1-1-2003

Hardware software partitioning using directed acyclic data dependence graph with precedence

Matthew Jin
Ryerson University

Hardware Software Partitioning Using Directed Acyclic Data Dependence Graph with Precedence

By

Matthew Jin

BASc. University of Toronto
2001

A thesis presented to Ryerson University in partial fulfillment to the requirements for the degree of
Master of Applied Science
in the Program of
Electrical and Computer Engineering.

Toronto, Ontario, Canada, 2003-09-10

Instructions on Borrowers

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

In this thesis, we present a system partitioning technique that employs C/C++ as input specification language for hardware/software co-design. The proposed algorithm is able to explore a number of partitioning solutions as compared to other partitioning research. This benefit is obtained by processing data dependency and precedence dependency simultaneously in a new representation called Directed Acyclic Data dependency Graph with Precedence (DADGP). DADGP is an extension of Directed Acyclic Graph (DAG) structure frequently used in the past for partitioning.

The DADGP based partitioning algorithm minimizes communication overhead, overall system execution time as well as system cost in terms of hardware area. The algorithm analyzes the DADGP and tries to expose parallelism between processing elements and repeated tasks. The benefits of exposing parallelism with minimum inter PE communication overhead are shown in the experimental results. However, such benefits come with increase in cost due to additional hardware units and their inter-connections. DADGP-based partitioning technique is also employed to implement block matching and SOBEL edge detection techniques. Overall, the proposed system partitioning algorithm is fast and powerful enough to handle complicated and large system designs.

Acknowledgements

I would like to acknowledge and give many thanks to Professor G. N. Khan for his patience and support. Without him I could not have accomplished this work. His constant guidance and professionalism has been the driving force of my research. I would also like to acknowledge the financial support from the NSERC research discovery grant awarded to my supervisor, and Canadian Microelectronic Corporation (CMC) for providing Rapid Prototyping Platform as well as co-design tools.

My special thanks to the professors and the department for their feedback and participation before and during my thesis defense. I am also grateful to the School of Graduate Studies and the Department of Electrical and Computer Engineering of Ryerson University for their support financially and academically.

I would also like to thank my family and friends who have supported and encouraged me. Especially to my parents who have been always there for me to give courage and motivation during my times of doubt.

Finally, my deepest thanks to my girlfriend Christina for her encouragement, patience and understanding. I could not have been where I am today without her support and love.

Contents

List of Figures

List of Tables and Equations

Chapter 1
Introduction

1.1 Overview

There are many embedded systems surrounding us that we do not even realize their presence. Video game units, DVD players, televisions, microwaves, scanners, cellular phones, and many more contain some sort of embedded processor(s). Using embedded computers in devices that previously relied on analog circuitry such as digital cameras, camcorders, Internet radios, and telephones provide revolutionary performance and functionality that any analog design improvement can not achieve. Most of the embedded computer systems are designed for just one particular application, and it generally provides cost effective solution by employing specialized architecture rather than using a general purpose computing system.

Until now the embedded system design has taken brute force approach. Hardware and software were designed separately where correctness and comparability of the two domains were left to integration stage. If problems arise during the integration stage, the design cycle spin begins and it results in a frequent struggle to

make a sub optimal architecture. Sometimes even the overall project is delayed or even terminated.

Designing of embedded systems today requires working with several million gates of logic and millions of lines of software code. In order to efficiently design these systems, it is desirable to move to higher levels of abstraction for system design automation. Furthermore, rapid improvements in microprocessors performance are changing the balance between embedded software and hardware. What use to be the efficient and cost effective hardware solution can now be transferred into software, due to high performance microprocessors. In this environment, it is necessary to adapt the system design tools that encompass these fast microprocessors rather than to compete with them.

The current hardware/software design methodologies do not effectively handle the massive software-hardware integration necessary [1]. Waiting until a system implementation before understanding the hardware-software interactions is no longer an option. To meet the current market demand, designers now need to produce more complex computing architecture in a shorter period of time. The previous approach of

independent hardware and software development methodology is not acceptable. Hardware software tradeoff must be analyzed early in the system design to reduce the iterative design cycle.

The main question is "how can we design with several hundred million transistors effectively and quickly?" Hardware software co-design is said to provide the answer for designing such large systems [2, 3, 4, 5]. Hardware software co-design is a wide area of research consisting of specification, simulation and estimation, validation, synthesis, and other components. Hardware software co-design concept has been proposed, and being researched for a number of years. Many EDA vendors and researchers have employed dedicated efforts to develop viable hardware software co-design methodologies and tools, yet no standards has been adapted to streamline and coordinate their design efforts.

The main objective of hardware software co-design are to shorten the development cycle, minimize bugs, manage cost, and to produce competitive embedded computing systems that meet today's requirements. Figure 1.1 shows a generalized design methodology for hardware/software co-design. Most of the past and recent

research and development has been built around this model. Hardware/software co-design problem is broken into three major components given below.

- System specification design for describing the system level behavior.
- Hardware/software co-simulation and analysis (validation).
- Rapid hardware software integration by co-synthesis.

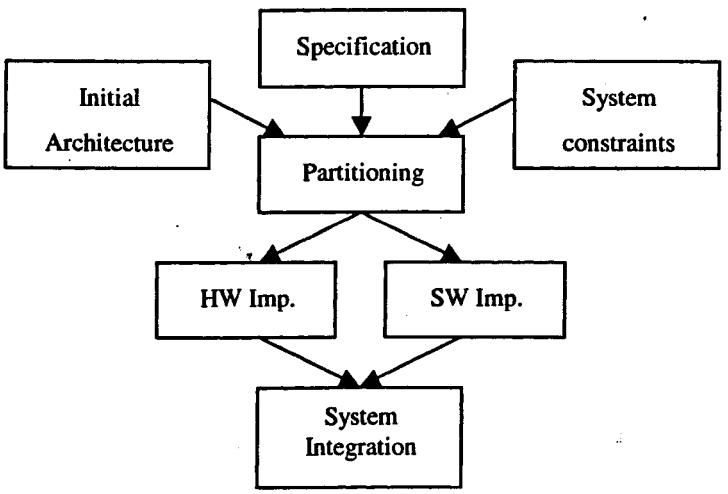


Figure 1.1: Hardware-Software Co-design Methodology

Specification design for describing the system level behavior is a difficult and challenging problem as it needs high level of abstraction as well as fine details to reduce ambiguities during co-synthesis. It is very important to capture the system specification correctly early in the design cycle. Many research projects have been conducted to create a unified co-design environment by proposing a system

specification language [6, 7]. The idea is to capture the specification with more details by augmenting the HDL (Hardware Descriptive Language) and other programming languages like C to describe the entire system. MSC, SDL, PMSC and SystemC are the main examples of such languages.

Design specification language concept has lead to co-synthesis that involves automation of hardware and software architecture synthesis. The function of co-synthesis is to take a system specification language as input and generate competitive hardware and software architecture. A number of co-synthesis systems are under research, where PICO [8] (program in chipout), Corsair [9], Polis [1] are examples of some of these systems.

Hardware/software partitioning is a sub problem of co-synthesis, which is also very difficult due to many conflicting factors affecting the outcome of partitioning decision. Generally speaking, hardware and software are interchangeable in terms of their functionality. To correctly partition the system, expertise of both hardware and software design communities must communicate with the help of diverse tools that can evaluate the trade offs of using hardware or software.

In this thesis, complete hardware software co-design methodology is proposed including: specification, analysis, partitioning and scheduling (co-synthesis) leading to final system implementation is presented by using a Rapid Prototyping Platform. The proposed methodology includes system-partitioning technique with a system specification based on C/C++. The proposed technique also processes data and precedence dependency simultaneously by employing a new structure, Directed Acyclic Data dependency Graph with Precedence (DADGP) that is an extension of Directed Acyclic Graph (DAG). The DADGP based partitioning technique minimizes the communication overhead as well as overall system execution time. Furthermore, the partitioning algorithm minimizes the system cost in terms of hardware area. The partitioning algorithm presented in this thesis analyzes the DADGP and tries to expose parallelism between tasks and repeated tasks. The benefits of exposing parallelism and considering inter processing element (PE) communication overhead are also explained in this thesis. However, these benefits come with an increase in price due to additional hardware modules and their inter-connection structure. The proposed partitioning algorithm is powerful enough to handle complex designs, and is easily extendable for future requirements as explained in the future work section. The partitioning technique provides promising system partitioning solution as compared to

previous methods in terms of efficiency of the over all system design. However, before we get to the proposed methodology, it is important to overview each components of hardware software Co-design process and their advancements until today.

1.2 Motivation

The current hardware software design flow has hard time meeting today's market demand. The separate design of hardware and software results in an error prone integration that leads to a design cycle spins delaying the final product. In order to design these hardware and software systems more efficiently, a proper partitioned hardware software module must be decided before implementation. Currently there are no commercially available tools to help designers with the partitioning of complex hardware software embedded systems.

1.3 Original Contributions

This thesis proposes a new design methodology for designing hardware and software systems for embedded and System on Chip (SoC) application. In particularly,

computationally intensive embedded systems which require multiple hardware and software are considered. A full hardware software co-design methodology is presented with implementation results, which shows the validity of the new DADGP partitioning algorithm. The publication of the proposed research can be found in [10].

The Major contributions for the thesis are summarized as follows:

- Development of software profiling tool for ARM7 processor running under a Rapid Prototyping System has been developed
- Design of a new hardware software interaction representation graph called Directed Acyclic Data dependence Graph with Precedence (DADGP)
- Development of new DADGP partitioning algorithm that includes iterative mapping and scheduling method has been implemented
- In-depth comparative analyses of DADGP algorithm with other system partitioning methods.

1.4 Thesis Organization

This thesis consists of six chapters and it is organized as follows:

In chapter 1, we provide a brief overview of current issues of embedded system design and their solutions (hardware software co-design). In chapter two, a more detailed design of hardware software co-design methodology is presented. The chapter will help readers to understand the basic idea behind the notion of designing hardware software systems.

Chapter 3 will survey previous works related to hardware software partitioning. Partitioning research presented in this chapter is implemented and compared to DADGP partitioning algorithm in later chapter. This chapter also serves to present introductory knowledge for readers who are interested in different partitioning algorithms.

Chapter 4 describes the newly proposed DADGP based hardware software co-design methodology in depth from specifications to partitioning. This chapter discusses the proposed system specifications, a formal definition of DADGP and a detailed explanation of mapping and scheduling process using DADGP.

Chapter 5 presents the experimental results of the proposed DADGP partitioning algorithm. In this chapter, two other system partitioning algorithms are presented in detail for comparison purposes to DADGP based partitioning algorithm. The software comparison result is also presented with many randomly generated task graphs, and their performance is recorded. Furthermore, two computationally intensive multimedia application has been implemented using Rapid Prototyping Platform (RPP) and their results are presented. Some details of the RPP design flow is also presented.

In chapter 6, the thesis is summarized and the directions of future research are proposed.

Chapter 2
Hardware Software Co-design

In a Hardware software co-design problem, the hardware and software must be designed together to make sure that the target system not only functions properly but also meets performance, cost and reliability goals. While a great deal of research has addressed the individual design methods for software and hardware, not much is known about the joint design of these two domains. Due to advancement in VLSI technology, high performance microprocessors are cheap enough to be used in consumer products, and have stimulated research in embedded system co-design. To be able to make use of high performance CPUs, we must develop new design methodologies that allow designers to predict implementation costs, refine an embedded system incrementally over multiple levels of abstraction, and create a working implementation. The current Hardware Software Co-design process involves solving sub-problems of: specifications, validation, and synthesis. Although hardware software co-design problem cannot be entirely separated, it is divided into three separate sections for the purpose of discussion.

2.1 System Specifications

The system design specifications that describe system level behavior is a difficult and challenging problem because it needs high abstraction yet requires fine details to reduce ambiguities during synthesis. Traditionally, these specifications were

written in plain English describing the system constraints and functionalities. The problem with English as a specification language is that very often, designers can misinterpret the meaning of intended idea. These ambiguities and misinterpretation can occur more frequently when different people with different professional background consult the specification to start hardware and software design. It is very important to capture the specification correctly early in the design cycle to reduce the Non Recurring Expenses (NRE) [8]. Many research works have been done to create a unified co-design environment by proposing a design specification language. The idea is to capture the specification with more details by augmenting the HDL and programming language like C. Such a method is known as homogeneous modeling, where hardware and software is represented by a common unified language [11].

