in a two-step process. A flow graph from the source language can be first generated and, after a detailed dependency analysis, a dependency graph can be generated. In the case of a flow graph, the input and output sets of the nodes are formed and the dependency sets of the nodes are empty.

An example flow graph and a corresponding input specification are shown in Figure 3.8. The edges in the graph are annotated by the storage names that they represent.

The nodes in a CDFG can be broadly classified into these two types:

- *Control Nodes:* These nodes are used to represent the flow of control and the CDFG is organized as a collection of control nodes connected by control edges. Conditional constructs such as if-then-else and case are represented using the SELECT - END_SELECT node pairs. The WUR (wait until raised) and WUL (wait until lowered) nodes are used to represent inter-task dependencies. The LEAVE node is used to represent the exit point from loops. Further an operation subgraph can be encompassed within a BEGIN_CONTROL_BLOCK - END_CONTROL_BLOCK node pair. The semantics of these will become clear in the following section.

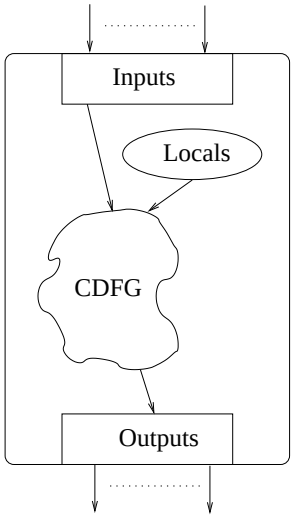


Figure 3.7: Task Model

- *Simple Nodes:* These represent relational, arithmetic, and logical operations. They are binary operations and an expression can be decomposed into a tree of nodes as shown in Figure 3.8. Also, a ST node is used to represent the assignment of a value to a storage element.

The flow graph shown in Figure 3.8 captures only the flow of data, but fails to capture all the concurrencies among the operations. This can be done by performing a detailed dependency analysis.

Dependency analysis The edges in the CDFG represent dependencies among the nodes. Dependencies can broadly be classified as: data and control dependencies. Data dependencies capture the flow of data and hence the order of execution assignment statements. Control dependencies capture the semantics of sequencing, conditional, and loop constructs. Both these dependencies should be properly represented in a correct CDFG representation.

A data_dependency_set is a set of 2-tuples, (n,d) where n is an ancestor node and d is one of the three types of data dependencies (described below) between the two nodes. The following paragraphs discuss data and control dependencies in some detail. [65] provides a more detailed discussion on dependencies in programming languages.

Data Dependencies: Given any two operations represented by nodes N1 and N2, where N1 is a predecessor or ancestor of N2 in the flow graph:

$$N_2 \; is \; flow \; dependent \; on \; N_1 \; if \;\; output\_set(N_1) \cap input\_set(N_2) \neq \emptyset$$

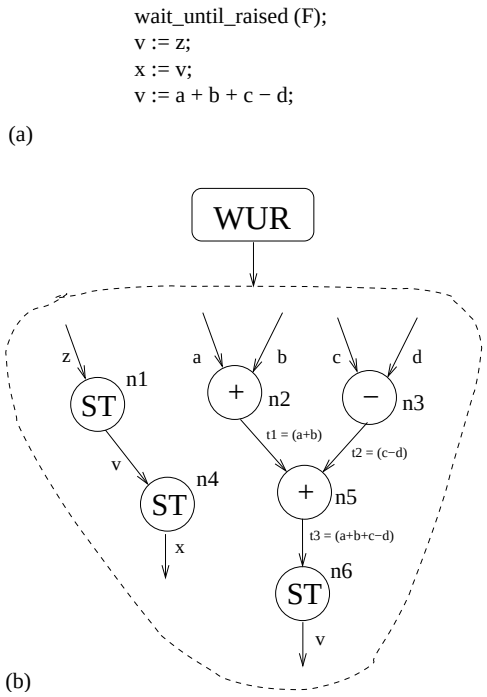


Figure 3.8: Example Flow Graph

$$N_2 \textit{ is anti dependent on } N_1 \textit{ if } \textit{ input\_set}(N_1) \cap \textit{output\_set}(N_2) \neq \emptyset$$

$$N_2 \textit{ is output dependent on } N_1 \textit{ if } \textit{ output\_set}(N_1) \cap \textit{output\_set}(N_2) \neq \emptyset$$

A data dependency is said to exist between two nodes in the operation graph if any of the above three types of dependencies holds between them. These dependencies can be extracted from VHDL variable assignment statements. Consider a VHDL specification where A and B are any two sequential variable assignment statements with A occurring before B. B is said to be flow dependent on A if a variable that is written to in A is read in B. On the other hand, if the variable is first read in A and later written to in B, then B is said to be anti-dependent on A. If the same variable is written to in both A and B, B is said to be output dependent on A.

Control Dependencies: Gajski, Dutt, and Wu in [57] proposed a way to handle control constructs, by mapping a control-flow representation to an equivalent data-flow representation. This has the advantage of making the concurrencies in the design explicit but generates a complicated graph representation that could get too large and has little correspondence to the original specification. This section describes a representation that efficiently incorporates control constructs and alleviates some of the problems that arise in a pure data flow style representation.

Control dependencies are handled with the aid of the *control block* demarcated by BEGIN_CONTROL_BLK - END_CONTROL_BLK node pairs. A *control block* is a suitably chosen subgraph of an operation flow graph with no other control nodes in it. Thus, edges within a *control block* always denote data flow. A *control block* is said to have executed if the control flow reaches its END_CONTROL_BLK node. Control flow enters a *control block* only after all previous control blocks have executed.

Thus, the CDFG of any specification is viewed as a series of control blocks. A control block automatically enforces a control dependency, and thus reduces the overhead of maintaining input, output, and dependency sets of the nodes in the operation graph. This simplifies, to a large extent, the complexities involved in generating dependencies although there is some loss in exploiting to the fullest the concurrencies present in the design[2].

The dependency graph for the VHDL specification of Figure 3.8 after dependency analysis is shown in Figure 3.9. Differences between this representation and the earlier flow graph representation can be observed. The subgraphs enclosed by dashed lines show the control blocks. The WUR node specifies a control boundary. Therefore the control block encapsulating the first subgraph ends before the WUR node and the second subgraph falls within a new control block. The control block ensures that all preceding nodes are executed before the WUR node is executed and all succeeding nodes are executed only after the WUR node is executed. The operation graph correctly detects dependencies between nodes otherwise missed by the flow graph representation.

From the flow graph in Figure 3.8, it appears that nodes *n4* and *n6* can be executed in parallel but the anti-dependence between them is clearly captured in the dependency graph in Figure 3.9. Further, output dependency between nodes *n1* and *n6* now enforces an order in their execution, even in the absence of node *n4*. On the other hand, no order is specified in the execution of the nodes in the set {*n1, n2, n3*} and in the set {*n4, n5*}. As a result, the nodes in each of these sets can be executed in parallel. Thus, dependency analysis serves to faithfully capture the semantics and the concurrencies in the specification.

Translation of Conditionals, Loops, and Wait Constructs The following paragraphs explain in some detail the translation of conditionals, loops, and wait constructs into a CDFG.

Conditionals: The most popularly used conditional constructs are the 'if–then–else' statement and the 'case' statement. These two are easily translated with the help of control

[2]This loss can be recovered to a large extent by the use of suitable behavioral transformations that facilitate code motion across control blocks [68]

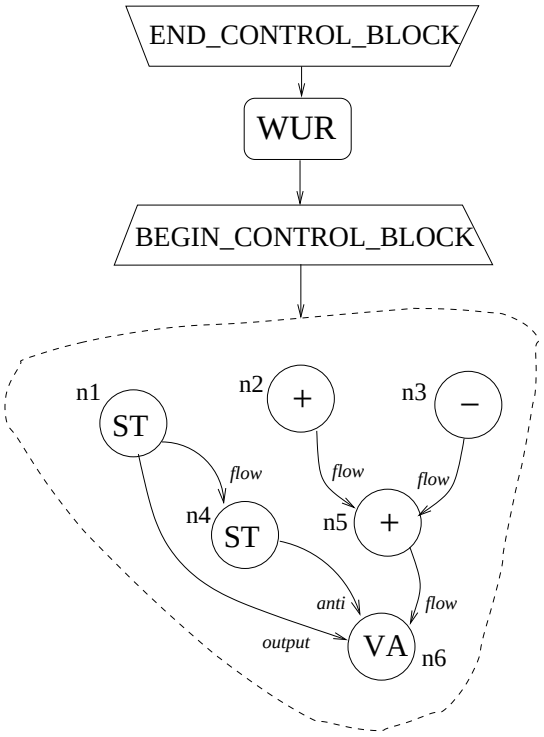


Figure 3.9: Dependency Graph

blocks, the semantics of which was explained earlier in Section 2.3.

Case Statement: A typical case statement is shown in Figure 3.10(a). The translated operation graph of the case statement is shown in Figure 3.10(c). The case statement is translated into an expression tree followed by a SELECT node whose input set has the final data flow edge in the expression tree. Therefore, the select node is flow dependent on the final node of the case expression tree. The select node has several branches with a set of select values for each branch. The control flows only into that branch whose set of values match with the result of the case expression. The select node in an operation graph represents this comparison operation that determines the choice of the branch for control flow.

The statements in a case branch are translated into a subgraph, which is enclosed in a control block, as shown in Figure 3.10.

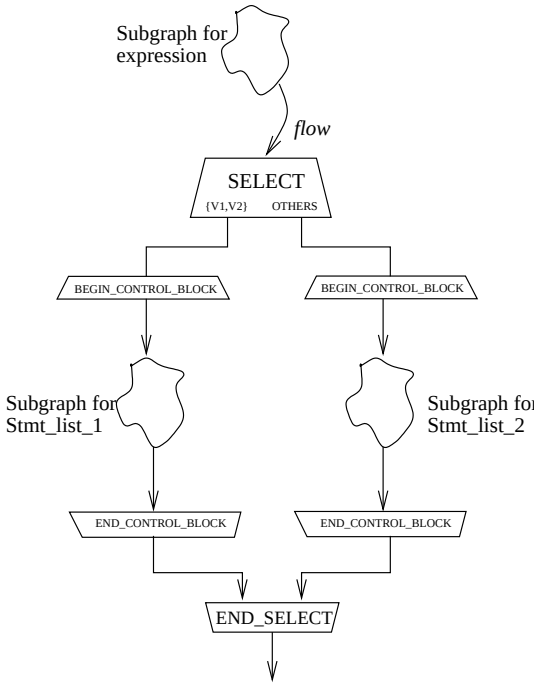
Therefore, the select node corresponding to a case statement would have as many branches as are in the case statement, each branch having a control block. This ensures that the case expression and the branching operation (represented by the select node) are executed before control can flow into any one of the branches. This enforced control dependency also helps in restricting data dependency analysis within each branch.

```
case <expression>
      when v1 | v2 =>
            <stmt_list_1>
      when others =>
            <stmt_list_2>
end case ;
```

(a)

```
if <conditional_expression>
then
     <branch-1>
else
     <branch-2>
end if;
```

(b)



(c)

Figure 3.10: Conditional Constructs

If–then–else Statement: A typical 'if–then–else' statement is shown in Figure 3.10(b). An 'if–then–else' statement can be treated as a case statement with two branches and the case expression replaced by the conditional-expression. The select values for the first and the second branches are the Boolean constants TRUE and FALSE, respectively. Any other kind of branching constructs like the 'if–then–elsif–elsif–else–end if' construct can be transformed into an 'if–then–else' statement, the else part of which having another 'if–then–else' statement.

Loops: Loops are usually of three types. The infinite loop with no loop condition, the 'for' loop, and the 'while' loop. All three loop variations can be represented by the generic form shown in Figure 3.11.

The following paragraphs describe the translation of the generic loop statement whose operation graph is shown in Figure 3.11. The loop condition is translated into an expression tree followed by a select node whose input is the resulting edge of the tree. The select node has two branches: (1) The true select branch has the subgraph for the loop_body enclosed in a control block. (2) The false select branch has a LEAVE node within a control block. These control blocks ensure that the loop condition evaluation takes place before the control flows into any of these branches. Semantics of the loop-block imply that control flow keeps

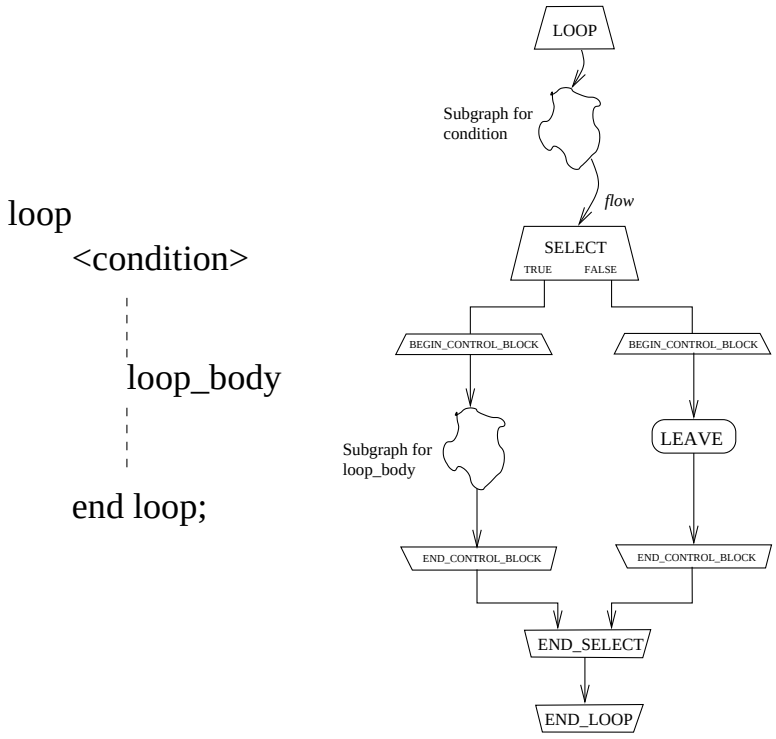


Figure 3.11: Loop Statement

looping inside until the loop condition evaluates to false and the leave node is reached.

Wait: The Wait_Until_Raised (WUR) and Wait_Until_Lowered (WUL) are implemented as follows:

WUR(flag) = while (not(IsRaised(flag))) nop;

WUL(flag) = while (IsRaised(flag)) nop;

Essentially, both the waits are special cases of the loop construct, where the loop condition subgraph has a simple comparison operation for the raised/lowered value of the flag, and the subgraph for the loop body is empty.

3.3.3 Summary of USM Specification

In this section, an overview of the Uniform Specification Model was presented. The USM provides a hierarchical representation to succinctly capture inter-task level control and dataflow, as well as intra-task operation-level dependencies. More importantly in this work, the USM allows easy representation of the data structures in an application and provides a good memory synthesis environment. Data structures are clearly delimited and defined in the

USM, resource arbitration is easily achievable, and the USM allows memory mapping across different levels of design abstraction. The BBIF along with the USM specification styles provide an adequate environment to clearly characterize data structures of a design and allow efficient mapping and synthesis.

Chapter 4

Memory Synthesis Using ILP for Single-Configuration Memories

4.1 Introduction

This chapter presents an automatic memory mapping methodology which takes into account: the number of words and word size of design data segments and physical memory banks, number of ports on the banks, access latency of the banks, proximity of the banks to the processing unit, and the life cycle analysis of data segments. The solution presented in this chapter assumes a single processing element or FPGA on the RC system. In chapter 7, the memory mapping methodology is extended to handle multiple processing elements. Along with the mapping of data structures in the design, the logic tasks will also be mapped onto the processing elements of the RC board.

The rest of this chapter is organized as follows: Section 4.2 depicts the Integer Linear Programming approach for memory mapping. Section 4.3 shows some results obtained and a discussion of the approach. Section 4.4 concludes the chapter.

4.2 ILP Formulation

Starting with the definitions and assumptions stated in Section 2.4, the following describes the ILP formulation of the flat approach, or complete approach, where the mapping occurs in a single step. Only single configuration banks are handled in these formulations.

4.2.1 Constraints Formulation

The following are some of the important constraints in the memory mapping problem.

- **Uniqueness constraints:** Each data structure should be mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS}, \sum_{t \in PB} Z_{dt} = 1$$

- **Port constraints:** Each data structure should be mapped to at most one port of an instance:

$$\forall_{d \in DS}, \ \forall_{t \in PB} \ \forall_{1 \leq i \leq I_t} \ \sum_{p=1}^{P_t} X_{dtip} \leq 1$$

- **Capacity constraints:** Each data structure should be fully stored on the hardware:

$$\forall_{d \in DS}, \ \forall_{t \in PB},$$

$$\sum_{i=1}^{I_t} \sum_{p=1}^{P_t} X_{dtip} = \left\lceil \frac{D_d}{D_t} \right\rceil * \left\lceil \frac{W_d}{W_t} \right\rceil * Z_{dt}$$

- **Lifecycle conflict constraints:** When there is a conflict between two data structures, they cannot be mapped onto the same port of a physical bank:

$$\forall_{t \in PB}, \ \forall_{1 \leq i \leq I_t}, \ \forall_{1 \leq p \leq P_t},$$

$$\forall_{1 \leq q \leq Q}, \ \forall_{1 \leq r \leq Q} \quad X_{qtip} + X_{rtip} \leq 1$$

  Note that if all data structures conflict with each other, the formulation can be simplified by condensing the above into a single constraint:

$$\forall_{t \in PB}, \ \forall_{1 \leq i \leq I_t}, \ \forall_{1 \leq p \leq P_t} :$$

$$\sum_{d \in DS} X_{dtip} \leq 1$$

  Another formulation for lifecycle conflict constraints is presented in Section 7.4.2.

- **Physical storage constraints:** In multi-ported banks, the sum of the storage space mapped to every port must fit within the bank:

$$\forall_{t \in PB}, \ \forall_{1 \leq i \leq I_t} :$$

$$\sum_{p=1}^{P_t} max(\sum_{d \in DS} X_{dtip} * \lceil D_d \rceil) \leq D_t$$

In the above equation, the $max()$ operator can be dropped if the design assumes that all data structures conflict with each others (i.e. no space overlapping). This makes the formulation much simpler since it avoids any non-linearity in the model.

Note that when the number of physical banks is very large, the model might not need to consider all available banks. An upper bound on this number, for each type of physical banks, can be used instead. Also, several other simplifications could be made in order to speed the execution while still producing good results.

4.2.2 Objective Formulation

The objective of the ILP model is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost function takes the form:

$$minimize[Cost_1 * \alpha_1 + Cost_2 * \alpha_2 + ... + Cost_n * \alpha_n]$$

where α_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components.

The objective formulation in an ILP model is compact and allows easy expansions. Three cost components are depicted below. They try to cover the speed performance and the pin limitation constraints while assuming that the depth of a data segment is proportional to the number of times the segment is accessed:

- **Latency cost:** Assuming that RA_d and WA_d provide a good estimation on the number of reads and writes from and to data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [RA_d * RL_t + WA_d * WL_t]$$

- **Pin delay cost:** Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (RA_d + WA_d) * T_t$$

- **Pin I/O cost:** The larger the width of a data structure the more pins it will need in

the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [W_d + log_2(D_d)] * T_t$$

4.3 Discussion

The ILP formulations presented above were implemented in the CPLEX environment, a commercial linear programming solver [69]. Both the number and types of logical segments as well as physical banks were varied. Given the constraints and objective functions, the quality of the mapping produced was optimal.

Figure 4.1 shows the execution times for several mapping problems where both the number of physical memory banks as well as the number of data structures were varied. The execution times were obtained on a SUN Ultra-30 (248MHz with 128MB RAM). For logical memories, the number of data structures represents the main complexity parameter in the ILP formulation. Similarly, for physical memories, the two complexity parameters are: the total number of physical banks and the total number of ports summed over all instances of all bank types. Note that the missing points on the plot refer to cases where the solution was unfeasible; the data structures could not fit on the RC. Also, The lines connecting the data points are used for the sake of clarity; they do not suggest the predictability of the mapping solutions between test cases that were used.The biggest example in this plot contains 100 data structures and 110 physical banks. Note that beyond a certain limit, the number of physical banks does not affect the ILP solution. In fact, an upper bound for each physical bank type can be initially computed based on the data structures available and used in the ILP formulation. This will reduce the solution time especially when the hardware resources far outnumber the data structures.

It is important to note that the memory structure of the RC directly affects the solution time of the solver. All tested designs included RC platforms that were based on currently available architectures. And because memory banks that are on-chip tend to be very fast while not requiring off-chip interconnection, the ILP objective formulation can be easily met since there are few contradicting objectives. In other words, when a hardware architecture was given to the tool that has on-chip memory banks with slower latency characteristics than other off-chip banks, the solution time of the ILP solver increased dramatically.

Note the following:

- The number of data structures can be larger than the total number of available physical

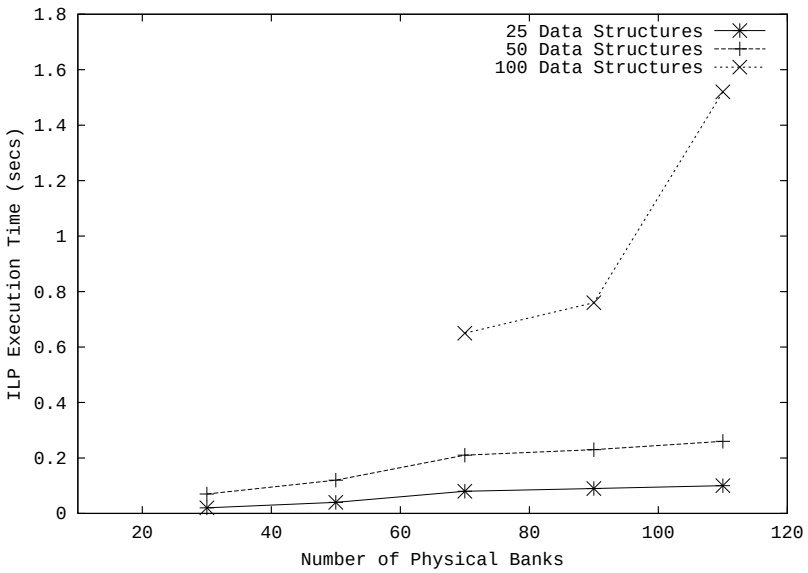


Figure 4.1: ILP Execution Times for Single Configuration Banks

banks. There are several ways of coping with this problem. First, if a physical bank has more than one port, then the mapper could assign more than one logical segment to this bank (one logical segment per port). Second, if there are no life cycle conflicts between two or more logical segments, then they could be mapped to the same port of a bank. Third, two or more conflicting data segments could be mapped to the same port of a physical bank provided an arbitration mechanism is introduced (See Chapter 9). The mapper decides to arbitrate segments based on the added area and delay estimates due to arbitration.

- A footprint analysis or software profiling information depicting the number of memory accesses is not available to the mapper. As a rough guide, the number of words in each data structure was assumed to be proportional to the frequency of accesses. A software profiling approach could lead to a lower cost implementation since the mapper would have knowledge of the number of memory accesses for every data structure.

- In all objective functions presented in the thesis, the weight coefficients α_i were set by first finding a solution with initial coefficients set to one, then analyzing the values of the cost components obtained and setting the weight coefficients appropriately.

- The current work did not take into account constraints posed by the data structures. In other words, there was no way of specifying a performance constraint on specific segments. At the cost of added complexity, the ILP formulation could handle such constraints.

- The execution time of the ILP solver is as accurate as the time resolution reported by the operating systems in the host computer. When the ILP solver time is close to the time resolution of the operating system, the execution time reported is not very accurate.

- In all results obtained in the thesis, data structures were constructed either randomly or based on typical signal processing applications such as Fast Fourier Transformation (FFT) and Discrete Cosine Transform (DCT). In other words, the depths and widths of data structures were set either randomly or according to actual data structures from different DSP applications. When the width and depth were generated randomly, sizes varied between 10x1 and 10,000x256. Similarly, the physical memories used in the results were constructed either randomly or based on available RC platforms. For random generation, sizes of each instance of a memory bank type varied between 4096x1 and several megabytes.

- Finally, validation of the techniques presented in this chapter and in the entire thesis was performed on existing RC platforms. However, since the available hardware offered a simple memory hierarchy and/or a limited set of memory banks, only very small designs were validated on the hardware.

4.4 Conclusion

This chapter presented an ILP formulation to memory mapping for single configuration memory banks. The solutions obtained with this technique are optimal with respect to the objective function issued to the ILP solver.

The technique presented in this chapter is limited by RC platforms that offer single configuration memory banks. Since multi-configuration memory banks constitute the majority of RC memory banks and are typically on-chip, RC platforms with single configuration banks usually have a very limited number of banks. For this reason, the memory mapping problem size that can be solved by this technique is limited.

As an extension to this work, the following chapter incorporates multi-configuration memories in the mapping formulation. This will enable data assignment for large hardware architectures as well as small and medium sized ones. Hence, the following chapter utilizes the Y_{tipc} ILP variable to assign an exact configuration to each multi-configuration memory bank.

Chapter 5

Memory Synthesis Using ILP for Multi-Configuration Memories

5.1 Introduction

The limitation of the technique presented in Chapter 4 is with respect to multi-configuration memories. The technique only targeted single configuration memory banks and was restricted to RC platforms that did not contain FPGAs with on-chip memories. Since typical on-chip memory banks of FPGAs offer multiple configurations, this chapter presents an automatic memory mapping methodology [70] which takes into account: the number of words and word size of design data segments and physical memory banks, number of ports on the banks, access latency of the banks, proximity of the banks to the processing unit, and life cycle analysis of data segments. It also incorporates configuration selection from the multiple configurations available in BlockRAMs of Virtex series FPGAs or other FPGAs' on-chip memory banks.

The rest of this chapter is organized as follows: Section 5.2 depicts the Integer Linear Programming approach for memory mapping. Section 5.3 shows some results obtained and a discussion of the approach. Section 5.4 provides clues on how to expand the formulations. Finally, Section 5.5 concludes the chapter.

5.2 ILP Formulation

Starting with the definitions and assumptions stated in Section 2.4, the following describes the ILP formulation of the flat approach, or complete approach, where the mapping occurs in a single step.

5.2.1 Constraints Formulation

The following are some of the important constraints in the memory mapping problem.

- **Uniqueness constraints:** Each data structure should be mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS}, \sum_{t \in PB} Z_{dt} = 1$$

- **Port constraints:** Each data structure should be mapped to at most one port of an instance:

$$\forall_{d \in DS}, \forall_{t \in PB} \forall_{1 \leq i \leq I_t} : \sum_{p=1}^{P_t} X_{dtip} \leq 1$$

- **Capacity constraints:** Each data structure should be fully stored on the hardware. In the case of physical banks with *fixed configuration*:

$$\forall_{d \in DS}, \forall_{t \in PB},$$

$$\sum_{i=1}^{I_t} \sum_{p=1}^{P_t} X_{dtip} = \left\lceil \frac{D_d}{D_t} \right\rceil * \left\lceil \frac{W_d}{W_t} \right\rceil * Z_{dt}$$

  For physical banks with multiple configurations, the formulation is more complex. First, each data structure must fit in the storage area assigned to it:

$$\forall_{d \in DS},$$

$$\lceil D_d \rceil * \lceil W_d \rceil \leq \sum_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} X_{dtip} * D_t[1] * W_t[1]$$

  This equation is based on the assumption of Equation 2.1. In other words,

$$\forall_i, \ , \forall_j, \ 1 \leq i, j \leq C_t : \ D_t[i] * W_t[i] = D_t[j] * W_t[j]$$

  Second, the depth of each segment must be less than the combined depth of the assigned

banks:

$$\forall_{d \in DS},\ D_d \leq \sum_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} \sum_{c=1}^{C_t} D_t[c] * Y_{tipc} * Z_{dt}$$

Third, the width of each segment must be less than the combined width of the assigned banks:

$$\forall_{d \in DS},\ W_d \leq \sum_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} \sum_{c=1}^{C_t} W_t[c] * Y_{tipc} * Z_{dt}$$

Since the last two formulations are non-linear; i.e. they include the non-linear terms $Y_{tipc} * Z_{dt}$, they are linearized by using the following linearization method [71]. For each product term $X * Y$, a new real-valued variable Z with an upper-bound of 1 is introduced such that:

$$Z = A * B$$

Subject to:

$$A + B - Z \leq 1$$

and to:

$$\begin{cases} A \geq Z \\ B \geq Z \end{cases}$$

- **Lifecycle conflict constraints:** When there is a conflict between two data structures, they cannot be mapped onto the same port of a physical bank:

$$\forall_{t \in PB},\ \forall_{1 \leq i \leq I_t},\ \forall_{1 \leq p \leq P_t},$$

$$\forall_{1 \leq q \leq Q},\ \forall_{1 \leq r \leq Q} \quad X_{qtip} + X_{rtip} \leq 1$$

Note that if all data structures conflict with each other, the formulation can be simplified by condensing the above into a single constraint:

$$\forall_{t \in PB},\ \forall_{1 \leq i \leq I_t},\ \forall_{p, 1 \leq p \leq P_t}:$$

$$\sum_{d \in DS} X_{dtip} \leq 1$$

- **Configuration uniqueness constraints:** In multi-configuration banks, at most one configuration can be selected at a time:

$$\forall_{t \in PB},\ \forall_{1 \leq i \leq I_t},\ \forall_{1 \leq p \leq P_t}:$$

$$\sum_{c=1}^{C_t} Y_{tipc} \leq 1$$

- **Physical storage constraints:** In multi-ported banks, the sum of the storage space mapped to every port must fit within the bank. For single configuration banks, the constraint can be formulated as:

$$\forall_{t \in PB},\ \forall_{1 \leq i \leq I_t} :$$

$$\sum_{p=1}^{P_t} max(\sum_{d \in DS} X_{dtip} * \lceil D_d \rceil) \leq D_t$$

And for multi-configuration banks, the constraint becomes:

$$\forall_{t \in PB},\ \forall_{1 \leq i \leq I_t} :$$

$$\sum_{p=1}^{P_t} max(\sum_{d \in DS} X_{dtip} * \lceil D_d \rceil * \lceil W_d \rceil) \leq D_t * W_t$$

In the above two equations, the $max()$ operator can be dropped if the design assumes that all data structures conflict with each others (i.e. no space overlapping). This makes the formulation much simpler since it avoids any non-linearity in the model.

Note that when the number of physical banks is very large, the model might not need to consider all available banks. An upper bound on this number, for each type of physical banks, can be used instead. Also, several other simplifications could be made in order to speed the execution while still producing good results.

5.2.2 Objective Formulation

Similar to Chapter 4, the objective of the ILP model is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost function takes the form:

$$minimize[Cost_1 * \alpha_1 + Cost_2 * \alpha_2 + ... + Cost_n * \alpha_n]$$

where α_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components.

The objective formulation in an ILP model is compact and allows easy expansions. Three cost components are depicted below. They try to cover the speed performance and the pin limitation constraints while assuming that the depth of a data segment is proportional to the number of times the segment is accessed:

- **Latency cost:** Assuming that RA_d and WA_d provide a good estimation on the number of reads and writes from and to data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [RA_d * RL_t + WA_d * WL_t]$$

- **Pin delay cost:** Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (RA_d + WA_d) * T_t$$

- **Pin I/O cost:** The larger the width of a data structure the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [W_d + log_2(D_d)] * T_t$$

5.3 Results and Discussion

The ILP model presented in the previous section was executed for designs of different sizes. CPLEX, a commercial linear programming solver [69], was used. Both the number and types of logical segments as well as physical banks were varied. Given the constraints and objective functions, the quality of the mapping produced was optimal. Table 5.1 shows the execution time on a SUN Ultra-30 (248MHz with 128MB RAM) for designs of various sizes: For logical memories, the number of segments represents the main complexity parameter in the ILP formulation. Similarly, for physical memories, the three complexity parameters are: the total number of physical banks, the total number of ports summed over all instances of all bank types, and the total number of possible configuration settings. The total number of possible configurations is the product of number of configurations of all configurable bank *instances*; i.e. the total number of possible ways that all memory bank instances can be programmed. The execution time is given in seconds.

It is clear that for large designs, the ILP formulation becomes impractical. However, based on the current RC technology and current applications, the ILP execution is satisfactory in several cases. It can also be integrated with heuristical approaches to limit the search space. By choosing proper heuristics, the quality of the final mapping could be preserved to a large extent while gaining execution speed. The ILP formulation used to obtain the results in Table 5.1 contains all constraints stated in this chapter; it did, however, assume

Logical Memories	Physical Memories			Execution Time
#segments	Total #banks	Total #ports	Total #configs	(in seconds)
22	13	25	$\sim 10^6$	8.1
32	23	45	$\sim 10^{13}$	29.4
32	45	77	$\sim 10^{20}$	99.3
42	45	77	$\sim 10^{20}$	130.4
32	65	105	$\sim 10^{20}$	172.7
62	65	105	$\sim 10^{20}$	411.0
32	180	265	$\sim 3 * 10^{52}$	518.3
62	180	265	$\sim 3 * 10^{52}$	1225.0
132	180	265	$\sim 3 * 10^{52}$	2989.0

Table 5.1: ILP Execution Times

that all data structures had conflicting life cycles, thus there was no overlapping of memory segments. A few additional constraints were also introduced and the formulation of some constraints was slightly modified to enhance the ILP execution speed.

Finally, a comparison of the ILP technique, used in this chapter, with a heuristic-based approach was performed. A Tabu search meta-heuristic approach was implemented [72] to map a data structures onto several instances of a memory type on the RC. In both techniques, the same cost function was utilized to validate the quality of the mappings. The applications, characterized by the number of logical memories, and the target architectures, characterized by the total number of available ports, were randomly chosen. The selected target architectures contained a mixture of on-chip and off-chip memories.

The ILP approach generates optimal results and is expected to outperform the heuristic approach in terms of lower cost function. Table 5.2 and Figure 5.1 provide a comparison of both techniques for ten different mapping problems. Given that the design problems were relatively large, in most of the examples, the ILP approach did not converge and had to be terminated forcefully. Hence, the value of the ILP cost function is not necessarily optimal. However, it is important to note that the ILP cost values reported in the table were generated instantly after the ILP solver started processing the problems and that these values are, in all cases, superior to the ones obtained by the heuristic approach. Furthermore, the average improvement of the cost value was about 3.5%, and the maximum was about 13% with execution times averaging 21 seconds for the heuristic approach, compared to milli-seconds to obtain the ILP costs.

Design Number	# Data Structures	# Ports	ILP Cost	Heuristic Cost	% Difference
1	8	18	422	422	0.0
2	18	39	770	777	0.9
3	32	63	1292	1319	2.0
4	27	29	1302	1381	5.7
5	27	37	1327	1411	5.9
6	39	95	1584	1622	2.3
7	42	77	1841	1872	1.6
8	49	99	1997	2280	12.4
9	18	52	3139	3143	0.1
10	32	60	4702	4931	4.6

Table 5.2: Flat-ILP Versus Heuristic Comparison

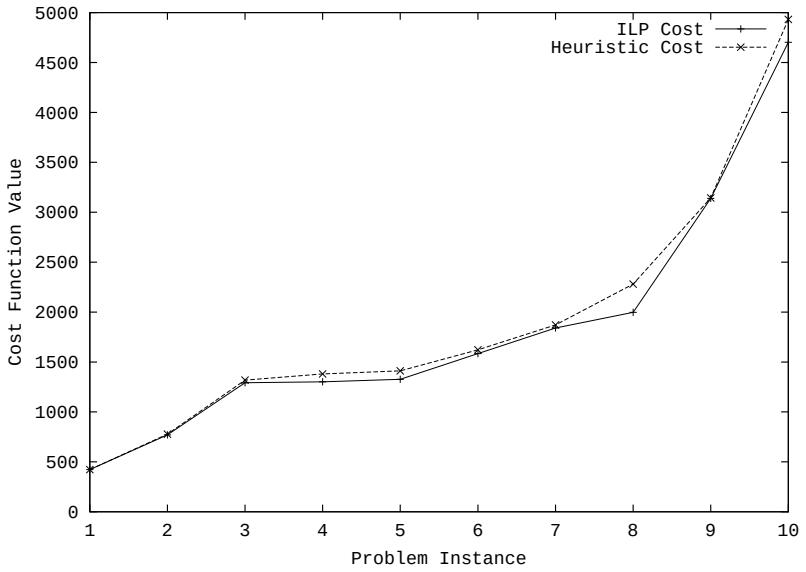


Figure 5.1: ILP and Heuristic Cost Comparison

5.4 Discussion

As part of extensions to this chapter, the following issues can be addressed:

- In the case of a single processing unit, all design logic is mapped onto one hardware area, and all logic areas are assumed equidistant from each physical bank. The model need to be enhanced to support multiple processing units. This extension is provided in Chapter 7.
- The interconnection structure connecting the processing unit to the physical banks is fixed. In practice, some RC boards might offer limited programmability. New constraints should be added to support programmable interconnect. Insight to the interconnection synthesis problem can be found in [73].
- Arbitration is not taken into consideration in this chapter. In other words, if two logical segments conflict, they will be mapped on two different ports. Support for arbitration could be based on the arbitration scheme introduced in Chapter 9. This scheme interacts with the synthesis process to provide performance estimates due to arbitration and introduce arbiter tasks in the design.
- It is seen that heuristic approaches could be used to prune the design space before or while running the ILP model. Such approaches include limiting the number of available physical banks and pre-processing of the input logical segments.
- Finally, an interfacing mechanism between the memory mapper and the synthesis tools will allow the design space exploration phase to take into account different memory assignments.

5.5 Conclusion

It is clear that the ILP formulation followed in this chapter yields an optimal solution to the mapping problem. However, it is only efficient for small to medium size problems where the design space is relatively small. For applications with an abundance of data structures and/or a hardware platform with an abundance of memory banks, the execution time of the technique proposed in this chapter becomes quite large, and sometimes depletes the resources of the host computer before converging to a solution.

In order to obtain optimal or near optimal solutions using ILP, a different methodology needs to be introduced to solve large mapping problems. The next chapter describes the

two-layered approach that was proposed to solve this problem; it is based on the formulations set forth in this chapter but introduces an estimation mechanism that ensures success of both mapping layers.

Chapter 6

Two-Layered ILP Formulation for Multi-Configuration Memories

6.1 Introduction

An intelligent memory assignment minimizes the total latency of the design and the interconnection requirements due to memory accesses. A complete Integer Linear Programming formulation of the problem results in an optimized memory mapping; however, as seen in the previous chapter, the formulation is complex and takes a very long time to produce a solution. In order to efficiently solve the problem, the concept of global/detailed memory mapping is introduced in this chapter. An ILP formulation of the global mapping process is described. This formulation is simpler and faster than the complete formulation, and it leaves the task of detailed mapping to a post-ILP tool that does not affect the optimality of the memory assignment. As a result, larger designs can be handled at a faster rate and more constraints can be introduced to the formulation.

The need of having a memory mapping technique that can handle larger design problems is due to the abundance of physical memory banks as well as the amount and importance of data structures in signal processing algorithms. The on-chip memory banks of Xilinx Virtex devices [12], called *BlockRAMs*, vary from 8 BlockRAMs for the XCV-50 device up to 208 BlockRAMs for the XCV-3200E device. On-chip memory banks of Altera FLEX 10K devices [41], called *Embedded Array Blocks (EABs)*, vary from 9 EABs for the EPF10K70 device up to 20 EABs for the EPF10K250A device. On-chip memory banks of Altera APEX E devices [42], called *Embedded System Blocks (ESBs)*, vary from 12 ESBs for the EP20K30E device up to 216 ESBs for the EP20K1500E device. Chapter 2 provided insight into the memory mapping

Device	RAM Name	RAMs (# banks)	Size (# bits)	Configurations
Xilinx Virtex	BlockRAM	8 → 280	4096	4096x1 2048x2 1024x4 512x8 256x16
Xilinx Virtex II	BlockRAM	4 → 192	18432	16384x1 8192x2 4096x4 2048x9 1024x18 512x36
Altera Flex 10K	Embedded Array Block	9 → 20	2048	2048x1 1024x2 512x4 256x8
Altera Flex 10KE	Embedded Array Block	9 → 20	4096	2048x2 1024x4 512x8 256x16
Altera Apex E	Embedded System Block	12 → 216	2048	2048x1 1024x2 512x4 256x8 128x16

Table 6.1: FPGA On-Chip RAMs

problem, and Table 6.1 further shows the complexity of the memory structures across several types of FPGA devices.