System-level specification languages may not always be textual. Visual or combination of visual and textual languages can be used to organize the overall system architecture. After all, humans do work and process images better than just plain textual description. SDL (Specification and Description Languages) is one of the common languages of choice in this area, and PARSE process graph is another interesting approach to describe both hardware and software [12]. However, as more and more details are added, graphical representations might not be suitable, and textual languages become mandatory to express system details completely. Nevertheless, graphical representation is always good to have early in the design cycle for high abstraction and easy visualization for the overall structure of an entire or its sub systems.

The exchange of system-level intellectual property (IP) models for creating executable specifications has become a key strategic element for efficient system to silicon design flows. As C/C++ is the dominant language used by chip architects, system and software engineers prefer C-based approach to hardware modeling [11]. This demand has lead to a much popular homogeneous modeling open source system language known as SystemC [7]. The goal of SystemC is to facilitate the co-verification of hardware-software systems by supplying a single language framework, where designer describes both hardware and software completely. An immediate advantage of having homogeneous modeling is that, it eliminates the need for complex programming language interfaces (PLIs) or remote procedure calls (RPCs) interfaces, which will speed up the co-simulation process. SystemC also allows the user to successively refine models without translating it to an HDL representation. When sufficient implementation details are available, the design can be handed to synthesis tools for circuit generation [3]. SystemC synthesis tools are still under development by Synopsys Inc. and many other EDA vendors.

Heterogeneous modeling is another approach that can be used to model hardware and software early in the design cycle. The hardware and software are modeled using two different languages such as VHDL and C/C++. These representations can then be ported to CAD tools, which allow hardware-software co-simulation for mixed language descriptions. Seamless, Eaglei, CoWare N2C are examples of such tools [13, 14]. Appendix A contains Seamless tutorials to get started in co-simulation environment using C and Verilog.

2.2 Validation and Co-simulation

Validation loosely refers to the process of determining the correctness of a design. Simulation remains the main tool to validate a model; however the importance of formal verification is growing especially for safety-critical embedded systems [15]. Formal verification is the process of mathematically checking the behavior of a system described by a formal model to satisfy a given property. Simulating embedded systems is challenging because they are heterogeneous. In particular, these simulations contain both software and hardware components that must be simulated concurrently. This simulation challenge is known as the co-simulation problem. The basic co-simulation problem is to reconcile two apparently conflicting requirements:

- To execute the software as fast as possible, often on a host machine that may be faster than the final embedded processor and certainly very different from it.
- To keep the hardware and software simulations synchronized, so that they interact just as they will in the actual target system.

In hardware-software co-simulation, software execution is simulated as being executed on the target hardware. Since gate and register transfer level hardware simulations are too slow for practical purposes, a more abstract execution model is needed [16]. Moreover, as systems become complex, validation is necessary to insure that correct functionality and required performance levels are achieved in the implementation of a system model. Different models can be employed with a tradeoff between accuracy and performance.

- Gate-level models are viable only for small validation problems, where either the processor is a simple one or very little code needs to be executed, or both.
- Instruction-set architecture (ISA) models augmented with hardware interfaces. An ISA model is a standard processor simulator (often written in C) augmented with hardware interface information for coupling to a logic simulator.
- Bus-functional models are only hardware models of the processor interface that cannot run any software. Instead, they are configured (programmed) to make the interface appear as if software is running on the processor. A stochastic model of the processor and the program can be used to determine the mix of bus transactions.
- Translation-based models convert the target CPU code into a code that can be executed natively on the computer system executing the simulation. Preserving timing information and coupling the translated code to a hardware simulator are the major challenges.

When more accuracy is required and acceptable simulation performance is not achievable on standard computers, designers sometimes resort to emulation. In this case, configurable hardware can emulate the behavior of the system being designed. There are two types of validation during co-simulation of heterogeneous models: functional verification and performance evaluation. During functional verification, software part is executed on the host processor that communicates with the hardware part through HDL functional model of the system processor bus. In such simulation

environments, the simulation time is very fast because software executes independent to the hardware simulator. To evaluate the performance of a system, VHDL model of a generic instruction set simulator that executes the software part on a functional model of the particular microprocessor bus can be implemented. However, the simulation is somewhat slower because software and hardware are synchronized to evaluate the correct timing of system operation. A more detailed discussion in functional verification and performance evaluation and their simulation results can be found else where [17].

One popular commercial co-simulation tool is Seamless CVS by Mentor Graphics, which uses instruction set simulator, adapted memory, bus models and a target processor model to create virtual hardware environment [13]. The co-simulation environment of Seamless is well modeled and gives accurate results in terms of timing and functionality of the entire system. Seamless tool has also brought new optimization algorithm during co-simulation to allow faster Instruction Set Simulation (ISS). The basic idea of faster ISS optimization scheme is that once a certain part of the system has been verified, the simulation can bypassed the verified components to validate other parts of the system (i.e. instruction fetch, memory reads and write cycles etc). Another similar co-simulating tool is called CoWare's N2C and the methodology of N2C can be found in [18]. In Virtual Component Co-design (VCC) by Cadence, the system behavior is verified separately from the system architecture. Once system behavior is verified, each functional block is mapped to the system architecture. Depending on the choice of partitioning (mapping function), the VCC system calculates the overall performance, and refines the architecture [19].

These simulation environments rely heavily on the availability of a library containing processor or co-processor, communication, interface and memory models that may not be available. To avoid such problems, a more abstract approach is used by Eaglei tool to simulate only functional behavior of the system. In this case, the simulation is more flexible and efficient at the cost of higher abstraction and fewer information details (such as timing and performance). Such simulation environment is called Link Processor Model, where the software runs on the host computer and communicates with another hardware simulating software.

Rapid prototyping is another approach taken to design time dominated systems that require more than just functional verification. FPGA prototyping allows validating a target system yet to be manufactured. Such validating environment provides design engineers with a more realistic data on correctness and performance than the system level simulation. The simulation now has a physical hardware prototype in the loop, emulating the physical behavior of a system, which is implemented using FPGA technology for fast synthesis (e.g. Corsair) [9]. The only downside of FPGA based prototyping is its limited flexibility during co-simulation. Unlike system level simulation environment, FPGA prototyping does not allow single stepping, register value checks or break points in the middle of operations. However, the Rapid Prototyping Platform used by us solves this problem with the help of ARMs Integrator board and Multi-ICE technology. A more detail description of this technology is described in Chapter 5 RPP section.

2.3 Synthesis

Synthesis can be broadly described as a stage in the design refinement where a high level specification is translated into a less abstract specification. For embedded systems, synthesis combines the manual and automatic processes, and it is often divided into three stages:

- Mapping to architecture, in which the general structure of an implementation is chosen.
- Partitioning, in which the sections of a specification are bound to the architectural units.
- Hardware and software synthesis, where the details of the units are filled.

Mapping from specification to architectural design is one of the key aspects of embedded system design. Supporting a designer to choose the right mix of components and implementation technologies is essential for the success of a final product. Generally speaking, the mapping problem takes functional specification as input and produces system architecture as an output and assignment of functions to architectural units. Architecture is generally composed of the following components. Partitioning determines which parts of the specification will be implemented on the above components, and their actual implementation will be created by software and hardware synthesis tool.

- Hardware components (e.g. microprocessors, microcontrollers, memories, I/O devices, ASIC's and FPGA's).
- Software components (e.g. operating system, device drivers, procedures, and concurrent programs).
- Interconnection media (e.g. abstract channels, busses, and shared memories).

The cost function optimized by the mapping process includes a mixture of time, hardware area and power consumption, where the relative importance depends heavily on the application type. Cost of time may be measured either as execution time for an algorithm or as missed deadlines for a soft real-time system. Hardware area cost may be measured as chip, board, or memory size. The components of the cost function may take the form of a hard constraint or a quantity to be minimized. Current synthesis-based methods almost invariably impose some restrictions on the target architecture in order to make the mapping problem manageable. For example, the architecture may be limited to a library of pre-defined components due to vendor restrictions or interfacing constraints.

There are several types of architectural models, which use both processors and ASICs. Models included a processor with an ASIC, single processor with several ASICs, several processors with several ASICs. All systems that automatically synthesize circuits based on these models include an estimation system and a partitioning system. The estimation system allows the quick evaluation of alternative partitioning solutions in the design space. System partitioning allows the total task to

be optimally shared by processors and ASICs, according to a given set of criteria including speed, cost or power.

System partitioning is required for any system design using more than one component. It is a particularly interesting problem in embedded systems design because of their heterogeneous hardware and software unit mixtures. Furthermore, as we rapidly move towards an era of low cost high-speed processors and their cores, the boundary between software and hardware changes rapidly. What someone would have said with certainty should have been implemented in hardware just a year ago, is probably implemented today in software for a fraction of the cost without sacrificing performance.

Estimation tools have been notoriously ineffective in the past. Three of the most widely used estimation tools have been profiling, hardware area estimation and execution time estimation. Profiling tools are a necessity to get information on how long a particular segment of code takes to execute and how many times a loop is executed. Area estimation tools are used to assess the probable size of the hardware (ASIC) when it is implemented. Lastly, execution time estimation estimates the execution time of hardware (ASIC). The tools used to estimate the size of area can be extremely error prone, particularly since it is difficult to estimate the interconnection area. The estimation error can be costly as the cost of chips (ASICS) is a step functional rather than linear. This situation is improved as more and more pre-fabricated cores are used in the design, which would then reduce the total amount of unknown interconnection area. The time taken for a particular ASIC to execute is

difficult to estimate without the final layout, since the clock frequency cannot be predicted. Often predictions are based on the number of cycles, but this is almost useless without the clock cycle information.

Exploring the most common partitioning algorithms includes greedy heuristics, clustering methods, iterative improvement, and mathematical programming [16, 11]. These partitioning algorithms are usually effective and fast. However, there seems to be no clear winners among theses partitioning methods. This is due to early research efforts in this area and the intrinsic complexity of the problem, which seems to preclude an exact formulation with realistic cost functions. Furthermore, these partitioning techniques depend on estimation and profiling tools to produce optimal partitions that make it quite unreliable if the estimation tools themselves are not accurate.

After partitioning and sometimes before partitioning in order to provide cost estimates, the hardware and software components of the embedded system must be implemented. This process is also known as co-synthesis because it involves synthesizing both hardware and software. Generally speaking, the constraints and optimization criteria for co-synthesis step are the same as those used during partitioning. Area and code size must be traded off against performance, which often dominates due to the real-time characteristics of many embedded systems. Cost considerations generally suggest the use of software running on off-the shelf processors, whenever possible. This choice, among other things, allows one to separate the software from the hardware during the synthesis process, relying on some form of pre-designed or customized interfacing mechanism. However, commercial tools for system synthesis

are not as mature as modeling and analysis tools. Yet, because of its continuous demand, EDA industry and researchers are working together to meet today's market demand.

Chapter 3
System Partitioning Overview

3.1 Introduction

The key phase in the discipline of hardware-software co-design is the partitioning of system specifications into hardware and software modules for implementation, while keeping the system cost at the minimum. In other words, the end goal of system partitioning is to minimize the hardware area, subjected to architectural and performance constraints such as memory size, timing constraints, power, etc. It is also known that such partitioning problem is NP complete, and many algorithms and heuristics have been developed to solve this problem. This chapter will discuss partitioning algorithms and heuristics that have been proposed by many researchers around the world to familiarize the reader in many methods and problems with the current co-synthesis approach.

3.2 Survey

A great deal of research work has addressed the co-synthesis and partitioning of system with one-CPU and ASIC hardware engine architecture [20, 21]. Given such constraint, Potkonjak and Wolf [20] have addressed the problem of combining several concurrent tasks onto a single ASIC instead of designing a separate ASIC for each task. They discussed an iterative algorithm that combines tasks for a single ASIC implementation based on the bit-width requirements, register counts, source and destination locations of the task. The application-specific instruction processor synthesis problem is to design a domain-specific processor by selecting the optimal "instruction set" for a class of applications. Typically, the class of applications is analyzed to find the most commonly used instructions, and a data path and controller for that instruction set is designed. Several bodies of research have addressed this problem. For instance, the optimal instruction set selection problem is formulated as an integer linear program described by Ngoc and others [22].

Edwards and Forrest addressed the hardware-software partitioning for performance enhancement by finding the bottleneck in the software and moving that critical region to hardware [5]. Given a C code and taking the "performance profiler"

of it, the hot spots of the source code can be captured. It then tries to accelerate the software execution by implementing the hot spots to hardware components. However, before such transformation, one must calculate the performance gain versus the cost factor. If the solution is feasible (a good performance gain), new hardware and modified software are generated and simulated. However, because the algorithm never takes the transfer of parameters and data from memory or to other hardware into consideration, the overall improvement has not been as great as originally expected. In some hardware software systems (when software is exchanged to hardware), data transfer times accounts for almost 50% of the HW/SW execution time.

Another interesting solution was presented where formation of genetics was used to model the HW/SW partitioning procedure [23]. The algorithm takes the control data flow graph where nodes represent functional elements and the edges represent control or data flow dependencies. First the HW/SW partitioning program is mapped to a constraint satisfaction problem. Then the genetic algorithm is mapped to the constraint satisfaction problem by using a fitness function to generate successive chromosomes. In genetic hardware-software partitioning, three types of constraints were used and they are cost, timing, and concurrency. The genetic algorithm uses the

fitness function to generate the next generation of chromosomes. Selecting two parents by performing crossover and mutations according to a given probability produces each generation. Such chromosomes are decoded to calculate their fitness. The main idea of genetic partitioning is that as the algorithm progresses, more stronger chromosomes will survive and their children will also have higher probability of being fitter. The fitness functions are:

Fitness= (FIT - cost_penalty) / FIT
(if time_penalty=False and Concurrency_penalty=False)
Fitness=(FIT - cost_penalty) / (FIT x Time_pen_wt)
(if time_penalty=True and Concurrency_penalty=False)
Fitness=(FIT – cost_penalty) / (FIT x Con_pen_wt)
(if time_penalty=False and Concurrency_penalty=True)
Fitness=(FIT – cost_penalty) / (FIT x Time_pen_wt x Con_pen_wt)
(if time_penalty=True and Concurrency_penalty=True)

where

- FIT= Maximum of the cost_penalties in a population
- Time_pen_wt and Con_pen_wt are the weight values put to emphasize the violation of time and concurrency constraints.
- Cost_penalty is the sum of all the devices being used as Hardware.

The method shows improvement of the average cost improving (decreasing) as time progresses. It is also concluded that as the search space (design space) increases, the genetic algorithm performs better as compared to other greedy and forward search approaches. However, as of yet the algorithm does not take into account the overhead

of the hardware interface and inter process communication.

Mapping and Implementation-Bin Selection (MIBS) is another way of solving the hardware-software partitioning problem by heuristics [24]. The MIBS partitioning also work with a graph similar to control flow graph known as DAG where nodes represent computations and arcs represent the data and control precedence between nodes. The general structure of MIBS is described in Figure 3.1. The GCLP (Global Criticality/Local Phase) algorithm first traverses the DAG and maps each node to either hardware or software, such that an objective function is minimized. The two objective functions of GCLP algorithm are:

- Minimize finish time of the node (execution time).
- Minimize the percentage of resource consumed by the nodes (HW area and SW size).

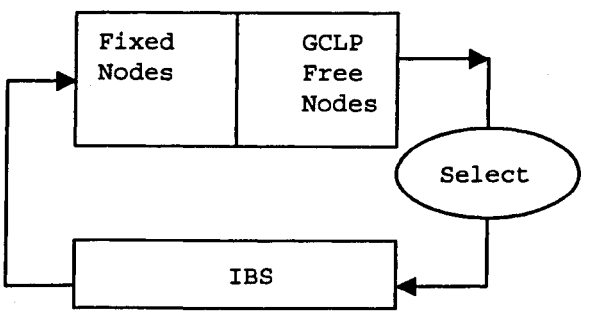


Figure 3.1: MIBS algorithm flow chart

However, the above objective functions contradict each other as if one would like to optimize the completion time, it will require more hardware and then the percentage of resources consumed will not be optimum. Therefore, the GCLP algorithm selects an appropriate optimization objective at each step. For example, if time is more critical factor then the objective function with the minimum completion time is selected; otherwise the one that minimizes the hardware area is selected. After the completion of GCLP, all the nodes in the graph are selected as hardware and software. Then the Implementation-Bin Selection (IBS) further selects the type of implementation for each node. To select an appropriate implementation for the tagged node T, Bin Fraction Curve (BFC) is constructed for that tagged node. BFC is the curve obtained by calculating the bin fraction (BF) for each implementation is:

BF = No. of free nodes to L bin to meet time constraint / No. of free node

Free nodes are the nodes that have not yet been mapped to either hardware or software, therefore for such nodes the algorithm tries to map them to H bin for minimizing the hardware area as long as the timing constraints are satisfactory. L bin are the implementation of hardware-software which takes a lot of area/size but shorter computation time, and H bin are the implementation of hardware/software that takes little area/size but longer time to compute. The free node that is not mapped to H bin

belongs to L bin as long as the hardware area constraints are satisfied. The main idea of BFC is to find the most variation in the BFC curve and selects that variable limit point for implementation that will result in the largest reduction in the area of free nodes. The MIBS algorithm does not consider the communication overhead when calculating the objective function to minimize the completion time. Implementing nodes to L bins of hardware at first seems to minimize the completion time; however, introduction of more hardware to the system can increase the communication overhead that degenerates the completion time for the overall system.

Another similar partitioning algorithm is proposed by Ondghiri and others, where the difference lies in the search technique [25]. Instead of using objective functions to map nodes to hardware and software for a particular solution, the hierarchical design space is explored to provide various solutions. The hierarchical design space search is done by varying its granularity level. Variation in granularity allows the designer to start with an input behavior at the process level. If the performance constraints are not satisfied, finer granularity is selected to increase the number of basic blocks. This operation is performed gradually by accessing successive levels in the hierarchy of the input system. The most complex model of the

input system (operation level) is used only when the performance constraints are not satisfied by the higher complex model. Granularity feature in a co-design tool provides an enlargement of the design space and avoids the use of a detailed and complex model unless the required performance is not reached by using simpler models. The analysis of this algorithm showed that too abstract or complex model did not provide an optimal balanced solution in general, and there exist an optimal solution at some level between the two extreme levels.

Most of the partitioning algorithm worked first from the software side and tried to move the critical region of the software to the hardware component. The next partitioning algorithm introduced by Togawa and others tries to do the opposite [26]. Given an input assembly code generated by the compiler, the hardware-software partitioning algorithm first determines the types and number of required hardware units as an initial resource allocation for a processor core (such as multiple functional units, hardware loop units, and particular addressing units). Then the hardware units determined at initial resource allocation are reduced one by one while the assembly code meets a given timing constraint. The execution time of the assembly code becomes longer but the hardware cost for a processor core to execute it becomes smaller.

Finally, it outputs an optimized assembly code and a processor configuration. In this partitioning environment, the solution will produce an optimal processor core for a particular assembly code and only for that assembly code. Hence, the processor's performance is limited to one particular type of application.

It is clear now that hardware-software partitioning can be considered as a process that can be performed by means of different algorithms, like adaptation of classical circuit partitioning algorithms [21, 27], standard optimization methods of simulated annealing [28] and Tabu search [12]. The constraint-driven system partitioning algorithm presented by Lopez-Vallejo and others however suggest the use of a powerful cost function to consider system constraints in the hardware-software partitioning process [29]. This is performed by formulating different cost functions that will drive the partitioning process. The use of complex cost function allows the algorithm to capture more aspects of the design. Another strong point of the proposed cost function is its generality and therefore, it does not depend on the problem and can be easily extended for considering new design constraints. The cost matrix function described by Lopez-Vallejo is general enough to be used in any partitioning algorithm that considers partitioning as a constraint satisfying problem [29].

The other researchers have also considered the hardware-software partitioning problem as constraint satisfying problem. The partitioning method by Hardt is based on design specification analysis under the restriction of defined architecture and interface in order to make hardware-software partitioning problem feasible [4]. This restriction is acceptable as there are many systems built from a standard architecture for general purpose computations. During specification analysis, a design is thought of as a set of interacting modules. The suitability of each module for hardware implementation is examined during the four phases.

- The analysis phases take static aspects (SA)
- Dynamic runtime characteristics (DA)
- Parameter transportation costs (PA)
- Main memory access (MA).

These specification analysis phases result in a cost vector Ψ=(SA,DA,PA,MA) in which the partitioning algorithm tries to minimize Ψ.

3.3 Summary

From all the partitioning technique discussed, most of the methods take static aspect form of the constraint satisfying problem by optimizing certain cost matrix function. Furthermore, most of the partitioning algorithms would take either dependency graph or execution graph as an input to generate a new set of partitioned graph of hardware and software. The solutions presented, however, only seems to work in ideal cases, as the hardware-software partitioning problem is still too complicated to deal with the actual implementation issues such as delays, communication overhead, interface overhead, etc. Hence, for a commercial product to exist, hardware-software partitioning solution must deal with the implementation issues. The DADGP algorithm presented in this thesis verifies the performance of the algorithm in simulation, and also in actual implementation to fully validate the proposed approach.

Chapter 4
DADGP-BASED HW/SW PARTITONING

4.1 System Specifications

We use textual representation for specification and graphical representation for system partitioning and scheduling to incorporate the possibility of ever-changing demand. C/C++ is used as the initial system specification language, and through profiling, the system specification is converted into DADGP representation for partitioning. SystemC can also be easily incorporated into our approach for future improvements. The high level system specification is translated into C/C++ in the form of modules so that each module can be evaluated and mapped to the process space during profiling (similar to SystemC). The translation levels of specification to modules are also referred to as granularity level. Every system is made up of small and large modules, and in order to partition a system, the level of system modules must be decided. For example, during the block matching step of MPEG, sum-of-absolute-differences are calculated to measure the similarity between the macroblock and image search area. If one is to develop an IP block to calculate the sum-of-absolute-differences, equation 4.1 is first translated into C/C++ specification, where the granularity level is selected as sub, abs and sum modules. The partitioner will then use the information provided in Figures 4.1a and 4.1b to generate the initial DADGP as shown is Figure 4.1c (initial DADGP solution). Other granularity level can be selected to gain different sub optimal partitioning results.

$$\sum_{1 \le i} \sum_{j \le n} |M(i,j) - S(i_x, j_y)| \qquad (4.1)$$

sum-of-absolute-differences

Figure 4.1a: Data table

Processing Elements (PE) component	PE0	PE1
sub	6	5
abs	4	8
sum	3	2

```
for
( i=0;i<=16;i++)
{
        for (j=0;j<=16;j++)
        {
                temp=sub (M(i,j),S(i,j));
                temp1=abs(temp);
                result=sum(result,temp1);
        }
} // M(i,j) is the object matrix, S(i,j) is the search space matrix
```

Figure 4.1b: C specification of block matching algorithm

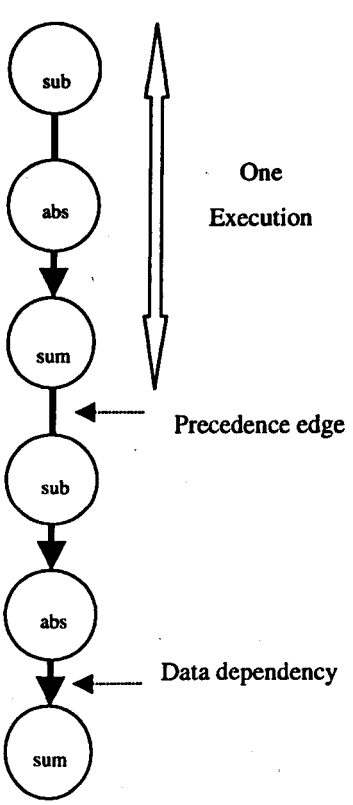


Figure 4.1c: Initial DADGP

4.2 DADGP Graph Representation

Graphs are discrete structures consisting of vertices and edges that connect these vertices. There are several different types of graphs that differ in terms of type and number of edges that connect pairs of vertices. Problems related to most of the disciplines can be solved using graph models. We show in this thesis how a DADGP graph is used to solve a hardware software partitioning problem. First of all, we introduce a formal definition of DADGP to understand the structure of this graph.