This chapter proposes a novel approach to memory mapping: it divides the data assignment into two sequential steps [74]:

1. *Global Mapping:* this step maps each data structure of the application onto a unique memory type. It estimates the fitness of all data structures assigned to each memory type, but it does not perform the actual mapping of a data structure onto a specific set of ports.

2. *Detailed Mapping:* this step performs the lower-level assignment. Given a set of data structures assigned to a memory type, detailed mapping these data structures to spe-

cific ports of specific instances. It also selects the optimal configuration on multi-configuration memory banks.

6.2 Two-Layer Approach

For the formulation presented in this section, we assume the notations and assumptions stated in Chapter 2.

6.2.1 Global Memory Mapping

Global mapping only considers the task of assigning a data structure to exactly one type of memory bank. It does not deal with the assignment of the data structure to *specific* instances and ports of the type. However, global mapping ensures a successful detailed mapping by taking into account the architecture specification while avoiding non-optimizing factors in the formulation.

While a complete memory mapper makes use of all three X_{dtip}, Z_{dt}, and Y_{tipc} parameters, a global memory mapper requires only the Z_{dt} parameter. Z_{dt} assigns a data structure to a memory type, whereas X_{dtip} and Y_{tipc} assign data structures and configurations to specific bank instances.

The execution time savings obtained by using a global mapper could be lost if the detailed mapper fails. If this occurs, the global and detailed mappers need to execute multiple times until a solution is found. Thus, it is very important to ensure that the global mapper produces an assignment that can be successfully detailed mapped. In an ILP formulation, this translates to having constraints in the global mapper that take into account the number of instances of each type of memory banks, the number of ports of each instance, and the available width/depth configurations of the type.

ILP Pre-processing

For each design being mapped, the global mapper initially pre-processes some information in order to produce an ILP formulation of the problem. This formulation will result in a fast ILP solution that will successfully go through detailed mapping.

Three main parameters need to be computed to allow a simple yet powerful constraint formulation: CP_{dt} or the total number of consumed ports of memory type t if data structure

d is assigned to it. CW_{dt} or the "ceiling" value of the width of data structure d if assigned to bank type t. And finally, CD_{dt} or the "ceiling" value of the depth of data structure d if assigned to bank type t.

First, the total number of consumed ports CP_{dt} depends on the size of data structure d with respect to the size and number of ports of bank type t. There are four components to CP_{dt} and they are illustrated by the following example.

A 55x17 data structure is to be mapped onto one type of memory bank that has 3 ports, and four ratio configurations: 128x1, 64x2, 32x4, 16x8. Since the data structure requires more than one instance, Figure 6.1 shows a mapping where the assigned instances can be visualized as a rectangular area. The width, 17, of the data structure will be divided into: 8, 8, and 1; and the depth will thus have to be 16 to match the 16x8 configuration. Hence, the upper left instances in the rectangle are *fully* utilized with configuration 16x8 selected. The upper right instances, that form a single column, are *partially* utilized with configuration 128x1 selected since there remained one bit from the width of the structure. The lower left instances, that form a single row, are *partially* utilized with configuration 16x8 selected. Finally, the lower right instance, that is a single instance, is *partially* utilized with configuration 128x1 selected. Note also that, since the memory type has 3 ports, *all* ports in the upper left instances are consumed, and *some* ports in all other instances are consumed. The "O" next to a port signifies that the port is used by this data structure. The "X" next to a port denotes that the port is wasted. And, the check mark next to a port indicates that the port is available for other data structures. Finally, the number between parentheses next to available ports indicates the total number of bits that are still unused in the instances.

Thus, following the scheme of Figure 6.1:

$$\forall_{d \in DS},\ \forall_{t \in PB},$$

$$CP_{dt} = FP_{dt} + WP_{dt} + DP_{dt} + WDP_{dt}$$

where

$$FP_{dt} = \left\lfloor \frac{D_d}{D_{t\sigma}} \right\rfloor * \left\lfloor \frac{W_d}{W_{t\sigma}} \right\rfloor * P_t$$

σ refers to the configuration with the smallest width such that $W_{t\sigma}$ is greater than or equal to W_d. If W_d is larger than all configuration widths, then σ is the configuration with the largest width. For multi-configuration banks, the best configuration yields the smallest numbers of instances used while trying to match the width of the bank with the width of the data structure.

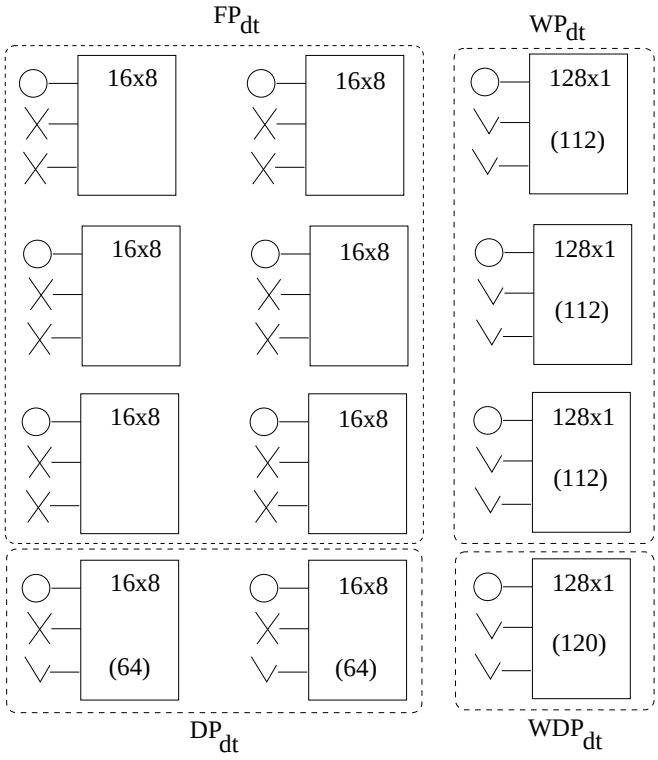


Figure 6.1: Space and Ports Allocation Example

if $((W_d \; mod \; W_{t\sigma}) == 0)$ then $WP_{dt} = 0$ else:

$$WP_{dt} = \left\lfloor \frac{D_d}{D_{t\sigma}} \right\rfloor * consumed\_ports\left(D_{t\sigma}, D_{t\beta}, P_t\right)$$

where β refers to the configuration with the smallest width such that:

$$W_{t\beta} \geq W_d mod W_{t\sigma}$$

consumed_ports() is defined in Figure 6.2. And:

$$DP_{dt} = \left\lfloor \frac{W_d}{W_{t\sigma}} \right\rfloor * consumed\_ports\left(\left(D_d mod D_{t\sigma}\right), D_{t\sigma}, P_t\right)$$

Finally, if $((W_d mod W_{t\sigma}) == 0)$ then $WDP_{dt} = 0$ else:

$$WDP_{dt} = consumed\_ports\left(\left(D_d mod D_{t\sigma}\right), D_{t\beta}, P_t\right)$$

The second parameter, CW_{dt}, indicates the total width that is consumed by data structure

```
function consumed_ports(D_d, D_t, P_t)
begin
    depth = round(D_d, pow(2))
    fraction = depth / D_t
    EP = ⌈fraction * P_t⌉
    returns EP
end
```

Figure 6.2: Fractional Port Consumption

d if assigned to bank type t. After finding configuration σ and β as described above:

$$CW_{dt} = \left\lfloor \frac{W_d}{W_{t\sigma}} \right\rfloor * W_{t\sigma} + W_{t\beta}$$

Similarly, the third parameter, CD_{dt}, indicates the total depth that is consumed by data structure d if assigned to bank type t:

$$CD_{dt} = \left\lfloor \frac{D_d}{D_{t\sigma}} \right\rfloor * D_{t\sigma} + \lceil D_d mod D_{t\sigma} \rceil_{pow(2)}$$

For data structures to co-exist on the same instance of a bank type (on different ports of the bank), *consumed_ports*() is used to compute the fractional number of ports consumed by a data structure. If the entire data structure fits on a single instance, then *consumed_ports*() actually is the total number of ports consumed by the data structure. However, when multiple instances are required, *consumed_ports*() returns the number of ports consumed on each used instance.

Thus, for an n-ported memory, the algorithm in Figure 6.2 computes *consumed_ports*(). The main purpose of this algorithm is to make sure that once data structures are mapped onto physical banks, no adders or other logic would be required to perform memory accesses. To do so, each assigned fraction in an instance is rounded to the closest power-of-two depth that corresponds to the configuration with the largest width. In addition, the port assignment follows the order of decreasing fraction sizes. This ensures that several fractions can be assigned to different ports of an instance without the need of extra logic for base address generation. It also ensures that the memory space for each fraction is mutually exclusive, thus avoiding unwanted conflicts.

Note that *consumed_ports*() in Figure 6.2 is optimal for $P_t = 2$. There is a waste of ports when $P_t > 2$; but since the majority of on-board and on-chip memory banks are either

3-port 16-word bank		
Port 1 (# words)	Port 2 (# words)	Port 3 (# words)
16	0	0
8	8	0
8	4	4,2,1,0
8	2	2,1,0
8	1	1,0
8	0	0
4	4	4,2,1,0
4	2	2,1,0
4	1	1,0
4	0	0
2	2	2,1,0
2	1	1,0
2	0	0
1	1	0,1
1	0	0
0	0	0

Table 6.2: Example on Allocation Options

single or dual ported, this problem is not very pronounced currently. Improvement to this algorithm is part of future work.

Hence, a multi-ported memory bank can only be divided in a fixed number of ways. For instance, the general space allocation for a 3-port memory with 16-word depth is shown in Table 6.2. The algorithm in Figure 6.2 rejects the (8, 8, 0) configuration since it estimates that 8 words require two ports each thus requiring a total of 4 ports. This over-estimation does not occur when a bank type has only two ports.

Constraints Formulation

Next, the ILP formulation is constructed based on the following constraints:

- **Uniqueness constraints:** Each data structure should be mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS}, \sum_{t \in PB} Z_{dt} = 1$$

- **Port constraints:** Each memory type should have enough ports for all the data structures assigned to it:

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * CP_{dt} \leq P_t * I_t$$

From the pre-processing step, the number of ports that each data structure would consume from each type of memory is computed. The sum of all consumed ports of a type should be less than or equal to the total number of available ports of the type.

- **Capacity constraints:** Each bank type must have enough space to contain all data structures assigned to it.

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * (CW_{dt} * CD_{dt}) \leq I_t * W_{t[1]} * D_{t[1]}$$

In the event where the life-cycles of different data structures do not conflict, the capacity constraint is slightly modified to allow overlapping in the memory space.

Note: The uniqueness constraint ignores the specifications of the memory banks; whereas the port and capacity constraints cater to the specific features of each type. It is these latter constraints in addition to some parameters computed in the pre-processing step, that ensure successful detailed mapping.

Objective Formulation

The objective of the ILP model is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost function takes the form:

$$minimize[Cost_1 * \alpha_1 + Cost_2 * \alpha_2 + ... + Cost_n * \alpha_n]$$

where α_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components.

Three cost components are depicted below.

- **Latency cost:** Assuming that RA_d and WA_d provide a good estimation on the number of reads and writes from and to data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [RA_d * RL_t + WA_d * WL_t]$$

- **Pin delay cost:** Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (RA_d + WA_d) * T_t$$

- **Pin I/O cost:** The larger the depth and width of a data structure the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (\lceil log_2(CD_{dt}) \rceil + CW_{dt}) * T_t$$

6.2.2 Detailed Memory Mapping

Once global mapping is complete, the task of detailed mapping is to consider each bank type at a time and all data structures assigned to it. It might re-shape the data structures in case they are larger than a single instance of the bank type, it might assign configurations for each port of each bank instance of the type, and it will assign specific instances for each data structure segment. Thus, the entire problem is serialized into one main formulation (global mapping) followed by a collection of formulations (detailed mapping on each bank type).

Since global memory mapping ensures a successful detailed assignment, the task of the detailed mapper is simplified. It cannot further optimize the assignment based on the optimization criteria established in global mapping. Instead, it can aim at optimizing the detailed assignment based on different optimization factors.

The ILP-based formulation for the detailed memory mapper was developed based on the formulations of Chapter 5. The constraints formulations are identical to the ones stated in the flat approach, to the exception that they do not include the Z_{dt} variable since the detailed mapper assumes that the data structures are already assigned to a memory type. For every type of memory bank, an ILP problem is formed and solved. The aim is to assign data structures to specific ports of specific instances of the bank, possibly requiring the fragmentation of the data structure to fit on several instances. Since the objective function only included Z_{dt} variables in the formulation, the old formulations become constant values in the detailed mapping. Thus, the detailed mapper can be executed as non-optimizing tool that simply assigns data structure chunks onto specific ports of specific instances of the bank type. On the other hand, new objective functions can be introduced to optimize other costs in the model. For instance, if net congestion affects the routing of the synthesized design, an objective cost can be added that attempts to minimize the congestion by balancing the load of data structures on instances of a bank type.

Data Structures	Physical Banks			Complete Approach	Global Approach
#segments	Total #banks	Total #ports	Total #configs	Execution Time (sec)	Execution Time (sec)
22	13	25	$\sim 10^6$	8.1	7.8
32	23	45	$\sim 10^{13}$	29.4	25.3
32	45	77	$\sim 10^{20}$	99.3	50.7
42	45	77	$\sim 10^{20}$	130.4	59.2
32	65	105	$\sim 10^{20}$	172.7	105.1
62	65	105	$\sim 10^{20}$	411.0	140.4
32	180	265	$\sim 3 * 10^{52}$	518.3	216.4
62	180	265	$\sim 3 * 10^{52}$	1225.0	309.0
132	180	265	$\sim 3 * 10^{52}$	2989.0	489.0

Table 6.3: ILP Execution Times for Complete and Global/Detailed Approaches

6.3 Results

The ILP model presented in the previous section was executed for designs of different sizes. Also, the complete (flat approach) memory mapper [70] was executed on the same designs. CPLEX, a commercial linear programming solver [69], was used. Both the number and types of logical segments as well as physical banks were varied. Table 6.3 shows the execution time for both the complete formulation approach and the global/detailed formulation approach. The platform was a SUN Ultra-30 (248MHz with 128MB RAM) for designs of various sizes: For logical memories, the number of segments represents the main complexity parameter in the ILP formulation. Similarly, for physical memories, the three complexity parameters are: the total number of physical banks, the total number of ports summed over all instances of all bank types, and the total number of possible configuration settings. The total number of possible configurations is the product of number of configurations of all configurable bank *instances*; i.e. the total number of possible ways that all memory bank instances can be programmed. The execution time is given in seconds.

It is clear that for large designs, the complete approach becomes impractical compared to the global/detailed approach. Note that the execution times shown for the global/detailed formulation include all pre-processing steps.

It can be seen that for relatively small designs, the difference in the two approaches decreases. This is because for the global/detailed technique, the setup time required for pre-processing and for ILP-processing becomes the dominating factor.

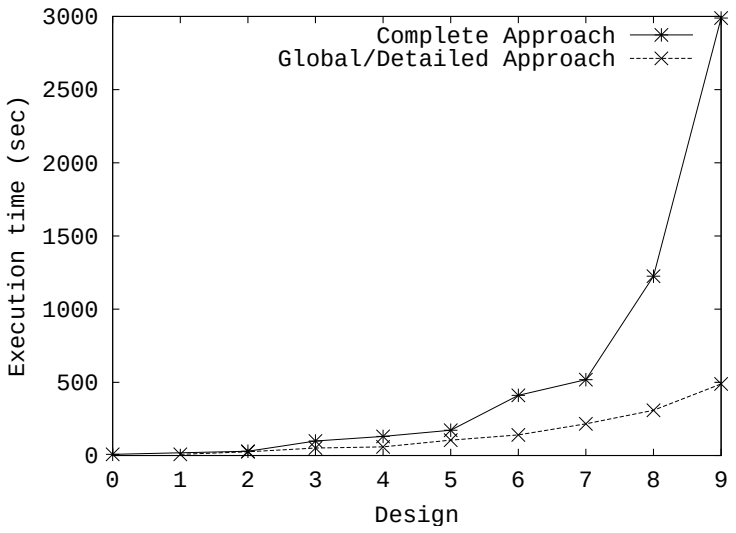


Figure 6.3: Complete Versus Global/Detailed Execution Times

The plot in Figure 6.3 gives a visual rendering of the results. The X-axis represents the different design points that are ordered in the increasing size of the problem (corresponding to the increasing row number in Table 6.3). For the sake of clarity, a line is plotted to connect the data points; it does not represent the performance of the algorithm between the test cases.

Finally, the difference in cost function values obtained in the two-layer and flat approaches were within 4% for the examples shown in Table 6.3. This difference would not have existed if the RC platform in the examples only had single or dual-port memory banks. The degradation obtained is due to the *consumed_ports*() not being optimal for memory banks with more than two ports.

6.4 Conclusion

The global versus detailed memory mapping paradigm, introduced in this work, eased the ILP formulation while conserving the quality of the resulting assignment. Large designs can be mapped faster and more efficiently than in the case of complete ILP formulation. In addition, the pre-processing of the data simplified the complexity of the global mapping formulation and ensured a successful detailed mapping.

As mentioned in the previous section, the *consumed_ports*() algorithm needs to be improved for memory banks with more than two ports. However, since the degradation in the solutions is within 4% from optimal ones, and since typical RC platforms are limited to single-port

or dual-port memory banks, the two-layer approach provides a fast and efficient solution to large mapping problems.

Finally, in the event of RAM ports limitation, the model could allow data structures to overlap at the price of adding conflict resolution to the objective function. This, along with what is presented in Chapter 9, can solve any resource conflicts that might arise due to port sharing amongst data structures.

Chapter 7

Spatial Partitioning: Logic and Memory Mapping

7.1 Introduction

In RCs containing multiple processing elements, the memory mapping problem becomes more complicated. The memory mapper needs to optimize the placement of the design's data structures with respect to the placement of their accessing logic tasks and not with respect to the processing elements. Knowledge of which part of the design's logic accesses what data structures is therefore needed for the mapping process.

Logic partitioning is the task of assigning logic in a design to specific processing elements of the RC system. In this work, the design's logic is assumed to be grouped into computational tasks (as described in Chapter 3). By examining the input/output ports of each computational task, it is easy to extract the access patterns of the computational tasks with respect to data structures in the design.

This chapter presents a framework in which the task of memory mapping interacts with logic partitioning in different ways to provide best memory mapping. Chapter 8 uses the formulations presented in this chapter to construct robust spatial partitioning formulations (both memory and logic mapping).

By extending memory mapping to include logic partitioning, the ILP formulations become quite complex. ILP formulations presented in Chapters 4, 5, and 6 need to be simplified, at the price of a relatively minor loss in the quality of mapping, in order to extend the formulations and still be able to handle large spatial partitioning problems. This chapter revisits the memory mapping formulations and introduces changes that simplify the problem.

Furthermore, the ILP formulations presented so far as well as the heuristic approach that was used for performance comparison did not optimize the data mapping with respect to the addressing logic. Restrictions should be placed on the data assignment in order to reduce the complexity in terms of addressing logic introduced to the design.

The rest of this chapter is organized as follows: Section 7.2 reviews the available literature on partitioning. Section 7.3 describes both roles of memory mapping and logic partitioning and the interaction between them. The next three sections lay the foundation for an ILP-based spatial partitioning solution. Section 7.4 starts by modifying the Integer Linear Programming approach presented in Chapter 4 for memory mapping on systems with single processing elements and physical memory banks with a single configuration. Section 7.5 extends the ILP approach to handle multiple configurations physical banks, as described in Chapter 5, however still targeting RCs with single processing elements. Section 7.6 reformulates the global/detailed memory mapping technique described in Chapter 6. These three sections lead to the ILP solution constructed for spatial partitioning that includes both the placement of logic and the assignment of data structures on the RC system. Finally, Section 7.7 concludes the chapter.

7.2 Background on Partitioning

In multi-FPGA RC systems, partitioning is the task of assigning interconnected components of a design to the available processing elements on the RC. This is done while ensuring that all tasks assigned to each processing element fit in that processing element and that the number of interconnections between components placed on different processing elements is less than or equal to the number of available pins on the processing elements.

The partitioning problem is known to be an NP-hard combinatorial problem [75] and available solutions in the literature target both bi-partitioning (assuming an RC system with only two processing elements) and multi-level partitioning (an RC system with any number of processing elements).

Heuristic and combinatorial optimization algorithms are available in the literature that address the problem of partitioning. The following lists five general partitioning solutions and briefly describes them:

1. *Genetic Algorithms:* These algorithms work on the basis of natural genetics. The basic concept consists of an iterative procedure in which a series of generations of populations, one per iteration, are created. Each member of the population, also

called chromosome, represents a solution of the problem being solved. The solution representation is based on a suitable encoding of the solution space. The evolution process needed to produce new generations of populations consists of three operators: *Selection Operators* probabilistically select some of the highly fit members of the current population that move to the next generation. *Crossover Operators* attempt to combine the features of highly fit member of a population in the hope of creating new members that are likely to be better fit than either of the parents. Finally, *Mutation Operators* randomly alter the structure of the chromosomes in an attempt to introduce new features into the population that did not exist in the previous generation. Note that mutation is analogous to a neighborhood move in the case of Simulated Annealing algorithms. The genetic search procedure was developed in 1975 [76], and since then has been used successfully for solving several combinatorial problems in VLSI design automation [77], [78], and [79].

2. *Simulated Annealing:* Simulated Annealing algorithms work on the principles of thermodynamics [80]. In contrast to greedy algorithms that can make only downhill moves (moves to a better solution) and get trapped in a local minima, Simulated Annealing easily climbs out of the local minima. The algorithms allow localized negative moves (moves that will increase the cost of the solution) in the hope to get out of any local minima; hence finding the global minima. With time, the algorithm tightens the range of negative solutions that it can consider by reducing the number of moves attempted. Due to the ease of adapting Simulated Annealing to a wide range of problems, when given a sufficient time, the algorithm does converge to a near-optimal solution. Simulated Annealing has been extensively used in VLSI problems [81], [82], and [83].

3. *Kernighan and Lin Algorithm:* The basic Kernighan and Lin algorithm [84] is a bi-partitioning algorithm that forms an initial partition and then swaps pairs of nodes between partitions according to some cost criteria and moves towards the best partition. In practice, the complexity of this algorithm is reduced by maintaining sorted lists in the programming implementation.

4. *Fiduccia-Mattheyses Algorithm:* This algorithm [85], that has it roots in the Kernighan and Lin algorithm, introduces two modifications: it moves only one vertex at a time and keeps a sorted list of candidates for moving onto the other side. By using efficient data structures, the time complexity is also reduced. Several enhancements exist to the Fiduccia-Mattheyses algorithm that further reduce the time complexity of the partitioning [86], [87], and [88].

5. *Clustering Algorithms:* Clustering algorithms group nodes in a graph into clusters

based on some measure of closeness [89]. These algorithms are often used to reduce the problem size to a more tractable problem instance. Given a large graph, clustering heuristic can be applied to cluster nodes together and form a new graph where each cluster is a node. It is shown that the Fiduccia-Mattheyses algorithm performs near-optimal after clustering while they perform relatively poorly before clustering. [90], [91], and [92] have motivated several two-phase approaches to move-based algorithms with an initial clustering phase.

The last three techniques discussed above are heuristic algorithms that are fast and efficient but are deterministic and rely heavily on the initial solution. On the other hand, the first two techniques are general purpose combinatorial optimization algorithms that are suitable to handle multiple costs and complex cost functions and, arbitrary combinations of constraint-satisfaction and optimization goals. For a more comprehensive review of various partitioning algorithms, refer to the survey article by Alpert and Kahng [93]. In this chapter, and as will be seen in Chapter 8, Integer Linear Programming optimization techniques are used to interact with memory mapping in order to partition the logic in the design. Compared to the above mentioned techniques, the ILP technique produces an optimal solution based on the set of constraints and the objective function of a problem. Furthermore, the ILP methodology allows easy insertion of new constraints and objective costs to the problem and it provides a framework in which a spatial partitioning problem can be clearly analyzed and decomposed.

7.3 Spatial Partitioning Methodology

This section describes interaction between the memory mapper and the logic partitioner and, considers the different approaches that can be achieved.

Figure 7.1 shows the interaction of the memory mapper/synthesis tool with an existing synthesis and partitioning tool for multi-FPGA RC boards [59].

In the previous chapters, a single processing element was considered, hence the task of memory synthesis can occur after logic placement. Once all tasks are placed within the processing element, memory mapping (or memory synthesis in Figure 7.1) is invoked to assign data structures of the design to the available physical memory banks.

In a RC system containing multiple processing units, the memory mapping process can occur at different stages of synthesis. Figure 7.2 shows three ways of incorporating memory mapping with logic partitioning:

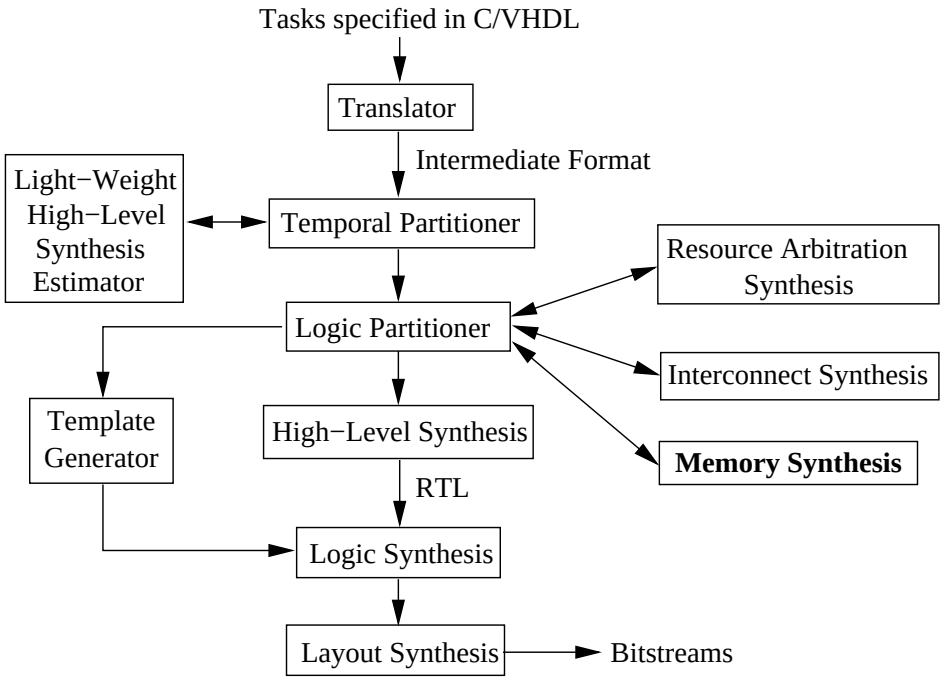


Figure 7.1: Interaction with the Synthesis Flow

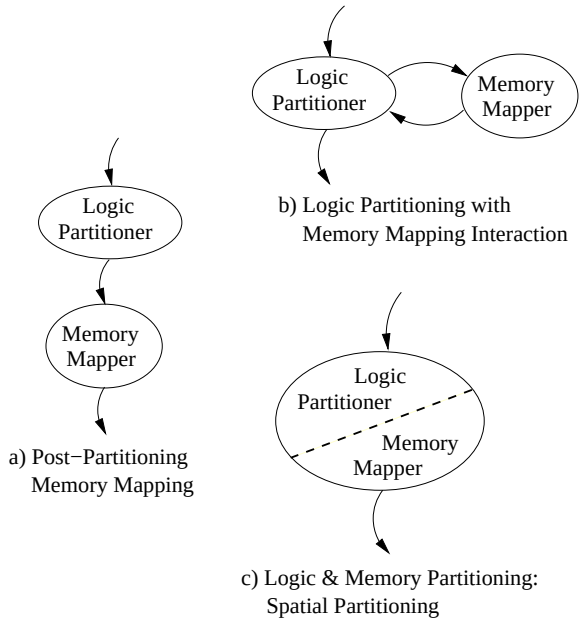


Figure 7.2: Partitioning and Memory Mapping

- Figure 7.2a depicts a *post-partitioning* scheme where memory mapping occurs after the logic partitioning is complete. The logic partitioner assumes that at least one memory mapping is possible but does not take into account how and where data structures are placed on the board. It also cannot estimate the number of pins that will be required for each processing element, in the event logic tasks are assigned to a processing element that has on-chip memory banks (i.e. if the data structures are assigned locally, there will be no need for off-chip communication).

- Figure 7.2b shows an *interaction* between the logic partitioner and the memory mapper. In this scenario, the logic partitioner contemplates logic partitions while taking into account possible memory assignments. Depending on how many times the logic partitioner invokes the memory mapper, a *lighter-weight* memory mapper technique could be implemented in order to speed the interaction. A lighter-weight mapper could be one with fewer or more relaxed constraints.

- In Figure 7.2c, the processes of logic partitioning and of memory mapping are merged. The resultant *merged* spatial partitioning engine considers both logic and data structures at the same time. Both problems are tackled as part of the formulation or of the heuristic algorithm.

In the previous chapters, since a single processing unit was used, the three views are equivalent to the post-partitioning approach where memory mapping occurs after logic partitioning. However, it will be seen in Chapter 8, that each view is handled differently.

7.4 Improving the Formulation for Single Configuration Banks

In order to include the task of logic partitioning while performing memory mapping, the ILP formulations stated in Chapters 4, 5, and 6 were used initially, but lead to poor results. In this chapter, the ILP formulation is analyzed from a slightly different perspective in order to achieve better results in view of the logic partitioning.

Different 0-1 variables are defined in the following formulation that do not quite match the definitions stated in Section 2.4.

As before, Z_{dt} associates a data structure to a memory type:

$$Z_{dt} = \begin{cases} 1 & \text{if data structure } d \text{ is assigned to some} \\ & \text{instance(s) of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

Z_{dt} is used to force an over-sized data structure to be split across banks of the *same type*.

Then, two sets of variables, X_{dti} and V_{dtip} are defined. Both associate chunks of a data structure to specific instances of a physical bank type. However, the first refers to instances that will be *fully* occupied, while the second refers to instances that will be *partially* occupied by the data structure. As an example, consider a data structure of 25 16-bit words, and instances of one physical memory bank type where each instance has 10 16-bit words. Obviously, more than one instance is required to fit the data structure onto this type of bank; in fact, 3 instances are required: 2 that will be fully occupied (for the first 20 words of the data structure), and a third instance that will be partially occupied (for the last 5 words of the data structure).

Following the example given above, Figure 7.3 shows the placement of a data structure with 55 18-bit words onto physical memory banks with 16 8-bit words. As can be seen from the dashed box in the figure, 6 banks will be fully occupied (banks 1, 2, 4, 5, 7, and 8). In Addition, the first three rows of the last column (banks 3, 6, and 9) will also be fully occupied although not all bits will be used. In other words, all locations in these instances will be used, but not all bits at each location. Thus, there will be 9 fully occupied banks. Finally, the last row of banks will be partially occupied since only 7 out of 16 available words will be used. Note that the first 2 banks of this row (banks 10 and 11) will utilize all bits of each 7 words, but the last bank (bank 12) will use only part of the 7 words. In summary, 9 banks were fully occupied (the first three rows of banks) and 3 banks were partially occupied (the last row of banks). Note that the part assigned to bank 11 (or bank 12) could actually be assigned to the second or third port of bank 10.

In general, the X variables denote all the memory banks that will be used by a data structure except for the last row of banks that will be associated to the V variables.

Thus, for the 0-1 variables, X_{dti} associates a data structure to a specific physical bank and to a specific physical bank instance of that type, such that those instances are fully occupied by that data structure:

$$X_{dti} = \begin{cases} 1 & \text{if instance } i \text{ in bank type } t \text{ is fully occupied by data structure } d. \\ 0 & \text{otherwise.} \end{cases}$$

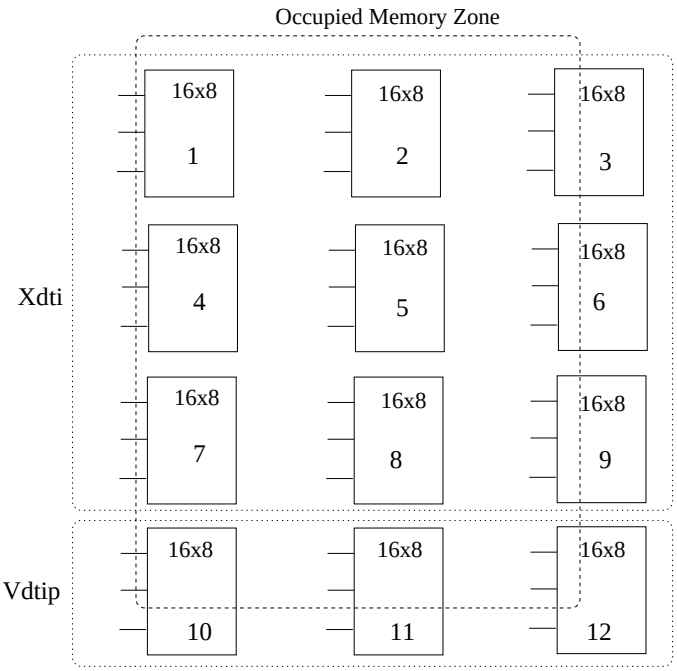


Figure 7.3: Mapping a 55x18 Data Structure Onto 16x8 Instances

Similarly, V_{dtip} associates a data structure to a specific physical bank type and to a specific physical bank instance of that type, but such that those instances are partially occupied by that data structure:

$$V_{dtip} = \begin{cases} 1 & \text{if port } p \text{ of instance } i \text{ in bank type } t \text{ is partially occupied by data structure } d. \\ 0 & \text{otherwise.} \end{cases}$$

Contrary to the definitions stated in Section 2.4, note that in the X variables, there is no need to keep track of the port number, since the instance will be fully utilized by the associated data structure, hence X will use one port but make all other ports of that instance useless. On the other hand, in the V variables, there is a need to keep track of the port number since the bank will not be fully occupied by the associated memory, hence leaving space for other data structures to be assigned to the free ports of that bank. In other words, once an X variable is set to one, the bank instance it is being mapped to is completely used by the associated data structure and no other data structure can be simultaneously mapped onto that bank. However, multiple V variables belonging to different data structures can be mapped onto the same physical bank. The port number index in the V variable specifies which port of the bank a data structure is assigned to.

In addition, multiple V variables belonging to the same data structure *could* be assigned

to the same physical bank. For example, the chunk of data assigned to memory bank 11 in Figure 7.3 could have been assigned on the second port of memory bank 10; the only constraint being that this chunk and the chunk already existing on the first port of bank 10 must fit within the available memory space of the bank. This flexibility allows to place chunks of a data structure in an arbitrary manner, taking advantage of the available space.

The following section describes the improved ILP formulation of the flat approach that will benefit the spatial partitioning process. Only single configuration banks are handled in these formulations.

7.4.1 Constraints Formulation

The following are the important constraints in the memory mapping problem for single configuration memory banks.

- **Uniqueness constraints:** As before, each data structure should be mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS} \sum_{t \in PB} Z_{dt} = 1$$

- **Capacity constraints:** Each data structure should be fully stored on the hardware. For that, the number of X and V variables is computed:

$$\forall_{d \in DS}\ \forall_{t \in PB}\ \sum_{i=1}^{I_t} X_{dti} = \left\lfloor \frac{D_d}{D_t} \right\rfloor * \left\lceil \frac{W_d}{W_t} \right\rceil * Z_{dt}$$

  where the floor operator is used to ignore the last row of memory banks.

  For the V variables:

$$\forall_{d \in DS}\ \forall_{t \in PB}\ \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} V_{dtip} = \left\lceil \frac{W_d}{W_t} \right\rceil * Z_{dt} * \begin{cases} 1 & \text{if } (D_d \ mod \ D_t) \neq 0 \\ 0 & \text{if } (D_d \ mod \ D_t) = 0 \end{cases}$$

  Note that if there is no remainder from the depth ratio, there is no need for V variables. This can be known before formulating the problem, thus the last part of the equation will not be processed by the ILP solver. This also applies to the X variables although the formula does not have any "if" statements – when the data structure depth is less than the physical bank depth, there will be no X variables (since the floor() function would be zero).

- **Port constraints:** Each X variable will occupy the entire bank, thus every instance

will have at most one X variable assigned to it:

$$\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} : \sum_{d \in DS} X_{dti} \leq 1$$

It will be seen in the physical storage constraints below that this constraint is redundant.

Each V variable will occupy one port of a physical bank, so the number of V variables assigned to one bank instance must be less than or equal to the number of available ports:

$$\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t}\ \forall_{1 \leq p \leq P_t} : \sum_{d \in DS} V_{dtip} \leq 1$$

- **Physical storage constraints:** In multi-ported banks, the sum of the storage space mapped to every port must fit within the bank:

$$\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t} X_{dti} * D_t + V_{dtip} * (D_d\ mod\ D_t) \leq D_t$$

In this inequality, not more than a single X variable can be one, thus the first port constraint listed above is implicitly stated here.

Note that, if there exists one X variable set to one on a certain bank instance, then all V variables on all ports of that instance should be zero; whereas if all X variables are set to zero on a certain bank instance, then V variables could either be zero or one based on the mapping. In a different perspective, this is equivalent to the following:

$$\begin{aligned}
\text{if} \quad & \left(\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} : \sum_{d \in DS} X_{dti} = 1\right) \\
& \text{then} \left(\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} : \sum_{d \in DS} \sum_{p=1}^{P_t} V_{dtip} = 0\right) \\
\text{else if} \quad & \left(\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} : \sum_{d \in DS} X_{dti} = 0\right) \\
& \text{then } \textit{no constraints on} \left(\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t} : \sum_{d \in DS} \sum_{p=1}^{P_t} V_{dtip}\right)
\end{aligned}$$

By linearizing this "if-then-else" statement which is equivalent to a constraint in the form $[\alpha * (1 - X)]$, the physical storage constraint stated initially is obtained, hence validating the formulation.

- **Lifecycle conflict constraints:** When there is a conflict between two data structures,

they cannot occupy the same memory space or ports of bank instances. In this chapter, no new constraints will be added to the formulation due to lifecycle conflicts. Instead, a pre-processing mechanism is introduced in the following section that resolves lifecycle conflicts *before* the ILP problem is solved. This reduces the complexity of the ILP problem, and as can be seen from the next section, retains the high quality of the mapping.

The constraints formulation stated above assumes that a data structure will be split in a pre-defined way. A data segment of 20 words will *not* be split if the bank instances that it is being mapped to have 20 word locations or more, with the assumption that the width of each word in the data segment is less than or equal to the width of the memory locations in the banks. This introduces one minor limitation to the mapping that is illustrated in the following example. Consider the data structures and bank instances shown in Figure 7.4. Since the three data structures are smaller than any one of the two instances, all X variables will be zero, and exactly three V variables will be set to one. If data structure 1 is assigned to bank A, then data structure 2 will be assigned to bank B. Data structure 3 will have to be split in two chunks, one that goes on the second port of bank A and one that goes on the second port of bank B. For this to happen, data structure 3 must have two V variables set to one instead of just one. The current model does not allow this splitting since by doing so, the mapping can end up heavily fragmenting data structures, leading to highly complex addressing logic.