4.2.1 Formal Definition

A graph G is a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices such that:

$E = \{(u,v) \mid u, v \in V\}$.

There are also two types of edges E, directed edge (D and undirected edge (U)

Where:

$D = \{d_1 \ldots d_n\}$, and the other non-directed precedence edge

$U = \{u_1 \ldots u_m\}$ to connect vertex V.

The graph G is connected with a directed edge if and only if the two vertices have producer consumer relationship. For example if vertex v_1 produces data that is required by v_2 then v_1 and v_2 are connected by a directed edge d_1. The graph vertices are connected with undirected precedence dependency edge if and only if the two vertices are independent of each other in terms of data generation and consumption.

For example, if vertex v_1 produces data and v_3 does not require such data to compute its own, then vertex v_1 and v_3 are said to be independent.

The vertex and edge of a graph G can also be weighted. Vertices are weighted in terms of their execution time. Furthermore, a vertex can represent either software functional node or some hardware functional node (ASIC), and its appropriate execution time values are assigned to each vertex. Every edge is also weighted to model the communication between two vertices. For example, if vertex v_1 produces 32 bits of data that is required by v_2 then the weighted value of edge d_1 (that connects v_1 and v_2) is 32 divided by the data transfer rate. Undirected edges will have weighted value of zero since there is no data transfer required between vertices that are connected with undirected precedence edge U.

Finally, the graph G can not have any circuits or circular path. For example, from any vertex of G, there can not be any path that has the same starting vertex and ending vertex (starting vertex is a node where the path begins, and ending vertex is a node where no more edges exists). Where path, $P = \{p_1 \ldots p_z\}$, and p is a set of distinct vertices and edges that are connected to each other with edges D or U. The graph that follows these definitions can be called Directed Acyclic data Dependency Graph with Precedence (DADGP). A more detailed operation that can be performed with DADGP will be discussed in the following chapter.

4.3 Hardware-Software Partitioning Algorithm

System partitioning algorithm described in this section can be divided into three major components.

- Profiling of C++
- LD path search (Longest Delay)
- Mapping of LD path and Scheduling

The last two steps are repeated until an adequate solution is reached. A more detailed explanation of LD path search method is provided in this section. The proposed partitioning system flow chart is also shown in Figure 4.2 with the following assumptions:

- Initial target architecture of one processor core.
- Every node is executable with at least one PE.
- All nodes can be implemented as either a hardware or a software module.
- Inter PE communication is measured by the amount of data transferred, where transfer rate is same for all nodes. Communication overhead is zero for nodes that are executed by the same PE.
- The partitioner has all the necessary information including execution time of each node on different PEs, cost of adding PE, inter PE communication between all PEs, and initial system constraints (required system execution time and maximum hardware area).

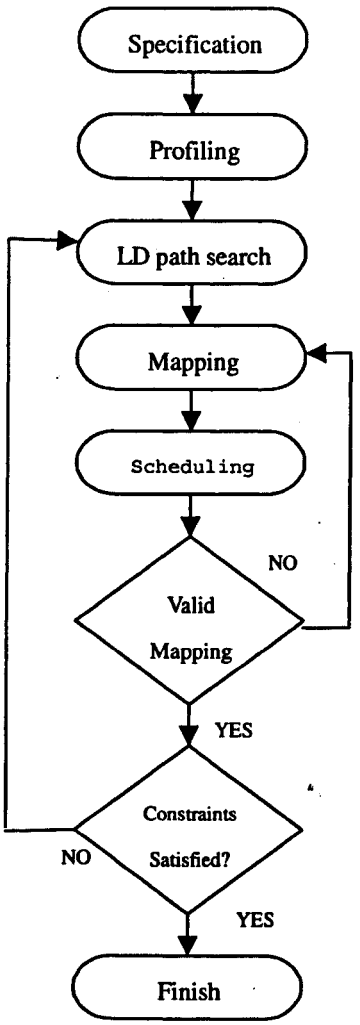


Figure 4.2: DADGP design flow

4.3.1 Profiling

Profiling is performed by executing C++ based system specification on the target processor. The software profiling is a useful step as we must check whether an all software solution is acceptable. If we translate the system specification into software that can fulfill the deadline requirements on the target platform then system

partitioning will not be necessary. However, in most of the cases, all software solution may not be possible for high performance real-time embedded systems and hardware software partitioning step needs to be performed. It is also vital to execute the C++ based system specification on the actual target platform to accurately collect the profiling data. This method also naturally considers SW-SW communication overhead between two software tasks, and is included as part of the module execution time.

The profiler translates each module of C++ system specification into nodes with the following information:

- Execution time of each module
- Start and end time of each module
- Amount of data transfer
- The caller(s) of the module
- The child(s) of the module
- Module identification
- Execution order

The profiler uses the above information to generate the DADGP. The unique characteristics of DADGP is that it contains two types of edges to represent the system.

- Data dependency edge
- Precedence dependency edge

The data dependency edge is represented with an arrow symbol as shown in Figure 4.1c.

Two nodes are connected with data dependency edge when they have a producer-consumer relationship. The precedence dependency edge is represented by a normal line to connect two independent nodes. The precedence dependency captures the order of execution between nodes and such nodes can be executed in parallel if desired. A more detailed discussion on DADGP is provided as part of mapping and scheduling.

4.3.2 Mapping and Scheduling

DADGP is a super set of Directed Acyclic Graph (DAG), with the only difference of having two types of connecting edges. Our contribution to DAG is the introduction of precedence dependency edges to explicitly represent the independence and the execution orders between nodes. The DAG representation is not algorithmically friendly to capture non-parallel executions of independent nodes for hardware-software partitioning. Exposing the independence and introducing parallelism between independent nodes are not always the best decision when optimizing the execution time of a system due to inter PE communication overheads. The incorporation of DADGP to our partitioning method has allowed us to only expose the necessary parallelism for capturing wide rage of solutions.

The Longest Delay path (LD_path) represents the longest execution route in a DADGP. LD_path is not just determined by the number of node hops, but it also depends on the execution time of each node and corresponding inter PE communication overhead as seen in equation 4.2.

$$\textbf{LD\_time} = \sum_{i=1}^{N} n_i + \sum_{j=1}^{Q} e_j \qquad (4.2)$$

where node $n_i \in \{ n_1 \ldots n_N \}$ and they must be connected with any type of edges $e_j \in$ E (all edges D or U).

p=$\{ n_i \ldots n_N , e_j \ldots e_{N-1} \}$ is a path from one root to any ending node.

P=$\{ p_1 \ldots p_M \}$ is a collection of all paths from one root to any ending node.
M is the total number of paths in a given graph.
LD_path can be found with the following equation.

LD Path= Max [LD time (p_k)] (4.3)
where k varies from 1 to M.

Finding an LD_path in a DADGP is similar to finding a bottleneck in the system. The LD_paths are used to improve the overall execution time by mapping one of the LD_path nodes to hardware. A repeated searching and mapping of DADGP reduces the search space and improves the convergence rate for an optimal solution.

The LD_path searching algorithm is given below.

```
L = { l_1... l_N }; //set of all leaf nodes in a DADGP
for ( i = 1; i++; i <= N ) //N is the total # of leaf nodes
        P = Find_path(l_i); //Finds all unique path from l_i to root(s)
                        // now set P has all the paths from leaf to root
max = 0;
for ( i = 1; i++; i <= M ) //M is the total # of path{
        temp = LD_time( p_i );
        if ( temp > max ) then{
                max = temp;
                path = p_i ;
        }
} // After checking all path maximum delay path is found
```

4.3.3 Mapping and Scheduling of DADGP

This is the most sophisticated and important step of our partitioning algorithm. After finding the LD_path, one of the nodes in the path is mapped to the optimal hardware. However, to make such critical decision, following factors must be taken into consideration.

- Parallelism in DADGP nodes
- PE resource limitation
- PE Execution time of nodes
- Inter PE communication
- Hardware area (cost)

The PE can be a dedicated hardware unit or a software task being executed on a processor. The algorithm starts by finding the LD_path from a given DADGP and the execution time of LD_path is also calculated. All the nodes in the LD_path are mapped to appropriate hardware, one at a time and scheduled to calculate the overall system execution time. A node that allows maximum improvement of system execution time is finally mapped as hardware and the DADGP is updated according to the final mapping decision. This process is repeated for all the LD_paths as explained in the partitioning algorithm given below.

```
LD_path = { n_1 ... n_A , e_1 ... e_B } //A is the total number of node, and
                    //B is the total number of edges
Previous_system_EXE = ∞;    // highest allowable value by the system
while (( System_EXE > Required_EXE) AND (Current_HW_area >
Required_HW_area))
{
        LD_path = Find_LD_path (graph);    //find the LD_path from a given graph
        LD_path_EXE = LD_time (LD_path);    //calculated the execution time of
        LD_path
        Min_EXE = ∞;    // highest allowable value by the system
        for ( i = 1 ; i++ ; i <= A ) {
                G = map (DADGP,  n_i );
                //where G is a new DADGP with the node  n_i ∈ LD_path nodes mapped
                //as HW.

                (EXE, S, G_prime) = schedule (G);

                // G_prime = updated G
                // S = schedule of G
                // EXE execution time of G
                // schedule details are given in the schedule algorithm below


                if ( EXE < Min_EXE ) then {

                    Min_EXE = EXE;//save the current optimal execution solution

                    graph = G_prime;// save the current partitioned DADGP

                Final_S = S;//save the current schedule result after a valid //mapping

                  }

        } // successfully found a node to be implemented as hardware

       if ( previous_system_EXE < Min_EXE ) then

                return (graph); //return current partitioned graph

                    //system could not be improved any further, and it could //not satisfy
                    the given constraints.

         previous_system_EXE = Min_EXE;    // save previous solution

} // now the solution is found and the algorithm is terminated normally
return (G, Final_S); // return currently partitioned graph with its scheduling
```

The scheduling function "schedule (G)" is summarized as:

a) Start scheduling from the root of DADGP.

b) Traverse down the tree and schedule the earliest starting time node.

c) If a node is connected with precedence dependency edge, check to see whether exposing parallelism can eliminate that edge. If a precedence dependency edge is eliminated, the structure of the DADGP may change and some nodes can be disconnected from the original graph resulting in two separate DADGP. In this case, the new root of the disconnected DADGP must be combined to make one DADGP by connecting it self and the original root to a new dummy node called "start".

d) If multiple descendents (or roots) exist, force schedule all descendents (or roots) by adding necessary PE if required.

e) Update PE resource library and generate the total execution time by using the following equation:

$$EXE = LD\_path\_EXE - n_i\_exe + HW\_exe + IPC - Hidden\_node\_EXE$$

where, n_i_exe = execution time of a node that is currently in interest, HW_exe = HW execution time of node n_i.
IPC = Communication value introduced by mapping node n_i to additional HW, hidden_node_EXE = smaller execution time value between two parallel modules.

4.3.4 Complexity of the Algorithm

One measure of efficiency of an algorithm is the time used by a computer to solve a problem by using the algorithm for a specified input value. A second measure is the amount of memory required to implement the algorithm when input values are of a specified size. An analysis of the time required to solve a problem of a particular size determine the time complexity while analysis of the computer memory required involves the space complexity of the algorithm. The time and space complexity analysis of an algorithm is essential for its implementation. It is obviously important to know whether an algorithm will produce the answer in microseconds, minutes, or years. Likewise, the required memory must be available to solve the problem. Considerations of space complexity are tied with the data structures used to implement the algorithm. We assume that enough memory resources are available as the proposed partitioning algorithm has already been implemented and executed without a memory deficiency. Hence, space complexity will not be considered and we will restrict our attention to time complexity.

Time complexity is described in terms of number of operations required instead of actual computer time because of the difference in time needed for different computers. Moreover, it is quite complicated to break down all the operations to the basic computer operations. Furthermore, the fastest computers in existence can perform basic bit operations (for instance add, multiply, compare, or exchanging) in nano seconds but personal computers may require micro seconds that takes 1000 times longer to perform the same operation.

For simplicity, assume that there are $n=2^k$ paths in the DADGP graph where k is a nonnegative integer and k=log(n). If n is the number of paths in the graph and it is not a power of 2, then the graph can be considered as part of a larger graph with 2^{k+1} paths where $2^k<n<2^{k+1}$ where 2^{k+1} is the smallest power of 2 larger than n. Therefore, at maximum the algorithm will take log(n) times to find the LD_Path. Mapping procedure only considers nodes in the LD_path and therefore the complexity of mapping algorithm is just the number of vertices in the LD_path, and this is bounded by N, which is the total number of nodes in the DADGP graph.

Scheduling algorithm also traverses down the DADGP and schedules the entire vertices (nodes) according to the resource availability and vertex start time. Therefore, the complexity of the scheduling algorithm is also bounded by N because it can not consider more vertices than what is in the DADGP. In such cases, the complexity of the entire algorithm from LD_path search to mapping and scheduling is N x N x log(N) because LD path search, mapping and scheduling is iterative process of the partitioning algorithm.

Chapter 5
Experimental Results

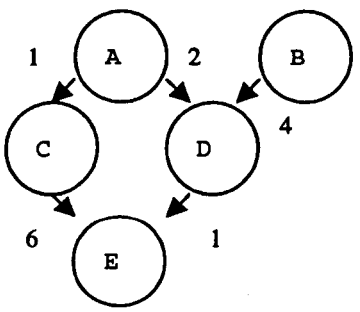
Heterogeneous system architectures are commonly found in high performance embedded systems. They are application specific systems that contain hardware and/or software tailored for a particular application. General and special purpose processors, digital signal processors, and ASICs are among the many components of these systems. In these systems, heterogeneous processors are tightly coupled with low inter-process communication (IPC) overhead but with heavy resource constraints. Therefore, the schedulers for such heterogeneous systems need to account for IPC overhead and processor heterogeneity where a task has different execution times on different processors. Task scheduling for homogeneous multi-processors has been a difficult problem in the past while scheduling problem for heterogeneous processors is much more difficult.

In this chapter we present experimental results showing the validity of DADGP based partitioning algorithm. First of all, we begin with a brief overview of two simple yet well known partitioning and scheduling algorithms: General Dynamic Level [18] and Simulated Annealing technique [12]. Secondly, these two techniques are implemented and a performance comparison of these algorithms with DADGP technique is presented by using randomly generated graphs as their input. Lastly, the results from two application implementations to the actual hardware-software system are shown to compare the simulated partitioned results. Block matching and SOBEL

edge detection applications are partitioned using the DADGP technique. They are implemented using a Rapid Prototyping Platform (RPP) containing an ARM7 processor and Xilinx FPGA for custom hardware.

5.1 GDL Scheduling Technique

The GDL scheduling takes a standard DAG as an input task graph. A DAG node represents a task to be executed on a processor and a directed edge indicates the data dependency between two nodes. Each edge is associated with a number that specifies the amount of IPC overhead. The algorithm assumes that the execution time of a node N_i on a processor P_j, $E(N_i,P_j)$, is known at compile-time for each processing element. If node N_i (task) cannot be executed on processor P_j, the value of $E(N_i,P_j)$ is infinite (a very high number). This occurs when processor P_j has no resource or instruction for node N_i. The algorithm also assumes that the target architecture has a dedicated hardware for IPC so that inter-processor communication time can be overlapped with computation time in a schedule. An example DAG and its node execution-time table is shown in Figure 5.1.



	PE0 Cost= 20	PE1 Cost=40	PE2 Cost=50
A	3	6	3
B	2	7	9
C	5	5	1000
D	1	2	1
E	10	2	1000

Figure 5.1: An example of DAG and its node execution time table

The scheduling objective of GDL is to minimize the scheduling length of the input DAG. The scheduling problem is NP-hard in the strong sense and therefore it relies on heuristics. GDL algorithm is one of the existing scheduling heuristics for heterogeneous scheduling problem and it is based on the list scheduling idea [18]. Each node is assigned a priority and GDL calls a node runnable when its ancestor nodes have been already scheduled. The list scheduling schedules the runnable nodes in the decreasing order of priority. The variants of list scheduling techniques differ in terms of how to assign priorities to the nodes and on which processor a selected node is to be scheduled.

The schedule length can not be less than the maximum length (or critical path length) of a node to the terminal node. Therefore, to minimize the schedule length, it is intuitive to assign the highest priority to a node from the longest critical path length. The critical path length of a node is the sum of execution time of nodes on the critical path and the IPC overheads incurred if these nodes are assigned to different processors. The IPC overheads are not known before the descendant nodes are scheduled, therefore a typical list scheduling technique considers only the sum of execution times on the critical path to determine the priority of a node. We call it the static level SL(Ni) of a node. The static level of a node becomes the critical path length if all nodes on the critical path are scheduled to the same processor. In a heterogeneous system, however, a node has different execution times on different processors. If we choose the smallest execution time of a node in the static level computation, it may not be possible to assign the node to the same processor as its ancestor. In this case, ignoring the IPC overhead is problematic for heterogeneous systems. Therefore, the GDL scheduler defines the

assumed execution time of node N_i, denoted by $E^*(N_i)$, as the median execution time of node N_i over all the processors. If the median execution time is infinite, the largest finite execution time is substituted.

The GDL scheduler considers the effects of IPC overhead by adjusting the priority level when the node becomes runnable. $DA(N_i,P_j)$ is defined to be the earliest time that all data required by node N_i are available at Processor P_j, where IPC overhead is accounted for. $TF(P_j)$ is further defined as the time that the last node assigned to the jth processor finishes its execution. The node N_i can not be scheduled before $T(N_i,P_j)$, which is the maximum of $DA(N_i,P_j)$ and $TF(P_j)$ for processor P_j. To account for the varying processing speeds, they also defined $\Delta(N_i,P_j) = E^*(N_i) - E(N_i,P_j)$. A large positive $\Delta(N_i,P_j)$ indicates that processor P_j executes node N_i more rapidly than most of the other processors, while negative value of $\Delta(N_i,P_j)$ indicates the opposite. By incorporating these terms, they first extended the static priority level to the dynamic priority level $DL1(N_i,P_j)$ as given below:

$$DL1(N_i,P_j) = SL(N_i) - T(N_i,P_j) + \Delta(N_i,P_j).$$

$DL1(N_i,P_j)$ level gives a higher priority to a node with regard to processor P_j. Higher static level means that it can be scheduled earlier or the it can be executed faster. Although $DL1(N_i,P_j)$ indicates how well node Ni is matched with processor P_j, but it fails to consider how well the descendants of N_i are matched with P_j. If a descendant node of N_i can not be scheduled on P_j due to resource constraints, the IPC overhead between node N_i and its descendant should be considered to estimate the cost of assigning N_i to processor P_j. From this observation, GDL scheduler selects a

descendant of a node with the largest IPC overhead $D(N_i)$ and another function $F[N_i,D(N_i),P_j]$ is defined to indicate how quickly $D(N_i)$ can be completed on any other processor if N_i is executed on P_j. Then, the effects of descendants on the level of node N_i becomes: $DC(N_i,P_j) = E^*[D(N_i)] - \min(E[D(N_i),P_j],F[N_i,D(N_i),P_j])$

$DC(N_i,P_j)$ roughly indicates how well the "most expensive" descendant of node N_i matches with processor P_j, if N_i is scheduled on P_j. Incorporating the descendant consideration term modifies the level of a node on processor P_j as:

$$DL2(N_i,P_j) = DL1(N_i,P_j) + DC(N_i,P_j).$$

In addition to the descendant consideration effect, a heterogeneous processing environment also introduces a cost associated with a node if it is unable to be executed on a certain processor. To characterize this resource scarcity cost, they selected an optimal processor Pj* on the second best processor to maximize DL2(Ni,Pj). Then the "generalized" dynamic level is defined as: $GDL(Ni,P) \doteq DL2(Ni, Pj^*) + C(Ni)$.

The second term indicates an increase in the dynamic level if the node is forced to be scheduled on the second optimal processor. Now GDL scheduling technique selects the runnable node of highest priority or highest GDL value. Figure 5.2 shows the overall flow of GDL algorithm.

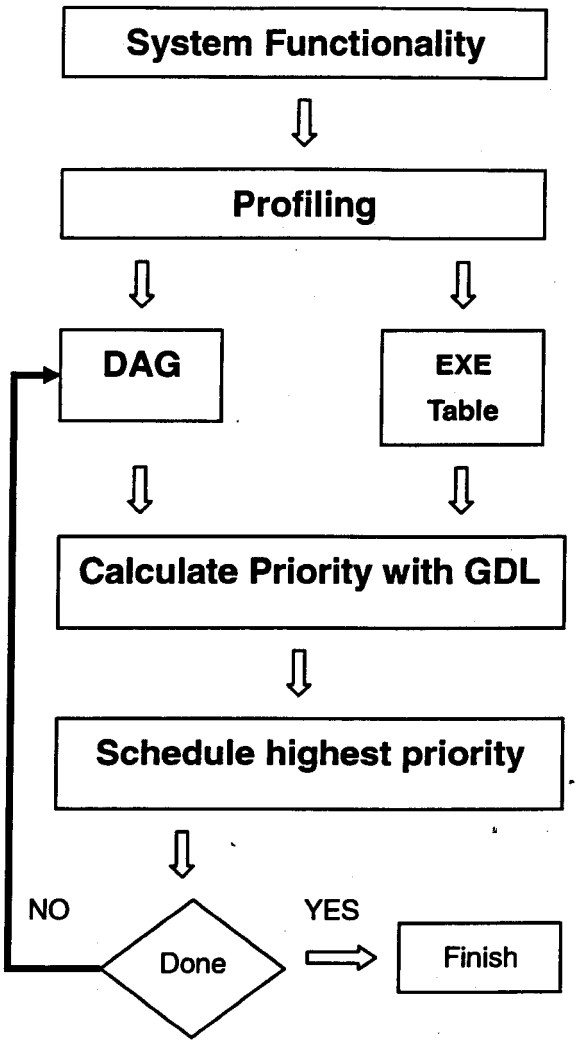


Figure 5.2: GDL algorithm flow chart

5.2 Simulated Annealing Based Partitioning

The idea of simulated annealing originated from metal processing. Annealing in metal processing is how a liquid becomes progressively more organized into a solid as the temperature falls slowly. Metropolis formalized this into an algorithm shown in Figure 5.3 [30]. Simulated annealing is similar to the well-known greedy algorithm [31] except for two key differences. Both incorporate the concept of neighbor (nodes),

which is created by making some changes to the current solution. However, the greedy algorithm always follows the neighbor with the largest gain, whereas simulated annealing picks random neighbors. In addition, the greedy algorithm will never accept a neighbor with a higher cost. On the other hand, simulated annealing may accept a neighbor with higher cost, depending on the gain, temperature, or a randomly generated value. These differences have important implications; while the greedy algorithm will always find the local minimum, simulated annealing attempts to find the global minimum. The other implication is that simulated annealing depends on random numbers that makes it probabilistic in nature. As simulated annealing can be adapted to a multitude of problems, one has to adapt the algorithm for system partitioning. Our basic implementation of the simulated annealing algorithm is fairly straightforward as shown in Figure 5.3.