As a workaround to this problem, if a mapping fails, the assignment can repeat by ignoring one or more small data structures. If the new assignment succeeds, a post-processing tool can then split the remaining non-assigned data structure(s) into chunks that fit on the available ports/memory space of the banks. This solution will increase the complexity of accessing the data structure since a memory request has to be routed to the appropriate memory bank that hold the requested address.

Note also that when the number of physical banks is very large, the model might not need to consider all available banks. An upper bound on this number, for each type of physical banks, can be used instead. As in the previous formulations, several other simplifications could be made in order to speed the execution while still producing same quality results.

7.4.2 Lifecycle Conflicts Analysis

In the previous chapters, lifecycle conflicts were handled in the constraints formulation, thus making the problem complex and very time consuming. In this chapter, a pre-processing of

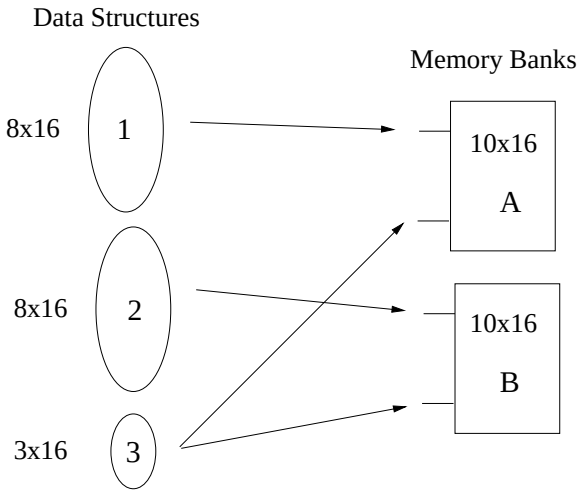


Figure 7.4: Mapping 3 Small Data Structures Onto 2 Bank Instances

all non-conflicting data structures is performed. The pre-processing reduces the size of the problem while retaining a high quality of mapping solution.

As seen in Section 2.4, a conflict pair indicates a lifecycle conflict between two data structures. Conflicting data structures *have to* be mapped onto separate memory spaces and separate memory ports since the information present in the two data structures need to be available in an overlapping time period. The constraints formulation stated in the previous section is fit for a problem where all data structures conflict in their lifecycles. In other words, all pairs $\forall_{i \in DS}\ \forall_{j \in DS}\ i \neq j :\ \{DS_i, DS_j\}$ are present in the problem formulation.

When only a subset of the above conflict pairs exist, the problem becomes a little more challenging. In the simplest case, assuming that no pairs exist in the set of conflict pairs and that all data structures in the design have the same dimensions (*depth* x *width*), the problem consists of mapping data structures that do not conflict at all. In this case, a single *equivalent data structure* with the same dimensions as that of the design's data structures can replace all data structures and the problem reduces to simply mapping this equivalent data structure. Once the equivalent data structure is mapped, all data structures in the design can be assigned to the ports and instances that the equivalent data structure was assigned to and the mapping is complete. In real problems, three complexities arise:

- **Selecting the Dimensions of the Equivalent Data Structure:** In the above example, all data structures assume the same depth and width, but how are the dimensions selected when the dimensions of data structures vary? The equivalent data structure must be chosen adequately so as to accommodate *any* design's data structure. Furthermore, it should not be too large and should fit in smaller memory types when

any design data structure can fit in a smaller memory type. As a first upper-bound, the equivalent data structure (labeled as e) can be chosen with the following width and depth:

$$W_e = max\left(\forall_{d \in DS} \lceil W_d \rceil^*\right)$$

and:

$$D_e = max\left(\forall_{d \in DS} \lceil D_d \rceil^*\right)$$

* The ceiling operator denotes the closest multiple of W_t (or D_t) greater than or equal to W_d (or D_d).

This equivalent data structure might be much larger than any of the data structures in the design. If the design has two data structures, one with 5x100 dimension and one with 100x5, the resulting equivalent data structure will have dimensions greater than or equal to 100x100, which makes it much larger than any one of the two data structure in the design.

As a better approximation – a tighter upper-bound, the X and V variables of the equivalent data structure can be computed by:

$$\forall_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} X_{eti} = max_{\forall_{d \in DS}} \left(X_{dti} + V_{dtip}\right)$$

and:

$$\forall_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} V_{etip} = 0$$

This upper-bound will result in an equivalent data structure that can accommodate any one of the data structures in the design, but still has some wastage of memory space. This is due to the rounding being done on the partially full instances of the maximum data structures and hence might not be beneficial in all cases.

Finally, a tight upper-bound can be obtained by setting the X and V variables of the equivalent data structure to:

$$\forall_{t \in PB} \sum_{i=1}^{I_t} X_{eti} = max_{\forall_{d \in DS}} \left(X_{dti}\right)$$

and:

$$\forall_{t \in PB} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} V_{etip} = max_{\forall_{d \in DS}} \left(X_{dti} + V_{dtip}\right) - max_{\forall_{d \in DS}} \left(X_{dti}\right)$$

Finally, the number of words that will be allocated in the instances where the V variable

is set to one is:

$$\forall_{t \in PB} \ (\#words_t) = (D_e \ mod \ D_t) =$$

$$max_{\forall_{d \in DS}} \left[(D_d \ mod \ D_t) \mid \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} (X_{dti} + V_{dtip}) > X_{eti} \right]$$

When the equivalent data structure is being processed in the physical storage constraint of the previous section, the variable $\#words$ will replace any occurrence of: $(D_e \ mod \ D_t)$

In this equivalent data structure, there will be exactly as many fully occupied instances as needed by the biggest data structure in the design. The number of words that are allocated to each V-assigned instance is computed as an upper-bound which might yield some wastage in memory space. However, this waste is insignificant compared to the previous two upper-bound techniques.

- **Data Structures Have Significantly Different Dimensions:** Second, data structures in a design have significantly different dimensions and this makes the selection of the equivalent data structure more complex. Since the equivalent data structure has to accommodate all data structures, it has to have dimensions larger than or equal to the largest data structure in the design. This has been described thoroughly in the previous paragraphs. Consider the example shown in Figure 7.5. If, due to size constraints, the equivalent data structure will only fit on memory type 2, this makes both data structures 1 and 2 mapped onto instances of memory type 2. However, if the data structures were mapped separately onto the architecture, data structure 2 will still be mapped onto instances of memory type 2, but data structure 1 can fit on instances of memory type 1. If the access time of memory type 1 is much lower than the access time of memory type 2 and/or the proximity of type 1 is much closer to the logic area than the proximity of type 2, then a large speed penalty is incurred by using the equivalent data structure.

  This problem occurs only when there is a large amount of memory available on the RC, so that the mapping does not need to overlap all the data structures. To address this problem, an analysis of all the non-conflicting data structures groups the data structures in multiple sets, and each set is replaced by an equivalent data structure. All data structures in a set are chosen so that they are very close in size to their equivalent data structure. The number of sets selected should take into account the total available memory space in the RC to ensure that all equivalent data structures fit on the system.

- **Not All Data Structures Can Overlap:** Finally, since not all data structures are

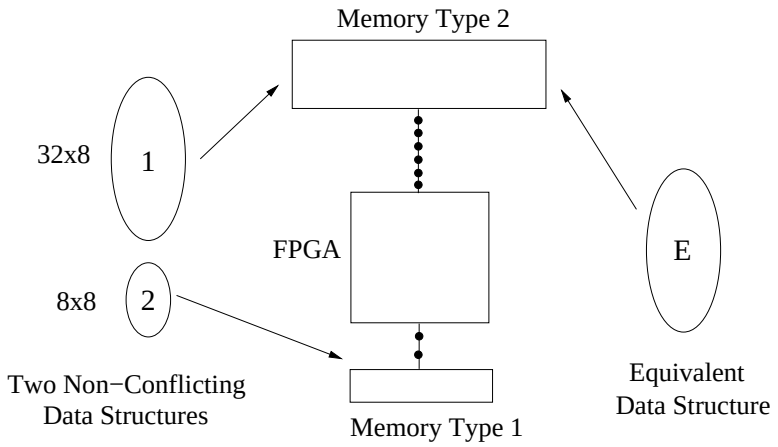


Figure 7.5: Mapping of Equivalent Data Structure

non-conflicting, i.e. not all conflict pairs exist in every problem, there are multiple sets of non-conflicting data structures. Each set contains data structures that do not conflict with each other. In order to form the sets, the following graph theory [94] steps are followed:

1. *Obtain conflict graph:* form a graph where nodes are data structures in the design and arcs connecting nodes represent the conflict pair between the two data structures.
2. *Complement conflict graph:* to obtain the non-conflicting pairs of data structures, the conflict graph is complemented.
3. *Form cliques:* Form non-overlapping cliques of fully-connected nodes in the complemented graph. Each clique corresponds to data structures that can completely overlap on the memory space.
4. *Select cliques:* From all the cliques obtained in the previous step, select non-overlapping cliques (no nodes in common) such that the equivalent data structure of each clique is close in dimensions to the nodes of that clique. The objective is two-fold. First, minimize the number of cliques since each clique will be represented by one equivalent data structure in the mapping problem. Second, optimize the selection of equivalent data structures by ensuring that each data structure in a clique is not much smaller than its corresponding equivalent data structure.

In conclusion, the lifecycle conflicts analysis replaces each subset of data structures of the design by an equivalent data structure. The subset has to be selected carefully so to minimize

loss in mapping quality. In addition, the equivalent data structure has to be constructed appropriately so to minimize wastage of memory space.

Once the equivalent data structures replace their corresponding data structures, the mapping resumes with the same constraints of Section 7.4.1. When the mapping is complete, all data structures that were replaced by an equivalent data structure will be mapped to the space assigned to their equivalent data structure. The selected size of the equivalent data structure ensures that any data structure in the subset fits in the assigned memory space.

7.4.3 Objective Formulation

As before, the objective of the ILP model is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost function takes the form:

$$minimize[Cost_1 * \alpha_1 + Cost_2 * \alpha_2 + ... + Cost_n * \alpha_n]$$

where α_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components.

The objective formulation in an ILP model is compact and allows easy expansions. Three cost components are depicted below. They try to cover the speed performance and the pin limitation constraints:

- **Latency cost:** Assuming that RA_d and WA_d provide a good estimation on the number of reads and writes from and to data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [RA_d * RL_t + WA_d * WL_t]$$

- **Pin delay cost:** Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (RA_d + WA_d) * T_t$$

- **Pin I/O cost:** The larger the width of a data structure the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * W_d * T_t$$

The α_i weight coefficient are used for two reasons:

1. *Normalization:* If one cost component is at a unit scale not proportional to the other components, the weight coefficient is used to normalize the components and make them uniform with respect to each other. E.g. if the latency cost is in the 10^6 order, the pin delay cost in the 10^4 order, and the pin I/O cost in the 10^2 order, the respective weight coefficients can be set to 10^{-4}, 10^{-2}, and 10^0 so as to have all cost components at the same scale.

2. *Emphasis:* If in the mapping process one cost component is more important than another, the emphasis can be manifested by altering the corresponding weight coefficient. E.g. if the latency cost is much more important than both pin delay and pin I/O costs, the respective weight coefficients can be set to 0.5, 0.25, and 0.25 so to stress the importance in the ILP solution.

As mentioned in Chapter 4, in all objective functions presented in the thesis, the weight coefficients α_i were obtained by first setting them to one, then finding a mapping solution and analyzing the values of the cost components obtained. Finally, the weight coefficients were appropriately set based on the magnitude and emphasis factor of each cost component.

7.4.4 Preliminary Results

For the improved single configuration bank mapping, initial results are shown in the plot of Figure 7.6. The plot shows execution times for the ILP solvers for problems including 25, 50, and 100 data structures, and 30, 50, 70, 90, and 110 physical bank instances. Again, results were obtained on a SUN Ultra-30 (248MHz with 128MB RAM). For the sake of clarity, lines are plotted to connect the data points; they do not represent the performance of the algorithm between the test cases.

At first sight, the results seem similar to the ones obtained in Chapter 4, however they are actually worse in terms of execution speed. The improvement to the formulation although simplified the X_{dtip} variables to X_{dti}, it introduced the new V_{dtip} variables. The improvement is not in terms of execution speed of the ILP solver, but in terms of the regularity and the efficiency in mapping the data structures to the RC platform. This improvement makes the process of synthesis much simpler in addition to reducing considerably the addressing logic introduced due to memory mapping. Furthermore, it avoids wasting memory space due to rounding in the formulations. Instead, the V_{dtip} along with the X_{dti} variables ensure that only required memory space is allocated to the data structure. Hence, the objective

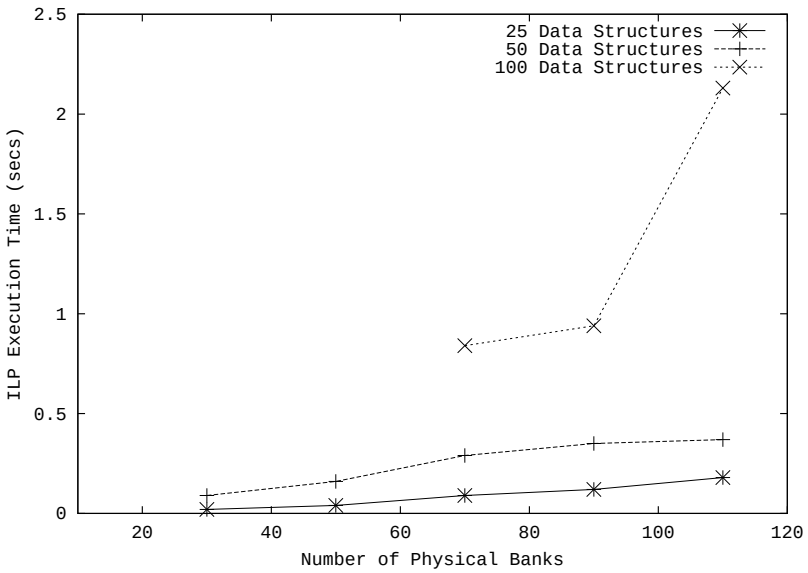


Figure 7.6: ILP Execution Times for Single Configuration Banks

function values obtained in the improved formulation are lower than the ones obtained in the formulation presented in Chapter 4. Since the objective function values are not meaningful, the percentage improvement factors obtained by comparing the old and improved formulations are shown in Table 7.1.

It can be seen from Table 7.1 that when the assignment is tight (i.e. many data structures for fewer physical banks), the improvement in objective function values are higher. Since the old formulation wastes space in rounding the data structure sizes, the mapper is forced to migrate to a slower physical bank type; whereas the improved formulation avoids wasted space and populates the faster/closer memory banks wisely.

In the case of data structures that can occupy the same memory space (i.e. their lifecycles do not overlap), the improvement in the technique does lead to smaller execution speeds. In the formulation of Chapter 6, the $max()$ operator makes the formulation more complex and slows down considerably the execution of the solver. However, with the improvements introduced in Section 7.4.2, a fast pre-processing technique that replaces sets of data structures with their equivalent data structure, makes the complexity of the ILP problem much simpler and solutions are generated much faster.

# of Data Structures	# of Physical Bank Types	Improvement (%)
25	30	4.4
25	50	3.9
25	70	3.5
25	90	2.2
25	110	2.2
50	30	8.8
50	50	6.0
50	70	5.6
50	90	5.9
50	110	5.0
100	70	11.2
100	90	7.5
100	110	5.4

Table 7.1: Improvement Factors in Memory Mapping for Single-Configuration Banks

7.5 Improving the Formulation for Multi-Config Banks

The previous section improved upon the memory mapping process for RC with single configuration banks. This section extends these formulations to handle multi-configuration banks.

The goal of the formulation in this section is to take advantage of the programmability feature of the multi-configuration banks in order to improve the access performance of logic tasks to physical memory banks. Since each port of each instance of a multi-configuration bank type can be configured separately to have its own width and depth dimensions, the formulation becomes quite more complex. When a data structure is mapped onto a memory bank type that handles multiple configurations, some restrictions should be placed so that the access to the data structure is not too complicated. The next section describes how multiple configurations are dealt with and how the optimal configuration is selected for each data structure and memory bank type pair. Once the optimal configuration is selected, the ILP formulation is detailed

7.5.1 Analysis of Multi-Configuration Banks

When a data structure is mapped on instances of a bank type that handle multiple configurations, there are several ways of mapping the data structure. For example, assuming instances of a dual-port bank type that handle four configurations: {32x1, 16x2, 8x4, 4x8},

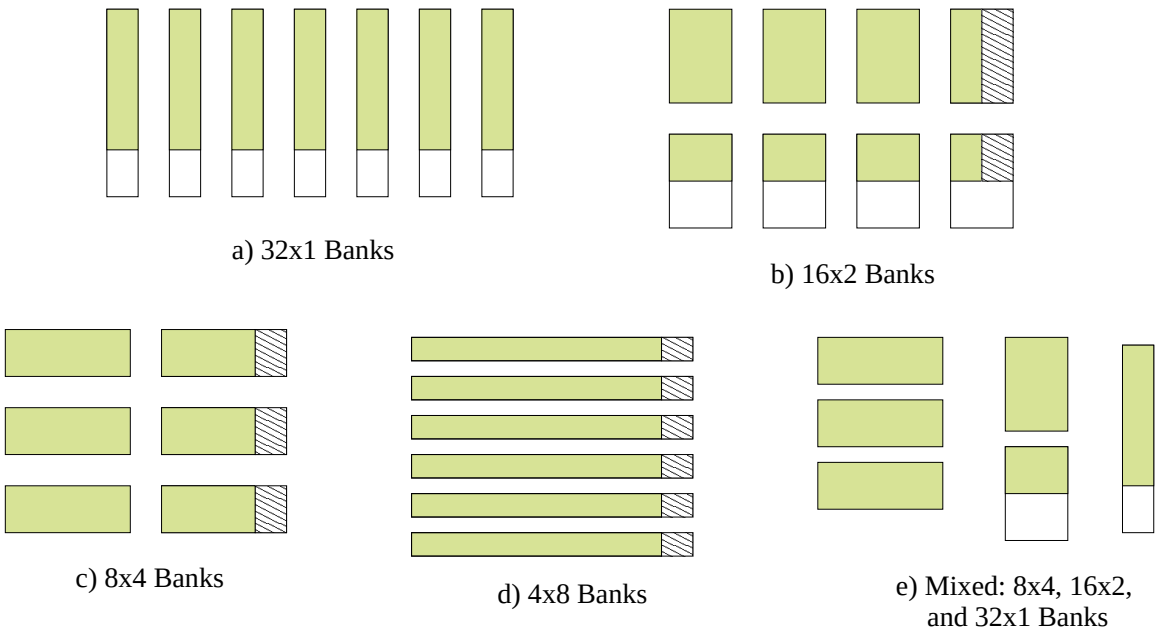


Figure 7.7: Mapping a 24x7 Data Structure Onto Multi-Configuration Instances

and a single data structure of size 24x7, Figure 7.7 shows five different mapping solutions that are available. Note that there are also other mapping solutions available for that data structure. In the Figure, the gray area of the bank instances denotes parts of the instances that are occupied by the data structure. The dashed area denotes parts of the instances that are wasted since the data structure did not utilize them nor are they available for other data structures to be assigned to. Finally, the empty area denotes parts of the instances that are not occupied by the data structure and that are available for other data structures.

It is difficult to judge which solution yields the best mapping in the overall partitioning problem. Several contradictory goals are to be achieved. However, it is clear that the main goal is of compactly fitting the data structures on the instances while minimizing the number of instances used and the amount of wasted space.

One of the most important issues that needs to be addressed by the mapper is the ease of synthesis of the resulting partitioned design. By dividing a data structure and assigning it to multiple memory instances, a complex addressing scheme is introduced to the design. Figure 7.8 shows some of the required logic that is added in order to access the data structure. The *write enable* input of each bank needs to be generated also – a combinational function of $A3$ and $A4$ for each row of banks is required. This is shown only for one bank in the Figure. Note that since the sizes of all configurations are powers of two, it is not required to decode and introduce adders/subtractors to compute the effective addresses within each bank instance.

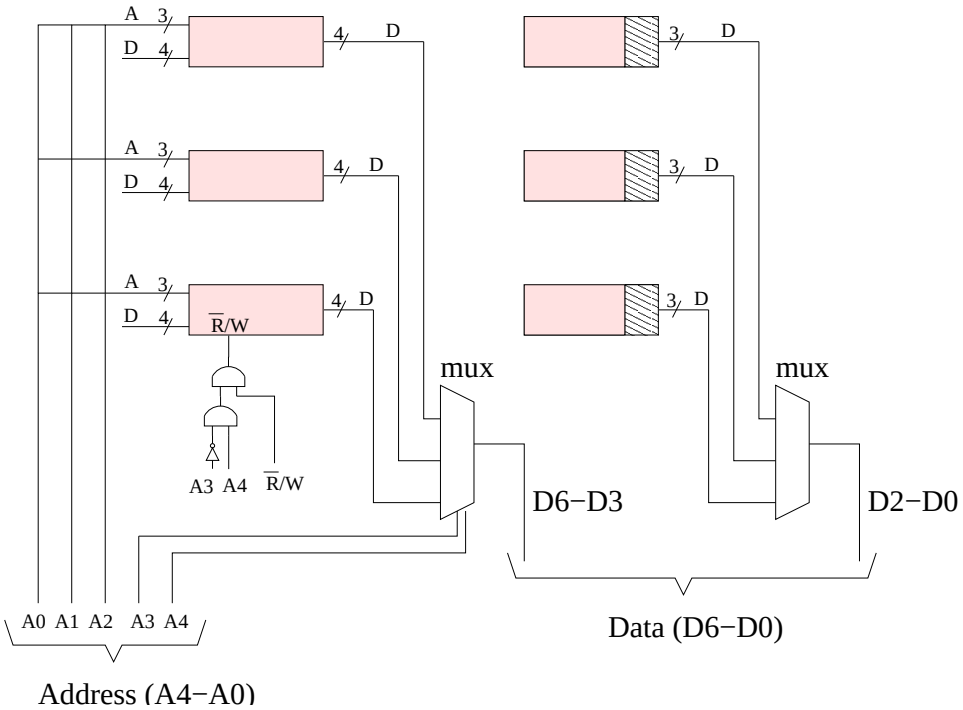


Figure 7.8: Addressing Logic For Mapping a 24x7 Data Structure onto 8x4 Banks

It is clear that the complexity of the addressing logic grows when more bank instances are needed for a data structure. In addition, when different configurations are used for a single data structure, the logic required for the addressing becomes asymmetrical across bank instances thus making the synthesis process quite complex as well as increasing the area required for addressing logic.

For the above reasons, the solutions followed in this chapter restrict each data structure to one configuration of memory instances. Hence, for each data structure and physical memory bank type pair, a single configuration is selected before the mapping is done. In Figure 7.7, this is equivalent to restricting the placement to cases a, b, c, and d, and forbidding case e.

The remaining task is to select which configuration will best fit each data structure on each physical bank type. In Figure 7.7, this is equivalent to choosing one of the four valid configurations (cases a, b, c, and d). Hence for each data structure and physical memory bank type pair, (d, t), and configuration c, the following is a list of criteria that guide the selection and compare configuration c with other configurations for the (d, t) pair:

1. Minimize the number of instances assigned to the data structure. The number of

instances is given by:

$$\#Instances = \left\lceil \frac{D_d}{D_{t[c]}} \right\rceil * \left\lceil \frac{W_d}{W_{t[c]}} \right\rceil$$

2. Minimize the multiplexor logic added for addressing logic. The number of multiplexors is given by:

$$\#Muxes = \left\lceil \frac{W_d}{W_{t[c]}} \right\rceil$$

The number of inputs to each multiplexor is:

$$\#MuxInputs = \left\lceil \frac{D_d}{D_{t[c]}} \right\rceil$$

And the number of bits per input/output to/from each multiplexor is:

$$\#BitsPerMuxInput = W_{t[c]}$$

3. Minimize the number of nets introduced. The number of nets that are going to the select lines of the multiplexors is:

$$\#MuxSelects = \#Muxes * log_2 \left(\#MuxInputs\right)$$

And the number of nets used for the address bits of the physical bank instances is:

$$\#AddressNets = \#Instances * log_2 \left(D_{t[c]}\right)$$

4. Minimize the number of wasted bits located in the last column of the mapping (e.g. last columns of bank instances in Figure 7.7c). This waste is due to the fact that since the same configuration is selected for all instances on which the data structure is assigned, it is probably that the total width of each row of instances is slightly larger than the width of the data structure. The number of wasted bits is:

$$\#ColumnWastedBits = D_{t[c]} * \left(W_{t[c]} - \left(W_d \bmod W_{t[c]}\right)\right)$$

Data structures assigned on a multi-port memory bank should be stored starting at address locations that are powers of two. This facilitates the address logic by avoiding the insertion of adders to compute the effective address. Furthermore, the beginning address locations should start at boundaries of words corresponding to the largest configuration width. Since all configuration widths are multiple of the smallest configuration width, this guarantees that any configuration chosen can start on the selected

boundary. For doing so, there will be a small amount of unused memory between the end of the memory space corresponding to one port of an instance and the beginning of the memory space corresponding to another port of the same instance. The number of wasted bits in this case is:

$$\begin{aligned} \#RowWastedBits &= \left\lceil \frac{\left(D_d \ mod \ D_{t[c]}\right) * W_{t[c]}}{max_{\forall_{m \in C_t}}\left(W_{t[m]}\right)} \right\rceil * max_{\forall_{m \in C_t}}\left(W_{t[m]}\right) \\ &\quad - \left(D_d \ mod \ D_{t[c]}\right) * W_{t[c]} \end{aligned}$$

5. Minimize the number of ports used. The number of ports used by the data structure d on bank type t includes ports that are used for accessing the data structure as well as the ports that are wasted in the event the data structure is occupying the entire multi-ported physical instance (i.e. since only one port is used for the data structure, the other ports of the instance are wasted). The number of consumed ports is given by:

$$\begin{aligned} \#UsedPorts &= P_t * (\#Xs\_set\_to\_one) + (\#Vs\_set\_to\_one) \\ &= P_t * \left\lceil \frac{W_d}{W_{t[c]}} \right\rceil * \left\lfloor \frac{D_d}{D_{t[c]}} \right\rfloor + \left\lceil \frac{W_d}{W_{t[c]}} \right\rceil * \begin{cases} 1 & \text{if } (D_d \ mod \ D_t) \neq 0 \\ 0 & \text{if } (D_d \ mod \ D_t) = 0 \end{cases} \end{aligned}$$

6. Finally, when the number of V variables set to one, grows, the mapping can end up assigning many different data structures onto the same instance (different ports of an instance). When this occurs, more addressing logic has to be inserted to account for the address offsets needed to address all ports of an instance. The number of V variables as described in Section 7.4 is:

$$\#Vs\_set\_to\_one = \left\lceil \frac{W_d}{W_{t[c]}} \right\rceil * \begin{cases} 1 & \text{if } (D_d \ mod \ D_t) \neq 0 \\ 0 & \text{if } (D_d \ mod \ D_t) = 0 \end{cases}$$

The above six criteria are general guidelines for selecting a configuration for each data structure and physical memory bank type, (d, t). In order to do that, an ILP formulation is generated for each (d, t) pair and the solution from the ILP program is the configuration number associated with the (d, t) pair. Since the number of configurations is usually very small – less than 10, and typically between 1 and 5 – the ILP solution time is negligible.

The only variable in the ILP formulation is F_c which is defined as:

$$F_c = \begin{cases} 1 & \text{if configuration } c \text{ is selected for the } (d, t) \text{ pair.} \\ 0 & \text{otherwise.} \end{cases}$$

Also, only one constraint exists in the ILP formulation. The constraint forces a single configuration to be selected for each (d, t) pair:

$$\sum_{c \in C_t} F_c = 1$$

Finally, the objective function is to optimize the six criteria listed above. The objective formulation takes the form:

$$minimize \left[\beta_1 * \left(\sum_{c=1}^{C_t} Cost_1 * F_c \right) + \beta_2 * \left(\sum_{c=1}^{C_t} Cost_2 * F_c \right) + ... + \beta_3 * \left(\sum_{c=1}^{C_t} Cost_6 * F_c \right) \right]$$

where β_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components and the $Cost_i$ components are the quantities listed in the six criteria above.

Initial Results

Table 7.2 shows ILP execution speeds for different types of memories. The third and fourth column refer to the number of configurations for each instance in the memory bank type and the time it takes to generate the ILP solution. It is assumed in these results that all instances across all physical bank types have the same number of configurations.

The ILP formulation stated above is simple since, for each data structure and physical bank type (d, t), only a single set of variables, F_c, exists. Furthermore, there are as many variables as there are configurations in one instance of a physical memory bank type. Since the number of configurations is typically in the $[1; 10]$ range, the ILP solution is obtained in a negligible amount of time, as shown in the table. The table includes pre-processing time required to generate the constraint and objective cost components.

The last few rows in Table 7.2 correspond to very large designs, where one hundred data structures exist and twenty different *types* of memory banks exist. In practice, the number of memory bank types is below twelve.

The efficiency of the formulation relies heavily on the cost components as well as the normalization and weighting factors used in the objective function. The results obtained in the following sections depend on the formulation provided in this section. Once the objective function is fully generated, it is seen that the pre-processing technique used to generate the best configuration for each (d, t) pair, leads to a simplified yet very efficient ILP mapping solutions.

# of Data Structures	# of Physical Bank Types	For every (d,t) pair		Total ILP Execution Time (in secs)
		Number of Configurations	ILP Execution Time (in secs)	
50	5	5	0.01	2.5
50	5	10	0.03	7.5
100	5	5	0.01	5
100	5	10	0.03	15
50	10	5	0.01	5
50	10	10	0.03	15
100	10	5	0.01	10
100	10	10	0.03	30
50	20	5	0.01	10
50	20	10	0.03	30
100	20	5	0.01	20
100	20	10	0.03	60

Table 7.2: ILP Execution Speed for Finding Best Configurations

7.5.2 Constraints and Objective Formulation

Once a configuration is selected for each (d,t) pair, the ILP formulation for mapping the data structures onto the physical memory banks is generated. The formulation of the constraints is similar to the one listed in Section 7.4.1 but with a couple of differences.

First, all occurrences of D_t and W_t are replaced by $D_{t[b]}$ and $W_{t[b]}$ respectively. The index b refers to the best configuration that pairs a data structure with a physical memory bank type as described in Section 7.5.1.

Second, the physical storage constraint is different. Since there exists multiple configurations for each instance, it is not enough to check on the depth of the data structures assigned to the instance. There is also a need to check on the total number of bits. Hence, the new constraint becomes:

Physical storage constraints: Depending on the configurability feature of the physical memory bank type, the sum of the storage space mapped to every port must fit within the bank:

$$\forall_{t \in PB} \; \forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t}$$

$$X_{dti} * \left(D_{t[1]} * W_{t[1]}\right) + V_{dtip} * \left\lceil \frac{\left(D_d \; mod \; D_{t[b]}\right) * W_{t[b]}}{max_{\forall_{m \in C_t}} \left(W_{t[m]}\right)} \right\rceil * max_{\forall_{m \in C_t}} \left(W_{t[m]}\right) \leq D_{t[1]} * W_{t[1]}$$

Again, this equation is based on the assumption of Equation 2.1. In other words,

$$\forall_i, \ ,\forall_j, \ 1 \leq i,j \leq C_t : \ D_t[i] * W_t[i] = D_t[j] * W_t[j]$$

If bank type t has a single configuration, then the physical storage constraints equation is replaced with:

$$\forall_{t \in PB} \ \forall_{1 \leq i \leq I_t} \ \sum_{d \in DS} \ \sum_{p=1}^{P_t} X_{dti} * D_t + V_{dtip} * (D_d \ mod \ D_t) \leq D_t$$

In the case of the multi-configuration constraint stated above, the equation ensures that the total number of bits assigned to an instance is less than or equal to the capacity of the bank. If any X variable is set to one, then all other variables corresponding to this bank (both X and V variables) should be set to zero. However, for the V variables, the equation rounds the parts of data structures that are smaller than the instance to the closest multiple of the largest width configuration as described in criteria #4 of Section 7.5.1.

Except for the physical storage constraint described above, no changes are required for the other constraints in the ILP formulation. In addition, the objective remains the same, trying to minimize the latency cost, the pin delay cost, and the pin I/O cost. One worthy remark about the objective function: besides criteria #6 of Section 7.5.1 which aim is to reduce the number of V variables for each (d,t) pair, it is beneficial to map data structures to different instances when available. When more than one data structure is assigned to a physical memory bank instance, more work has to be done by the synthesis process in order to decode the addresses of the data structures since all but one data structure will be assigned to a non-zero initial position within the instance (similar problem as in criteria #6). However, the criteria listed in the previous section aimed at finding the best configuration for a (d,t) pair by reducing the total number of V variables for each (d,t) pair but did not take into consideration the interaction between data structures once they are mapped on the RC system. To decrease the number of data structures mapped onto a *same* physical memory instance, the objective function has to be modified. The objective function must have a new component which will minimize the number of data structures assigned to a same instance. In an ILP notation, this is equivalent to:

$$minimize \left[max \left(\forall_{t \in PB} \ \forall_{1 \leq i \leq I_t} \ \sum_{d \in DS} \ \sum_{p=1}^{P_t} V_{dtip} \right) \right]$$

Since this is a non-linear objective, a simpler method is followed to ensure the *scattering* of data structures onto different bank instances.

Data Scattering

The method of data scattering consists by first performing the placement of data structures as usual without the objective stated above. Then, once the placement is done, a post-processing of the mapping solution is performed that distributes V portions of data structures assigned to the same instances over unused bank instances. This post-processing is very fast and efficient since it has exact knowledge of the mapping solution and the available banks on the RC system. Hence the process of mapping the data can be summarized as:

1. Perform the memory mapping with the ILP formulation that uses the constraints and objective functions described so far.
2. Go through the ILP solution obtained and find all V_{dtip} variables that were set to one, and add them to a list, $PartialList$.
3. For any bank type and instance number combination, (t, i), do the following:
   (a) Find all V_{dtip} in $PartialList$ that match with both t and i, and add them to a list, $CoexistantDS$.
   (b) Sort the list $CoexistantDS$ in the decreasing order of access frequency of the data structures. Thus, the first element of the $CoexistantDS$ list refers the data structure with the most number of read/write accesses to it.
   (c) Starting by the beginning of the $CoexistantDS$ list, find an empty physical bank instance of memory type t, if any. If successful, set V_{dtip} to zero and set V_{dtj1} to one, where instance j is the empty instance that was found, and the index 1 refers to the first port of that instance.
   (d) Repeat step (c) until either all elements of the $CoexistantDS$ list are assigned to other instances or no more empty instances are found.
4. repeat step (3) until all (t, i) combinations are visited.

There are two useful remarks that can be made about this algorithm:

- First, it might be beneficial to sort the list of V_{dtip} variables set to one not only among the variables corresponding to a single (t, i) pair, but among all the variables. This will ensure that the data structures with highest access frequency have the priority of being relocated to empty bank instances.

- Step (3d) can be modified. When the number of empty physical banks is scarce, assigning one data structure to a physical bank instance can result with exiting the algorithm fairly quickly. The resulting solution might have several instances with a single data structure assigned to them but might also have several instances with a large number of data structures. Although there is no gain in reducing the addressing logic by balancing the number of data structures assigned to each bank instance, the benefit of the balancing is to reduce the net congestion on bank instances. By balancing the assignment of data structures, the density of nets reduces around each bank instance and becomes more regular, thus facilitating the task of routing.

Initial Results

Results for the improved ILP mapping for multi-configuration banks are shown in Section 8.3 since the underlying formulation is similar.

7.6 Improving the Global/Detailed Formulation

Figure 7.9 shows the complete flow of execution when the global/detailed method is used. The top part of the figure shows the pre-processing of all (d, t) pairs. As described in Section 7.5.1, the best configuration is selected for every (d, t) pair where type t allows multiple configurations.

At the end of pre-processing, the ILP problem for the memory mapping can be formulated without the added complexity of selecting configurations for each port of each instance in multi-configuration memory banks. This reduces the problem considerably, especially when the process of logic partitioning is interfaced with memory mapping.

The global mapping process can then proceed to assign each data structure to a unique physical memory bank type. This process does not map data structures to specific instances and specific ports of instances, but simply assign the data structures to types of memory banks.

The output of global mapping is an assignment of each data structure on a single physical memory bank type. The detailed mapper proceeds by assigning each data structure to specific bank instances and specific ports of the assigned bank type.

Finally, as shown in Figure 7.9, a post-processing procedure visits all instances of all types that were occupied by more than one data structure and attempts to distribute these data

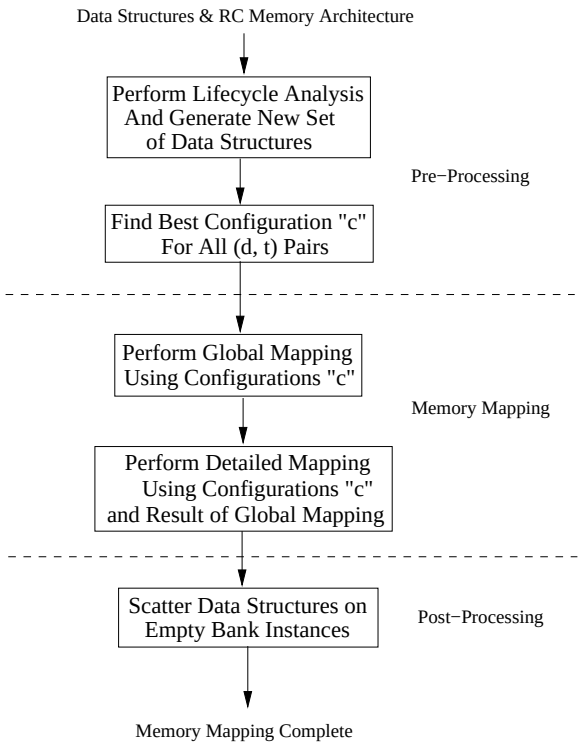


Figure 7.9: Global/Detailed Memory Mapping Methodology

structures onto bank instances that remained empty after the complete mapping. This procedure was described in Section 7.5.2.

The following sections describe in detail both the global and detailed mapping formulations adapted to the improved ILP formulations presented in Sections 7.4 and 7.5.

7.6.1 Global Memory Mapping Pre-Processing

As described in Chapter 6, global mapping only sets the Z_{dt} variables while predicting the fitness of data structures on each type of memory bank. The prediction takes into account the number of ports that are consumed by each data structure by carefully analyzing how much space each data structure will occupy (utilize efficiently and waste) on the physical instances of each memory type.

Since in this chapter, all multi-configuration instances assigned to a single data structure have the same configuration selection, the global mapping formulation is slightly different than the formulation generated in Chapter 6. This is depicted in Figure 7.10 where a 55x17 data structure is to be mapped onto one type of memory bank that has 3 ports, and four ratio configurations: 128x1, 64x2, 32x4, 16x8. Since a single configuration will be selected for this

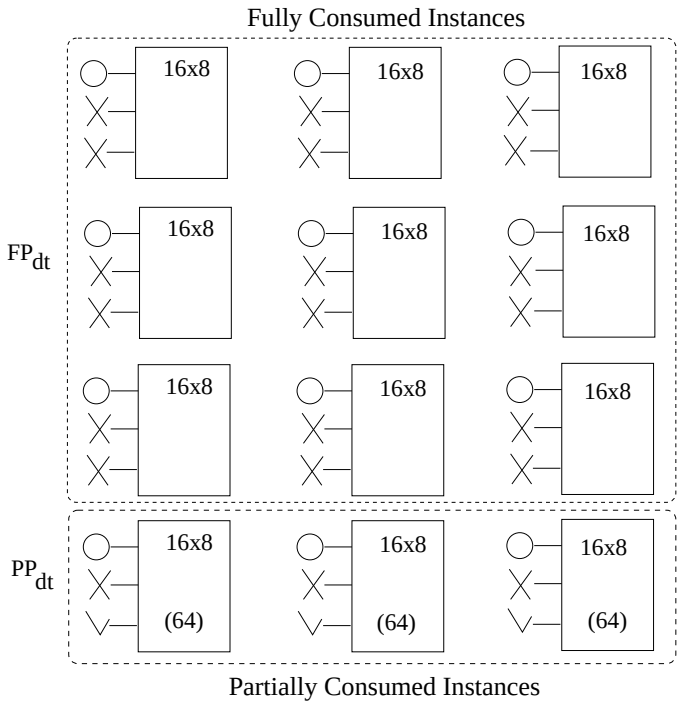


Figure 7.10: Restricted Space and Ports Allocation Example

(d, t) pair, and assuming that the configuration selected by the procedure of Section 7.5.1 is configuration four (i.e. 16x8), two main areas are identified in the figure: the area on top contains all instances that will be fully utilized by the data structure, whereas the area on the bottom contains all instances that are partially utilized. The "O" mark next to a port signifies that the port is used by this data structure. (The "X" next to a port denotes that the port is wasted. And, the check mark next to a port indicates that the port is available for other data structures. Finally, the number between parentheses next to available ports indicates the total number of bits that are still unused in the instances).

Note that the data structure will be divided in width into 8, 8, and 8 bits although the last column of instances will occupy only one of the 8 bits and waste the other 7 bits.

Based on this difference in configuration selection, for each design being mapped, the global mapper initially pre-processes some information in order to produce an ILP formulation.

Along the lines of the formulation presented in Chapter 6, three main parameters need to be computed to allow a simple yet powerful constraint formulation: CP_{dt} or the total number of consumed ports of memory type t if data structure d is assigned to it. CW_{dt} or the "ceiling" value of the width of data structure d if assigned to bank type t. And finally, CD_{dt} or the "ceiling" value of the depth of data structure d if assigned to bank type t.

First, the total number of consumed ports CP_{dt} depends on the size of data structure d with respect to the size and number of ports of bank type t. There are two components to CP_{dt} corresponding to the two areas of Figure 7.10 discussed above.

Thus, following the scheme of Figure 7.10:

$$\forall_{d \in DS},\ \forall_{t \in PB},\ CP_{dt} = FP_{dt} + PP_{dt}$$

FP_{dt} denotes the number of ports consumed by all except the last row of bank instances (fully occupied instances):

$$FP_{dt} = \left\lfloor \frac{D_d}{D_{t[b]}} \right\rfloor * \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * P_t$$

where b refers to the configuration selected by the procedure of Section 7.5.1.

PP_{dt} denotes the number of ports consumed by only the last row of bank instances (partially full instances):

$$PP_{dt} = \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * consumed\_ports\left(\left(D_d mod D_{t[b]}\right), D_{t[b]}, P_t\right)$$

consumed_ports() was defined in Figure 6.2 in Chapter 6.

The second parameter, CW_{dt}, indicates the total width that is consumed by data structure d if assigned to bank type t. After finding configuration b:

$$CW_{dt} = \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * W_{t[b]}$$

Similarly, the third parameter, CD_{dt}, indicates the total depth that is consumed by data structure d if assigned to bank type t:

$$CD_{dt} = \left\lfloor \frac{D_d}{D_{t[b]}} \right\rfloor * D_{t[b]} + \left\lceil \frac{\left(D_d\ mod\ D_{t[b]}\right) * W_{t[b]}}{max_{\forall_{m \in C_t}}\left(W_{t[m]}\right)} \right\rceil * max_{\forall_{m \in C_t}}\left(W_{t[m]}\right)$$

CW_{dt} includes the wasted bits located in the last column of memory bank instances whereas CD_{dt} includes the wasted bits due to unused memory between the end of the memory space corresponding to one port of an instance and the beginning of the memory space corresponding to another port of the same instance.

7.6.2 Global Mapping Constraints and Objective Formulations

Constraints Formulation

Similar to the formulations given in Chapter 6, the ILP formulation of the global mapper is constructed based on the following constraints:

- **Uniqueness constraints:** This constraint does not take into account the architectural details of the memory bank types, it just ensures that each data structure is mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS}, \sum_{t \in PB} Z_{dt} = 1$$

- **Port constraints:** Each memory type should have enough ports for all the data structures assigned to it. The sum of all *estimated* consumed ports of all data structures assigned to one physical bank type should be less than or equal to the total number of available ports on the bank type:

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * CP_{dt} \leq P_t * I_t$$

- **Physical Storage constraints:** Each bank type must have enough space to contain all data structures assigned to it. The product of the width and depth that each data structure would occupy on any type of physical bank should be less than or equal to the total number of bits available on the bank type:

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * (CW_{dt} * CD_{dt}) \leq I_t * W_{t[1]} * D_{t[1]}$$

Objective Formulation

The objective is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost components are:

- **Latency cost:** This cost component computes the total number of clock cycles required for accessing all data structures. It is based on the profiling information given in the problem specification:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * [RA_d * RL_t + WA_d * WL_t]$$

- **Pin delay cost:** In addition to the number of clock cycles required to accessing the data in the design, the effect of the memory mapping on the clock cycle speed is captured in the pin delay constraint. Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed, then:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (RA_d + WA_d) * T_t$$

- **Pin I/O cost:** The last cost component tries to measure the number of interconnect wires that is required between processing elements and memory banks. The number of wires is the sum of the *data* wires, the *address* wires, and any memory control wires such as the read/write enable. Since memory control wires are unavoidable, the pin I/O cost takes into account only the *data* and *address* lines. The larger the depth and width of a data structure the more pins it will need in the event of off-chip physical banks. Also, the further away memory banks are from the processing element, the higher the cost of pin I/O. Thus, the pin I/O cost is formulated as:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (\lceil log_2(CD_{dt}) \rceil + CW_{dt}) * T_t$$

  Note that CW_{dt} can be replaced by W_d if the synthesis process allows a subset of pins to be connected to a memory bank.

Once the ILP formulation is solved, scattering of data can occur as described in Section 7.5.2. Finally, the output of the global mapper associates each data structure to a single memory bank type and passes this information along to the detailed mapper.

7.6.3 Detailed Memory Mapping Pre-Processing

Detailed memory mapping takes the output of the global memory mapping, as seen in Figure 7.9, and for each physical memory bank type, performs a detailed mapping of all data structures that were assigned to this memory type. The resulting detailed mapping procedure is depicted in Figure 7.11.

It should be stressed that since global memory mapping ensures a successful detailed assignment, the task of the detailed mapper is simplified. It cannot further optimize the assignment based on the optimization criteria established in global mapping. Instead, it can aim at optimizing the detailed assignment based on the different detailed constraints described below.

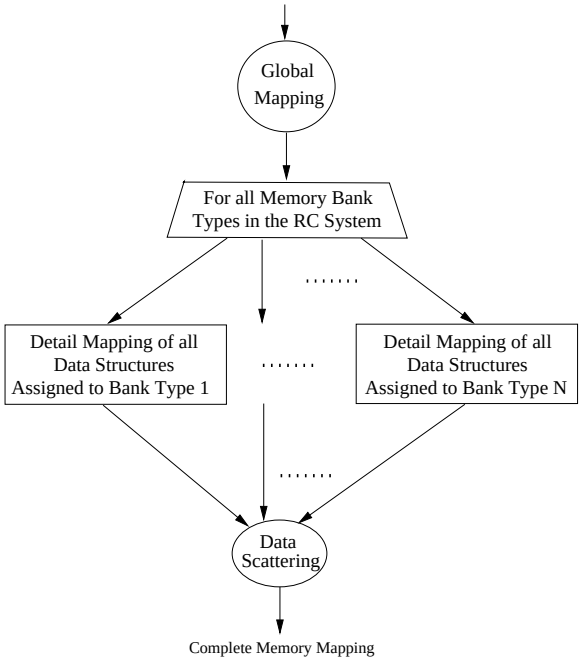


Figure 7.11: Detailed Memory Mapping

There are as many detailed mapping ILP problems as there are physical bank types. The problem sizes are reduced since the t index previously part of the ILP variables X_{dti} and V_{dtip} is no longer required. Furthermore, the number of problems is typically less than ten or fifteen.

The pre-processing required in detailed mapping consists of going through each physical bank type, grouping in a list all the data structures that are assigned to it and passing the list along with information about the physical bank type (described extensively in Section 2.4 to the detailed mapping ILP model described below.

7.6.4 Detailed Mapping Constraints and Objective Formulations

In the constraints and objective formulations of detailed mapping, the memory bank type index, t, is dropped from the ILP variables since it is a constant for each run of the mapper. In addition, lifecycle conflicts are resolved as described in Section 7.4.2, before the ILP problem is formulated.

Constraints Formulation

The following are the important constraints in the detailed memory mapping problem catered to single and multi-configuration physical memory banks:

- **Capacity constraints:** Each data structure should be fully stored on the hardware. For that, the number of X and V variables is computed:

$$\forall_{d \in DS} \sum_{i=1}^{I_t} X_{di} = \left\lfloor \frac{D_d}{D_{t[b]}} \right\rfloor * \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil$$

  where the floor operator is used to ignore the last row of memory banks.

  For the V variables:

$$\forall_{d \in DS} \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} V_{dip} = \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * \begin{cases} 1 & \text{if } \left(D_d \; mod \; D_{t[b]}\right) \neq 0 \\ 0 & \text{if } \left(D_d \; mod \; D_{t[b]}\right) = 0 \end{cases}$$

  Note that if there is no remainder from the depth ratio, there is no need for V variables. This can be known before formulating the problem, thus the last part of the equation will not be processed by the ILP solver. This also applies to the X variables although the formula does not have any "if" statements: when the data structure depth is less than the physical bank depth, there will be no X variables (since the floor() function would be zero).

- **Port constraints:** Each X variable will occupy the entire bank, thus every instance will have at most one X variable assigned to it:

$$\forall_{1 \leq i \leq I_t} : \sum_{d \in DS} X_{di} \leq 1$$

  It will be seen in the physical storage constraints below that this constraint is redundant.

  Also, each V variable will occupy one port of physical bank, so the number of V variables assigned to one bank instance must be less than or equal to the number of available ports:

$$\forall_{1 \leq i \leq I_t} \forall_{1 \leq p \leq P_t} : \sum_{d \in DS} V_{dip} \leq 1$$

- **Physical storage constraints:** Depending on the configurability feature of the physical memory bank type, the sum of the storage space mapped to every port must fit

within the bank:

$$\forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t}$$

$$X_{di} * \left(D_{t[1]} * W_{t[1]}\right) + V_{dip} * \left\lceil \frac{\left(D_d \ mod \ D_{t[b]}\right) * W_{t[b]}}{max_{\forall_{m \in C_t}} \left(W_{t[m]}\right)} \right\rceil * max_{\forall_{m \in C_t}} \left(W_{t[m]}\right) \leq D_{t[1]} * W_{t[1]}$$

Again, this equation is based on the assumption of Equation 2.1. On the other hand, if bank type t has a single configuration, then the physical storage constraints equation is replaced with:

$$\forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t} X_{di} * D_t + V_{dip} * (D_d \ mod \ D_t) \leq D_t$$

Objective Formulation

The objective function can be avoided altogether since all three cost components of the objective function discussed so far rely on the Z_{dt} variables. Since the Z_{dt} variables are already set during detailed mapping, the objective function reduces to a constant, thus, a non-optimizable goal. The detailed mapper can be formulated with no objective function, just to perform a feasible assignment of data structures to specific ports of specific instances. Since there are no objective functions, other techniques can be used to perform the detailed mapping, but the ILP formulation is chosen here since it allows very simple addition of cost components such as guiding the placement of data structures on physical bank instances with respect to placement and routing that occurs in the later stages of the synthesis process.

Preliminary Results

Results for the improved ILP global/detailed memory mapping are provided in Section 8.4 since the underlying formulation is similar.

7.7 Conclusion

This chapter presented the general spatial partitioning methodology. After reviewing the partitioning techniques available in the literature, three different views of spatial partitioning were described. A foundation to ILP formulations for the three views was established by re-visiting and improving upon the formulations presented in Chapters 4, 5, and 6. The

improvements provided faster mapping execution while retaining a high quality and accuracy of mapping.

Chapter 8

ILP Solutions for Spatial Partitioning

8.1 Introduction

This chapter presents solutions to the problem of spatial partitioning where both the tasks of data mapping and logic mapping are considered. In previous chapters, it was assumed that all logic representing the design are placed in a single FPGA or processing element, and physical memory banks were scattered around and inside that processing element. Since all logic was assigned to one processing element, the task of the memory mapper was to optimize the placement of the design's data structures with respect to the processing element.

As depicted in Figure 7.2, there are three ways that a memory mapper can interface with logic partitioning. This chapter provides solutions for each one of the three views by using, as a foundation, the formulations of Sections 7.4, 7.5, and 7.6.

In the case of a single processing element, memory mapping consists of placing the design's data structures in the physical memory banks of the RC system without considering the logic (or computational tasks) of the design. Since all computational tasks exist on a single processing element, the cost of accessing a data structure is the same for all computational tasks. This is not the case when multiple processing elements are available on the RC system; computational tasks can be assigned to different processing elements, thus making the cost of accessing a data structure different. For example, if both computational tasks 1 and 2 access data structure A, and the logic partitioning places task 1 on FPGA1, task 2 on FPGA2, and the memory mapper places data structure A in the on-chip RAMs of FPGA1, then the cost of accessing data structure A is smaller for task 1 than it is for task 2. The problem definition and ILP formulation need to be altered to accommodate the presence of multiple processing elements and the knowledge of access patterns between computational tasks and

data structures in a design.

Section 8.2 defines new variables and parameters used in this chapter. The following three sections, Section 8.3, Section 8.4, and Section 8.5, model the three spatial partitioning problems of Figure 7.2. First, the multi-configuration formulation presented in Section 7.5 is extended to support RC systems with multiple processing elements. Second, the global/detailed technique of Section 7.6 is adapted to handle multiple processing elements. And third, a new ILP model is presented to include both the tasks of logic partitioning as well as memory mapping. Finally, Section 8.6 concludes the chapter.

8.2 Definitions

The definitions presented in Section 2.4 need to include the information about multiple processing elements on the RC system. Appendix A provides a description summary of all variables and parameters used in the thesis. The extensions required are the following:

- A list of L computational tasks or logic tasks in the design:

  $$CT = \{CT_1, CT_2, ..., CT_L\}$$

  For each computational task l, $PE[l]$ denotes the processing element that is assigned to task l.

- For each physical memory bank type t, T_t was defined as the number of pins traversed from *the* processing unit to a bank of type t. Since multiple processing units exist in the current model, T_t is re-defined as an array:

  $$T_t = \{T_{t1}, T_{t2}, ..., T_{tF}\}$$

  where F is the number of processing elements available in the RC system, and T_{tf} denotes the number of traversed pins from processing element f to a bank of type t. This array provides complete knowledge of the physical memories proximity to every single processing element in the RC system.

- For each data structure d, RA_d was defined as the total number of estimated reads by *the* processing unit from data structure d; and WA_d was defined as the total number of estimated writes by *the* processing unit to data structure d. Again, since multiple processing elements exist and computational tasks can be mapped to them arbitrarily,

knowledge of the frequency of access from each processing element to a data structure is needed. For this reason, RA_d and WA_d are re-defined as arrays:

$$RA_d = \{RA_{d1}, RA_{d2}, ..., RA_{dL}\}$$

and:

$$WA_d = \{WA_{d1}, WA_{d2}, ..., WA_{dL}\}$$

where L is the number of computational tasks in the design, RA_{dl} and WA_{dl} denote the number of read and write accesses respectively that computational task l performs on data structure d. These two arrays provide complete knowledge of the access frequency of all computational tasks to all data structures in the design.

In addition to the above definitions, it should be noted that the lifecycle conflicts analysis presented in Section 7.4.2 still applies to the general spatial partitioning problem (all three views of Figure 7.2); though, one new complexity is introduced in the event of multiple processing elements. When a single processing element exists, the equivalent data structure, replacing a set of data structures, is equidistant from all data structures in the set. This is not valid for RC boards with multiple processing elements. Since computational tasks accessing the set of data structures can be distributed on different processing elements of the RC system, the equivalent data structure can be closer to some processing elements than to others. As a solution to this problem, a post-processing mechanism can easily *replicate* equivalent data structures, if memory space permits, in order to minimize the number of interconnect wires and the latency penalty. This mechanism is shown in Figure 8.1.

8.3 Logic Partitioning with Memory Post-Partitioning

Based on the information introduced in the previous section, the ILP problem can be specified. Hence, the following are the constraints and objective formulation for the logic partitioning with memory post-partitioning view depicted in Figure 8.2.

Note that once the logic partitioning completes execution, the memory mapper has the logic mapping information. In other words, $PE[l]$ – the processing element that is assigned to logic task l – is available for all computational tasks. This information is obtained from the *taskgraph* described thoroughly in Chapter 10.

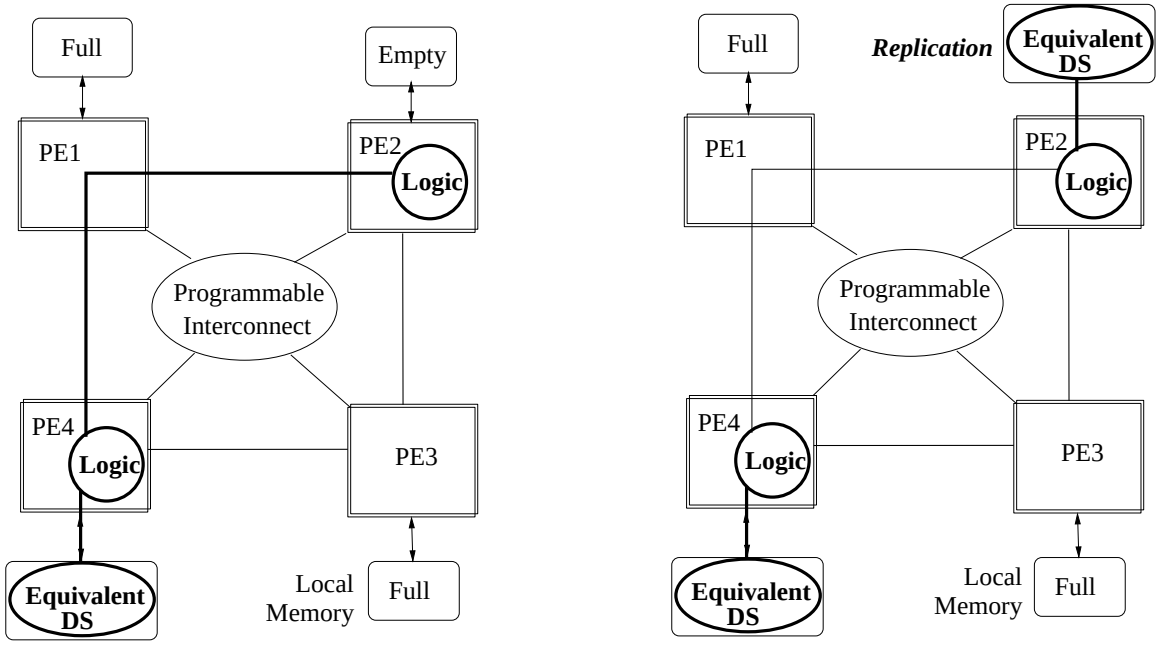


Figure 8.1: Lifecycle Conflicts Analysis in RCs with Multiple Processing Units

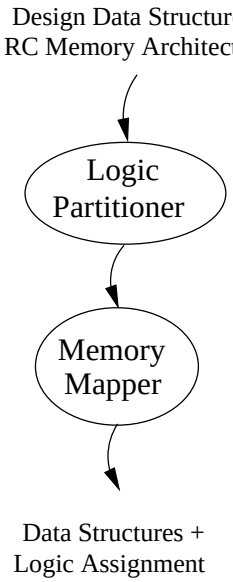


Figure 8.2: Post-Partitioning Memory Mapping

8.3.1 Constraints Formulation

Since the constraints only target the presence and fitness of the data structures onto the physical memory banks available on the RC system, the constraints formulation for the post-partitioning case is similar to the one presented in Section 7.4.1 except that the D_t and W_t variables are replaced with $D_{t[b]}$ and $W_{t[b]}$ respectively to take into account the multiple configurations. Furthermore, the constraints include the changes introduced in Section 7.5.2. Thus, the complete ILP constraints are:

- **Uniqueness constraints:** Irrespective of the number of processing elements available on the RC system, each data structure should be mapped to exactly one type of physical bank:

$$\forall_{d \in DS} \sum_{t \in PB} Z_{dt} = 1$$

- **Capacity constraints:** Similarly, irrespective of the number of processing elements, each data structure should be fully stored on the hardware:

$$\forall_{d \in DS}\ \forall_{t \in PB}\ \sum_{i=1}^{I_t} X_{dti} = \left\lfloor \frac{D_d}{D_{t[b]}} \right\rfloor * \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * Z_{dt}$$

  where the b index refers to the best configuration that was found for the (d, t) pair by the procedure of Section 7.5.1.

  For the V variables:

$$\forall_{d \in DS}\ \forall_{t \in PB}\ \sum_{i=1}^{I_t} \sum_{p=1}^{P_t} V_{dtip} = \left\lceil \frac{W_d}{W_{t[b]}} \right\rceil * Z_{dt} * \begin{cases} 1 & \text{if } \left(D_d \ mod\ D_{t[b]}\right) \neq 0 \\ 0 & \text{if } \left(D_d \ mod\ D_{t[b]}\right) = 0 \end{cases}$$

- **Port constraints:** There are no port constraints on the X variables since the physical storage constraints presented below ensures that at most one X variable is set to one for any instance of any bank type. However, each V variable will occupy one port of a physical bank, so the number of V variables assigned to one bank instance must be less than or equal to the number of available ports:

$$\forall_{t \in PB}\ \forall_{1 \leq i \leq I_t}\ \forall_{1 \leq p \leq P_t} : \sum_{d \in DS} V_{dtip} \leq 1$$

- **Physical storage constraints:** Depending on the configurability feature of the physical memory bank type, the sum of the storage space mapped to every port must fit

within the bank:

$$\forall_{t \in PB} \; \forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t}$$

$$X_{dti} * \left(D_{t[1]} * W_{t[1]}\right) + V_{dtip} * \left\lceil \frac{\left(D_d \; mod \; D_{t[b]}\right) * W_{t[b]}}{max_{\forall_{m \in C_t}} \left(W_{t[m]}\right)} \right\rceil * max_{\forall_{m \in C_t}} \left(W_{t[m]}\right) \leq D_{t[1]} * W_{t[1]}$$

assuming Equation 2.1 holds. In other words,

$$\forall_i, \; , \forall_j, \; 1 \leq i, j \leq C_t : \; D_{t[i]} * W_{t[i]} = D_{t[j]} * W_{t[j]}$$

If bank type t has a single configuration, then the physical storage constraints equation is replaced with:

$$\forall_{t \in PB} \; \forall_{1 \leq i \leq I_t} \sum_{d \in DS} \sum_{p=1}^{P_t} X_{dti} * D_{t[b]} + V_{dtip} * \left(D_d \; mod \; D_{t[b]}\right) \leq D_{t[b]}$$

In the case of the multi-configuration constraint stated above, the equation ensures that the total number of bits assigned to an instance is less than or equal to the capacity of the bank. If any X variable is set to one, then all other variables corresponding to this bank (both X and V variables) should be set to zero. However, for the V variables, the equation rounds the parts of data structures that are smaller than the instance to the closest multiple of the largest width configuration as described in criteria #4 of Section 7.5.1.

8.3.2 Objective Formulation

Constraints were not affected by the multiple processing unit complexity since they solely ensure the placement of data structures on the physical memory banks without worrying about the number and location of processing elements in the system. However, the location of processing elements with respect to the physical memory banks as well as the placement of computational tasks in the available processing elements make the objective formulation different.

The objective of the ILP model is to still optimize the performance and minimize the interconnection cost of the memory assignment. Taking into account the multiple processing elements and the definitions stated above, the different cost components of the objective function become:

- **Latency cost:** Assuming that RA_{dl} and WA_{dl} provide a good estimation on the

number of reads and writes that computational task l performs on data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} Z_{dt} * [RA_{dl} * RL_t + WA_{dl} * WL_t]$$

- **Pin delay cost:** This cost component depends on the placement of the computational tasks, thus $PE[l]$ is used to find the processing element that was assigned to computational task l. Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} Z_{dt} * (RA_{dl} + WA_{dl}) * T_{t[PE[l]]}$$

  where $T_{t[PE[l]]}$ refers to the number of pins traversed from the processing element on which computational task l was placed, to physical memory bank type t.

- **Pin I/O cost:** Finally, the pin I/O cost component is modified to include interconnects between memory type t that data structure d is assigned to and all processing elements $PE[l]$ that have been assigned all computational tasks l that access the data structure. The larger the width of a data structure, the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} Z_{dt} * W_d * T_{t[PE[l]]} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases}$$

  The value $(RA_{dl} + WA_{dl})$ is the total number of times (reads and writes) that data structure d is accessed by computational task l. If this value is zero, then the logic task does *not* access the data structure, hence no need to introduce interconnect wires between the logic and the data segment. However, if the value is any number greater than zero, then the logic task *does* access the data structure, thus requiring interconnect wires to connect the physical memory bank where the data structure is mapped to the processing element where the logic task is placed.

8.3.3 Results

The results for the improved multi-configuration ILP memory mapping technique are shown in Table 8.1. The number of data structures, the overall total number of physical bank instances and ports, and total number of configurations as a product of all possible configurations, are shown in the table. In addition, a comparison in ILP solving speed and value

Data Structures	Physical Memories			Execution Time (secs)		Improvement (%)	
# data structures	Total # banks	Total # ports	Total # configs	New ILP	Old ILP	Solution Speed	Objective Value
22	13	25	$\sim 10^6$	5.2	8.1	36	-2.3
32	23	45	$\sim 10^{13}$	9.0	29.4	69	-2.7
32	45	77	$\sim 10^{20}$	15.3	99.3	85	2.0
42	45	77	$\sim 10^{20}$	16.5	130.4	87	-5.6
32	65	105	$\sim 10^{20}$	20.7	172.7	88	-3.6
62	65	105	$\sim 10^{20}$	52.8	411.0	87	-5.4
32	180	265	$\sim 3 * 10^{52}$	69.3	518.3	87	-2.3
62	180	265	$\sim 3 * 10^{52}$	140.5	1225.0	89	-3.2
132	180	265	$\sim 3 * 10^{52}$	531.7	2989.0	82	-4.9

Table 8.1: ILP Results for Logic Partitioning with Memory Post-Partitioning

of the objective functions between the improved formulation and the ILP formulation discussed in Chapter 5 is included. Since in the post-partitioning case, knowledge of logic task assignment to processing elements is known before memory mapping, the logic partitioning does not add any complexity to the memory mapping problem, hence the validity of the comparison.

The efficiency of the multi-configuration analysis approach presented earlier is readily seen. At a price of 3.6% deterioration in the value of the objective function obtained, the execution speed is decreased dramatically: an average decrease of 79% in the ILP solution execution time. This allows easy expansion of the ILP formulation to include different constraints pertaining to the presence of multiple processing elements on the RC platform.

8.4 Logic and Memory Partitioning Interaction

Figure 8.3 shows the second view of spatial partitioning. While the logic partitioner is executing, it invokes the memory mapper to ensure that a feasible mapping of the data structures exists.

A spatial partitioner based on the view of Figure 8.3 is presented in [95]. Every time the memory mapper is invoked, a feasible solution is generated and feedback is returned to the logic partitioner. Components of the feedback include:

- First, feedback on whether a feasible memory mapping solution was achieved. If a

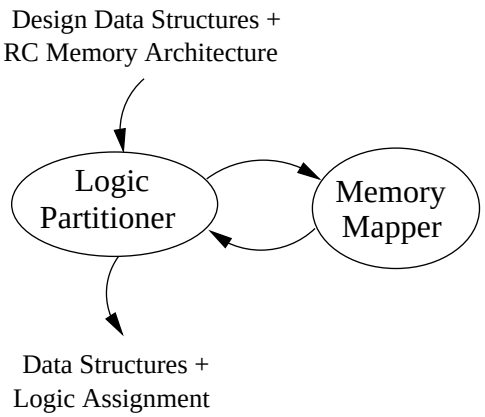


Figure 8.3: Logic Partitioning with Memory Mapping Interaction

feasible solution is not possible, the mapper can provide clues on where it failed. In other words, the mapper indicates which data structures did not fit on the RC system's memory banks.

- Second, the mapper returns the extra logic required for all the addressing logic. As discussed earlier in the chapter, addressing logic consisting of multiplexors and simple logic gates are needed for address offsetting as well as for the decoding of address lines.
- Third, in the event arbitration is allowed (refer to Chapter 9), the mapper returns the location and size of the arbiters introduced along with any extra logic required to multiplex/demultiplex common address and data lines.

When the memory mapper is invoked, knowledge of the computational tasks placement is available and the mapping problem is similar to the one presented in the global/detailed approach in Section 7.6. Since the memory mapper is typically involved several times in this scenario, it is therefore required to have fast memory mapping so that the overall partitioning can explore more solutions in less amount of time. Global memory mapping (without its detailed mapping counterpart) fits this role perfectly. Since, for all but the last call of the memory mapper, only estimates are required, global mapping can be used to provide the estimates without going through the detailed mapping process. Only when the logic partitioner is satisfied with its placement can it invoke the detailed memory mapper.

In summary, by using the global/detailed memory mapping technique, the new partitioning paradigm is depicted in Figure 8.4.

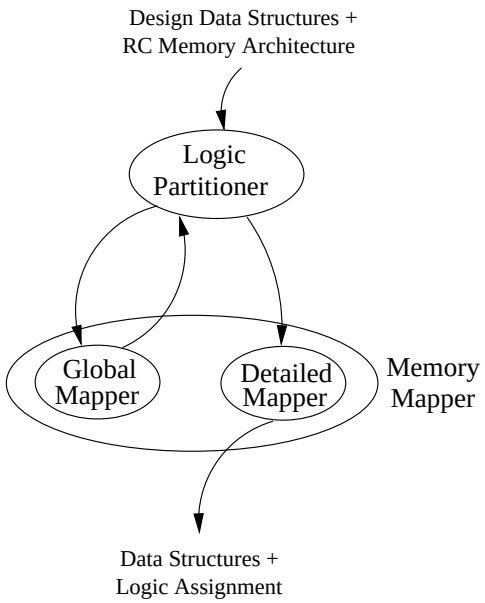


Figure 8.4: Global/Detailed Memory Mapping with Logic Partitioning Interaction

8.4.1 Constraints Formulation

- *Constraints for global mapping:* The ILP formulation of the global mapping constraints is similar to the formulation provided in Section 7.6.2 since the knowledge of the number of processing units is ignored in the constraints. Note that the goal of the current ILP solution is to *minimize* the number of overall interconnect wires without having rigid constraints on the number of wires available between processing elements and memory banks or other processing elements. The task of routing is omitted from this work and it is assumed that a separate routing tool performs the routing feasibility and assignment.

- *Constraints for detailed mapping:* The ILP formulation of the detailed mapping constraints is also similar to the formulation provided in Section 7.6.4.

8.4.2 Objective Formulation

- *Objective for global mapping:* The objective formulation in Section 7.6.2 needs to be extended to handle multiple processing elements. This leads to the constraints stated in Section 8.3.

- *Objective for detailed mapping:* Since the global mapping generated an optimized placement of the data structures on the physical memory bank types, the objective of de-

Data Structures	Physical Memories		Execution Time (secs)		Improvement (%)	
# data structures	Total # banks	Total # ports	Interaction ILP	Post ILP	Solution Speed	Objective Value
22	13	25	6.1	5.2	-17	-0.7
32	23	45	9.3	9.0	-3	0.0
32	45	77	13.8	15.3	10	-1.7
42	45	77	15.9	16.5	4	-2.7
32	65	105	17.9	20.7	14	-0.8
62	65	105	27.7	52.8	47.5	-4.1
32	180	265	33.3	69.3	52	-0.9
62	180	265	42.4	140.5	70	-2.3
132	180	265	206.8	531.7	61	-3.8

Table 8.2: Performance Results of Logic and Memory Partitioning Interaction

tailed mapping can be ignored in order to generate any detailed mapping solution. Other objective components can be introduced as mentioned in Section 7.6.4.

8.4.3 Results

Table 8.2 shows execution speeds and improvements obtained in the objective function values for both the logic and memory logic partitioning interaction and the logic partitioning with memory post-partitioning. Column 4 shows the execution speed of the ILP solver when the problem is stated in the formulation presented in this section. Column 5 shows the execution speed of the ILP solver when the problem is stated in the formulation presented in Section 8.3. Columns 6 and 7 show the improvement (or loss) in both the ILP convergence speed and the final value of the objective function.

As expected, it can be seen from the results that as the problem size increases, the partitioning interaction approach does not explode since the mapping problem is divided into two serialized problems: global and detailed mapping. This is achieved at a loss of optimality in the resulting mapping, hence the negative numbers in the objective function improvement. However, this loss is within 4.1%, justifying the gain obtained in the fast ILP mapping solution. The plot shown in Figure 8.5 is a visual rendering of the execution speed for both approaches. The lines connecting the data points are used for the sake of clarity; they do not suggest the predictability of the mapping solutions between the test cases that were used.

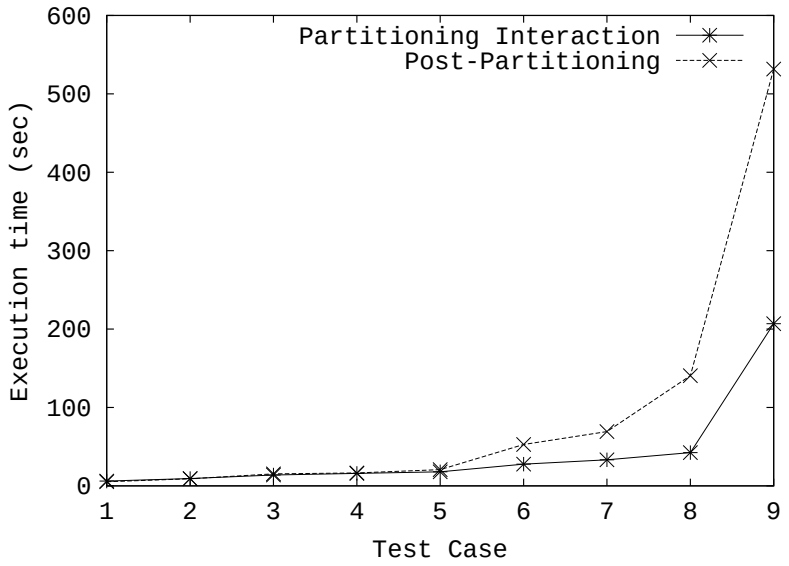


Figure 8.5: ILP Execution Times for Logic and Memory Partitioning Interaction

8.5 Merged Logic and Memory Spatial Partitioning

Figure 8.6 shows the third spatial paradigm where both the tasks of logic partitioning and memory mapping are combined into one formulation. This section describes the ILP formulation to solve this problem. In this scenario, the spatial partitioner not only assigns data structures to physical memory banks but assigns computational tasks to processing elements as well. A new variable is introduced for this purpose along with a few required constants.

First, a new set of variables is introduced in the ILP problem that assigns a computational or logic task to a single processing element in the RC system. It is assumed that a logic task must fit on at least one processing element and that logic tasks are considered as atomic units of placement: A logic task cannot be divided or mapped onto two or more processing elements.

The set of variables S_{lf} is introduced to perform the logic partitioning. It associates a logic task l with a processing unit f:

$$S_{lf} = \begin{cases} 1 & \text{if computational task } l \text{ is assigned to processing element } f. \\ 0 & \text{otherwise.} \end{cases}$$

Knowledge of the estimated area of each logic task is required for performing the partitioning. The paramter $E(l)$ returns the estimated area of logic task l. Typically, the area is measured in terms of configurable logic blocks, function generators, or lookup tables.

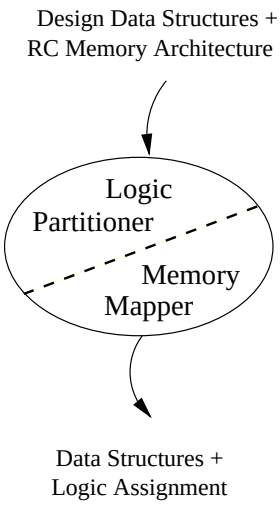


Figure 8.6: Combined Logic and Memory Partitioning

Finally, knowledge of the size of each processing element in the RC system is also required. The parameter $C(f)$ returns the total area of processing element f in the RC system.

In the combined logic and memory partitioner, the global/detailed approach is used to perform the mapping. This approach leads to the least complex formulation while achieving a very high quality of placement. Once global mapping is performed, both data structures and computational tasks are mapped to memory banks and processing units respectively. The detailed mapping then assigns the data structures to exact instances and exact ports of instances.

The detailed memory mapping phase followed in this section is similar to the one described in Section 7.6.4 and for the sake of brevity, it is not reproduced here. However, the global memory mapping phase is altered to include the S_{lf} variables in the formulations and optimize the partitioning with both the logic tasks and data segments. The following is a description of the main constraints and objective formulation of the combined logic and memory partitioner.

8.5.1 Constraints Formulation

Two categories of constraints exist in this partitioning problem: the first deals with finding a target processing unit for each computational task in the design, while the second assigns each data structure in the design to a physical memory bank. No interaction exists between these two categories, hence the absence of constraints that combine S variables with Z variables. On the other hand, the objective function optimizes the relative placement of logic tasks

with respect to data structures. Thus, the objective formulation includes components that combine the set of S variables with the set of Z variables.

The first two constraints assign computational tasks to processing elements while the remaining constraints map data structures to memory banks same as the formulations given in Section 7.6.2:

- **Computational Tasks Uniqueness Constraints:** Each logic task l should be mapped to exactly one processing unit f:

$$\forall_{1 \leq l \leq L}, \sum_{f=1}^{F} S_{lf} = 1$$

- **Computational Tasks Physical Storage Constraints:** The area obtained by the sum of all logic tasks assigned to processing element f must be less than or equal to the area of processing element f:

$$\forall_{1 \leq f \leq F} \sum_{l=1}^{L} S_{lf} * E(l) \leq K * C(f)$$

$$0 < K \leq 1$$

  As defined above, $E(l)$ is the estimated area of task l, $C(f)$ is the total available area of processing element f, and K is a constant reduction factor. Besides the computational tasks, extra logic for arbitration and addressing logic might be assigned to processing element l. The area associated with the extra logic is ensured by the reduction factor K. Furthermore, K also ensures that the processing elements will not be tightly packed since typical low-level routing tools fail in that event.