Main course of Simulated Annealing events can be summarized in 4 steps:

i. Starts with an initial partition and an initial simulated temperature

ii. The temperature is slowly decreased

iii. For each temperature, random moves are generated

iv. Any cost-improving move is accepted. Otherwise, it may still accept the move, but acceptance becomes less likely at lower temperatures as given below

- Accept(Δcost, temp) returns 1 if the move should be accepted, otherwise it returns a value in the range of [0,1]
- Decrease Temp (temp) is often defined as

  $$temp\_new = \alpha \times temp\_old; \text{ where } 0 < \alpha < 1$$

- Equilibrium can be approximated as no improvements for some number of iterations

```
temp= initial temperature, cost=Objfct(P)
while not Frozen loop
      while not Equilibrium loop
            P_tentative = RandomMove(P)
            cost_tentative = Objfct(P_tentative)
            Δcost = cost_tentative - cost
            if( Accept(Δcost,temp) ) > Random(0,1) ) then
                  P = P_tentativecost
                  cost = cost_tentative
            end if
      end loop
      temp=DecreaseTemp(temp)
end loop
```

where: Accept(Δcost, temp) = min(1, $e^{-(\Delta cost/temp)}$)

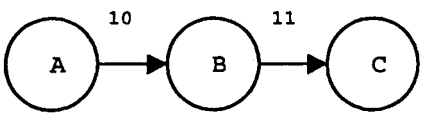
Objfct(P) returns a cost value of current partition P, and cost represents systems execution time and hardware area

Figure 5.3: The basic simulated annealing algorithm.

5.3 Software Simulation

To compare the performance of our DADGP based partitioning algorithm, we have implemented GDL and Simulated Annealing algorithms. Each partitioning algorithm have been given the same set of randomly generated DAG ranging from size 3, 9, 100, 250, 500 to 1000, and their results are presented here. The DADGP based partitioning algorithm has also been given the same set of graphs except that the DAG graph is first converted into DADGP (refer to Chapter 4.2 for detail).

The experimental results for DAG graph of size 3 is shown in Figure 5.4, which demonstrates the ability of DADGP partitioning algorithm that considers multiple descendants indirectly without the added complexity of their calculations. Considering GDL algorithm [18] and its complexity, the scheduling inherits the traditional weakness of conventional list scheduling where global effect of scheduling decision is not measured properly. The GDL scheduler improved on list scheduling by quantifying the scheduling effects on the descendants of a node. However, considering just the first descendants is not enough to measure the global effect. Consider DAG graph of Figure 5.4 and its corresponding parameters for GDL algorithm whose scheduler results are presented in Figure 5.5 in the form of Gantt chart. After scheduling node A, the GDL fails to consider the effect of processor selection for node C. GDL algorithm can only look up to node B (first descendent) as it schedules node A on processing element PE0 hoping that it will also assign node B to PE0 in the next iteration. However, the final scheduling result suffers from a large inter-PE communication overhead between A and B as node B is scheduled to processor PE1 considering node C's execution time on processor PE0 in the next iteration.



Tasks	PE0 Cost=20	PE1 Cost= 50
A	1	2
B	2	2
C	20	1

Figure 5.4: DADGP without Precedence

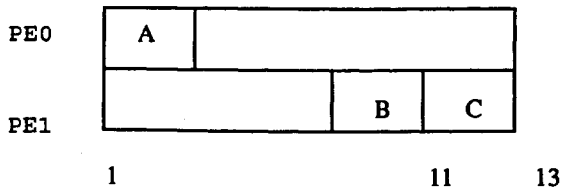


Figure 5.5: GDL Scheduled Results

The DADGP initial graph is obtained in Figure 5.6 which is exactly the same as the DAG of Figure 5.4 because the graph does not have any precedence dependency edges and hence no parallel execution is possible. Obtaining the LD_path is also simple because there is only one path. DADGP then tries to find a node that can improve the overall system execution by mapping tasks to another PE (Processing Element) assuming that PE0 is the target processor (software) and PE1 is the additional PE (ASIC, Processor, etc). Then the initial DADGP is given in Figure 5.6 our algorithm assumes that all nodes are initially executed as software (on PE0). The partitioner will first move node C to PE1 to reduce the total execution time (C is the Min[EXE]). In the next iteration, node B is moved to PE1 to reduce inter PE communication overhead between B and C. In the third iteration, node A is finally moved to PE1 to reduce the inter PE communication overhead, and the optimal scheduling is obtained as shown in figure 5.7.

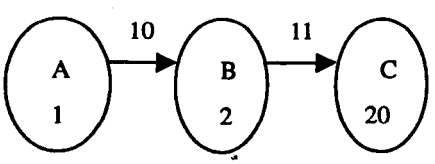


Figure 5.6: Initial all software DADGP solution

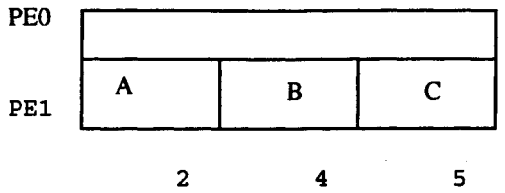
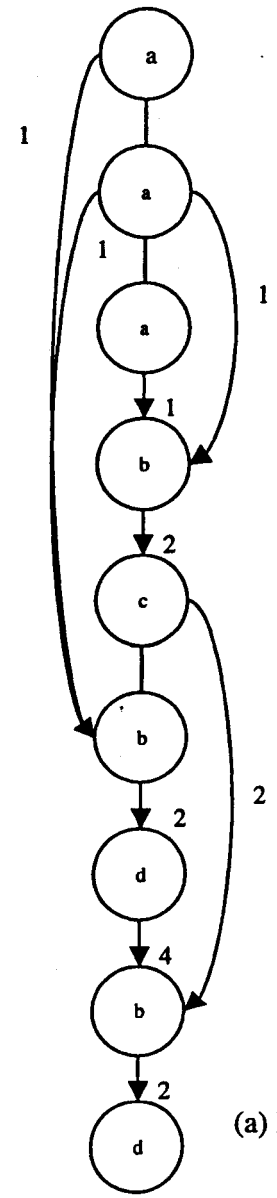


Figure 5.7: DADGP Solution

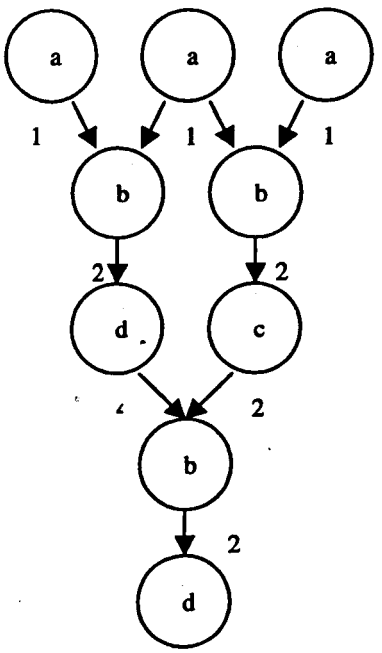
Simulated Annealing algorithm also arrives to the optimal solution by assigning and scheduling the entire task to processor P1. However, due to the selection of random neighbors in simulated annealing, execution of simulation multiple times with the same data has given different results. But only the best results have been recorded. The experimental results for bigger sized graph have been recorded and are shown in Table 5.1. The performance gain is defined as the ratio of non-optimized execution time over optimized execution time of DADGP. Cost is the amount of hardware required, and simulated time is the time it takes for completion of simulation, and execution time is the target system execution time based on the partitioned results. The random DADGP graph generated for 9 nodes can be seen in Figure 5.8a. The optimal solution obtained with DADGP based partitioning algorithm is shown in Figure 5.8b.

In comparing the performance gain of these three algorithms, DADGP has shown outstanding results as shown in Figure 5.9. For smaller size graphs, all three algorithm have shown similar results in terms of performance gain ratio. This is expected because the solution space for such a small graph is limited. The difference appeared when the graph size is increased to 9 or more nodes. In fact, GDL algorithm has shown lower performance gain with the increase in graph size. These results

show that GDL is suffering from wrong decision making as it can not consider multiple levels of descendants. Simulated Annealing has shown almost constant performance gain across different sized graphs. DADGP, however, shows increase in performance gain as number of nodes are increased. This characteristic is very desirable because increase in graph size means that there are more opportunities for improvements, and DADGP algorithm is exposing various solution spaces.



(a) Initial DADGP



(b) Optimal Solution

Figure 5.8: Randomly generated Graph (9 nodes)

Table 5.1: Software Simulated Comparison Experiment Results

DADGP	# of Nodes	Cost	Exe Time	Simulated Time	Performance Gain Ratio
	3	50	5	0.2s	4.4
	9	195	18	0.43s	5.5
	100	400	7350	15.4s	9.9
	250	350	12250	71.3s	14.86
	500	600	19015	271.25s	19.14
	1000	550	29400	486.1s	24.76
GDL					
	3	70	14	0.6s	1.57
	9	125	38	2.01s	2.63
	100	220	20472	237.16s	3.56
	250	320	60142	601.2s	3.03
	500	370	205422	1363.67s	1.77
	1000	570	314721	3954.64s	2.31
SA					
	3	70	5	0.72s	4.4
	9	95	34	0.89s	2.94
	100	200	18021	20.1s	4.04
	250	360	40124	100.5s	4.53
	500	550	80804	324.4s	4.5
	1000	770	80984	501.4s	8.99

Where Exe Time is execution time of partitioned system when scheduled.
Simulation time is the time it takes for an algorithm to partition a given system.
Performance gain ratio is Exe Time of all software solution over partitioned solution.

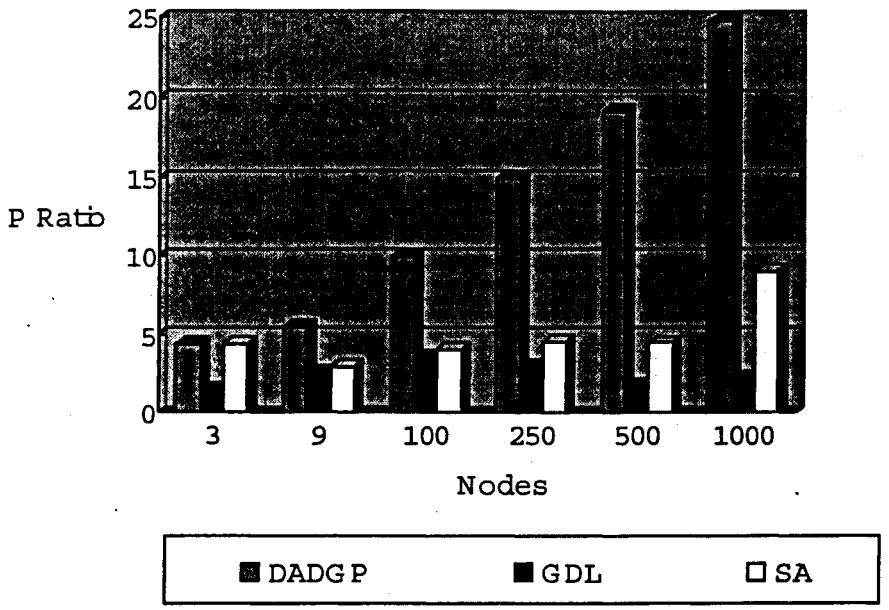


Figure 5.9: Performance gains

The next important result is the simulated time. DADGP algorithm shows improvement on all graphs, but it is also important that the algorithm is executed in a reasonable time. Figure 5.10 shows the simulated time for all three techniques vs. graph size, where DADGP method again indicated its superiority. The GDL algorithm indicated slower simulation time due to its $O(N^3)$ complexity. As the number of nodes increases in the graph, the simulated time increases exponentially. The runtime of DADGP and Simulated Annealing provide good results through out the experiment, keeping the simulated time below 10 minutes for a Pentium III 600Mhz processor under the worse case scenario.

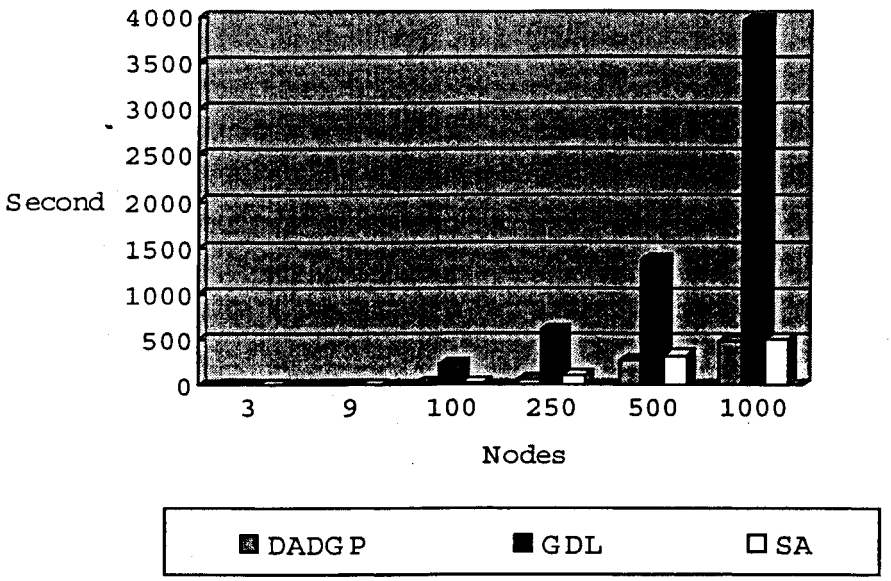


Figure 5.10: Simulated time

It is important to note that performance gain must come with a reasonable price, and this is evident in Figure 5.11 that shows the performance gain for DADGP algorithm. On average, DADGP solution is always expensive than GDL, however, increase in performance gain can compensate for its cost. The DADGP algorithm explored parallelism in the system to increase the performance; therefore, there must be additional components in the system to accommodate tasks concurrently. The difficulty is to explore those parallelisms in the order of maximum performance gain with minimum cost. GDL algorithm seems to show the inverse relationship to maximize performance gain. It provides improvements to the system, but with more hardware components than necessary. Combining the results of Figures 5.9 and 5.11, we can derive another important hardware to performance ratio as shown in Figure 5.12. The hardware cost to performance ratio indicates the performance gain of the system by

adding more gates to the system. The results indicate that as system becomes more complicated (bigger), it is more difficult to increase its performance by adding extra hardware as shown with SA and GDL algorithms. However, DADGP algorithm has shown increase in this ratio indicating that the algorithm can add necessary hardware to increase the performance. The results are also related to maximum performance bound as addition of more hardware after curtain performance boundary will only degenerate the system.

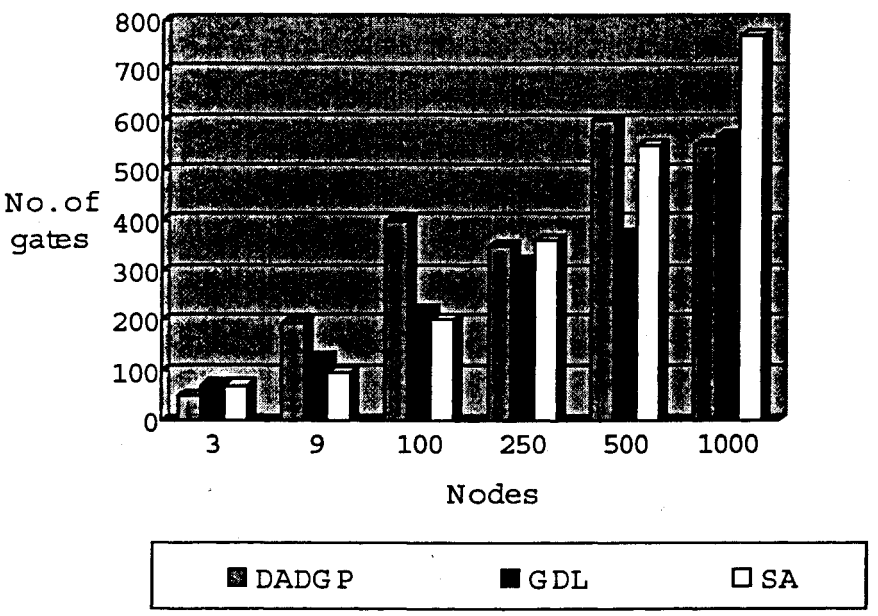


Figure 5.11: Hardware area cost

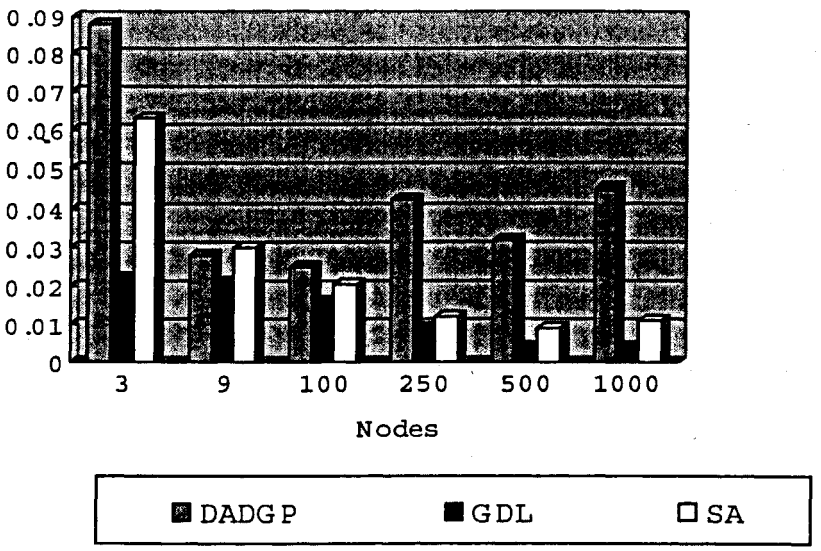


Figure 5.12: Hardware cost – Performance ratio

5.4 Rapid Prototyping Platform

Rapid-Prototyping Platform (RPP) consists of an ARM7 CPU and FPGA based hardware components to enable the prototyping and design of embedded systems. The RPP features two or more daughter cards, housed on a motherboard (ARM's Integrator/AP board) that are:

- The ARM7TDMI core
- A re-programmable hardware module, featuring a Xilinx Virtex-2000E FPGA.

The Integrator board allows stacking multiple cores (e.g. ARM7, ARM9) and logic modules (Xilinx or Altera) as shown in Figure 5.13. RPP provides a software-programmable processor as well as hardware modules. The design flow for the RPP

involves both software and hardware design tools. Figure 5.14 demonstrates the flow from specification to integration of hardware and software designs on the RPP.

Chip capacity continues to increase at an exponential rate and designs increase in size and complexity (e.g. embedded processors, third-party IP, mixed-signal components), design verification and proof of concept have become significant bottlenecks. The Rapid Prototyping Platform by ARM makes use of several key system-on-chip (SOC) concepts to enable researchers to prototype their designs quickly and with higher confidence. In the past, simulation was often sufficient means to verify the proof of concept. However, with increase in chip complexity, simulation of large designs require many millions of clock cycles. The incorporation of embedded processors on a chip amplifies the problem of simulating embedded software on multiple processors. To address these problems, rapid prototyping systems with high-capacity FPGAs and embedded processors are challenging the role of simulation for system-level verification. What may take days to simulate at cycle level accuracy can be reduced to several hours through the hardware-assisted simulation of hardware-software system.

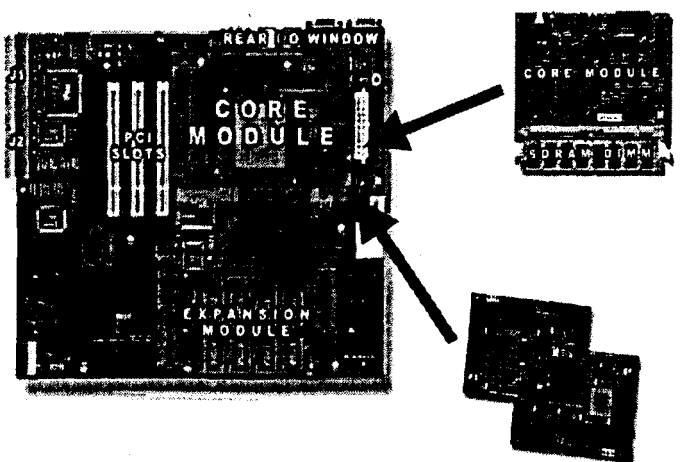


Figure 5.13: RPP (source CMC website)

In the research community, ASIC implementation has often been used to demonstrate the proof of concept. A design that can be implemented on an ASIC may be ported quickly to an FPGA-based rapid prototyping system, and what is lost in design performance can be regained in engineering time. Assuming first-pass success of the design, ASIC implementation of a typical industry-capacity design requires longer engineering time, while FPGA implementation requires much shorter time. Furthermore, with almost 40% likelihood of a design re-spin, implementing to FPGA again saves engineering time, as each re-spin adds approximately 20% more engineering time to the entire design cycle [9].

To accelerate the process of achieving proof of concept, rapid prototyping systems effectively utilize the concept of reuse, which is the driving force behind the SOC revolution. In SOC context, re-use takes two forms: software re-use and hardware re-use. Both of these are achievable with the rapid prototyping platform and both are presented in RPP design flow in the form of Intellectual Property (IP) libraries (see Figure 5.14). Rapid prototyping system is based on an ARM AMBA bus to support the re-use of hardware. A user needs to customize the rapid prototyping system to meet specific design requirements, and may require custom IP to execute some of the design functionality. A library of bus-independent IP is also available through Xilinx's LogiCORE system, in addition to the blocks already present in the system. Designers can also create a hardware block and later incorporate the block to an intellectual property library for re-use in other designs. The RPP design flow supports the use of Seamless, a co-design tool from Mentor Graphics for hardware/software co-verification. Furthermore, developers targeting the RPP environment can use C/C++ language to

program embedded software. Programming at such an abstract level opens the door for software re-use, as related standards evolve in the SOC community.

5.5 RPP Design Flow

Designing system with RPP is simple with the help from many EDA design tools. The RPP design environment is flexible as one can incorporate many different design tools to program and synthesize the ARM7 processor and its programmable logic device. The platform also supports JTAG and Multi-ICE tools for debugging. This section will introduce the RPP system environment and its capabilities. A detailed diagram of RPP design flow is given in Figure 5.14.

5.5.1 Design Specifications

The design flow starts with a set of design requirements and system specifications, detailing the function of the system as well as constraints such as clock rate, power and operating conditions. These design specifications are usually set out on a written document or spreadsheet. This specification is used to derive design constraints, which guide the designer (and design tools) and provide a basis for evaluating the quality of the design throughout the flow.