- **Data Structures Uniqueness Constraints:** This constraint does not take into account the architectural details of the memory bank types, it just ensures that each data structure is mapped to exactly one *type* of physical bank:

$$\forall_{d \in DS}, \sum_{t \in PB} Z_{dt} = 1$$

- **Port Constraints:** Each memory type should have enough ports for all the data structures assigned to it. The sum of all *estimated* consumed ports of all data structures assigned to one physical bank type should be less than or equal to the total number of

available ports on the bank type:

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * CP_{dt} \leq P_t * I_t$$

- **Physical Storage Constraints:** Each bank type must have enough space to contain all data structures assigned to it. The product of the width and depth that each data structure would occupy on any type of physical bank should be less than or equal to the total number of bits available on the bank type:

$$\forall_{t \in PB}, \sum_{d \in DS} Z_{dt} * (CW_{dt} * CD_{dt}) \leq I_t * W_{t[1]} * D_{t[1]}$$

8.5.2 Objective Formulation

The constraints formulation of the global mapping phase did not combine X and S variables, however the objective is affected by the multiple processing unit's complexity since all costs related to the RC architecture (namely, pin delay and pin I/O) take into account the relative positioning of logic tasks with respect to data structures. The following lists the three main cost components of the global mapping objective formulation:

- **Latency cost:** Assuming that RA_{dl} and WA_{dl} provide a good estimation on the number of reads and writes that computational task l performs on data segment d:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} Z_{dt} * [RA_{dl} * RL_t + WA_{dl} * WL_t]$$

  This cost component computes the number of clock cycles required for all data read and writes. Since the placement of computational tasks does not affect it, the set of S variables is not present in the formulation.

- **Pin delay cost:** This cost component depends on the placement of the computational tasks, thus S variables are used in conjunction with the Z variables to find the relative placement of data structures and the processing units assignments of their corresponding accessing logic tasks. Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed, the pin delay cost is formulated as:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} \sum_{f=1}^{F} Z_{dt} * (RA_{dl} + WA_{dl}) * S_{lf} * T_{tf}$$

As described above, T_{tf} refers to the number of pins traversed from processing element f to physical memory bank type t. S_{lf} is used to select the correct number of pins traversed based on the placement of logic task l on a processing element.

The above cost formulation is non-linear since it contains $Z_{dt} * S_{lf}$ terms. Furthermore, the two variables in the non-linear terms have different indices so for each (d, t) term, there are: $\sum_{l=1}^{L} \sum_{f=1}^{F} S_{lf}$ terms. To linearize such product terms, a new set of variables along with new constraints are added to the problem.

As an illustrative example, if two variables X and Y are present in the objective function:

$$minimize\,(X * Y)$$

Then, a new positive *integer* variable Z is introduced such that:

$$Z = X * Y$$

The new objective becomes:

$$minimize\,(Z)$$

Subject to new constraints:

$$Z \leq X * max(Y)$$

$$Z \leq Y$$

$$Z \geq (X - 1) * max(Y) + Y$$

It can be seen from the above constraints that when X is zero, Z is constrained by a zero upper-bound and the Y variable is unconstrained. On the other hand, when X is one, Z has Y as both upper-bound and lower-bound. Thus, in the case of X being one, minimizing Z is equivalent to minimizing Y.

To linearize the global mapping cost term using this technique, two equivalent formulations exist:

1. *Linearization with respect to dt:* The equivalent linear representation replaces the objective cost presented above by a new cost based on a new set of integer variables, M_{dt}:

$$M_{dt} = Z_{dt} * \sum_{l=1}^{L} \sum_{f=1}^{F} (RA_{dl} + WA_{dl}) * S_{lf} * T_{tf}$$

The new objective cost is:

$$\sum_{d \in DS} \sum_{t \in PB} M_{dt}$$

Besides M_{dt} being a positive integer (i.e. ≥ 0), it is subject to three constraints. First, when Z_{dt} is zero, M_{dt} should be set to zero; and when Z_{dt} is one, there should not be any constraints on M_{dt}:

$$\forall_{d \in DS} \, \forall_{t \in PB} \; M_{dt} \leq Z_{dt} * \kappa_{dt}$$

In this constraint, although M_{dt} can be unconstrained in the event Z_{dt} is one (i.e. $\kappa \gg 0$), κ is usually selected to be an upper-bound above which M_{dt} will never reach. This upper-bound speeds the ILP convergence by providing a tighter range on the set of M_{dt} variables. In this case, κ can be selected during the pre-processing stage as:

$$\kappa_{dt} = \sum_{l=1}^{L} \; (RA_{dl} + WA_{dl}) * max_{\forall_{1 \leq f \leq F}}(T_{tf})$$

Second, when Z_{dt} is set to one, the constraint on the upper-bound of M_{dt} is:

$$\forall_{d \in DS} \, \forall_{t \in PB} \; M_{dt} \leq \sum_{l=1}^{L} \; \sum_{f=1}^{F} \; (RA_{dl} + WA_{dl}) * S_{lf} * T_{tf}$$

Finally, when Z_{dt} is set to one, the constraint on the lower-bound of M_{dt} is:

$$\forall_{d \in DS} \, \forall_{t \in PB} \; M_{dt} \geq (Z_{dt} - 1) * \kappa_{dt} + \sum_{l=1}^{L} \; \sum_{f=1}^{F} \; (RA_{dl} + WA_{dl}) * S_{lf} * T_{tf}$$

The last two constraints ensure that, in the event Z_{dt} is set to one, minimizing M_{dt} is equivalent to minimizing the other term in the non-linear term.

2. *Linearization with respect to lf:* A second way of linearizing the pin delay cost component is with respect to the lf indices. A new set of positive integer variables, N_{lf}, is introduced:

$$N_{lf} = S_{lf} * \sum_{d \in DS} \; \left[(RA_{dl} + WA_{dl}) * \sum_{t \in PB} \; Z_{dt} * T_{tf} \right]$$

The new objective component is:

$$\sum_{l=1}^{L} \sum_{f=1}^{F} N_{lf}$$

Again, besides N_{lf} being a positive integer (i.e. ≥ 0), it is subject to three constraints. First, when S_{lf} is zero, N_{lf} should be set to zero; and when S_{lf} is one, there should not be any constraints on N_{lf}:

$$\forall_{1 \leq l \leq L} \, \forall_{1 \leq f \leq F} \, N_{lf} \leq S_{lf} * \sigma_{lf}$$

In this case, the upper-bound σ can be selected during the pre-processing stage as:

$$\sigma_{lf} = \sum_{d \in DS} (RA_{dl} + WA_{dl}) * max_{\forall_{t \in PB}}(T_{tf})$$

Second, when S_{lf} is set to one, the constraint on the upper-bound of N_{lf} is:

$$\forall_{1 \leq l \leq L} \, \forall_{1 \leq f \leq F} \, N_{dt} \leq \sum_{d \in DS} \left[(RA_{dl} + WA_{dl}) * \sum_{t \in PB} S_{lf} * T_{tf} \right]$$

Finally, when S_{lf} is set to one, the constraint on the lower-bound of N_{lf} is:

$$\forall_{1 \leq l \leq L} \, \forall_{1 \leq f \leq F} \, N_{lf} \geq (S_{lf} - 1) * \sigma_{lf} + \sum_{d \in DS} \left[(RA_{dl} + WA_{dl}) * \sum_{t \in PB} S_{lf} * T_{tf} \right]$$

The last two constraints ensure that, in the event S_{lf} is set to one, minimizing N_{lf} is equivalent to minimizing the other term in the non-linear term.

- **Pin I/O cost:** Finally, the pin I/O cost component also depends on the placement of the computational tasks, thus Z variables are combined with S variables to formulate the pin I/O cost component. The larger the width of a data structure the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} \sum_{l=1}^{L} \sum_{f=1}^{F} Z_{dt} * W_d * S_{lf} * T_{tf} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases}$$

Again, this cost component includes non-linear terms $Z_{dt} * S_{lf}$. The linearization technique presented above is used for obtaining a formulation suitable for an ILP solver. Similar to the pin delay cost component, two formulations exist to linearize the pin I/O cost component of the spatial partitioning global mapping problem. Only the first of the two formulations is shown below, the second transformation being similar

to the second formulation of the pin delay cost component. The next section provides insight on which formulation to select for both pin delay and pin I/O cost components.

Linearization with respect to dt: The equivalent linear representation replaces the objective cost presented above by a new cost based on a new set of integer variables, G_{dt}:

$$G_{dt} = Z_{dt} * W_d * \sum_{l=1}^{L} \sum_{f=1}^{F} S_{lf} * T_{tf} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases}$$

The new objective cost is:

$$\sum_{d \in DS} \sum_{t \in PB} G_{dt}$$

Besides G_{dt} being a positive integer (i.e. ≥ 0), it is subject to three constraints. First, when Z_{dt} is zero, G_{dt} should be set to zero; and when Z_{dt} is one, there should not be any constraints on G_{dt}:

$$\forall_{d \in DS} \, \forall_{t \in PB} \, G_{dt} \leq Z_{dt} * \lambda_{dt}$$

λ provides an upper-bound for faster ILP convergence. In this case, λ can be selected during the pre-processing stage as:

$$\lambda_{dt} = W_d * \sum_{l=1}^{L} max_{\forall_{1 \leq f \leq F}} \left(T_{tf} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases} \right)$$

Second, when Z_{dt} is set to one, the constraint on the upper-bound of G_{dt} is:

$$\forall_{d \in DS} \, \forall_{t \in PB} \, G_{dt} \leq W_d * \sum_{l=1}^{L} \sum_{f=1}^{F} \left[S_{lf} * T_{tf} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases} \right]$$

Finally, when Z_{dt} is set to one, the constraint on the lower-bound of G_{dt} is:

$$\forall_{d \in DS} \, \forall_{t \in PB} \, G_{dt} \geq (Z_{dt} - 1) * \lambda_{dt} + W_d * \sum_{l=1}^{L} \sum_{f=1}^{F} \left[S_{lf} * T_{tf} * \begin{cases} 1 & \text{if } (RA_{dl} + WA_{dl}) > 0 \\ 0 & \text{if } (RA_{dl} + WA_{dl}) = 0 \end{cases} \right]$$

8.5.3 Selection of Linearization Formulation

Both linearization techniques presented above, with respect to dt and lf, produce a correct formulation, however their complexity differs slightly and can affect the speed of the ILP solver convergence. The following is an analysis of the two linearization techniques that provides a recommendation on which of the two linearizations to follow.

Complexity Element	Number of Variables	Number of Constraints	Number of Terms
M_{dt}	$d * t$	0	0
$M \leq X * max(Y)$	0	$d * t$	2
$M \leq Y$	0	$d * t$	$l * f + 1$
$M \geq (X - 1) * max(Y) + Y$	0	$d * t$	$l * f + 2$
Total	$d * t$	$3 * d * t$	$2 * l * f + 3$

Table 8.3: Complexity of the dt Linearization Technique

Complexity Element	Number of Variables	Number of Constraints	Number of Terms
N_{lf}	$l * f$	0	0
$N \leq X * max(Y)$	0	$l * f$	2
$N \leq Y$	0	$l * f$	$d * t + 1$
$N \geq (X - 1) * max(Y) + Y$	0	$l * f$	$d * t + 2$
Total	$l * f$	$3 * l * f$	$2 * d * t + 3$

Table 8.4: Complexity of the lf Linearization Technique

Both the number of variables and the number of constraints need to be computed in order to have a better understanding of the complexity of a formulation.

When the linearization is performed with respect to dt, there are $d * t$ new M variables introduced. Table 8.3 shows the number of new variables, new constraints and their corresponding number of variable terms. Note that the first column refers to the notation used to describe the linearization technique described earlier in the section. The last row of the table shows the total in each category for the dt approach.

Similarly, Table 8.4 shows all new complexity elements added to the formulation due to the lf linearization approach.

From the last row of the two tables, it can be seen that the terms $d * t$ and $l * f$ dictate the difference in complexity of the two formulations. When $d * t < l * f$ then the dt linearization should be followed, whereas when $d * t > l * f$, the lf linearization would lead to a faster solution. In a typical RC system, the number of physical memory bank types, t, is close to the number of processing elements, f. Thus, the number of data structures, d, and the number of computational tasks, l, in the design, govern the comparison.

In summary, for a typical RC system, if $d < l$, then the dt linearization should be implemented; whereas the lf linearization is carried out if $d > l$.

# Logic Tasks	# Data Structures	# Processing Elements	Physical Memory Banks		ILP Execution Time (secs)
			# Instances	# Ports	
10	10	2	40	60	26.6
15	15	2	40	60	44.2
20	20	2	40	60	61.9
25	25	4	200	400	97.0
40	40	4	200	400	139.1
40	40	8	2500	4500	206.3
50	50	8	2500	4500	309.8
100	100	8	2500	4500	425.3
150	150	8	2500	4500	659.0
200	200	8	2500	4500	3468

Table 8.5: Performance Results of Merged Logic and Memory Spatial Partitioning

8.5.4 Results

Finally, the results for the merged logic and memory spatial partitioning are presented in Table 8.5. The table shows the number of computational tasks in column 1, the number of data structures in column 2, the number of processing elements in column 3, the number of physical bank instances in column 4, the total number of ports across all instances of all memory bank types in column 5, and column 6 shows the execution time taken by the ILP to produce the logic and memory mapping.

In the test cases presented in Table 8.5, the number of processing elements and their sizes were selected so that a mapping was feasible. By having small tasks relative to the capacities of the processing elements, the test cases ensured that many mapping alternatives existed. Furthermore, random connections from computational tasks to other computational tasks or data structures were introduced. On an average, each task was connected to four other tasks or data segments.

As mentioned in Section 5.4, pruning techniques were used to to reduce the search space for the ILP solver. In the cases where the number of physical instances of a bank was much larger than the data structures requirement, an upper-bound on the maximum number of instances was computed and used instead of the number of available instances. In many cases, this pruning reduces considerably the number of variables as well as the number of terms in the constraints formulation of the ILP problem.

The plot of Figure 8.7 shows the execution speed of the ILP solver of the merged approach. Test cases are sorted on the X-axis such that an approximate constant increase in design

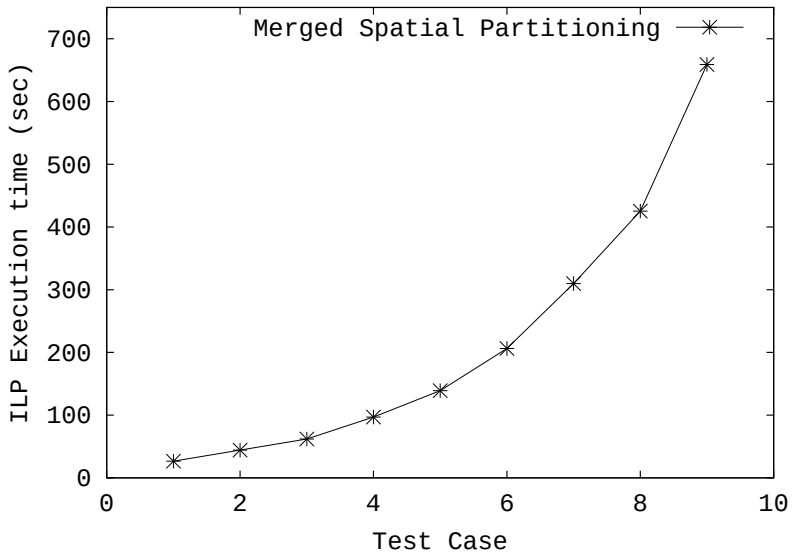


Figure 8.7: ILP Execution Times for Merged Logic and Memory Spatial Partitionin

and architecture size is introduced from one test case to another.

In conclusion, the technique presented in this section provided a solution for mapping logic tasks to processing elements and for assigning data structures to physical memory banks. The interaction between the logic and the data mappers ensured an overall efficient spatial partitioning solution. The ILP techniques and transformations that were used proved to be well fit for relatively large partitioning problems.

8.6 Conclusion

The spatial partitioning problem was tackled from the three different views presented in Chapter 7 and appropriate formulations were presented. The last section of the chapter, Section 8.5, is the culmination of all the mapping approaches where the global/detailed memory mapping approach was utilized and extended to include logic partitioning. This solution targeted a general spatial partitioning problem, where multiple types of physical memory bank types, multi-ported memory banks, multi-configuration banks, and multiple processing elements existed. The results obtained were encouraging since the ILP solver handled large behavioral (or structural) partitioning problems in reasonable amounts of time.

Besides obtaining a solution that was optimal with respect to the objective function used, the ILP provided an efficient framework where the problem was analyzed and understood

and where constraints and objective cost components were introduced and modified with ease.

Chapter 9

Arbitration

9.1 Introduction

FPGAs enable designers to perform fast prototyping of an ASIC design. Also, when a small number of design implementations is required, FPGAs provide a cheap and performance efficient alternative to hardware (ASICs) or software implementations. However, due to the rather limited programmable hardware area that FPGAs offer, many vendors introduced multi-FPGA reconfigurable computers [2, 4, 21].

These multi-FPGA reconfigurable computers eased the area constraint but introduced design complexities. One of the major problems in synthesis tools for reconfigurable computers is the lack of flexibility. Each tool is usually targeted for one specific Reconfigurable Computer (RC). Furthermore, the tools expect a tight relationship between the input design and the actual hardware. If the target RC changes, the design would require major modifications before it can be synthesized.

In an RC environment, the designer ideally has only an abstract view of the hardware architecture. This abstract modeling of the underlying hardware poses complex challenges to the synthesis and partitioning tools. Since the design specification is not constrained by the number of memory segments on the RC or the number of pins between FPGAs, it is difficult for the CAD tools to transform the design into one that maps onto a multi-FPGA RC. This chapter describes an arbitration mechanism that bridges the abstraction between the input design and the reconfigurable architecture. Since this mechanism allows such architecture abstraction between the design and the hardware, it becomes easier to port a design from one target architecture to another. This arbitration mechanism introduces very little overhead in terms of area and delay. It has been used in data-dominated applications;

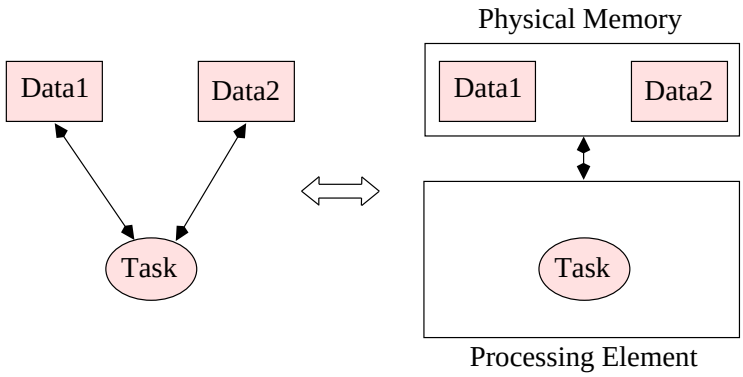


Figure 9.1: Memory Mapping

in this chapter, Fast Fourier Transform (FFT) is shown as an illustrative example.

In the context of memory synthesis, this chapter solves the problem of memory conflicts when mapping a number of data structures on a fewer number of RAM banks. Furthermore, as described in this chapter, the same problem exists when dealing with a limited set of pins between hardware devices. The methodology described is applicable to both problems.

9.1.1 Memory conflicts

If the design makes use of L logical data segments and the hardware has P physical memory segments, then two cases arise: when L is less than or equal to P, and when L is greater than P. It is assumed that the total amount of memory used at one time in the design must not exceed the total memory available in the hardware. Obviously, if L is less than or equal to P, then the mapping is straightforward: each data segment is mapped to an individual physical memory bank. It is then left up to the spatial partitioner tool to decide which data segments map to which physical banks.

On the other hand, when L is greater than P, there are more data segments in the design than there are physical memory banks on the multi-FPGA RC (refer to Figure 9.1). In this case, the mapping becomes difficult since more than one data segment has to be mapped to the same physical bank. Even if the two data segments can fit on a single physical bank, there might still be memory access conflicts.

So far, memory access arbitration has not posed a problem since very few synthesis systems cater to several target architectures [81]: the trend is to model a design at a lower level of abstraction where knowledge of the target hardware is given. For a tool to support different target architectures or for a design to be specified without implicit knowledge of the target

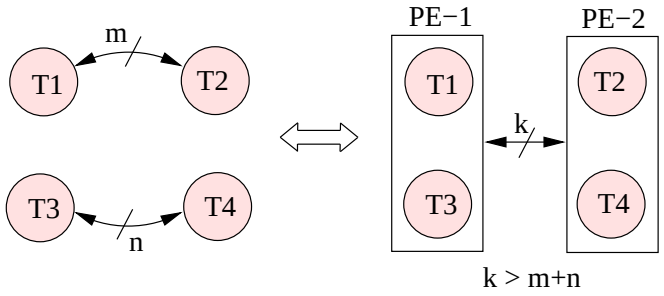


Figure 9.2: I/O Mapping

architecture, it is then necessary to provide resource mapping that ensures conflict resolution.

9.1.2 I/O pin limitations

FPGA pins limitation poses a great problem for partitioning and synthesis. The cutset between partitions limits how much the partitioner can fit on each partition (refer to Figure 9.2). FPGA pins limitation is a problem that is posed in any synthesis framework. With the advent of technology, the trend points to an increase in gates faster than an increase in pins. Effectively, FPGAs are getting bigger without an increase in the number of available pins. This might not pose a problem when the design fully fits on a single FPGA. However, when the design requires a multi-FPGA system, cutsets between the different partitions typically govern the amount of logic that can go in each FPGA: The bigger the partition, the larger the cutset between the partitions. Thus, the low number of pins on FPGAs (compared to the number of gates that the FPGA offers) forces designers to under-use the FPGA.

I/O pin management comes in two flavors: hardware specific and synthesis specific. Hardware specific management refers to RC architectures that have complex interconnect structures such as programmable crossbars or meshes [96, 97]. In order to bridge the gap between designs and variable RC architectures, software techniques are used to make the design hardware-independent. It becomes the partitioner and the synthesis tools' responsibility to adapt the design to the target RC architecture. Several mechanisms exist to reuse pins for several connections; Virtual wires [98] offer a way of overcoming pin limitations in FPGAs by statically scheduling data transfers so that multiple transfers re-use the same set of pins. This comes at the price of statically scheduling accesses. On the other hand, Vahid [99] uses functional partitioning and the concepts of FunctionBus interprocessor bus and port calling to reduce the I/O requirements. This solution came at the price of intrusive modifications to the partitioning and synthesis process.

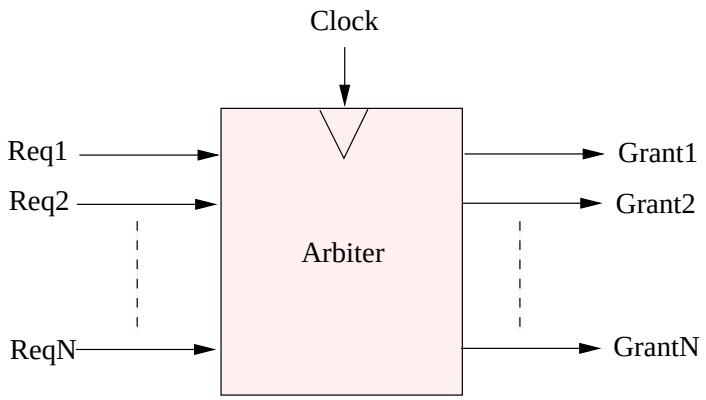


Figure 9.3: Generic N-bit Arbiter

9.1.3 Generic arbitration

It would be advantageous to have a mechanism that would solve both memory conflict and pin limitation problems. At the same time, this mechanism should not restrict scheduling of resource accesses or introduce complexity to the partitioning/synthesis process.

In the light of what is presented above, an arbitration mechanism can be introduced to solve both memory mapping and pin limitation problems. These problems can be solved with the same technique since they can be both viewed as resource sharing conflicts: multiple data segments are mapped to a single physical memory — the shared resource — and multiple I/O connections are mapped to a single set of I/O pins — the shared resource.

An arbiter should be introduced for each resource that is to be shared between processes executing in parallel. The size of the arbiter depends on the number of processes accessing that resource; and a general N-bit arbiter is shown in Figure 9.3. In the literature, arbiters are also referred to as mutual-exclusion circuits or interlocks [100].

For each process accessing a shared resource, two wires are introduced between the process and the resource's arbiter: *Request* and *Grant*. When a process wants to access the shared resource, it asserts its *Request* line and waits until its *Grant* is asserted. Thus, at any given point, the duty of the arbiter is to receive zero or more *Request*s from processes and issue zero or one *Grant*. If there are no requests, then the arbiter should not assert any grants. On the other hand, if there are one or more requests, the arbiter should then assert *exactly* one grant — hence the name mutual-exclusion circuit.

For the arbitration mechanism to be successful, it should be an automated step in the RC design environment. The advantages of this automation are two-fold: first, it allows the designer to produce architecture-independent designs; second, it allows the tools to

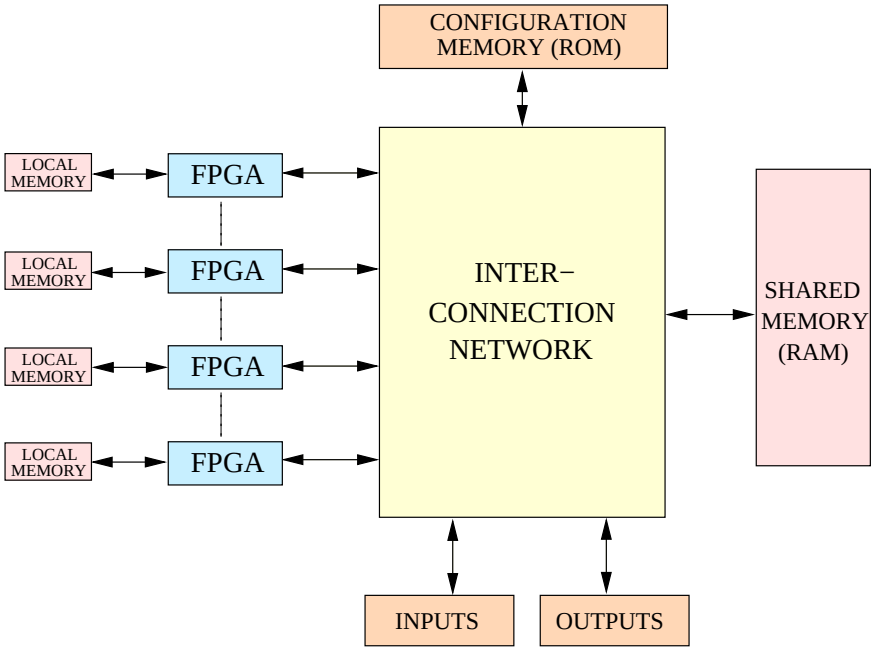


Figure 9.4: Generic Target Architecture

target a generic set of RCs. A generic target architecture is shown in Figure 9.4: the number of processing elements, the number of local memory banks, the number of shared memory banks, as well as the interconnection topology are variable. This chapter presents an automatic arbitration mechanism in the synthesis/partitioning RC environment.

The rest of this chapter is organized as follows: Section 9.2 describes the functioning of arbitration for both memory and I/O pins resolution. Section 9.3 lists the features that an arbiter should have. Section 9.4 introduces one implementation of an arbitration mechanism and discusses its suitability to this framework. Section 9.5 shows how arbitration fits in the RC framework and shows an actual implementation of arbitration in a popular digital signal-processing algorithm. Finally, Section 9.6 provides a brief conclusion.

9.2 Arbitration Mechanism

In this discussion, the input designs being partitioned and synthesized are assumed to be in the form of taskgraphs. An example taskgraph is shown in Figure 9.5. Taskgraphs contain two types of objects called tasks and memory segments. *Tasks* represent synthesizable elements of computation and *memory segments* represent elements of data storage. *Channels* are used to represent inter-task and task-to-memory communications.

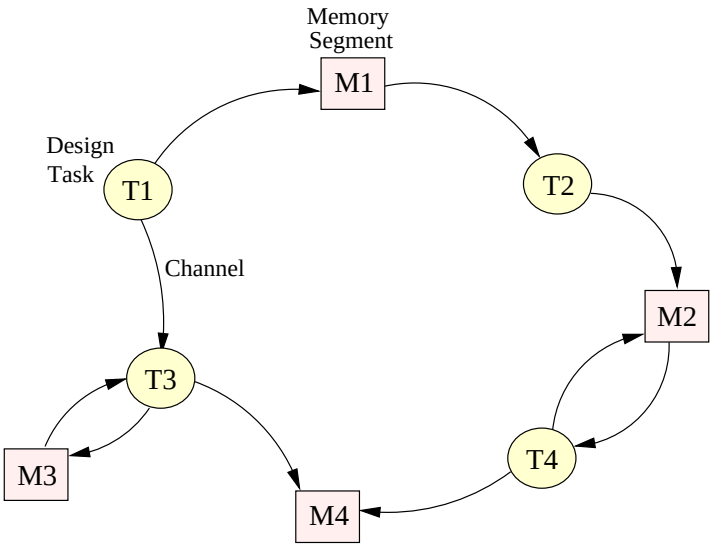


Figure 9.5: Design Specified as a Taskgraph

The Unified Specification Model format, USM [101], presented in chapter 3, is a candidate specification language that provides a hierarchical representation for specifying the behavior of a design. All tasks in the USM are simultaneously executing so as to model concurrency. Other specification languages based on the taskgraph representation are available in the literature [57, 61, 102].

By parsing an input behavior of a design written in a hardware description language, an equivalent taskgraph can be extracted and used as an input to the partitioning and synthesis tools. This format provides a reconfigurable computing environment in which partitioning tools as well as synthesis tools can co-exist [59]. This environment also exposes parallelism between independent (or partially independent) control threads in the design. The design thus becomes a collection of tasks executing in parallel. To preserve the parallelism between these tasks, an arbitration scheme cannot assume a fixed order of accesses to a shared resource. Instead, it has to *dynamically* ensure that only one task at a time is accessing the shared resource. Since the designer cannot predict how the partitioner will share the hardware resources, arbiters can only be introduced after partitioning occurs across the processing elements and the memory banks of the RC.

9.2.1 Memory arbitration

Consider the case when a task T1 reads/writes from data segment M1 and task T2 reads/writes from data segment M2 (Figure 9.6a). If the two memory segments M1 and M2 are assigned

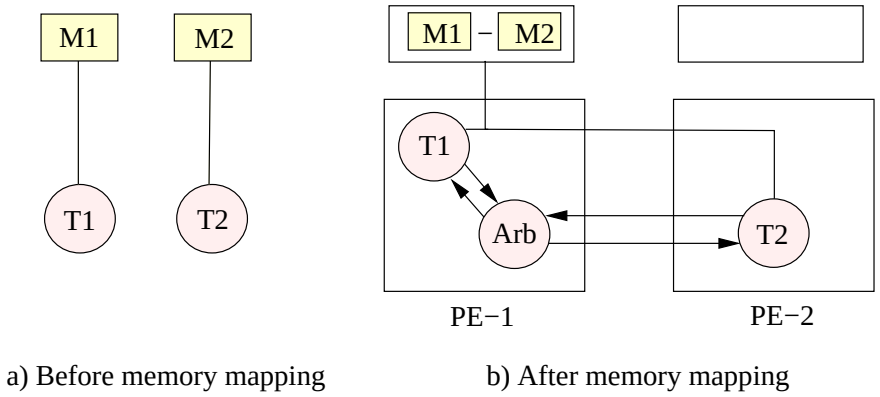


Figure 9.6: Memory Access Arbitration

to the same physical memory bank on the RC, then tasks T1 and T2 are sharing the same address lines, data lines, and read/write mode line of the memory bank. But this creates a conflict since tasks T1 and T2 might be independent from one another (i.e. executing in parallel).

Mutual exclusive access cannot be ensured for the address/data lines as well as the select mode line. So, if T1 is writing to the address lines in clock step c1, T2 cannot be accessing the memory during this step. Moreover, during clock step c1, T2 must tristate its access to the address lines. In conclusion, when two memory accesses are occurring through the same physical memory bank, an arbitration scheme has to be present to avoid any conflicts on the bank. For the example shown in Figure 9.6a, an arbiter solution is shown in Figure 9.6b.

9.2.2 Channel arbitration

Pin limitation between processing elements might cause a practical problem when a design has to be partitioned across several connected processing elements. In actual RCs, a limited amount of pins is available for interconnection. Typically, a large number of pins on each processing element is already dedicated to accessing a memory bank attached to the PE. Another set of pins is hardwired to adjacent processing elements. And finally, a limited set of pins might be dedicated to a programmable interconnect network that can connect processing elements with each others or with memory banks. Similar to the memory sharing mechanism described earlier, when the number of physical channels on the RC is less than the number of logical connections required, then physical channels can be re-used. A single physical channel can be used by more than one pair of writer/reader provided that arbitration is introduced to avoid access conflicts.

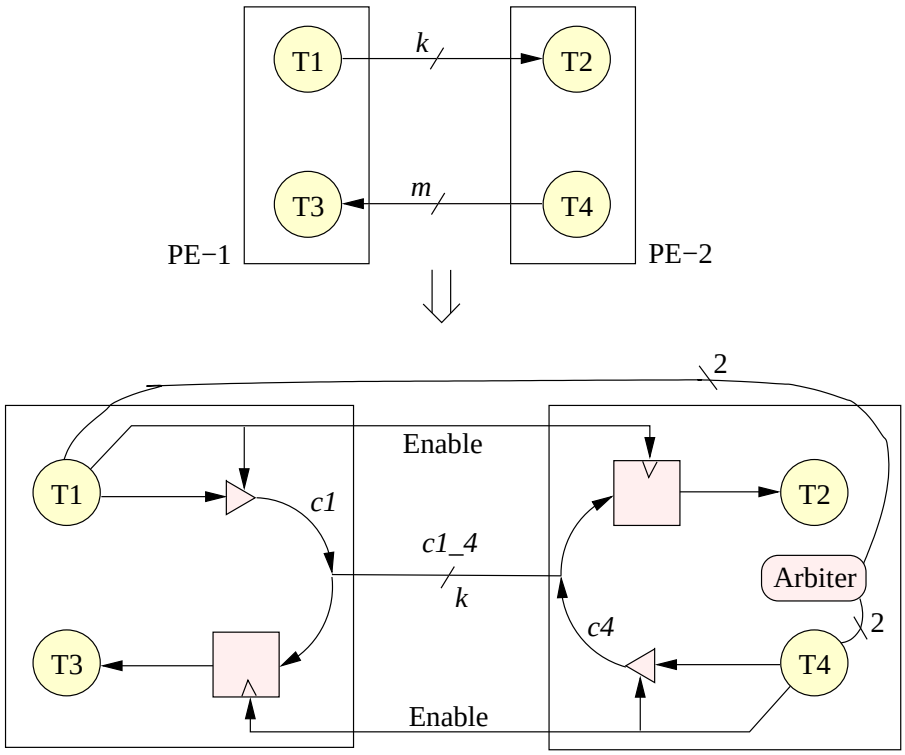


Figure 9.7: Channel Arbitration

An example of channel sharing is shown in Figure 9.7. Two logical channels (k-bit and m-bit wide, with $m<k$) are merged onto one k-bit physical channel. Arbitration circuitry (registers and tristate buffers) is required to ensure proper functioning of the shared channel; they are discussed in detail in Section 9.4.3.

Finally, irrespective of the type of resource that is being shared, tristating of shared lines is required when a task is not accessing the resource. This can be seen in Figure 9.8a where the processes' *Grant* lines control the "enable" lines of their tristates. Only when a process is granted access to the shared resource (i.e. its *Grant* is asserted), should the process drive the shared line. But what happens if all tasks are tristating their access to the shared line (in other words, none of the tasks is accessing the resource at a specific instant)? In Figure 9.8a, if both T1 and T2 are *not* accessing the shared resource, then the line connected to the shared resource is in a high- impedance state. This might cause the design to malfunction since the actual value of the shared line is unknown. In the case of address and data lines, this is not a problem, but for the select mode of a memory (write on high), for instance, it can produce unwanted effects. In this case, instead of tristating the shared line, a task should disable its access to the select mode of the memory by driving a zero, and all lines are *OR*-ed to drive the select mode of the shared memory. This ensures that even if the memory is idle at

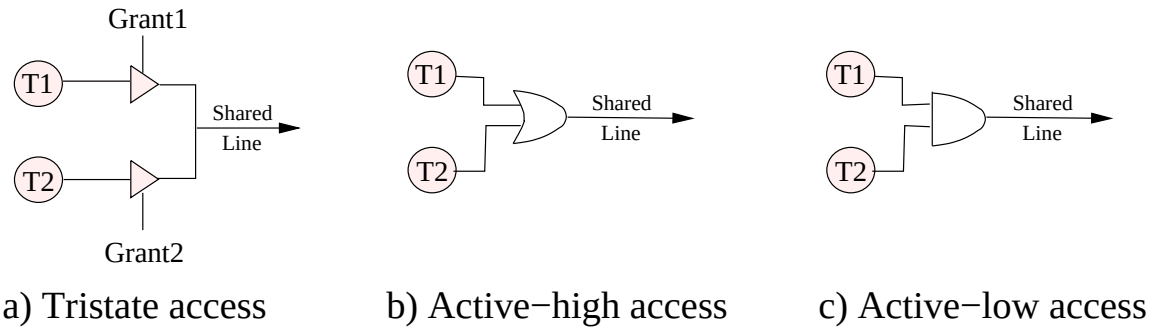


Figure 9.8: Accessing Shared Lines

some time, its select mode will be driven to zero (read mode) and no unwanted writes would occur. Hence, all resource inputs that are active-high should follow the scheme presented in Figure 9.8b; whereas active-low inputs should follow the one presented in Figure 9.8c.

In conclusion, if two or more tasks are accessing a single physical resource (memory, channel, etc.), arbitration has to resolve any access unless the tasks are globally scheduled to avoid conflicts. The latter statement refers to the case where scheduling of all tasks across *all* processing elements is done simultaneously so as to avoid resource conflicts. Global scheduling of the design is feasible but it requires a complicated controller model and it prohibits real parallelism in the execution when processes contain unpredictable loops and conditionals.

9.3 Choice of Arbiters

The implementation and functioning of arbiters depend on the environment that they will be used in as well as other constraints that the application imposes. In the RC framework, the constraints that an arbiter should follow are fairness, low overhead in terms of area and delay, and ease of insertion and synthesis.

1. *Fairness:* Similar to concerns in many aspects of multi-tasking operating systems, the arbiter should ensure mutual exclusion, prevent starvation, and prevent deadlock [103].

   Mutual exclusion: If process P is executing in its critical section (i.e. it is accessing the shared resource), then no other processes can be executing in their critical sections (i.e. they cannot access the shared resource).

   Starvation: occurs when a process that needs to access the shared resource has to wait indefinitely because the resource that it needs is always allocated to some other process.

Deadlock: occurs when waiting processes never get out of their wait states because the resources they have requested are held by other waiting processes. If the following four conditions hold simultaneously, a deadlock can occur:

(a) *Mutual exclusion* as defined above.

(b) *Hold and wait*: a process is holding a shared resource and is attempting to acquire one or more shared resources that are being held by other processes.

(c) *No preemption*: a shared resource being accessed by a task will not be available before the task voluntarily relinquishes control of it.

(d) *Circular wait*: a set of processes P_0, P_1, ..., P_K are waiting on each other in a circular manner: P_1 is waiting on P_0, P_2 is waiting on P_1, ..., and P_0 is waiting on P_K.

The reader is referred to [103] for an extensive explanation of the above concepts.

2. *Low overhead:* The introduction of arbitration to the design should not involve a substantial increase in area (function generators or CLBs) or a substantial slow-down in the design's clock speed. Also, the latency increase due to arbitration should be kept to a minimum.

3. *Extendibility and ease of insertion:* The process of introducing arbitration to the design should be simple, fast, and fully automatable. The arbiter generation should be parameterized such that the mechanism can be extended to any number of tasks being arbitered. Without these qualities, the insertion of arbitration becomes equivalent to manually modifying the design to cater to a specific RC architecture.

In the next section, a specific implementation of the arbitration mechanism is shown, and its conformity to the above requirements is analyzed.

9.4 Implementation of Arbitration

In the literature, there exist several algorithms for contention resolution each with its shares of advantages and drawbacks [100,103]. Techniques such as *random*, *FIFO*, *round-robin*, and *priority-based* were examined. Given its complexity and the type of applications that RC architectures help solve, the *round-robin* technique proved to best fit our RC framework. In this technique, requests are handled in a cyclic manner; whereas in the *random* technique, requests are handled in a random manner; for *FIFO*, requests are handled in the order in

which they arrive; and for *priority-based*, requests are handled in a statically-determined weighed order. With the exception of the *round-robin* technique, all other techniques introduced considerable complexity in the required hardware. In the RC framework, the required hardware made the arbiter either too slow or too large thus placing a considerable constraint on the synthesized design.

At anytime during the execution of a design, a *round-robin* arbiter — corresponding to a shared resource — resides in a single state. The number of states in the arbiter depends on the number of tasks accessing that resource. Each task being arbitrated introduces two states. For task *i*:
C_i corresponds to the state when task i is exclusively accessing the shared resource.
F_i corresponds to the state when none of the tasks are accessing the shared resource and task i has the highest access priority to the shared resource.

Thus, for N tasks accessing a single resource, the *round- robin* arbiter moves within the following set of states:

$\Phi = C_1, C_2, ..., C_N, F_1, F_2, ..., F_N$

The arbiter takes, as input, request signals from all tasks and produces, as output, a grant signal for each task. The set of input signals and output signals are respectively:

$\sigma = R_1, R_2, ..., R_N$

$\Omega = G_1, G_2, ..., G_N$

Based on the set of inputs (σ), outputs (Ω), and the possible set of states (Φ), the transition mechanism of the *round-robin* arbiter is shown in Figure 9.9.

The algorithm shown in Figure 9.9 is depicted in Figure 9.10 for $N = 2$ (an "X" denotes a *don't care*). Equation 9.1 and Equation 9.2 show the corresponding next states and outputs calculations. In general, for an *N*-input arbiter, there are 2**N* states: *N* "*T_i's turn*" states and *N* "*T_i in the future*" states. When the Finite State Machine (FSM) is in a "*T_i's turn*" state, task T_i is accessing the shared resource. The FSM will remain in this state until T_i releases its Request line. On the other hand, when the FSM is in a "*T_i in the future*" state, no task is currently accessing the shared resource, but task Ti has the highest priority for accessing the resource in the future. Note that after leaving state "*T_i's turn*", the FSM gives access priority to T_{i+1} by moving to state "*T_{i+1} in the future*".

$$\begin{cases} S_1(t+1) = Req_1.\overline{Req_0} + S_1(t).Req_1 + S_0(t).\overline{Req_0} \\ S_0(t+1) = \overline{Req_1}.Req_0 + \overline{S_1(t)}.Req_0 + S_0(t).Req_1 \end{cases} \quad (9.1)$$

```
case current_state is
    when F_i ⇒
       case σ is
          when zeroes ⇒
             next_state = F_i
             Ω = zeroes
          when R_i ⇒
             next_state = C_i
             Ω = G_i
          when not(R_i) and R_{i+1} ⇒
             next_state = C_{i+1}
             Ω = G_{i+1}
          when not(R_i) and not(R_{i+1}) and R_{i+2} ⇒
             next_state = C_{i+2}
             Ω = G_{i+2}
          when etc...
       end case
    when C_i ⇒
       case σ is
          when zeroes ⇒
             next_state = F_{i+1}
             Ω = zeroes
          when R_i ⇒
             next_state = C_i
             Ω = G_i
          when not(R_i) and R_{i+1} ⇒
             next_state = C_{i+1}
             Ω = G_{i+1}
          when not(R_i) and not(R_{i+1}) and R_{i+2} ⇒
             next_state = C_{i+2}
             Ω = G_{i+2}
          when etc...
       end case
end case
```

Figure 9.9: Round-Robin Transition Algorithm

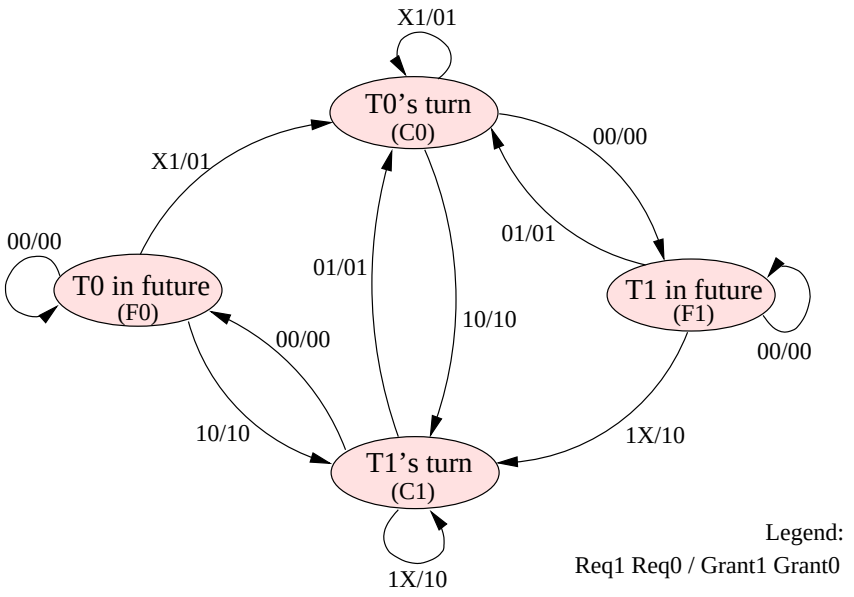


Figure 9.10: 2-Channel Round-Robin Arbiter

$$\begin{cases} Ack_1 = Req_1.\overline{Req_0} + S_1(t).Req_1 \\ Ack_0 = \overline{Req_1}.Req_0 + \overline{S_1(t)}.Req_0 \end{cases} \tag{9.2}$$

Since the *round-robin* arbiter is essentially a finite state machine, the complexity of its implementation is based on the number of equations and the number of terms in each equation. In Table 9.1, N is the number of FSMs to be arbitered, $|\sigma|$ is the number of arbiter inputs, and $|\Omega|$ is the number of arbiter outputs. Even for N=20, the complexity of the arbiter seems manageable.

9.4.1 Fairness

Since the *round-robin* arbiter is implemented as an FSM and since each state in the FSM acknowledges *at most* one request, mutual exclusion is ensured. Given the above assumption, the *round-robin* arbiter is designed such that starvation is avoided. Since the order of requests is cyclic, it is guaranteed that all tasks requesting access to the shared resource will be acknowledged. Furthermore, with the *N*-input arbiter implementation presented in this chapter, it is also guaranteed that a task requesting at a certain instant will have its grant at most after (*N*-1) tasks. This is the upper limit where not only all other (*N*-1) tasks happen to request access to the resource, but also the task in question happens to be at the end of the current order. Furthermore, the *round-robin* implementation prevents deadlock. The arbiter

N (=$\|\sigma\|$ =$\|\Omega\|$)	# States (S=2*N)	# State Variables (SV=$log_2 S$)	# Equation Inputs (=SV+$\|\sigma\|$)	# Equations (=SV + $\|\Omega\|$)
2	4	2	4	4
3	6	3	6	6
4	8	3	7	7
5	10	4	9	9
6	12	4	10	10
7	14	4	11	11
8	16	4	12	12
10	20	5	15	15
20	40	6	26	26

Table 9.1: Finite State Machine Scalability

can handle any number of requests occurring at the same time. Depending on the state in which the FSM is in, the task at the front of the list will be acknowledged. In the current form in which the *round-robin* arbiter is presented, it does not support preemption. Preemption is required when one of the tasks requests access to a shared resource, gets its grant, and never relinquishes access to this resource (i.e. task T_i makes Req_i =1, waits until $Grant_i$ =1, and never makes Req_i =0 thereafter). However, in this RC framework, since arbitration will be automated, the insertion of arbiters can ensure that the above scenario does not happen: for each "Req=1" occurrence, there exists a corresponding "Req=0" associated with it. In other words, the insertion process ensures that a task will have a limited access time to the resource (refer to Section 9.4.3 for further details).

This time-out preemption implementation is described in Figure 9.11. Apart of the little logic introduced by the counter circuitry, the FSM requires one additional input and two additional outputs.

9.4.2 Low overhead

In order to quantitatively evaluate the overhead introduced by the *round-robin* arbiter, an arbiter generator was implemented. It takes the number of tasks to be arbitrated (N) as input and it generates a corresponding VHDL file. The generator also has the option to produce different encoding schemes for the FSM (e.g. one-hot encoding, compact encoding, or synthesis tool's default encoding). Appendix C shows the PERL script used to generate arbiters of different sizes. The output of the script is a VHDL file containing the arbiter entity and its architecture. The arbiter generator was executed for N in the range [2; 10]

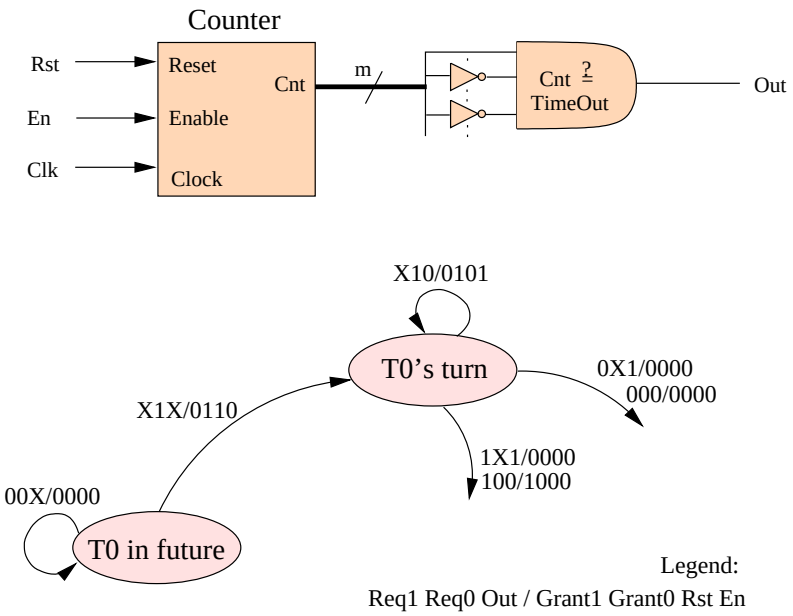


Figure 9.11: Time-Out Preemption Mechanism

and Table 9.2 lists the size of the generated VHDL files.

Each of the generated VHDL arbiters was then synthesized using two popular synthesis tools (Synplify 5.1.4 by Synplicity, Inc. and FPGA_express 2.1 by Synopsys, Inc.) targeted for the Xilinx XC4000e series with a -3 speed grade [1]. The synthesized files were then taken through Xilinx M1.5 logic and layout synthesis tools and the area values are reported in Figure 9.12 (in terms of CLBs). Note that Synplify used one-hot encoding regardless of what the VHDL files specified. FPGA_express, on the other hand, implemented both schemes. Also, note that for N=9 and N=10, even though the tool execution time of Synplify was very small compared to FPGA_express, its results were still satisfactory.

Similarly, maximum clocking speeds for each arbiter were obtained from Xilinx's estimates. These values are shown in Figure 9.13; they were obtained by placing timing constraints on the Xilinx partitioning and routing tools. It is important to note that no timing constraints were issued to Synplify or FPGA_express. Thus, it is possible to obtain even faster implementations.

It can be seen from the values reported in Figure 9.12 and Figure 9.13 that *round-robin* arbitration introduces very little overhead to the design. In terms of function generators, a 10-bit arbiter added about 40 CLBs to the design. In our experience, arbiters in the range [2; 6] were the arbiters mostly used for our example taskgraphs; larger arbiters were very seldom introduced. With the Xilinx XC4000e series FPGAs, this area overhead is

N	Size of generated VHDL file
2	1.4K
3	2.2K
4	3.5K
5	5.6K
6	9.6K
7	17.8K
8	35.7K
9	76K
10	169K

Table 9.2: Round-Robin Arbiter VHDL Filesizes

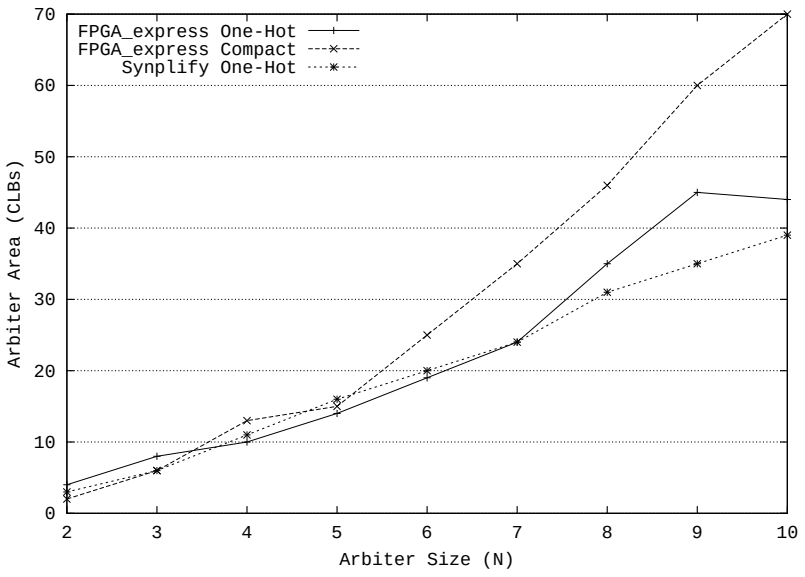


Figure 9.12: N-input Arbiter Sizes in CLBs

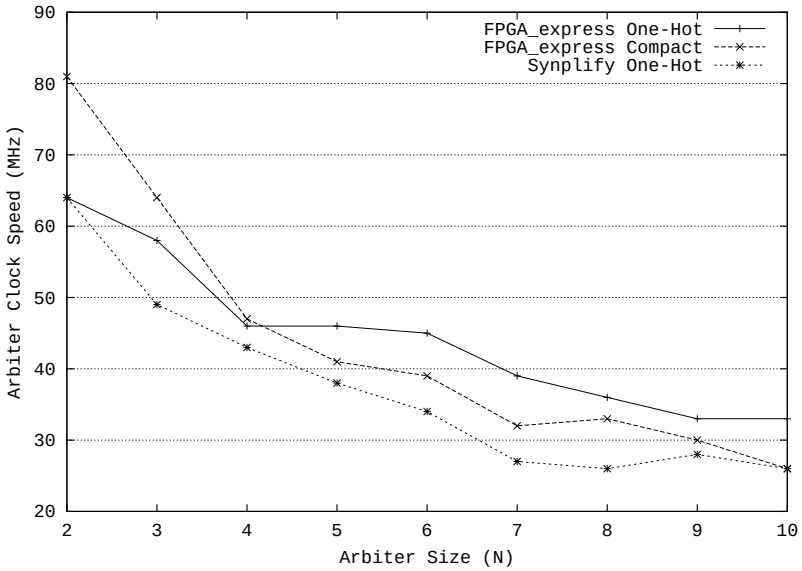


Figure 9.13: N-input Arbiter Clock Speed in MHz

very acceptable. Similarly, in terms of clocking speeds, the arbiters seem to outperform any design of reasonable size. Without, arbitration, designs occupying 50% or more of the FPGA usually reported clocking speeds of less than 25MHz (XC4000e -3 speed grade). Since 10-bit arbiters clocked at 26MHz, they did not introduce any overhead on the clock speed. As mentioned above, the arbiter frequencies shown in Figure 9.13 could be further decreased by forcing the RTL synthesis tools to produce time-efficient netlists.

9.4.3 Extendibility and ease of insertion

If a logical data segment is being accessed by more than one task, the designer is responsible for arbitrating access to the segment. However, an arbiter is automatically introduced when multiple data segments are mapped onto the same physical memory bank, and *multiple* tasks are accessing this physical memory bank.

Arbiters are introduced before calling any high-level synthesis routine; the tool flow is shown in Figure 9.14. Since arbiters are pre-characterized for the number of inputs and outputs, their area, and their delay, a precise estimation can be performed by the partitioners to ensure the fitness and speed of the contemplated design.

1. *N-bit extendibility:* The arbiter synthesis is extendible for all values of N. It follows the same process generation and insertion. The two main steps in this process are:

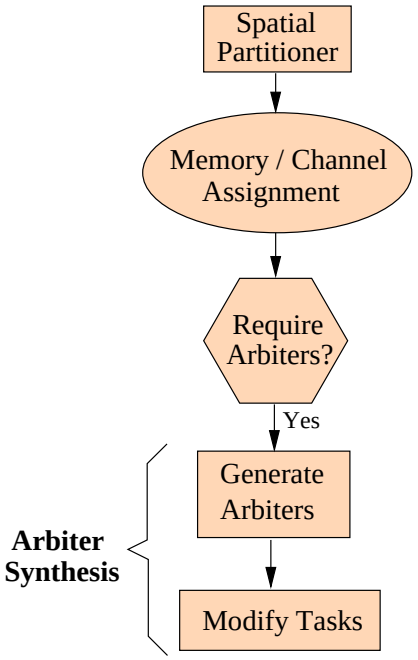


Figure 9.14: Arbiter Insertion Process

(a) It generates the appropriate arbiter based on the value of N. The arbiter generator is parameterizable: the number of arbiter inputs, N, is given as an input variable to the generator. Depending on the needs of the design, N is chosen, and the corresponding arbiter is generated. In practice, commonly used arbiter sizes would already be generated and area/delay estimates would be gathered to assist partitioning tools in their estimation process.

(b) It modifies all tasks accessing the shared resource: For each affected task, a (*Request*, *Grant*) pair of lines is added to the task's ports. And, within the task, for each access to the resource, the code is modified such that the task requests access from the arbiter, waits until it receives a grant, performs its usual access to the resource, then de-asserts its request.

A task that wants to continuously access a shared resource, has to make its *Request*=0 between each "M" accesses (in its simplest case, M can be restricted to 1). This is done in order to ensure that no task would have to wait a long time before it can access the resource (i.e. software support of preemption). An example is shown in Figure 9.15.

Note that when a task is not accessing the shared resource, it must set all shared lines to their default states; e.g. data and address lines are tri-stated, memory write/read select is set to read.

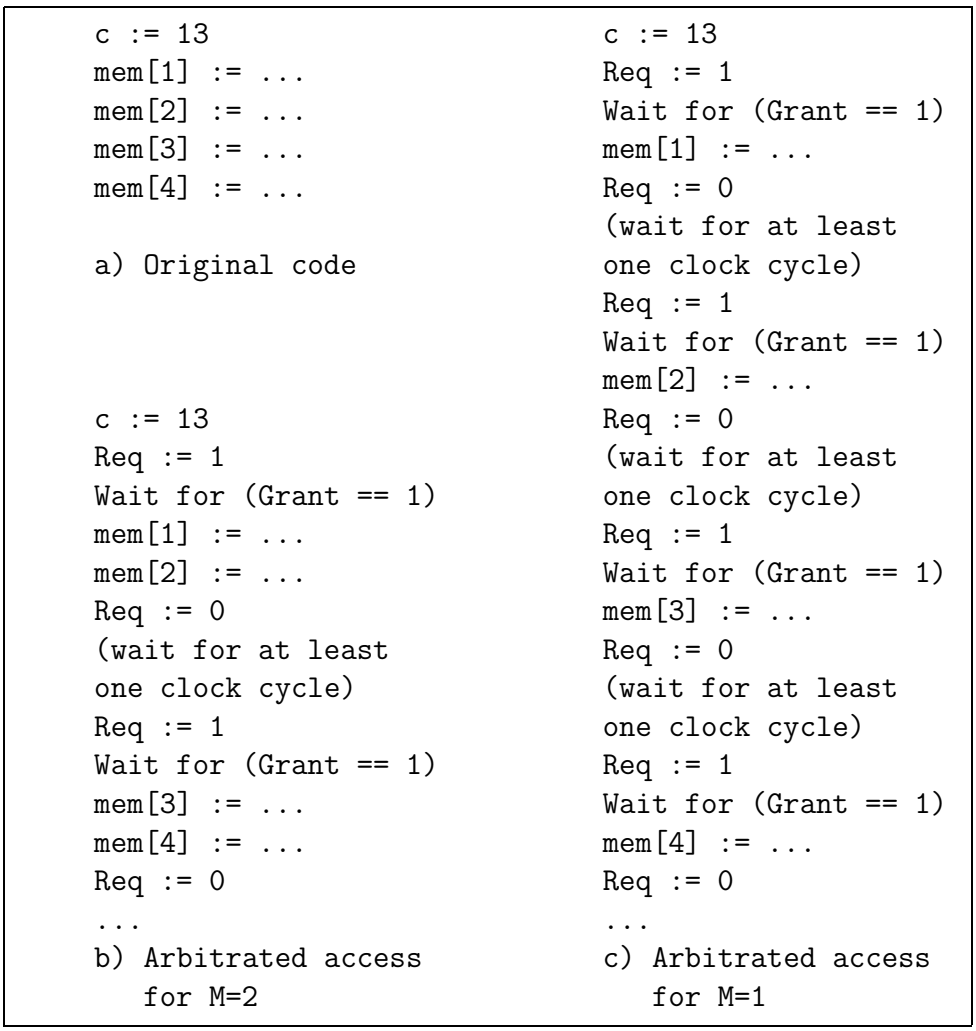


Figure 9.15: Task Modification Process

Time Step	Task 1	Task 2	Task 3	Task 4
1	c1 := 10	...	...	...
2	...	...	...	c4 := 102
3	...	x := c1	...	...

Table 9.3: Shared Channel Example

2. *Shared channel support:* The above discussion applies to all shared resources. In the case of channel sharing, however, an additional concern must be addressed: As seen in Figure 9.7, for each receiving end of a shared channel, a register will be introduced whose enable originates from the source task (whereas for non-shared channels, a register is introduced at the source end). The reason for having registers at the receiving ends of each transfer is to ensure that data going to one of the targets will not be overwritten by future transfers. In addition, the presence of the registers allows transferred data to be stored and subsequent transfers to take place immediately.

   For the example of Table 9.3, if c1 and c4 were to be merged into a single shared channel, c1_4, then we need to store the *c1 := 10* assignment from Task 1 before Task 4 performs the *c4 := 102* assignment. By having the register of c1 at Task 2's end, the value will remain indefinitely for Task 2 to consume regardless of when Task 4 writes to the shared channel.

   An arbiter is required when different sources of the shared channels belong to different tasks. If all sources belong to the same task, then there is no need to introduce an arbiter since the channel access would be implicitly arbitrated by the schedule of that task. Arbiter lines (*Request* & *Grant*) are added for every task containing one or more shared channel writes. Also, a tri-state buffer will be introduced at the output of each source task whose enable is the same as the one for the introduced register. The spatial partitioner tool marks a channel that is being shared; this way, the high-level synthesis tool can include in its estimations, the registers and buffers introduced by the sharing, and can tell whether a register should be inserted at the source or destination end.

   In addition to the task modification process described in Figure 9.9, channel sharing introduces control of the registers and tristate buffers. For example, in Figure 9.7, if we have:

   Task 1: c1 := α

   Task 4: c4 := β

   and we decide to merge channels c1 and c4 into the shared channel c1_4, then we get

the following:

Task 1:	Task 4:
Req1 := 1	Req4 := 1
Wait for (Grant1 == 1)	Wait for (Grant4 == 1)
c1_4 := α	c1_4 = β
Enable1 := 1	Enable4 := 1
clock	clock
Enable1 := 0	Enable4 := 0
Req1 :=0	Req4 := 0

In conclusion, arbiters are introduced after spatial partitioning occurs. Once the partitioner assigns data segments to physical memory banks and tasks to processing elements, arbiters are generated and introduced and tasks are modified appropriately. The hardware required for arbitration is pre-characterized for area and speed thus making the partitioners' estimation accurate. In addition, the number of clock cycles introduced by the task modification process is fixed and known prior to synthesis. The partitioners' estimation due to arbitration is accurate since the inserted hardware is pre-characterized for area and speed and the number of clock cycles introduced by the task modification process is fixed and known in advance.

9.5 Arbiter Synthesis in SPARCS

SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [59] is an integrated design system for automatically partitioning and synthesizing designs for reconfigurable computers with multiple field-programmable devices. The SPARCS system accepts design specifications at the behavior level, in the form of task graphs, where each task is specified in VHDL [104]. In SPARCS' view, a reconfigurable computer contains multiple FPGAs and multiple memory modules connected to each other through a static or reconfigurable interconnection fabric. This view admits the use of SPARCS to re-target the specification to a variety of RCs containing local and/or shared memories among the FPGAs and dedicated and/or shared connections among the memories and the FPGA units.

The SPARCS system automates the process of mapping task computations in the specification to the FPGA resources in the RC, abstract memories to physical memories and the data-flow among the tasks to the interconnection fabric. SPARCS system accomplishes this

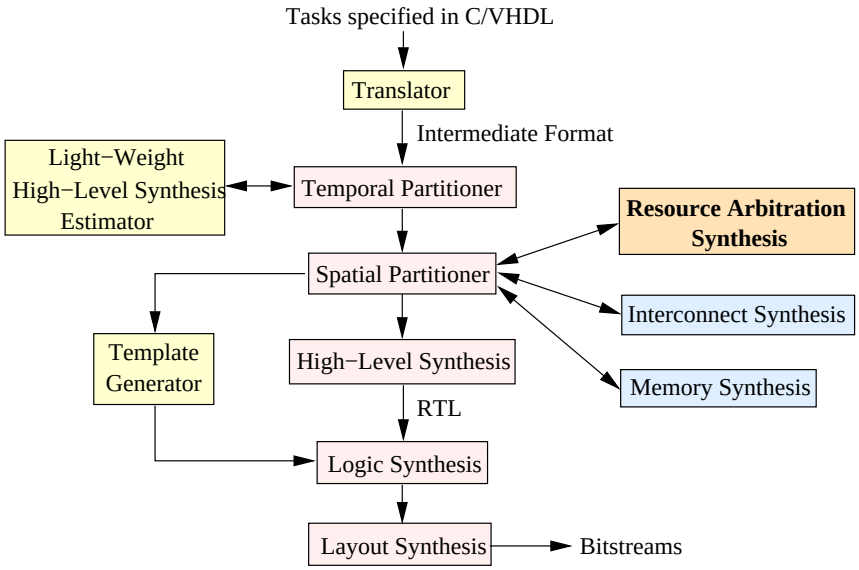


Figure 9.16: Arbiter Synthesis in SPARCS

while attempting to meet constraints on the RC clock-speed as well as time constraints on critical portions of the specification, in terms of the number of clock cycles. Following partitioning and synthesis, SPARCS generates bitmap files for each configuration of each FPGA and a reconfiguration schedule, which can be used to generate a driver program to control RC reconfiguration and execution from a host computer.

SPARCS contains: 1) a temporal partitioning tool to temporally divide and schedule the tasks on the reconfigurable architecture; 2) a spatial partitioning tool to map the tasks to individual FPGAs; and 3) a high-level synthesis tool to synthesize efficient register-transfer level designs for each set of tasks destined to be downloaded on each FPGA. Commercial logic and layout synthesis tools are used to complete logic synthesis, placement, and routing for each FPGA design segment. In addition to these tools, SPARCS automatically inserts resource arbitration, performs memory synthesis, and generates interconnection information.

A distinguishing feature of the SPARCS system is the tight integration of the partitioning and synthesis tools to accurately predict and control design performance and resource utilization.

Figure 9.16 shows the role of arbiter synthesis in the SPARCS flow. The arbiter synthesis tool can be adapted to a variety of synthesis/partitioning flows since it is contained as a separate module.

A variety of applications have been synthesized through SPARCS. In this chapter, we describe the Fast Fourier Transform (FFT) application. The 4x4 pixel, 2-dimension FFT algorithm was partitioned and synthesized in the integrated SPARCS environment [59]. The main

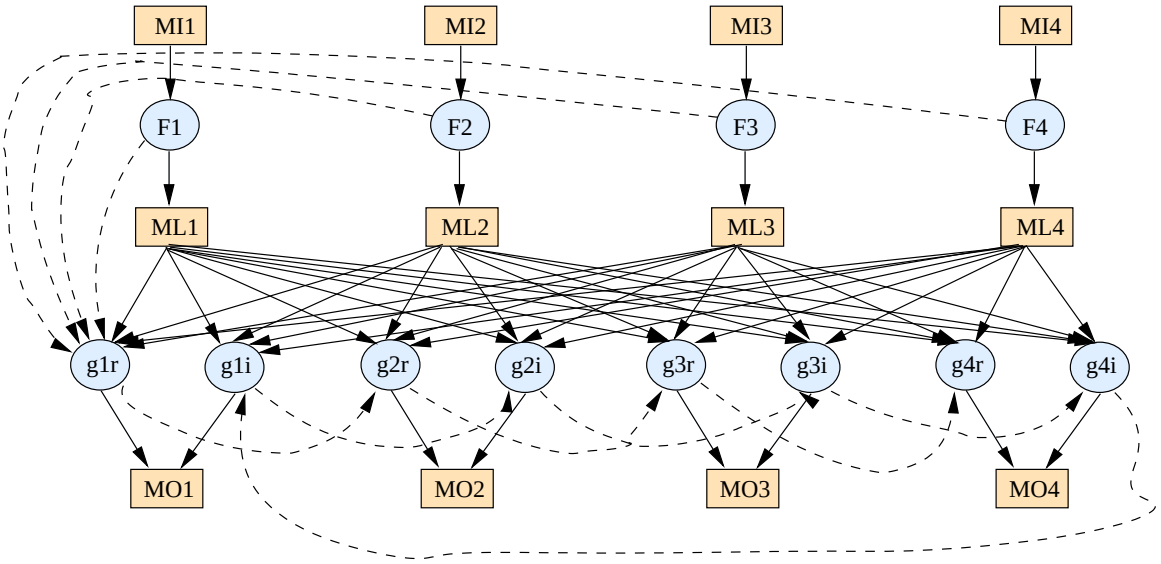


Figure 9.17: FFT Taskgraph

inputs to SPARCS consisted of:

1. *The FFT taskgraph:* The taskgraph for this application is shown in Figure 9.17. It follows the USM format [101] where the "F" tasks represent the first FFT dimension that is performed on an input image, whereas the "g" tasks represent the second FFT dimension performed on the complex-valued output of the first dimension. The solid arrows represent the data transfer between tasks and memory segments and the dashed arrows are used to specify control dependencies among tasks. Thus, in Figure 9.17, "MI" memory segments contain the input image, "MO" segments contain the final output transform, and the "ML" segments contain local data that, in this case, happens to be the output of the first FFT dimension.

2. *The target RC architecture:* The $Wildforce^{TM}$ platform [4] from Annapolis MicroSystems Inc. was used for this application. The RC (shown in Figure 9.18) has four processing elements (Xilinx XC4013e-3 FPGAs) with each a local memory (32Kbytes) attached to it. Each processing element is connected to its neighbor(s) by a set of 36 fixed pins. Also, each processing element has a 36-bit connection to a programmable crossbar interconnection structure. The crossbar can be programmed to connect any two or more processing elements together.

After partitioning, arbiter insertion, and synthesis, the tool produced three temporal partitions, of which temporal partition #0 is shown in Figure 9.19. This partition contains two

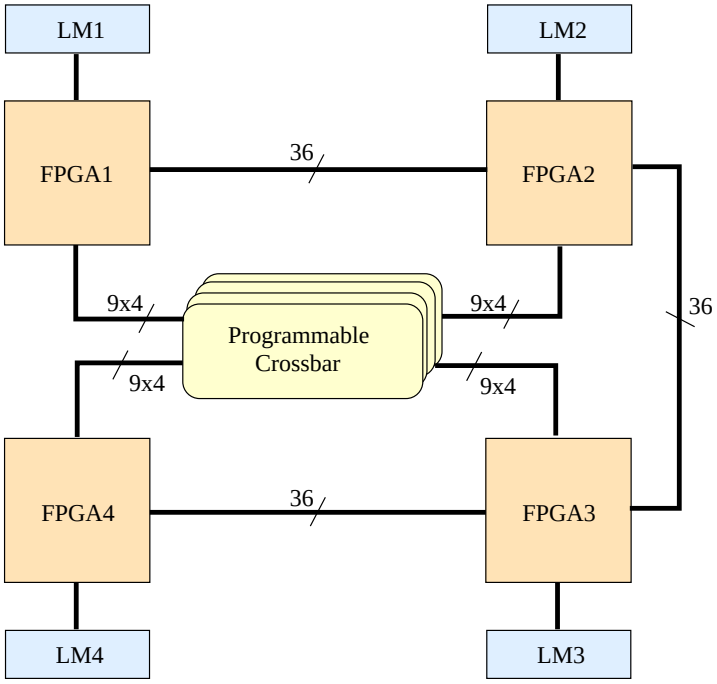


Figure 9.18: Wildforce^{TM} Architecture

arbiters: a 6-bit (Arb6) and a 2-bit (Arb2). The 6-bit arbitrates access to the local memory that contains all "ML" memory segments. Since all 6 tasks in this temporal partition access the "ML" memory segments (as can be seen in Figure 9.17), a 6-bit arbiter was introduced. The arbiter insertion assumed that all 6 tasks were executing in parallel, thus access should be arbitrated. In reality, since the "g" tasks execute after termination of the "F" tasks ("g" tasks have to wait until the "F" tasks finish writing their outputs), there is no memory conflicts between them. The arbiter insertion tool can easily detect this scenario based on the dependencies between the tasks. Instead of inserting an arbiter between these tasks, it should only ensure that the shared data, address, and select lines are appropriately set in tasks after they finish execution (tri-stated, OR-ed, or AND-ed as explained in Section 9.2.2). On the other hand, it should be noted that access of the "g" tasks to the "ML" segments is implicitly arbitrated by the designer. Since each "ML" data segment is assumed to be a single resource, the designer should ensure that not more than one "g" task (or any other task for that matter) can access it at one time.

Temporal partition #1 contained one 4-bit arbiter and partition #2 did not require arbitration. Thus, for the entire 4x4, 2-D FFT, a total of three arbiters were introduced and the design clocked at about 6MHz. Even with this low clocking frequency, the small amount of memory available in each bank, and the rather small size of the processing elements, the RC's hardware execution (4.4sec for a 512x512 image) proved faster than a software execution on a

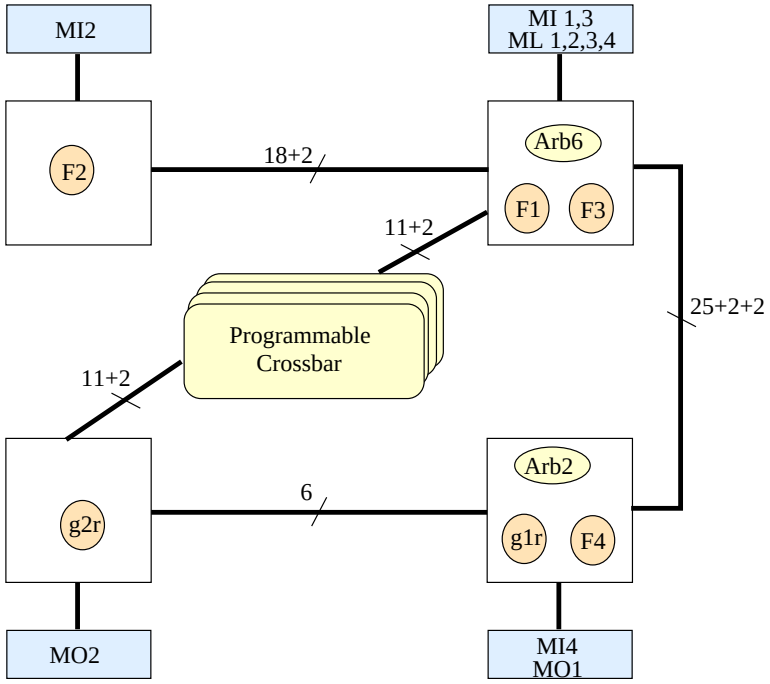


Figure 9.19: FFT Temporal Partition #0

Pentium system running at 150MHz, with 48MB of RAM (6.8sec execution time)! It should be noted that, even without arbitration, the number of temporal partitions produced would have remained the same as well as the clocking speed of the overall design.

Finally, several modifications to the partitioning and synthesis process could be made in order to obtain better results. For instance, providing constraints to the logic and layout synthesis tools could have resulted in faster designs. Also, as discussed above, by not arbitrating tasks "F" and tasks "g", the latency of the design could be reduced since tasks "F" do not have to go through the arbitration protocol in order to access the "ML" memory segments.

9.6 Conclusion

This chapter provided an explanation of resource arbitration in the RC framework and introduced the round-robin arbitration mechanism as a solution. The features of this arbitration mechanism are well-suited for an RC environment. A partitioning/synthesis system can freely distribute data segments onto physical memory banks, reuse FPGA pins if required, automatically recognize the need for arbiters and insert them, and ensure proper execution of the design without a substantial loss in area or speed. The Fast Fourier Transform application is a good candidate for such arbitration mechanism and this chapter showed how it

was synthesized for the Wildforce reconfigurable computer. With minimal user intervention, the synthesis process produced a solution for a low-end commercial RC that was faster than an equivalent software solution executing on a Pentium 150MHz platform.

Resource conflicts can arise in any memory mapping problem where a design is abstracted from the RC hardware. It remains the responsibility of the mapping and synthesis tools to allow resource sharing and introduce an arbitration protocol to ensure correct execution of the design once mapped on the reconfigurable computer. In Chapters 4, 5, and 6, the mapping techniques assumed that arbitration was not automatically inserted in the design after mapping and during synthesis: this can be seen from the *port constraints* that required distinct ports for each data structure. Arbitration can ease these constraints by allowing multiple data structures to be mapped to the same port of a memory bank, as long as the bank capacity can accommodate all data structures assigned to it. This constraint relaxation is very important since, otherwise, the number of data structures in the design is bound by the total number of available ports on the reconfigurable computer.

Chapter 10

Specification Synthesis

10.1 Introduction

This chapter describes how the synthesis process incorporates memory synthesis and how the features introduced in the earlier chapters of the thesis are handled.

Chapter 3 introduced a uniform specification model suitable to model an application at a behavioral level where data structures as well as computational tasks were easily modeled. In this chapter, the discussion is taken further and the USM specification is used to provide an efficient framework that supports memory synthesis. The goal of the chapter is to expose all transformations that memory mapping performs to a design and to provide clues on how synthesis can cater to them.

The overall synthesis flow of a design targeted on an RC system is shown in Figure 10.1. In general, the synthesis flow can be entered at any level of abstraction. So, un-partitioned and un-mapped designs can be available at the behavioral, register transfer, or gate level of abstraction. The ultimate goal is to have a synthesis system, such as the one shown in the figure, that can partition a design entered at any level of abstraction and pursue the partitioning flow before producing appropriate FPGA bitstream files used to program individual processing elements on the RC platform.

Figure 10.1 shows the generic view of the synthesis process that includes the following main components:

1. **Partitioning Engine:** The partitioning component handles the general tasks of partitioning the design in time and in space, and the partitioning of data structures with respect to the target architecture. The partitioning engine invokes the spatial parti-

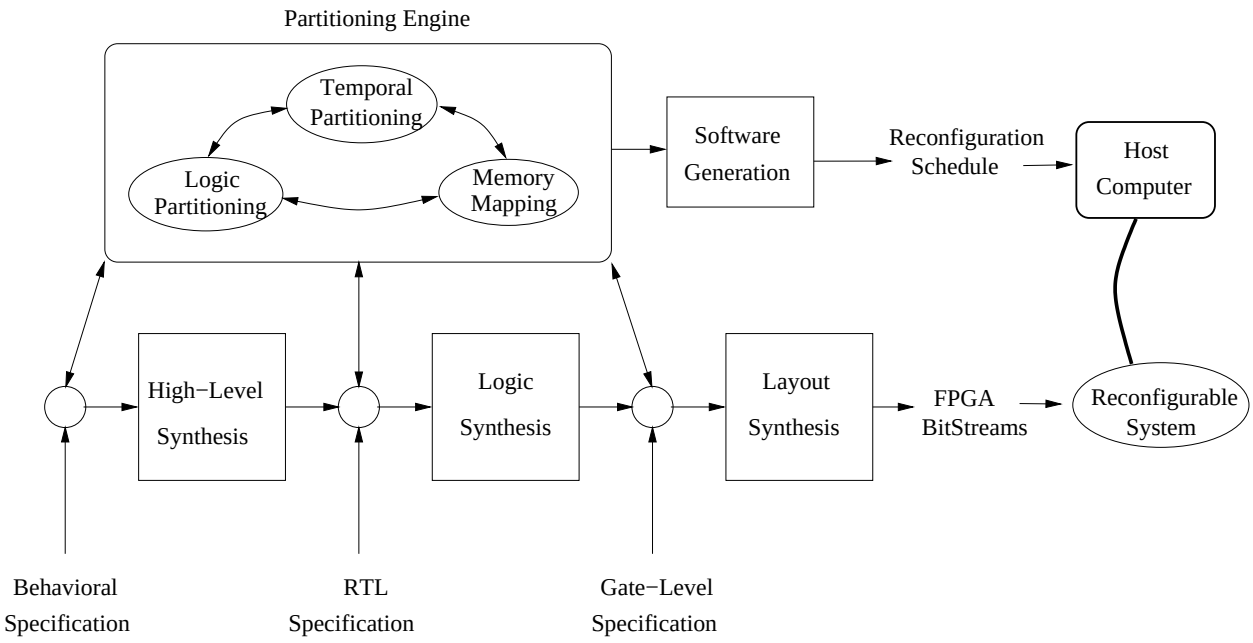


Figure 10.1: Overall Synthesis Flow

tioner when the design does not fit on a single processing element but fits on multiple processing units of the RC platform. It invokes the temporal partitioner when the design does not fit on the RC platform, thus requiring to multiplex the execution of the design on the RC platform over time.

(a) *Temporal Partitioning:* Since the logic capacity of programmable processing elements is typically far less than that of ASIC chips, the capacity of an RC platform is limited sometimes to execute only parts of the design. The temporal partitioner can extend the effective device capacity of the RC system by partitioning the design over time such that each segment fits on the platform. Temporal partitioning for RC systems is surveyed and discussed in [105].

(b) *Logic Partitioning:* Logic partitioning was presented and discussed in Chapter 7. The logic partitioner assigns the logic of the design to processing elements of the RC platform.

(c) *Memory Mapping:* When physical memory banks exist on the RC system, the memory mapper assigns the design's data structures to the banks of the platform.

2. **High-Level Synthesis:** High-level synthesis is the process of generating a structural implementation from the functional (or behavioral) specification of a design. High-level synthesis for reconfigurable computers is further surveyed and discussed in [106].

3. **Logic Synthesis:** Logic synthesis converts designs at the register transfer level into a gate-level equivalent mapped to the target FPGA architecture. Logic synthesis is described in [56].

4. **Layout Synthesis:** The last part of synthesis involves placing and routing the gate-level design onto the processing units. Gates are assigned to the atomic units of the target architecture in order to ease the routing of the interconnection nets and minimize the delay of the target circuit. Further descriptions of layout synthesis is provided in [107].

5. **Miscellaneous Software Generation:** To control the execution of the design on the target RC platform, a scheduling software is generated to monitor the execution of the design and to load FPGA bitstreams onto the processing units and the data structures onto the physical memory banks. Since temporal partitioning can produce more than one temporal segment, the scheduling software monitors the execution of a temporal segment, checks on the completion of the segment, then loads the next temporal segment and triggers its execution.

The SPARCS synthesis system [59] that was presented in Chapter 9 is based on the synthesis view of Figure 10.1. However, SPARCS limits the partitioning engine to the behavioral-level where impact on the final synthesized design is the greatest at small exploration time cost. In other words, when partitioning is performed at high levels of abstraction, many alternative solutions can be explored at a fraction of the time that it would require to explore partitioning alternatives at the lower levels of abstraction.

For all the partitioning and synthesis tools to co-exist and interact with each other, a specification model should be available to provide a vehicle in which the design can be passed from the output of a tool to the input of another tool. This vehicle mechanism sets the protocol in which tools interact and, if constructed carefully, can simplify the overall synthesis process.

In this chapter, a taskgraph representation, based on the USM model developed in Chapter 3, is presented. The taskgraph is the vehicle that allows efficient migration between the different partitioning engine components and high-level synthesis. The taskgraph format is presented in Section 10.2. Taskgraph, USM, and BBIF related synthesis issues are discussed in Section 10.3. Finally, Section 10.4 provides a conclusion.

10.2 Taskgraph Specification and Synthesis

The *taskgraph* is a representation of a design problem that includes information of both the functional design that is being synthesized and the RC system on which the design is targeted.

The taskgraph is a useful representation of the design that is used to first model a design and then take the design through several transformations including temporal partitioning, multi-FPGA logic partitioning, memory mapping, ending with high-level synthesis.

The taskgraph format can be easily extended to introduce new parameters in the RC architecture, constraints, or design specification; thus providing a flexible environment suitable for the fast changes occurring in the reconfigurable computing research.

The taskgraph is composed of two main categories: a general category and a design specific category. The design specific category represents the functional design being synthesized, whereas the general category contains RC architecture details and miscellaneous information related to synthesis. The following paragraphs provide a description of the taskgraph for a subset of RC architectures

- **General Category:** This category provides all the information required for the synthesis process except for the actual design.

    1. *RC architecture:* Figure 10.2 shows the RC architecture section of the taskgraph. It contains information about the hardware utilized – the number and type of processing units existing in the RC system, the size of each processing element (the number of configurable logic blocks), the size of memory banks attached to each processing element (the number of words in memory and the number of bits in each word), and the amount of time required to reconfigure the entire RC system. Knowledge about the number, type, and configurability options of on-chip memory banks is assumed to be obtained from an existing database that is queried based on the FPGA type selected.
    2. *Component library information:* This section points the synthesis tools to the library of components where estimates on the functional units and other components of a design exist. Figure 10.3 shows the corresponding taskgraph format.
    3. *Design constraints:* Figure 10.4 shows the constraints section that allows the designer to specify a clock constraint on the synthesized design.

       Based on the capabilities of the synthesis and partitioning tools, the design constraints section can be extended to handle other latency, speed, or area constraints.

```
BOARD
(
     % Name of board (Optional)
     BOARDNAME
     % Number of FPGAs (Required)
     FPGANUM
     % FPGA Type (Optional)
     FPGATYPE
     % Number of CLBs per FPGA (Required)
     FPGASIZE
     % Memory size per FPGA in memory words (Required)
     MEMSIZE
     % Number of bits per memory word (Required)
     WORDSIZE
     % Board Reconfiguration Time in ms (Required)
     CONFIGTIME
)
```

Figure 10.2: Board Information in Taskgraph

```
LIBRARY
(
     % File containing library of components (Required)
     LIBRARYNAME
)
```

Figure 10.3: Component Library Information in Taskgraph

```
CONSTRAINTS
(
     % Constraint on Clock Period in ns (Optional)
     CLOCK
)
```

Figure 10.4: Constraints Section in Taskgraph

```
TASK
(
     % Name of the task (Required)
     NAME
     % TP to which the task is assigned (Optional)
     TP
     % FPGA to which the task is assigned (Optional)
     FPGA
     % Controller to which the task is assigned (Optional)
     CONTROLLER
     % Name of file containing the module bag (Optional)
     MODULE_BAG
     % Total area (in CLBs) for the task (Optional)
     AREA
     % Total number of control cycles for the task (Optional)
     CYCLES
     % Name of file containing the actual task (Required)
     BBIF
)
```

Figure 10.5: Logic Task Specification in Taskgraph

- **Design Category:** The design structure is fully specified in this category. All tasks and data structures in the design as well as all communication among tasks and data structures is provided.

    1. *Tasks:* As presented in Chapter 3, tasks represent elements of computation. Figure 10.5 shows the taskgraph format of a task. This section in the taskgraph only holds the important characteristics of a task. The actual computations that the task performs are located in a file whose name is included in this section. Therefore, besides the name of the task and the corresponding filename that contains the computations in the task, the following information is present: which temporal segment and processing element the task is assigned to, an estimate of the area that the task will occupy on the processing element, the number of clock cycles required by the task to fully execute, and other synthesis-related details such as the controller that the task is assigned to, and the name of the file containing the task's module bag.

       To represent the design fully, there exist as many "task" sections in the taskgraph as there are tasks in the design. In Section 10.3, details are provided on why and

how the synthesis process introduces new tasks.

2. *Data segments:* Data structures in the design are also included in the taskgraph. Figure 10.6 shows the template used to model each data structure. The data segment section includes information about the size of each data structure in terms of total depth and width along with information about which tasks access it. The physical memory bank type, instance(s), and port to which the segment is assigned, and the offset at which the segment is placed in each memory bank instance is also provided.

   For each "accessor", a task accessing the data segment, information about whether the task is reading, writing, or both is required. In addition, the accessor subsection also provides the number of bits required to access the address and data buses along with optional pin numbers in case the buses cross boundary of the processing unit (to which the accessor task is assigned to), to reach the memory bank. The mode pin is the optional pin information for the read/write enable mode used to access the memory. Since one accessor might be accessing only a single bit from each word of the data structure, whereas another accessor might require all bits from each word, it is necessary to provide separate address and data buses bitwidths in order to minimize the number of wires that are required between tasks and data segments.

   Finally, the "mode" entry in the data segment section provides clues about the life of the data structure. An *IN* data structure refers to a data structure that needs to be present at the beginning of the design execution since an external source would be initially writing to the data structure (e.g. an input image to be processed). An *OUT* data structure refers to a data structure that needs to be present at the end of the design execution since an external device reads the data (e.g. the transformed output image). Finally, an *INOUT* data structure refers to a data structure that needs to be alive during the entire execution of the design. Based on the synthesis process and in the event they do not conflict, *IN*, *OUT*, and *LOCAL* data structures can overlap in the same memory space.

3. *Flags:* Flags are used to represent dependency edges between tasks. They are 1-bit equivalent communication wires that model the control flow among computations. Figure 10.7 depicts the template of each flag present in the design. Besides a name that is associated with each flag, only one task writing to the flag is allowed. However, multiple tasks can read the flag. The template holds the name of the writer task and can also include the optional pin information. In addition, multiple subsections of reader tasks exist to specify each task reading the value of the flag. Again, a pin number is associated with the processing unit in which the reader

```
MEMORY
(
     % Name of the memory segment (Required)
     NAME
     % Mode of the memory segment: IN, OUT, INOUT, or LOCAL (Required)
     MODE
     % Size of the memory segment (Required)
     SIZE
     % Memory type to which the memory segment is assigned (Optional)
     TYPE
     % Instances to which the memory segment is assigned (Optional)
     % (space separated list)
     INSTANCES
     % Port number to which the memory segment is assigned in each instance (Optional)
     % (space separated list.  Same length as the INSTANCES list)
     PORT
     % Base Address of the memory segment in the physical bank instances (Optional)
     % (space separated list.  Same length as the INSTANCES list)
     BASE
     % One or more accessor (required)
     ACCESSOR
     (
          % Accessor name (Required)
          NAME
          % Is this accessor a reader (R), writer (W), or both (RW)? (Required)
          MODE
          % Number of bits used by the address bus (Required)
          ADDR_WIDTH
          % Pins to which the address bus writer is assigned (Optional)
          % (space separated list)
          ADDR_PINS     ()
          % Number of bits used by the data bus (Required)
          DATA_WIDTH
          % Pins to which the data bus reader/writer is assigned (Optional)
          % (space separated list)
          DATA_PINS     ()
          % Pin to which the memory mode is assigned (Optional)
          MODE_PIN
     )
)
```

Figure 10.6: Data Structure Specification in Taskgraph

```
FLAG
(
     % Name of the flag (Required)
     NAME
     % Only one writer for each flag
     WRITER
     (
          % Signal name of the flag source (Required)
          NAME
          % Pin to which the flag source is assigned (Optional)
          PIN
     )
     % Multiple readers are allowed for each flag
     READER
     (
          % Signal name of the flag destination (Required)
          NAME
          % Pin to which the flag destination is assigned (Optional)
          PIN
     )
)
```

Figure 10.7: Flag Specification in Taskgraph

task exists. The names of the reader and writer tasks must match with the names provided in the task sections.

4. *Channels:* Figure 10.8 depicts the channels in the design. Channels are data communication links between computational tasks in the design. Contrary to the flags that are 1-bit wires, the bitwidth of a channel can vary. A corresponding bitwidth entry in the template exists. Furthermore, channels can be written to as well as read by one or more tasks. When two or more communication links are multiplexed on the same physical wires in the RC platform, the channel is shared among multiple writers. Hence, multiple "writer" subsections can exist that hold the name of the writer task, the maximum number of bits that the writer might use, as well as the optional pins information. Similarly, multiple reader tasks can exist, and each "reader" subsection carries the same information as the "writer" subsection.

   When a channel is multiplexed and used for several communication links, the design must ensure a proper arbitration scheme so that when one writer is sending

```
CHANNEL
(
     % Name of the channel (Required)
     NAME
     % Bitwidth of the channel (Optional)
     BITWIDTH
     % Multiple writers for each channel
     % (multiple writers => shared channel)
     WRITER
     (
          % Signal name of the channel writer (Required)
          NAME
          % Number of bits the writer requires (Optional)
          BITWIDTH
          % Pins to which the channel writer is assigned (Optional)
          % (space separated list)
          PINS     ()
     )
     % Multiple readers are allowed for each channel
     READER
     (
          % Signal name of the channel reader (Required)
          NAME
          % Number of bits the reader requires (Optional)
          BITWIDTH
          % Pins to which the channel reader is assigned (Optional)
          % (space separated list)
          PINS     ()
     )
)
```

Figure 10.8: Channel Specification in Taskgraph

data on the channel, all other writers tri-state their access to it. More synthesis details are provided in Section 10.3.

10.3 Taskgraph and Specification Synthesis

Initially, the input design specified in the USM specification model is converted to a taskgraph representation. This can be done manually or automatically by parsing the overall description of the design hierarchy provided in a descriptive language such as VHDL. A set of defined types and functions are grouped into a package that restricts the VHDL constructs in order to adhere to the USM methodology. By providing explicit "task", "memory", "flag", and "channel" types in VHDL, as well as built-in "memory" entities that emulate the physical memory banks of the RC hardware by allowing generic parameters such as the read or write latency of each bank, the taskgraph is easily extracted from the component declarations of all tasks. Furthermore, the overall entity that instantiates all tasks, and memory components is parsed to extract the inter-connectivity between all tasks and between tasks and data structures.

10.3.1 Taskgraph and USM Synthesis

Once the taskgraph is generated, it can go through the different components of partitioning. Temporal partitioning updates the taskgraph by assigning temporal partition numbers to each *task* in the taskgraph. Each task is annotated with a number reflecting the temporal segment in which it will be executed. This corresponds to the TP field in the task section of the taskgraph. All tasks that have the same number will co-exist on the RC platform and only when all tasks complete execution, will the next temporal segment be loaded onto the RC platform.

Similarly, spatial partitioning updates the taskgraph, but the updating process is more complex. The following updates are performed by spatial partitioning:

- *Spatial Assignment:* First, the spatial partitioner assigns *tasks* to specific processing elements of the RC platform. This corresponds to the FPGA field in the task field of the taskgraph. Second, it maps data structures to physical memory banks by assigning each *memory* in the taskgraph to a set of memory bank instances and a port for each memory bank instance. It also sets the addressing offset for each instance in the set. These correspond respectively to the INSTANCES, PORT, and BASE fields

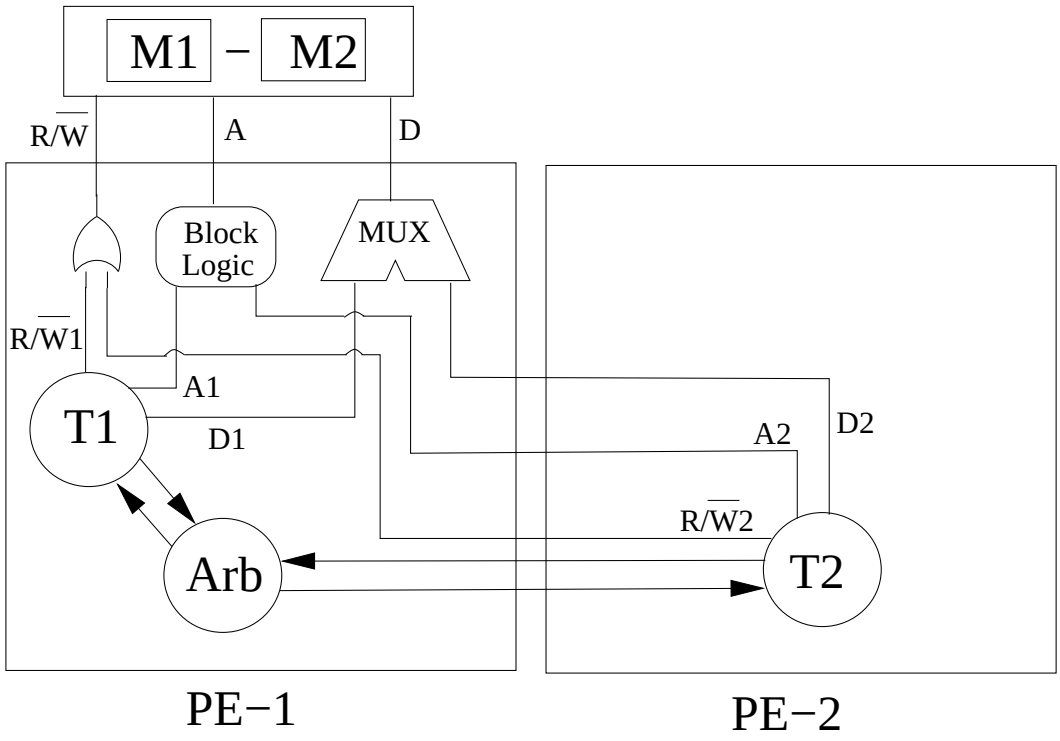


Figure 10.9: RC Memory Synthesis

in the memory section of the taskgraph. Third, the spatial partitioner assigns pin locations for all task-to-task and task-to-memory interconnections corresponding to the ADDR_PINS, DATA_PINS, and MODE_PIN in the memory section of the taskgraph, and the writer/reader PINS fields in the channel section of the taskgraph.

- *Memory bank arbitration:*

  Figure 10.9 shows two tasks, T1 and T2, that access two data structures respectively, M1 and M2. Since both M1 and M2 are placed onto the same port of a physical memory bank, arbitration is introduced as described in Chapter 9. Besides the introduction of an arbiter, additional logic is required for correct operation. The additional logic is shown in Figures 10.9 and 10.10. For the insertion of the arbiter, a new task is added to the taskgraph whose name is explicitly identified by the synthesis tools to indicate an automatically generated arbiter of a specific size. Similarly, for every additional logic block, a task is added to the taskgraph whose name identifies the logic required for the task. The spatial partitioner is provided with area and delay estimates for each introduced task, and hence can set the corresponding FPGA field. Finally, flag and/or channels are introduced to connect the new arbiter and logic tasks with the pre-existing computational tasks and with the data structures in the design.

In Figure 10.9, read/write enable of both tasks are OR-ed to produce the resulting signal that controls the write enable of the physical memory bank. It is assumed that when a task is not accessing its data structure, it sets the read enable signal to low. Thus, by OR-ing the two signals, a write enable will occur on the memory bank only when either one of the tasks is accessing the memory and performing a write operation. The data bus signals are multiplexed and selected by the arbiter acknowledge signals so that the correct data signals is channeled to the memory bank. If the tasks tri-state by their respective data bus signals when they are not granted access to the shared memory, then a multiplexor is not required. The data buses can be simply connected together. Lastly, the address buses of the tasks are processed by the block logic mechanism before being forwarded to the physical memory bank.

Figure 10.10 shows the block logic that is used to compute the physical address of the accessed data. Two different transformations occur before the physical address is generated. First, if the design is processing multiple sets of data, a technique also known as block processing, the block number is generated by a counter (as seen in Figure 10.9 and concatenated to the virtual address produced by the computational task. Second, an address offset is added in the event the data structure is placed at a non-zero address. This is not required when the data structure is assigned to the beginning of the physical memory bank. It can also be simplified and replaced by another concatenation operation when the offset is a power-of-two number.

- *Shared channel synthesis:* Similar to memory bank arbitration, when physical pins are shared among multiple connections in the design, an arbitration mechanism is introduced during partitioning to ensure proper operation of the design. Figure 10.11 shows how two connections, $c1$ and $c4$, connecting tasks ($T1$, $T2$) and ($T3$, $T4$) respectively, are overlapped on the same physical channel, $c1\_4$. As described in Chapter 9, resource access conflicts are resolved by an arbitration scheme during partitioning of the design.

  Arbiters and tri-state buffers are introduced as new tasks in the taskgraph, , and new flags and/or channels are added to connect the new tasks with the pre-existing computational tasks. Not only does the spatial partitioner estimates the area of the added logic when performing the logic partitioning, but it also computes the number of added interconnections that are added due to the arbitration mechanism.

- *Monitoring circuitry:* After the design is divided into several temporal partitions, a control mechanism ensures correct execution of each temporal partition by synchronizing all tasks within the partition and waiting for all tasks to finish execution. Each task is synthesized as a datapath-controller model where finite state machines control the execution of each task separately. This allows all tasks to execute separately and

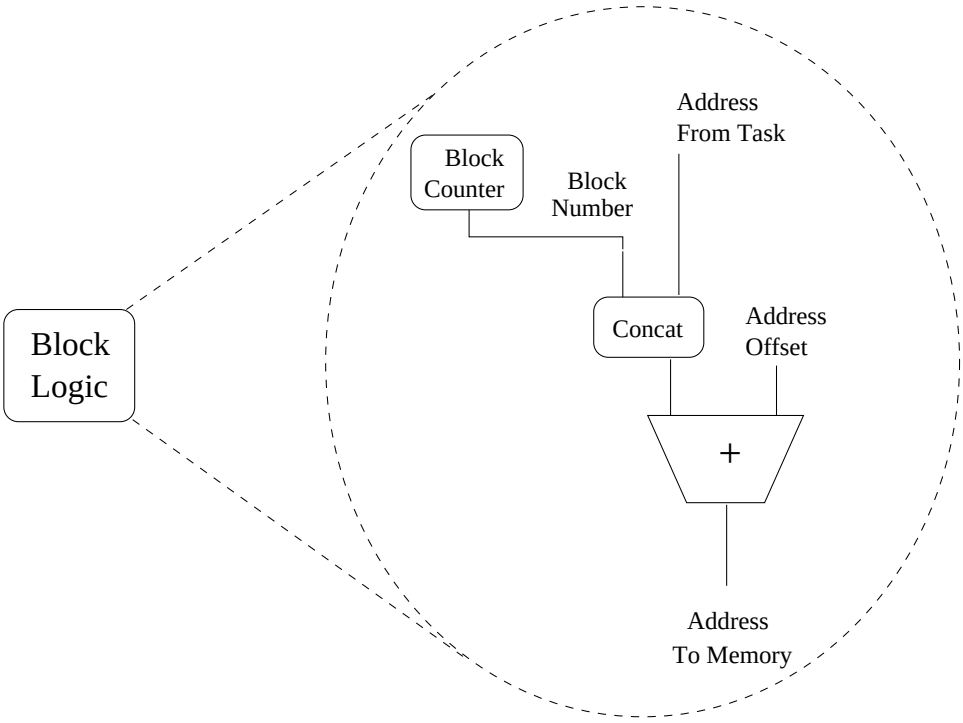


Figure 10.10: Block Logic in Memory Synthesis

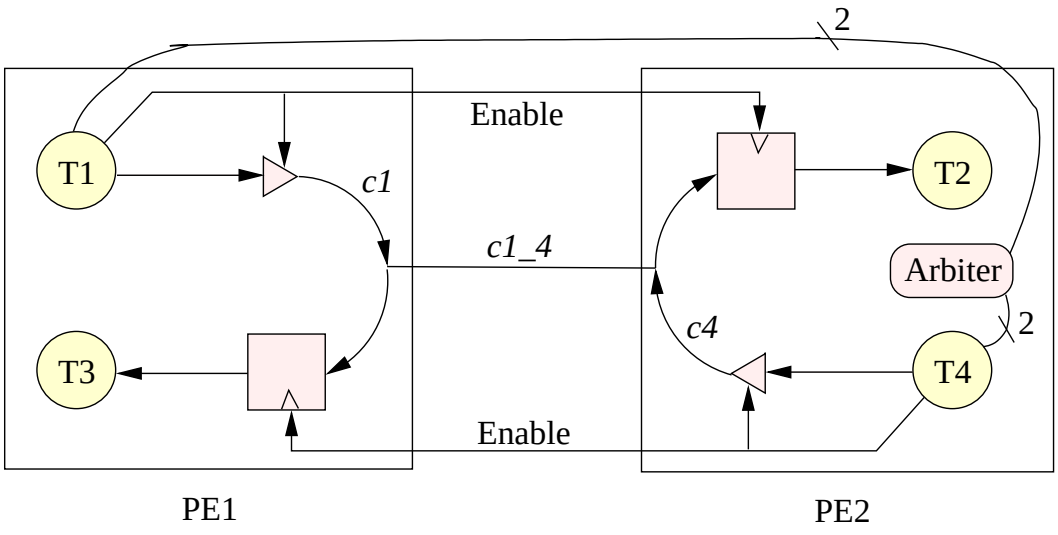


Figure 10.11: RC Shared Channel Synthesis

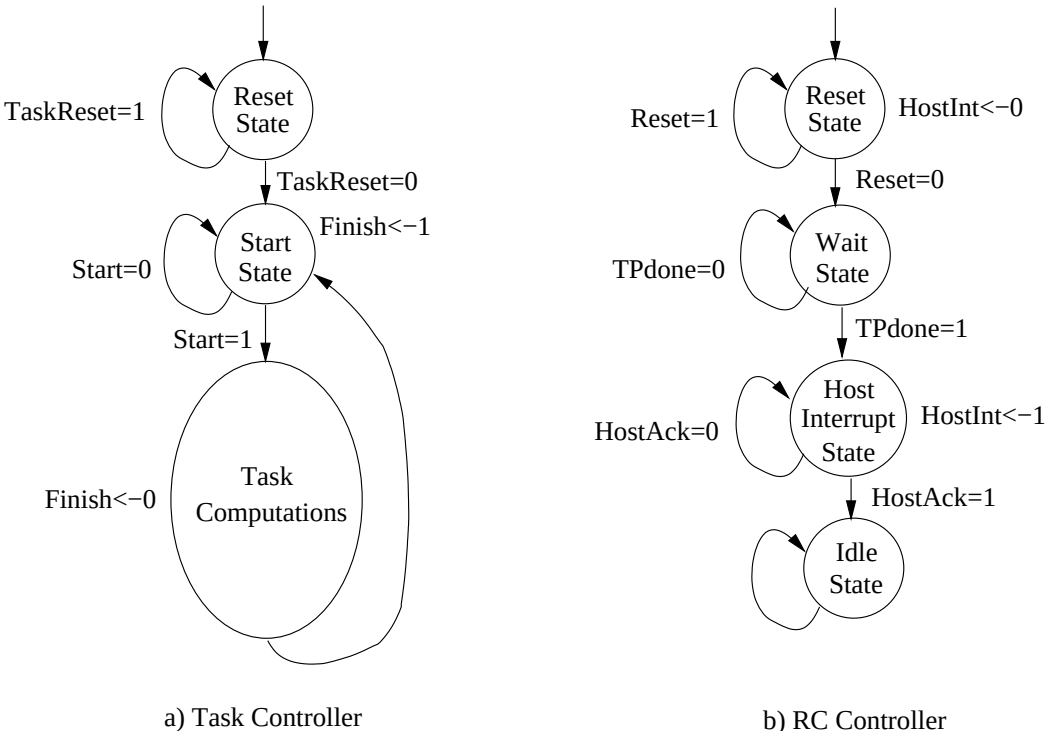


Figure 10.12: Task and RC State Machines

synchronize access to shared data structures and the exchange of data through the use of USM flags. There is no need for a global controller, however, a simple monitoring controller simply triggers all the tasks and waits until they all complete execution.

Figure 10.12 shows both the tasks' controller model and the overall monitoring finite state machine. In the task controller, the task waits until the TaskReset signal is de-asserted. The TaskReset is shown in Figure 10.13. It is generated by OR-ing the global RC reset and the the BlkDone signal: when either the global RC reset is asserted or when all tasks have finished processing one set of data and are ready to move on to the next set of data, should the task be reset. Once the task exits the reset state, it asserts its local finish signal to indicate that it is done with the previous set of data, then waits until its start signal is asserted. The start signal is asserted when all tasks are done processing the previous block of data and are ready to process a new block. Once the task gets the start signal, it resumes the execution of its computations.

The overall monitoring finite state machine, or RC controller shown in Figure 10.12b, is responsible for interacting with the host computer. After the host reset is de-asserted, the RC controller waits until TPdone becomes high, indicating the completion of the temporal partition. It then interrupts the host by asserting the HostInt signal, waits until it gets the HostAck acknowledge signal gets asserted, then goes into an idle state.

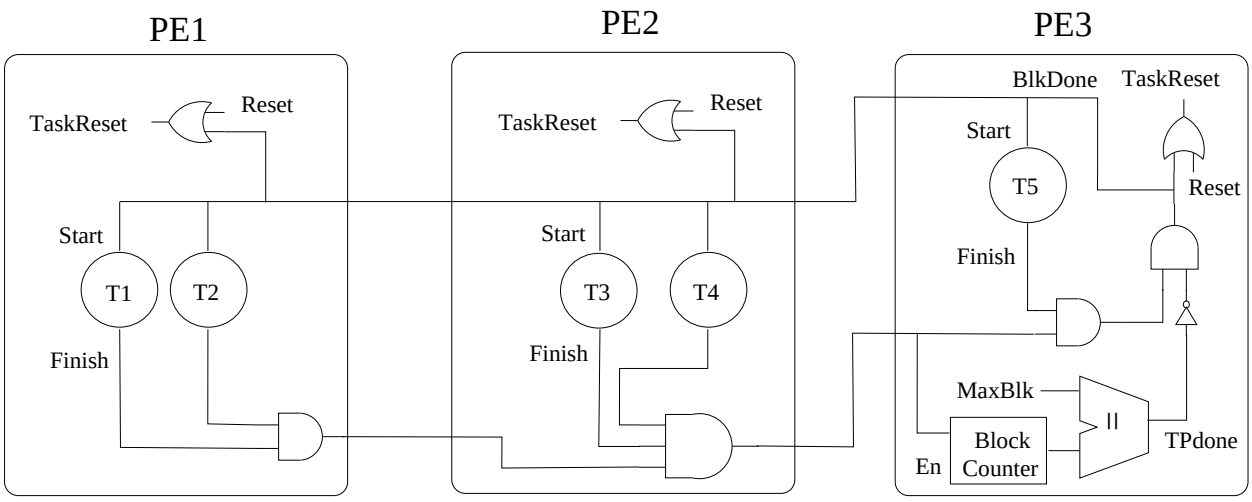


Figure 10.13: Task and RC Control

The TPdone signal is asserted only when all sets of data have been processed in the temporal partition. During partitioning, the spatial partitioning computes the number of data sets that can fit on the memory banks of the RC hardware, and introduces a comparator that checks the total number of data sets computed previously with the value of the current block of data being processed. If the comparison succeeds, the TPdone signal is asserted indicating the completion of the temporal segment. Furthermore, a counter is introduced to keep track of the current set of data. The value of the counter is used in the addressing logic described above to produce the physical address sent to the memory banks.

10.3.2 Lower-Level BBIF Synthesis

To be compatible with the USM specification, the lower-level BBIF representation used during high-level synthesis requires extensions to the design ports of a task. BBIF design ports do not have any associated synthesis semantics and hence are synthesized into hardware as wires. An extension is required to classify a design output port as a *defaulted* output port. Such an output port can be specified to have a particular default value. In every state of the RTL design where the behavior does not write a value to an output port, the port takes a default value according to its type. Four types of defaulted output ports are defined:

1. *Tri-stated*: Each bit of a tri-stated port remains disconnected when it is not accessed. This is implemented in hardware using a collection of tri-state buffers that drive the output port and are controlled by the RTL controller. When multiple data structures

are assigned onto the same physical memory bank port, the data bus lines issued from each accessing task need to be in the default tri-stated mode. All tasks that do not have access to the memory bank, tri-state their output by default so that only the accessing task is allowed to drive the physical data bus of the memory bank.

2. *Registered*: Each bit of a registered port maintains the last assigned value, when it is not accessed. This is implemented in hardware using a hardware register that drives the output port and is enabled by the RTL controller. Furthermore, the registered ports may also have standard set and reset signals, that will also be issued from the RTL controller. In the case of shared channels presented above, a register is inserted at the input of the receiving task. A value sent by the writer is automatically saved on the input of the receiving task allowing future access to that value even if the shared channel changes its value.

3. *Low*: Each bit of a defaulted low port takes the logic zero value when it is not accessed. This is implemented in hardware using a collection of AND-gates that drive the output port and are controlled by the RTL controller. In the case of the $\overline{read}/write$ enable signal that controls the mode of a memory bank, tasks that do not have access to the shared bank, should select the read mode in order to avoid inadvertent writes to the shared memory.

4. *High*: Each bit of a defaulted high port takes the logic one value when it is not accessed. This is implemented in hardware using a collection of OR-gates that drive the output port and are controlled by the RTL controller. Similar to the default low port, the default high is mainly used for $read/\overline{write}$ enable lines where the read mode is represented with a logic one. Note that the defaulted tri-state mode cannot be used for the read/write select signal of the memory because when none of the tasks is accessing the memory, the physical select line driving the memory bank will carry an unknown value which might inadvertently select the write mode.

10.4 Conclusion

This chapter presented the taskgraph model that is used to initially represent a design and transport it through the tasks of temporal and spatial partitioning. The flexibility of the taskgraph notation allows easy additions and extensions to both the design and the RC architecture as well as constraints placed on the solution.

Several synthesis issues were introduced in this chapter. The taskgraph model presented a

framework in which the synthesis issues can easily be incorporated in the overall partitioning and synthesis process. By automatically introducing new tasks and new flags to the taskgraph, access conflicts to physical pins and memory banks are resolved. In addition, the taskgraph easily introduces control circuitry used to monitor the execution of the several temporal segments of the design.

In conclusion, the Unified Specification Model along with the taskgraph representation provided a set of modeling features that are highly-suited for RC design synthesis.

Chapter 11

Conclusion

We have proposed a design framework that bridges the gap between behavioral descriptions of data structures and their hardware implementations on reconfigurable computers. The need to efficiently map data structures onto hardware is accentuated by the increasing complexity of existing reconfigurable architectures. In addition to a rich set of off-chip memories available on RC platforms, the recent capability of embedding a multitude of physical RAM banks into field-programmable devices makes the design automation of data mapping an indispensable task during synthesis. Furthermore, the need for efficient data mapping is driven by the ever-increasing memory requirements in the emerging area of multimedia applications. The overall performance of such applications is severely affected by their memory subsystems.

From addressing data specification at the front-end of the synthesis process to developing data mapping techniques that interact with logic partitioning at the back-end, this work proposed an automated and efficient design methodology that handles the storage requirements of an application as well as the interaction between data placement and logic partitioning.

11.1 Summary of Contributions

Four main aspects of data management were addressed in this work: specifications of data structures at the behavioral level, arbitration of resource accesses, logical-to-physical memory mapping algorithms, and general partitioning techniques that combine data placement with logic partitioning . The following briefly states the important contributions in each topic:

- *Specification Models:* In order to capture the behavior of computational tasks executing in parallel, the Unified Specification Model is introduced. The USM allows an easy

representation of multiple control threads, captures control dependencies between these threads, and offers an easy representation of data structures. Further, the Behavior Blocks Input Format is used to represent each control thread. BBIF modeling is characterized by a hierarchical control data flow graph representation with features fit for high-level RC synthesis. The combination of USM and BBIF representations yields applications that are well-suited for reconfigurable computing synthesis.

- *Resource Arbitration:* Since the quantity of RAM resources is fixed in RC platforms, RAM sharing is required when the number of data structures of an application exceeds the available memory banks. Resource arbitration is utilized to ensure exclusiveness among mutual accesses to the memories while allowing multiple threads of control to execute in parallel. A resource arbitration mechanism, highly suited for field-programmable devices, was introduced and successfully used in an automated high-level synthesis flow for RCs.

- *Data Mapping:* Several techniques were presented that assign data structures to physical RAMs. In all cases, Integer Linear Programming was used to formulate the mapping problem. First, the mapping was restricted to physical RAMs consisting of a variable number of instances and containing a variable number of ports. The banks, however, were limited to a single depth/width configuration. Second, in order to efficiently utilize on-chip RAMs available in recently manufactured programmable devices, the technique was extended to include memory banks that have multiple depth/width configurations. Even though a formulation was implemented, the complexity of the mapping was overloaded by the multiple configurations and the formulation only handled medium-size designs. Finally, by introducing a few assumptions on the way data structures can be partitioned, a new methodology was presented. A layered solution consisting of global mapping followed by detailed mapping was proposed. Not only do the assumptions reduce the complexity of the mapping problem, they also ensure a rigorous partitioning of the data structures: decoding logic and address generation are dramatically simplified.

- *Spatial Partitioning:* In the global view of synthesis, data mapping interacts closely with logic assignment to provide an overall optimized placement and performance of the synthesized design. By combining memory mapping with logic partitioning, the problem becomes more complex; hence a need for fast yet accurate partitioning methodologies.

  The goal of spatial partitioning is to map both data and logic such that the relative placement of data structures with respect to logic tasks is optimized within an RC

system containing multiple processing elements. Three views of the problem were discussed in this work; for each, an ILP-based solution was presented.

11.2 Future Extensions

Further improvements on the above topics are required to accommodate some features of reconfigurable computing. The following is a list of several issues that are extensions to the current work.

Interconnections Constraints The goal of the partitioning techniques in this work was to provide a mapping solution that minimizes the interconnection between all processing elements and memory banks or other processing elements. The formulation did not take into account hard constraints that can exist on the RC system. I.e. it might not be beneficial to minimize the overall number of interconnections when a bottleneck exists only in one location in the RC system. Hence, new variables and constraints need to be introduced to the ILP formulation to handle any hard constraint in the architecture.

Integration of Memory Synthesis with Register Binding in HLS Register binding assigns variables of an application to registers of the hardware. Since accessing off-chip memory banks is usually time-consuming, the HLS process targets on-chip registers to store variables. However, with the advent of fast on-chip RAM banks, it becomes possible to map the variables onto these banks.

By making register binding consider both registers and on-chip RAMs of the programmable device, new solutions can be explored during high-level synthesis.

Block Processing In digital signal processing and several other types of applications, computations are defined on very long streams of data. The design processes one input block at a time and produces the corresponding output block. This methodology, known as *block-processing*, can be efficiently used to increase the throughput of a system in parallel compilers [108] and VLSI processors [109].

While assigning data structures to physical memory banks, additional constraints and objective costs can be added to the ILP formulation to take block-processing into account. The mapper should balance the assignment in order to maximize the number of blocks that can be accessed in the available RAM of the RC platform.

Automatic Arbitration The work presented in Chapter 9 provided a solution for resource arbitration but did not include arbitration in the ILP formulations. For the memory mapping problem to allow overlapping of conflicting data structures onto the same physical memory port, the ILP problem needs to be extended to introduce the cost of arbitration, in terms of area and delay, into the constraints and objective formulations.

Guiding the Formation of Data Structures One very important issue encountered when dealing with memory management is the task of generating data structures by choosing their granularities. For instance, a 3x3 matrix in an application could be represented in several ways on the hardware: as one 9-word logical memory, as three 3-word logical memories, as nine 1-word logical memories, or as a combination of different length logical memories.

Ideally, knowledge of the RC hardware is useful while forming the logical memories of an application. The number and type of available RAMs on the platform can guide the formation of logical memories. However, since the purpose of the high-level specification languages is to hide hardware and implementation details, an efficient formation of the logical memories is not always feasible. To cope with this limitation, the synthesis process can be extended to interface with a memory partitioner to produce new logical memories based on the data access patterns and the hardware availability. Thus, the exploration phase of high-level synthesis would include varying the structure of the logical memories by invoking the memory partitioner.

The memory assignment techniques presented in this work allow the partitioning of the logical memories into chunks that fit in RAM instances. However, this partitioning does not consider the access patterns of the design. In practice, exploiting the pattern of memory accesses can optimize the synthesized design dramatically. Thus, data transformation techniques such as loop interchange, loop fusion, and loop unrolling [110] can be employed to partition the data structures into either finer or coarser logical memories and further exploit parallelism in computational tasks.

Bibliography

[1] Xilinx Corporation. "The Programmable Logic Data Book", 1998.

[2] Altera Corporation. "Reconfigurable Interconnect Peripheral Processor (RIPP10)".

[3] Atmel Corporation. "Data Book and Application Notes".

[4] Annapolis Micro Systems, Inc. "Wildforce multi-FPGA board".

[5] GigaOps. "http://www.gigaops.com".

[6] J. Hauser, J. Wawrzynek. "GARP: A MIPS processor with a Reconfigurable Coprocessor". In *International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Press, April 1997.

[7] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. "On Reconfigurable Coprocessing Units". In *Proceedings of the 5th Reconfigurable Architectures Workshop*, pages 67–72. Springer, March 1998.

[8] T. Miyamori and K. Olukotun. "REMARC: Reconfigurable Multimedia Array Coprocessor". In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*. ACM Press, 1998.

[9] T. Miyamori, K. Olukotun. "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications". In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 2–11. IEEE Computer Society Press, 1998.

[10] E. Waingold et al. "Baring it All to Software: Raw Machines". In *IEEE Computer*, pages 86–93, September 1997.

[11] Xilinx Corporation. "XC6200 FPGAs Product Description", April 1997.

[12] Xilinx Corporation. "Virtex 2.5V Field-Programmable Gate Arrays", October 2000.

[13] S. Trimberger, D. Carberry, A. Johsnon, and J. Wong. "A Time-Multiplexed FPGA". In *Proceedings of the ACM/SIGDA IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 22–28. IEEE Computer Society Press, 1997.

[14] Firefly XC6200-based single-FPGA board by Annapolis Micro Systems, Inc. "http://www.annapmicro.com".

[15] Virtual Workbench Virtex-based Rapid Prototyping Board. "http://www.vcc.com".

[16] TSI TelSys Corporation. "ACEcard Hardware Designer's Manual".

[17] Wildstar Virtex-based multi-FPGA board by Annapolis Micro Systems, Inc. "http://www.annapmicro.com".

[18] S. Trimberger. "Scheduling Designs into a Time-multiplexed FPGA". In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 153–160. ACM Press, 1998.

[19] S. M. Scalera and J. R. Vazquez. "The Design and Implementation of a Context Switching FPGA". In *Proceedings of the ACM/SIGDA IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 78–85. IEEE Computer Society Press, 1998.

[20] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. "A First Generation DPGA Implementation". In *the 3rd Canadian Workshop of Field-Programmable Devices*, May-June 1995.

[21] C. Rupp et al. "The NAPA Adaptive Processing Architecture". In *Proceedings of the Sixth Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, California, April 1998. IEEE Computer Society Press.

[22] G. Lu et al. "MorphoSys: a Reconfigurable Processor Targeted to High Performance Image Applications". In *Proceedings of the 6th Reconfigurable Architectures Workshop*, pages 660–669. Springer, April 1999.

[23] Xilinx Corporation. "http://www.xilinx.com".

[24] A. El Gamal. *"Protozone: The PC-Based ASIC Design Frame - User's Guide"*. Stanford University, Technical Report, Stanford, CA, 1992.

[25] B. Schott. "SLAAC: Presentation at the Loki Team Retreat", February 2000.

[26] Xilinx Corporation. "Virtex-E 1.8V Field-Programmable Gate Arrays", February 2001.

[27] Xilinx Corporation. "Virtex-E 1.8V Extended Memory Field-Programmable Gate Arrays", November 2000.

[28] Xilinx Corporation. "Virtex-II 1.5V Field-Programmable Gate Arrays", July 2002.

[29] Xilinx Corporation. "Virtex-II Pro Platform FPGAs: Introduction and Overview", June 2002.

[30] S. Y. Kung, H. J. Whitehouse, T. Kailath. *"VLSI and Modern Signal Processing"*. Prentice-Hall, Inc., 1985.

[31] L. B. Jackson. *"Digital Filters and Signal Processing"*. Kluwer Academic Publishers, second edition, 1989.

[32] S. R. Park, W. Burleson. "Configuration Cloning: Exploiting Regularity in Dynamic DSP Architectures". In *Proceedings of the ACM Symposium on FPGAs*, pages 81–89. ACM Press, 1999.

[33] D. E. Goldberg. *"Genetic Algorithms in Search, Optimization, and Machine Learning"*. Addison-Wesley, Reading, MA, 1989.

[34] J. M. Zurada. *"Introduction to Artificial Neural Systems"*. West Publishing Company, 1992.

[35] J. Koza et al. . "Evolving Computer Programs Using Rapidly Reconfigurable Field-Programmable Gate Arrays and Genetic Programming". In *Proceedings of the ACM Sixth International Symposium on Field Programmable Gate Arrays*. ACM Press, 1998.

[36] P. Zhong et al. "Accelerating Boolean Satisfiability with Configurable Hardware". In *Proceedings of the Sixth Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 186–195, Napa, California, April 1998. IEEE Computer Society Press.

[37] J. Eldredge and B. Hutchings. "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration". In *Proceedings of the Second Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 180–188, Napa, California, April 1994. IEEE Computer Society Press.

[38] T. Kean and A. Duncan. "A 800Mpixel/sec Reconfigurable Image Correlator on XC6216". In *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, 1997.

[39] K. Simha. "NEBULA: A Partially and Dynamically Reconfigurable Architecture". Master's thesis, University of Cincinnati, 1998.

[40] S. Goldstein and H. Schmit. "Reconfigurable Computing Seminar". In *Course 15-828/18-847*, http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15828-s98/www, Spring 1998.

[41] Altera Corporation. "FLEX 10K Embedded Programmable Logic Family Data Sheet", May 2000.

[42] Altera Corporation. "APEX 20K Programmable Logic Device Family Data Sheet", March 2000.

[43] M. Weinhardt and W. Luk. "Memory Access Optimization and RAM Inference for Pipeline Vectorization". In *Proceedings of International Workshop on Field-Programmable Logic and Applications*, pages 61–70. Springer, September 1999.

[44] C. J. Tseng and D. Siewiorek. "Automated Synthesis of Data Paths in Digital Systems". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 5, pages 379–395, July 1986.

[45] T. Kim and C. L. Liu. "Utilization of Multiport Memories in Data Path Synthesis". In *Proceedings of the 30th Design Automation Conference*, pages 298–302. ACM Press, June 1993.

[46] M. Balakrishnan, et al. "Allocation of Multiport Memories in Data Path Synthesis". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 7, pages 536–540, April 1988.

[47] I. Ahmad and C. Y. Chen. "Post-Process for Data Path Synthesis". In *Proceedings of International Conference on Computer Aided Design*, pages 276–279. ACM Press, 1991.

[48] J. Cong and K. Yan. "Synthesis for FPGAs with Embedded Memory Blocks". In *Proceedings of International Symposium on Field Programmable Gate Arrays*, pages 75–81. ACM press, February 2000.

[49] S. Wilton. "Heterogeneous Technology Mapping for FPGAs with Dual-Port Embedded Memory Arrays". In *Proceedings of International Symposium on Field Programmable Gate Arrays*, pages 67–74. ACM press, February 2000.

[50] S. Wilton. *"Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory"*. PhD thesis, University of Toronto, 1997.

[51] W. Ho and S. Wilton. "Logical-to-Physical Memory Mapping for FPGAs with Dual-Port Embedded Arrays". In *Proceedings of International Workshop on Field-Programmable Logic and Applications*, pages 111–123. Springer, September 1999.

[52] D. Karchmer and J. Rose. "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems". In *Proceedings of International Conference on Computer Aided Design*, pages 20–26. ACM Press, November 1994.

[53] P. Jha and N. Dutt. "High-Level Library Mapping for Memories". In *ACM Transactions on Design Automation of Electronic Systems*, pages 566–603. ACM Press, July 2000.

[54] F. Catthoor, et al. *"Custom Memory Management Methodology"*. Kluwer, 1998.

[55] P. R. Panda, N. Dutt, A. Nicalau. *"Memory Issues In Embedded Systems-On-Chip"*. Kluwer, 1999.

[56] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, 1994.

[57] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *"High-Level Synthesis, Introduction to Chip and System Design"*. Kluwer Academic Publishers, 1992.

[58] P. Hansen, B. Jaumard, and V. Mathon. "Constrained Nonlinear 0-1 Programming". In *ORSA Journal of Computing*, volume 5, pages 97–119, 1993.

[59] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures". In *Proceedings of the 5th Reconfigurable Architectures Workshop*, pages 31–36. Springer, March 1998.

[60] N. Narasimhan. *"Formal-Assertions Based Verification in a High-Level Synthesis System"*. PhD thesis, University of Cincinnati, ECECS Department, 1998.

[61] U. Steinhausen, R. Camposano, et al. "System-Synthesis Using Hardware / Software Codesign". In *International Workshop on Hardware-Software Co-Design*, October 1993.

[62] S. P. Levitan et al. "Using VHDL as a Language for Synthesis of CMOS VLSI Circuits". In *Proceedings of Computer Hardware Description Languages and their Applications*, pages 331–346. Elsevier, June 1989.

[63] M. J. Farland. *"Value Trace"*. Carnegie Mellon University, Internal Report, Pittsburgh, PA, 1978.

[64] P. Harper, S. Krolikoski, and O. Levia. "Using VHDL as a Synthesis Language in the Honeywell VSYNTH System". In *Proceedings of Computer Hardware Description Languages and their Applications*, pages 315–330. Elsevier, June 1989.

[65] A.E. Casavant, D.D. Gajski, and D.J. Kuck. "Automatic Design with Dependence Graph". In *17th Design Automation Conference"*, pages 506–515, 1980.

[66] J. Roy, N. Kumar, R. Dutta, and R. Vemuri. "DSS: A Distributed High-Level Synthesis System". In *IEEE Design and Test of Computers*, June 1992.

[67] R. Vemuri, H. Carter, and P. Alexander. "Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment". In *Proceedings of First Annual RASSP Conference*, August 1994.

[68] D.E. Thomas et al. *"Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench"*. Kluwer Academic Publishers, 1990.

[69] ILOG Incorporation. *"Using the CPLEX Callable Library"*. http://www.cplex.com.

[70] I. Ouaiss and R. Vemuri. "Hierarchical Memory Mapping During Synthesis in FPGA-Based Reconfigurable Computers". In *Proceedings of Design Automation and Test in Europe*. IEEE Computer Society Press, March 2001.

[71] F. Glover and E. Woolsey. "Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program". In *Operations Research*, volume 21 (1), pages 156–161, 1974.

[72] A. Kasat, I. Ouaiss, and R. Vemuri. "Memory Synthesis for FPGA Based Reconfigurable Computers". In *Proceedings of International Workshop on Field-Programmable Logic and Applications*. Springer, August 2001.

[73] V. Srinivasan, S. Radhakrishnan, R. Vemuri, and J. Walrath. "Interconnect Synthesis for Reconfigurable Multi-FPGA Architectures". In *Proceedings of the 6th Reconfigurable Architectures Workshop*, pages 597–605. Springer, April 1999.

[74] I. Ouaiss and R. Vemuri. "Global Memory Mapping for FPGA-Based Reconfigurable Systems". In *Proceedings of the 8th Reconfigurable Architectures Workshop*. IEEE CS Press, April 2001.

[75] B. Preas and M. Lorenzetti (editors). *"Physical Design Automation for VLSI Systems"*. Benjamin Cummings Pub., 1988.

[76] J. Holland. *"Adaptation in Natural and Artificial Systems"*. University of Michigan Press, 1975.

[77] R. Vemuri. *"Genetic Algorithms for Partitioning, Placement, and Layer Assignment for Mutichip Modules"*. PhD thesis, University of Cincinnati, ECECS Department, 1994.

[78] J. Cohoon and W. Paris. "Genetic Placement". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 6 (6), pages 956–964, November 1999.

[79] K. Shahookar and P. Mazumdar. "A Genetic Approach to Standard Cell Placement Using Metagenetic Parameter Optimization". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 9 (5), pages 500–511, November 1990.

[80] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In *Science*, volume 220 (4598), pages 671–680, 1983.

[81] A. A. Duncan, D. C. Hendry and P. Gray. "An Overview of the Cobra-ABS High-Level Synthesis System for Multi-FPGA Systems". In *Proceedings of FPGAs for Custom Computing Machines*, pages 106–115, Napa Valley, California, 1998. IEEE Computer Society Press.

[82] K. Roy-Neogi and C. Sechen. "Multiple FPGA Partitioning with Performance Optimization". In *Proceedings of the Third International Symposium on FPGAs*, pages 146–151, 1995.

[83] W. Sun and C. Sechen. "Efficient and Effective Placements for Very Large Circuits". In *IEEE International Conference on CAD*, pages 170–177, 1993.

[84] B. W. Kernighan and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs". In *Bell Systems Technical Journal*, volume 49 (2), pages 291–307, 1970.

[85] C. Fiduccia and R. Mattheyses. "A Linear Time Heuristic for Improving Network Partitions". In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[86] B. Krishnamurthy. "An Improved Min-cut Algorithm for Partitioning VLSI Networks". In *IEEE Transactions on Computers*, volume 33 (5), pages 438–446, 1984.

[87] L. A. Sahchis. "Multiple-Way Network Partitioning". In *IEEE Transactions on Computers*, volume 38 (1), pages 62–81, 1989.

[88] N. Kumar. *"High-Level VLSI Synthesis for Multichip Designs"*. PhD thesis, University of Cincinnati, ECECS Department, 1994.

[89] S. C. Johnson. "Hierarchical Clustering Schemes". In *Psychometrika*, volume 32 (3), September 1967.

[90] C. J. Alpert and A. B. Kahng. "Geometric Embeddings for Faster and Better Multi-Way Netlist Partitioning". In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 743–748, 1993.

[91] T. Bui et al. "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms". In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 773–778, 1989.

[92] H. Shin and C. Kim. "A Simple yet Effective Technique for Partitioning". In *IEEE Transactions on VSLI Systems*, volume 1 (3), 1993.

[93] C. J. Alpert and A. B. Kahng. "Recent Directions in Netlist Partitioning: A Survey". In *Integration, the VLSI Journal*, volume 19 (1-2), pages 1–81, 1995.

[94] R. J. Trudeau. *"Introduction to Graph Theory"*. Dover Books on Advanced Mathematics, 1993.

[95] V. Srinivasan. *"Partitioning for FPGA-Based Reconfigurable Computers"*. PhD thesis, University of Cincinnati, ECECS Department, 1999.

[96] S. Walters. "Computer-Aided Prototyping for ASIC-Based Systems". In *IEEE Design and Test of Computers*, June 1992.

[97] Brian Box. "Field Programmable Gate Array Based Reconfigurable Preprocessor". In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[98] J. Babb, R. Tessier, A. Agarwal. "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators". In *Proceedings of FPGAs for Custom Computing Machines*, 1993.

[99] Frank Vahid. "Techniques for Minimizing and Balancing I/O During Functional Partitioning". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 18, January 1999.

[100] J. Rabaey. *"Digital Integrated Cicuits: A Design Perspective"*. Prentice Hall, 1996.

[101] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. "A Unified Specification Model of Concurrency and Coordination for Synthesis from VHDL". In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis*, July 1998.

[102] C. Hoare. "Communicating Sequential Processes". In *ACM Communications*, volume 21, pages 666–677, 1978.

[103] A. Silberschatz, P. Galvin. *"Operating System Concepts"*. Addison-Wesley, 4th edition, 1994.

[104] IEEE Standard VHDL Language Reference Manual. IEEE Standards Office. New York, NY, 1993.

[105] M. Kaul. *" Optimal and Near-Optimal Temporal Partitioning Techniques for Reconfigurable Computers"*. PhD thesis, University of Cincinnati, ECECS Department, 2000.

[106] S. Govindarajan. *"Algorithms for Design Space Exploration and High-level Synthesis for Mutli-FPGA Reconfigurable Computers"*. PhD thesis, University of Cincinnati, ECECS Department, 2000.

[107] N. A. Sherwani. *"Algorithms for VLSI Physical Design Automation"*. Kluwer Academic Publishers, 1993.

[108] M. Wolf. *"High Performance Compilers for Parallel Computing"*. Addison-Wesley Publishers, 1996.

[109] S. Y. Kung. *"VLSI Array Processors"*. Prentice Hall, 1988.

[110] D. Bacon, S. Graham, and O. Sharp. "Compiler Transformations for High-Performance Computing". In *ACM Computing Surveys*, volume 26, pages 345–420, December 1994.

Appendix A

Nomenclature

This appendix provides a description summary of the important variables, parameters, functions, and indices defined in the ILP formulations of the thesis.

The following parameters are used in ILP formulations. Their values are explicitely stated in the problem definition:

$C(f)$ = The total area of processing element f.

C_t = The total number of depth/width configurations for each instance of memory bank type t.

CT = The list of computational tasks in the design, where $CT = \{CT_1, CT_2, ..., CT_L\}$

D_d = The number of words in data segment d.

D_t = Array of number of words on each instance of memory bank type t, where $D_t = \{d_1, d_2, ..., d_{C_t}\}$

I_t = The number of instances of memory bank type t.

DS = The set of all data structures in the design. There are a total of M data structures such that: $DS = \{DS_1, DS_2, ..., DS_M\}$.

(DS_x, DS_y) = Lifecycle conflict pair between the two data segments x and y.

$E(l)$ = The estimated area of computational task l.

K = Constant reduction factor for additional area required for addressing logic, arbitration logic, and routing purposes.

P_t = The number of ports on each instance of memory bank type t.

PB = The set of all physical memory bank *types* in the RC platform. There are a total of N types such that: $PB = \{PB_1, PB_2, ..., PB_M\}$.

$PE[l]$ = The processing element in which computational task l is placed.

RA_d = In the case of a single processing element, the total number of estimated reads from data segment d.

RA_d = In the case of multiple processing elements, the total number of estimated reads by each processing unit from data segment d, where $RA_d = \{RA_{d1}, RA_{d2}, ..., RA_{dL}\}$

RL_t = Number of clock cycles for the read latency of memory bank type t.

T_t = In the case of a single processing element, the number of pins traversed from the processing unit to an instance in memory bank type t.

T_t = In the case of multiple processing elements, the number of pins traversed from each processing unit to an instance in memory bank type t, where $T_t = \{T_{t1}, T_{t2}, ..., T_{tF}\}$

W_d = The number of bits per word in data segment d.

W_t = Array of number of bits per word on each instance of memory bank type t, where $W_t = \{w_1, w_2, ..., w_{C_t}\}$

WA_d = In the case of a single processing element, the total number of estimated writes to data segment d

WA_d = In the case of multiple processing elements, the total number of estimated writes by each processing unit to data segment d, where $WA_d = \{WA_{d1}, WA_{d2}, ..., WA_{dL}\}$

WL_t = Number of clock cycles for the write latency of memory bank type t.

The following parameters are used in ILP formulations but their values are computed before executing the ILP solver:

CD_{dt} = Estimated consumed number of words if data segment d is assigned to physical memory bank type t.

CP_{dt} = Estimated number of consumed ports if data segment d is assigned to physical memory bank type t.

CW_{dt} = Estimated consumed word width (in bits) if data segment d is assigned to

physical memory bank type t.

DP_{dt} = Estimated number of consumed ports in the last row of partially used instances if data segment d is assigned to physical memory bank type t.

FP_{dt} = Estimated number of consumed ports in fully used instances if data segment d is assigned to physical memory bank type t.

PP_{dt} = Estimated number of consumed ports in partially used instances if data segment d is assigned to physical memory bank type t.

WDP_{dt} = Estimated number of consumed ports in the bottom right corner partially used instance if data segment d is assigned to physical memory bank type t.

WP_{dt} = Estimated number of consumed ports in the last column of partially used instances if data segment d is assigned to physical memory bank type t.

κ_{dt} = Upperbound for the M_{dt} ILP variable.

σ_{lf} = Upperbound for the N_{lf} ILP variable.

λ_{dt} = Upperbound for the G_{dt} ILP variable.

The following parameters are used in ILP formulations for finding the best configuration for each data segment and physical memory bank type pair:

$\#AddressNets$ = Total number of address lines for all instances assigned to a data segment.

$\#BitsPerMuxInput$ = Number of bits per input for each multiplexer introduced due to addressing logic.

$\#ColumnWastedBits$ = Total number of bits wasted in the last column of assigned instances due to unmatched word widths.

$\#Instances$ = Number of instances assigned to the data structures.

$\#Muxes$ = Number of multiplexers introduced due to addressing logic.

$\#MuxInputs$ = Number of inputs for each multiplexer introduced due to addressing logic.

$\#MuxSelects$ = Number of select lines for each multiplexer introduced due to addressing logic.

$\#RowWastedBits$ = Total number of bits wasted due to rounding in the last

row of assigned instances.

$\#UsedPorts$ = Total number of ports consumed by a data segment.

$\#Vs\_set\_to\_one$ = Number of instances that are partially occupied by a data segment.

The following variables are used in ILP formulations and their value is set by the ILP solver:

F_c = ILP variable that selects the best configuration for a data segment and physical memory bank type pair.

G_{dt} = ILP variable used to linearize the pin I/O cost with respect to dt in the merged spatial partitioning paradigm.

M_{dt} = ILP variable used to linearize the pin delay cost with respect to dt in the merged spatial partitioning paradigm.

N_{lf} = ILP variable used to linearize the pin delay cost with respect to lf in the merged spatial partitioning paradigm.

V_{dip} = ILP variable that assigns the remaining parts of data segment d to *parts* of instance i on a pre-selected physical memory bank type.

V_{dtip} = ILP variable that assigns the remaining parts of data segment d to *parts* of instance i on physical memory bank type t.

X_{di} = ILP variable that assigns a part of data segment d to the *entire* instance i for a pre-selected physical memory bank type.

X_{dti} = ILP variable that assigns a part of data segment d to the *entire* instance i of physical memory bank type t.

X_{dtip} = ILP variable that assigns a part of data segment d to port p of instance i on physical memory bank type t.

Y_{tipc} = ILP variable that assigns configuration c to port p of instance i on physical memory bank type t.

Z_{dt} = ILP variable that assigns data segment d to physical memory bank type t.

The following are the important indices used in the variables and parameters described above:

b = refers to the best configuration chosen for a data structure and physical

memory bank type pair.

c = refers to a depth/width configuration of a physical memory bank type.

d = refers to a data structure.

e = refers to an equivalent data structure.

f = refers to a processing element in the RC platform.

i = refers to an instance of a physical memory bank type.

l = refers to a computational task.

p = refers to a port of an instance of a physical memory bank type.

t = refers to a physical memory bank type.

The following function is used in the pre-processing of ILP formulations:

$consumed\_ports()$ = Function that returns the estimated number of consumed ports based on the depth of data segment d, the depth and the number of ports in instances of physical memory bank type t

Appendix B

BBIF Specification

The following sections provide details on the BBIF model. A formal and more detailed description of the BBIF model is available in [60].

B.1 BBIF Model and Formal Notations

A BBIF model can be represented as a four-tuple:

$$BBIF < Blocks,\ ControlDeps,\ IN_{ports},\ OUT_{ports} >$$

where *Blocks* is a set of behavior blocks, *ControlDeps* is a set of *control dependency* edges, where each edge $< B_i, B_j >$ represents the control flow from block B_i to B_j, and IN_{ports} *and* OUT_{ports} represent the set of design input and output ports respectively.

In the BBIF model the atomic storage element is a *carrier* represented as a tuple,

$$Carrier < Id,\ Width >$$

consisting of an index *Id* and a non-zero positive integer *Width*. The carrier *Id* is a unique index used for carrier set operations such as union and intersection. The design input and output ports are essentially carriers sets. A behavior block is an 8-tuple

$$BehaviorBlock < BlkId,\ Type,\ \mathcal{I},\ \mathcal{O},\ \mathcal{L},\ \mathcal{C},\ \mathcal{F},\ FG >$$

consisting of a block index ($BlkId$), a block type ($Type$), five carrier sets and a flow graph (FG). A behavior block can be either of type *compute* or *io*. Computations in a task are

specified only within the *compute* blocks and interaction with the environment through the design ports are specified only within *io* blocks. The five carrier sets are:

- The set $\mathcal{I}(B_i)$ represents the set of *input* carriers of block B_i. These are input carriers that are passed from every parent block that branches to block B_i.
- The set $\mathcal{O}(B_i)$ represents the set of *output* carriers of block B_i. These are output carriers that are passed through the branches to every child of block B_i.
- The set $\mathcal{L}(B_i)$ represents the set of *local* carriers of block B_i. These carriers are visible only within block B_i and are used to capture the data flow across computations within the block.
- The set $\mathcal{C}(B_i)$ represents the set of *constants* that are visible only within block B_i. A constant is essentially a carrier with an additional *string* field that represents the actual constant value.
- The set $\mathcal{F}(B_i)$ represents the set of *flag* carriers of block B_i. These carriers are visible only within block B_i and are used to hold the resulting values of conditional expressions in the behavior. The flags are used for conditional branching at the end of the block.

In addition, a behavior block also consists of a *flow graph* FG, represented as a tuple,

$$FG < OprNodes,\ DataFlowDeps >$$

consisting of *operation nodes* ($OprNodes$) and *data dependency* edges ($DataFlowDeps$). An operation node is a 5-tuple

$$Operation < OprId,\ OprType,\ Inputs,\ Outputs,\ ConDeps >$$

consisting of a unique operation index ($OprId$), the operation type ($OprType$), the *input* carrier set, the *output* carrier set and an explicit control dependency set ($ConDeps$). Each operation O_i, has a set of input carriers $Inputs(O_i)$ that are *read* and a set of output carriers $Outputs(O_i)$ that are *written*. Both these sets may contain zero or more carriers. A data *flow dependency* is a directed edge between a parent operation O_i and a child operation O_j, represented as a tuple $< O_i,\ O_j >$. This dependency edge exists if and only if the following condition is satisfied:

$$DataFlowDependency < O_i,\ O_j > \iff (Outputs(O_i) \cap Inputs(O_j)) \neq \emptyset$$

The operation nodes in a block follow *single assignment semantics* by writing exactly once to a particular carrier. In other words, any *output, local, or flag* carrier in a block will appear exactly in only one *output* carrier set of an operation node. Therefore, there are no *anti* or *output* dependencies and the order of operations in the BBIF specification does not matter in deriving the flow graph.

The operations in the BBIF are classified as *pre-defined* and *user-defined*. There are three pre-defined operation types in the BBIF. The *io* block supports the two types namely, `getport` and `putport` to facilitate design port accesses. The third pre-defined operation type is the `transfer` that is supported in any behavior block. The transfer operation denotes an assignment of one carrier to another, and corresponds to an assignment statement in the behavior. The user-defined operations are uninterpreted. In other words, the synthesis system does not attach any functional semantics to the user-defined operations and expects the user to specify a component library that supports these operations.

B.2 Translation and Profiling

Figure B.1 shows the VHDL specification of an ALU example. The ALU takes two data inputs and a mode of operation, and generates a result. Depending on the mode of operation the ALU generates the *sum*, *difference*, *product*, or the *sum of squares* of the inputs. Figure B.2 shows the BBIF that was automatically translated from the VHDL specification of the ALU. The start *io* block `Blk_2` reads the design ports into the corresponding carriers and passes them to `Blk_3`. The TRUE_BRANCH statement represents an unconditional branch to a subsequent block. `Blk_3` has three inputs `a, b` and `m`, whereas `Blk_2` does not have any. The *compute* block `Blk_3` performs all the condition evaluations of the case statement and generates three flags. Based on the values of these flags, four branches arise from `Blk_3` leading to the blocks `Blk_4,Blk_5, Blk_6` and `Blk_7`. The four blocks perform the four types of the ALU operations. For example, the sum of squares is performed in the three operation statements of `Blk_7`. Each of these four blocks call the *io* block `Blk_8` to write the results to the output port. Note that `Blk_8` calls the start block `Blk_2` forming an overall infinite loop that represents the implicit loop of corresponding VHDL process.

```
entity ALU is
  port ( Data1, Data2 : in integer;
         Mode : in bit_vector(1 downto 0);
         RESULT : out integer
       );
end ALU;

architecture behavior of ALU is
begin
  compute: process
    variable A, B, value : integer;
    variable M : bit_vector(1 downto 0);
  begin
    A := Data1;
    B := Data2;
    M := Mode;
    case M is
      when "00" => value := A + B;
      when "01" => value := A - B;
      when "10" => value := A * B;
      when others => value := (A * A) + (B * B);
    end case;
    RESULT <= value;
  end process;
end behavior;
```

Figure B.1: VHDL Specification of an ALU

```
(INPORT (data1 16) (data2 16) (mode 2))
(OUTPORT (result 16))

(BB Blk_2
    (LOCAL (a 16) (b 16) (m 2))
1  (GET_PORT (data1) (a)) ()
2  (GET_PORT (data2) (b)) ()
3  (GET_PORT (mode) (m)) ()
(TRUE_BRANCH Blk_3(a b m))
)
(BB Blk_3 ( (a 16) (b 16) (m 2) )
    (LOCAL (flag_1 1) (flag_2 1) (flag_3 1))
    (CONSTANT (c12 2 00) (c15 2 01) (c18 2 10))
4  (eq (m c12) (flag_1)) ()
5  (eq (m c15) (flag_2)) ()
6  (eq (m c18) (flag_3)) ()
(flag_1 Blk_4(a b)
(flag_2 Blk_5(a b)
(flag_3 Blk_6(a b) Blk_7(a b))))
)
(BB Blk_4 ( (a 16) (b 16) )
    (LOCAL (value 16))
7  (plus (a b) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
⋮
(BB Blk_6 ( (a 16) (b 16) )
    (LOCAL (value 16))
9  (mult (a b) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
(BB Blk_7 ( (a 16) (b 16) )
    (LOCAL (t22 32) (t23 32) (value 16))
10  (mult (a a) (t22)) ()
11  (mult (b b) (t23)) ()
12  (plus (t22 t23) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
(BB Blk_8 ( (value 16) )
13  (PUT_PORT (value result) ()) ()
(TRUE_BRANCH Blk_2())
)
```

Figure B.2: BBIF Specification of the ALU Example

B.3 Component Library and Functional Unit Instantiation

The component library (C_{lib}), supplied by the user, specifies a list of combinational and sequential components with a list of operations supported by them. For every *user-defined* operation in the BBIF, there should exist at least one component in the library that supports that operation. For sequential components and for components that can support multiple operations, the user is also expected to provide the control signal that facilitates the selection of each of these operations. This information will be used by the synthesis system while generating the control logic.

Figure B.3 shows a portion of a typical component library. The class of a component denotes whether it is combinational (denoted by ALU) or sequential (denoted by REG). The first component `compare`, supports multiple operation types as specified in its MODE field, and its CONTROL field provides the control signal information for each operation type it supports. Components can be parameterized over their port sizes as well as over the number of ports. In the figure, the component that supports the bit-wise `and` operation is parameterized both on the number of inputs and port widths. Since all user-specified operations in the input description are uninterpreted, the library should provide all relevant information that the synthesis process might subsequently require. The SIGNATURE field specifies the ports of the component that are used to support an operation type. For example, `compare` has two inports and three outports while one of its operation type, `grt` uses the first two inports to read inputs and uses the third outport to write the output.

Given a BBIF specification and a component library, the HLS system performs *resource set generation*. For each unique operation type in the BBIF, one or more *functional units* are generated from the parameterized components in the given library. A functional unit is a library component that is instantiated with specific values to its generic parameters. This is done by matching the type of each BBIF operation with the MODE field of each component. Consequently, from the input and output carrier sets of the BBIF operation the generic parameters of the component are instantiated, resulting in a new functional unit. For example, the `eq` operations 4, 5 and 6 of block `Blk_3` in Figure B.2 would lead to a functional unit instantiation from component `compare` with generic parameter values of `width1` = 2, and `width2` = 1. The functional units are unique with respect to the component name and the parameter values. If *resource folding* needs to be performed, functional units may remain unique only based on the component name.

```
⋮
(COMP compare (width1 width2)
  (CLASS ALU)
  (MODE less grt eq)
  (INPORT (a width1) (b width1))
  (OUTPORT (c width2) (d width2) (e width2))
  (SIGNATURE
   (less (1 2) (1))
   (grt (1 2) (2))
   (eq (1 2) (3))
  )
  (CONTROL 2 (NOP 00) (less 01) (grt 10) (eq 11))
)

(COMP and (ins width1 width2)
  (CLASS ALU)
  (MODE and)
  (INPORT (ins width1))
  (OUTPORT (out width2))
  (CONTROL)
)

(COMP REG (bitwidth)
  (CLASS REGISTER)
  (MODE reg)
  (INPORT (input bitwidth))
  (OUTPORT (output bitwidth))
  (CONTROL 2 (NOP 00) (LOAD 01) (RESET 10))
)
⋮
```

Figure B.3: Snapshot of a Typical Component Library

Appendix C

Arbiter Generation Script

A PERL script was developed to automatically generate arbiters. The script is shown in Figures C.1, C.2, and C.3. It takes the number of tasks to be arbitrated (N) as input and it generates a corresponding VHDL file. The generator also has the option to produce different encoding schemes for the FSM (e.g. one-hot encoding, compact encoding, or synthesis tool's default encoding). The output of the script is a VHDL file containing the arbiter entity and its architecture.

```
# ***************************************************************
*
# Author:         Iyad Ouaiss
# File:           generate_arbiter.pl
# Description:    Generates an N-bit arbiter based on N and the
#                 encoding style requested;  The result is a VHDL
#                 file.
# ***************************************************************
*

if ($#ARGV != 2) {die "\nUsage: $0 number_of_bits encoding output_file\n\nWhere e
ncoding is:\n0 Encoding is not specified\n1 One-hot encoding\n2 Compact encoding\n\n";}

$N = $ARGV[0];
$NM1 = $N - 1;

$encoding = $ARGV[1];
$outfile = $ARGV[2];

open (OUTPUT, ">$outfile") || die "Error: Cannot open output file!!\n";

$string = 'type state_type is (F0, T0 ';
for ($i=1 ; $i<$N ; $i++) {
    $string = join (',', $string, " F$i, T$i");
}
$string = join ('', $string, ");");

if ($encoding == 1) {
    # This is for one hot encoding:
    $string2 = '';
    $M = 2 * $N;
    for ($i=0 ; $i<$M ; $i++) {
        $dec = pack('I', 2**$i);
        $bin = unpack('B*', $dec);
        $new = substr ($bin, -$M, $M);
        $string2 = join ('', $string2, $new, " ");
    }
}
else {
    # This is for compact encoding:
    $string2 = '';
    $M = 2 * $N;
    $Pfloat = log($M) / log(2);
    $Pint = sprintf ("%d", $Pfloat);
    $P = ($Pfloat > $Pint) ? ($Pint + 1) : $Pint;
    for ($i=0 ; $i<$M ; $i++) {
        $dec = pack('I', $i);
        $bin = unpack('B*', $dec);
        $new = substr ($bin, -$P, $P);
        $string2 = join ('', $string2, $new, " ");
    }
}
```

Figure C.1: PERL Script for Arbiter Generation (Part I)

```
$string3 = '';
for ($i=0 ; $i<$N ; $i++) {$string3 = join ('', $string3, "0");}

$string4 = ($encoding == 0) ? "--" : "";

print OUTPUT <<END1;
library IEEE;

use IEEE.std_logic_1164.all;

entity Arbiter$N is
   port (CLK : std_logic;
         RST : std_logic;
         Req : std_logic_vector ($NM1 downto 0);
         Ack : out std_logic_vector ($NM1 downto 0));
end Arbiter$N;

architecture structural of Arbiter$N is

$string

-- Do not use the following 2 lines for automatic FSM extraction:
$string4  attribute ENUM_ENCODING: STRING;
$string4  attribute ENUM_ENCODING of state_type: type is "$string2"
;


signal state : state_type;

begin
  process (CLK,RST)
  begin
    if RST= '1' then
      state <= F0;
      Ack <= "$string3";
    elsif CLK= '1' and CLK'event then
   case state is
END1

for ($n=0 ; $n<$N ; $n++) {
   print OUTPUT "\twhen F$n =>\n";
   for ($i=0 ; $i<$N ; $i++) {
      $nmod = ($n + $i) % $N;
            if ($i == 0) {
              print OUTPUT "\t  if (Req($nmod) = '1') then\n";
            }
            else {
              print OUTPUT "\t  elsif (Req($nmod) = '1') then\n";
            }
```

Figure C.2: PERL Script for Arbiter Generation (Part II)

```
        print OUTPUT "\t   state <= T$nmod;\n";
        $dec = pack ('I', (2**$nmod));
        $bin = unpack ('B*', $dec);
        $ack = substr ($bin, -$N, $N);
        print OUTPUT "\t   Ack  <= \"$ack\";\n";
    }
    print OUTPUT "\t  else\n";
    print OUTPUT "\t   state <= F$n;\n";
    $dec = pack ('I', 0);
    $bin = unpack ('B*', $dec);
    $ack = substr ($bin, -$N, $N);
    print OUTPUT "\t   Ack  <= \"$ack\";\n";
    print OUTPUT "\t  end if;\n";


    print OUTPUT "\twhen T$n =>\n";
    for ($i=0 ; $i<$N ; $i++) {
        $nmod = ($n + $i) % $N;
        if ($i == 0) {
           print OUTPUT "\t  if (Req($nmod) = '1') then\n";
        }
        else {
           print OUTPUT "\t  elsif (Req($nmod) = '1') then\n";
        }
        print OUTPUT "\t   state <= T$nmod;\n";
        $dec = pack ('I', (2**$nmod));
        $bin = unpack ('B*', $dec);
        $ack = substr ($bin, -$N, $N);
        print OUTPUT "\t   Ack  <= \"$ack\";\n";
    }
    $nmod = ($n + 1) % $N;
    print OUTPUT "\t  else\n";
    print OUTPUT "\t   state <= F$nmod;\n";
    $dec = pack ('I', 0);
    $bin = unpack ('B*', $dec);
    $ack = substr ($bin, -$N, $N);
    print OUTPUT "\t   Ack  <= \"$ack\";\n";
    print OUTPUT "\t  end if;\n";
}


print OUTPUT<<END2;
      end case;
    end if;
  end process;
end structural;
END2

close OUTPUT;
exit;
```

Figure C.3: PERL Script for Arbiter Generation (Part III)