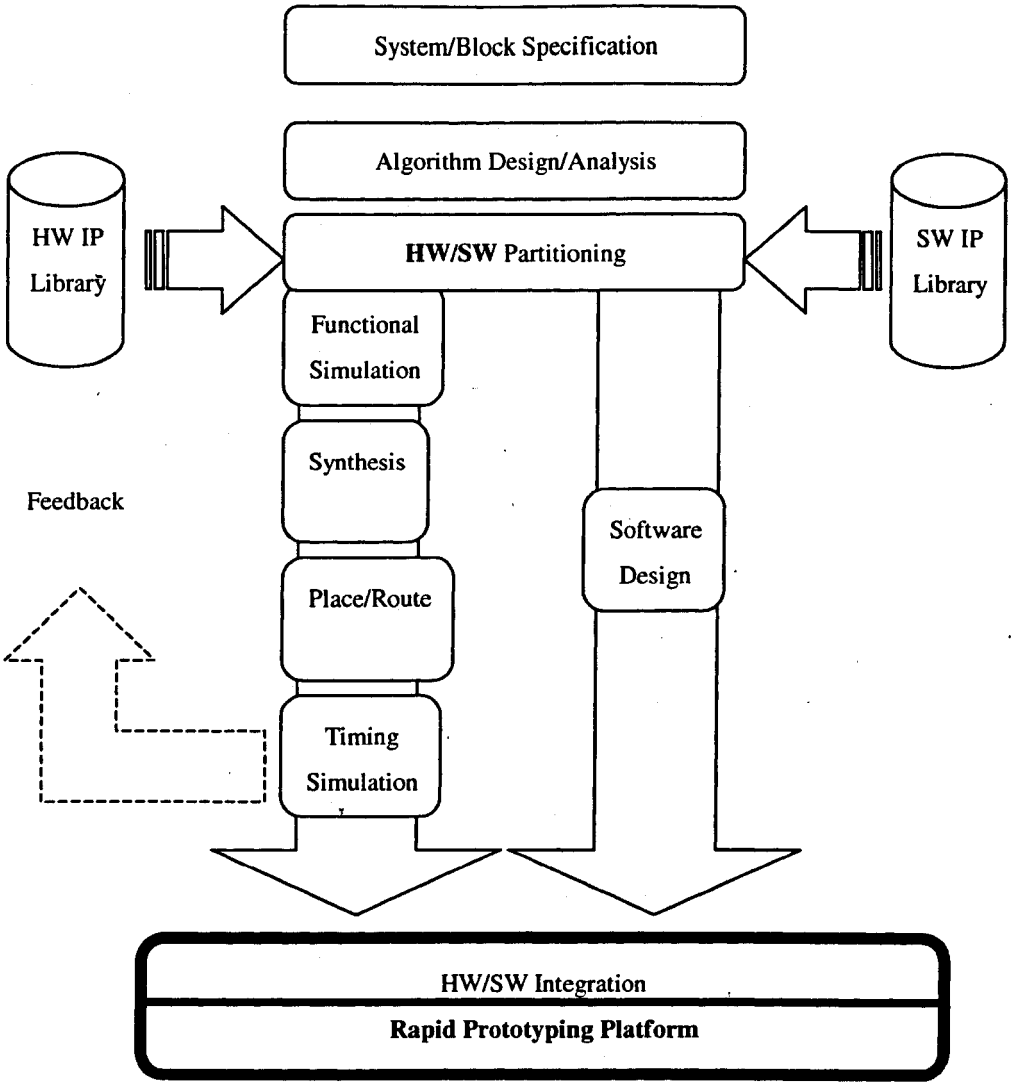


Figure 5.14: RPP Design Flow

5.5.2 Algorithmic Design and Analysis

In the next step of the process, the system designer translates the specifications into a high-level algorithmic description of the system. This algorithmic design and analysis step is usually implemented in C/C++ or in the case of DSP systems, tools like

Cadence SPW or Matlab can be used. The algorithmic description allows the designer to fully understand the system function before architectural details are developed. For example, in designing an audio filtering system using a finite-impulse response (FIR) digital filter, the designer can do the following steps before committing to implementation:

- Optimize the number of filter coefficients.
- Experiment with different windowing techniques to improve the filter response.
- Even test the effects of fixed-point arithmetic (e.g, overflows) versus floating-point.

A Cadence's SPW tool specifies the system as a set of connected, parameterized blocks (e.g, a FIR filter would be one block). A block diagram (filter block, signal sources and sinks, channel models, etc.) can be simulated many times using different parameter settings to investigate the algorithm and determine the optimum system.

5.5.3 System Architecture Design (HW/SW Partitioning)

Once the designer has thoroughly exercised and optimized the algorithm at this high level, the implementation process can begin. The first step is the system architecture design, where functional units are mapped to various architectural units. Traditionally this has been a manual process for relatively simple systems. However, as system-on-chip (SOC) designs become more and more complex and incorporate

more and more processing elements (e.g. embedded processors, DSPs, etc.), the design process requires tools like Cadence's Virtual Component Co-design (VCC). VCC allows system designers to specify the system function at a high level of abstraction (using C/C++ functional models). In a parallel activity, designers can model the system architecture (microprocessor, bus architecture, memory, ASIC components, RTOS, etc.) with appropriate estimates on timing, power and cost. The system designer can then map functional units onto architectural units and run a performance simulation to see if the function/architecture mapping is appropriate.

The designer can make changes to the system as well as mapping and re-simulate to further explore the system design space. For a large system, this kind of activity would be cumbersome and slow, if not impossible. Once this hardware/software partitioning stage is complete, the design is handed over to the hardware and software design teams. The system designer must create a specification for the hardware and software design teams (again stating constraints such as clock speed and power to guide the design processes).

5.5.4 HDL Hardware Coding

The hardware design flow uses a hardware description language, VHDL or Verilog, to create that portion of the system design. HDL coding can be done by hand (i.e. using a text editing program to write the code), or by using HDL design tools such as HDL Designer (formerly known as Renoir) from Mentor Graphics. HDL Designer allows the complete hardware sub-system to be specified as individual blocks,

interconnected by wires/buses. Design blocks can be regular RTL code, state diagrams, flow charts, truth tables or hierarchical block diagrams. HDL Designer can also create block diagrams from existing code, create testbenches and link to simulation and synthesis flow tools.

The HDL design process can be enhanced by the use of a hardware IP library. For example, Xilinx has a library of free IP cores in its LogiCORE product line. One of these cores is a parameterizable FIR filter core that designers can instantiate in their source code. The core comes with a simulation model and works in most commercial synthesis flows (e.g. Synopsys FPGA Compiler II). The LogiCORE library includes memory, DSP and mathematical functions.

5.5.5 Functional Simulation

Once the HDL is coded, it is verified through functional simulation (using a simulator such as Synopsys VSS or Cadence NC-Sim). The test bench is usually created alongside the HDL, but can also be derived earlier in the process (for example, reusing test data from the algorithmic design phase). This simulation is technology-independent and it does not contain timing or power data. The HDL code is modified and re-simulated until the function is verified. Some extra steps in this process (not shown in Figure 5.14) could include HDL linting (checking the code for compliance with coding standards), and test coverage. Test coverage involves evaluating how much of the actual code is tested by the test bench and thus the completeness of the verification strategy.

5.5.6 Synthesis

After functional simulation, synthesis process maps the RTL code to logic gates. Tools like FPGA Compiler II from Synopsys perform this task by using the constraints on timing to optimize the design. Designers can perform back-annotated timing simulation by using gate-level delays. Re-using the functional testbench step confirms that synthesis has not altered the design's functionality. This simulation also contains accurate timing information to confirm the operation of actual design within constraints

5.5.7 Place and Route

After synthesis, the designer performs place and route using the Xilinx design tools (e.g. Alliance 3.1i). This step maps the logic gates from synthesis to functional units on the FPGA. The output of this step is a bitstream file that is a complete map of the design, configured for a particular Xilinx part (e.g. Virtex 2000E-PQ540-6). This file can be downloaded to the corresponding Xilinx FPGA for operation.

5.5.7 Application Software

Occurring parallel to hardware design flow is the software development design flow that creates the code for execution on the processor in the system (in the case of RPP, this is an ARM7TDMI processor). Software development can involve several tools, including a real-time operating system (RTOS), instruction-set simulator (ISS) and code development tools (C/C++ compiler, linker, and assembler). An example of

an RTOS would be VRTX from Mentor Graphics. ARM has its own suite of code development tools called ARM Developer Suite (ADS). As in the hardware domain, pre-designed routines from a software IP library can speed up the design process.

5.5.8 HW/SW Integration

During the design of hardware and software, it is important to make sure that the hardware and software portions of the design will work together. They may be thoroughly tested separately by hardware and software simulation, but if the interfaces between software and hardware are not well-designed, the system may not function properly. Testing the interfaces can be left to the final stage that is the actual HW/SW integration on the rapid-prototyping platform. However, this can also be done earlier using a HW/SW co-verification tool like Seamless. This tool links a hardware simulation (e.g. NC-Sim) with simulation of the software on an ISS, this can verify the system earlier. Errors due to interactions between the HW and SW components can be caught early in the design cycle and save time and effort for both the hardware and software designers. After integration, the design is confirmed to be working. Experiments on the design that can execute in real-time with real-time data (e.g. audio data coming from a PC sound card) can fine-tune the design. Changes to hardware or software can be made, implemented and tested very quickly. At this stage, the design can be introduced into an ASIC design flow if desired, or simply stand as a proof-of-concept design where fabrication is not needed or perhaps not affordable.

5.6 Block Matching Implementation

5.6.1 Introduction to Block Matching

The block matching is an important computational part of MPEG video coding. MPEG has been by far the most popularly utilized motion estimation technique in video coding. It is interesting to note that even now a days, with the tremendous advancements in multimedia object and/or content-based manipulation of audio-visual information is very demanding particularly for multimedia data storage, retrieval, and distribution. The application of arbitrarily shaped objects has attracted significant research attention, and has been included in the MPEG activities.

Difficulties encountered in motion estimation and compensation with arbitrarily shaped blocks are enormous and to avoid these difficulties, the block matching technique was proposed by Jain and Jain [32]. An image is partitioned into a set of non-overlapped, equally spaced, fixed size, small rectangular blocks assuming a uniform translation motion within each block. This simple model considers translation motion while other types of motions such as rotation and zooming of large objects, may be closely approximated by piecewise translation of these small blocks provided that these blocks are small. As a compromise, a size of 16 x 16 is considered to be a good choice, (as specified in international video coding standards H.261, H.263, MPEG-1 and MPEG-2) while for finer estimation sometimes a block size of 8 x 8 is

also used. Displacement vectors for these blocks are estimated by finding their best matched counterparts in the previous frame. In this manner, motion estimation is significantly easier than that for arbitrarily shaped blocks. Furthermore, the rectangular shape information is known to both the encoder and the decoder, and hence does not need to be encoded, which saves both computation load and side information.

We use Figure 5.15 to illustrate the block matching technique. An image frame at moment t_n is segmented into non-overlapped p x q (block where) p = q = 16 are often used (Figure 5.15(a)). Consider one of the blocks centered at (x,y), it is assumed that the block is translated as a whole. Consequently, only one displacement vector needs to be estimated for this block. Figure 5.15(b) shows the previous frame at time t_{n-1}. In order to estimate the displacement vector, a rectangular search window is formed in frame t_{n-1} and centered at the pixel (x,y). Consider a pixel in the search window, a rectangular correlation window of the same size p x q is opened with the pixel located in its center and a similarity measure (correlation) is calculated. After the matching process completes for all candidate pixels in the search window, the correlation window corresponding to the largest similarity measure becomes the best match of the block under consideration in frame t_n. The relative position between these two blocks (the block and its best match) gives the displacement vector as shown in Figure 5.15(b).

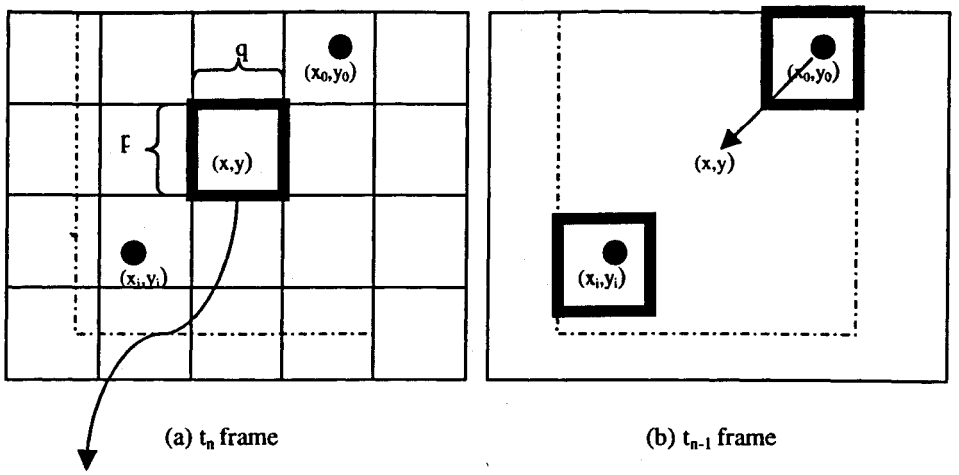


Figure 5.15: Block matching

The size of the search window is determined by the size of correlation window and the maximum possible displacement along four directions: up, down, right, and left. In Figure 5.16 these four quantities are assumed to be the same and are denoted by *d*, which is estimated from a priori knowledge about the translation motion. Translation motion includes the largest possible motion speed and the temporal interval between two consecutive frames ($t_n - t_{n-1}$).

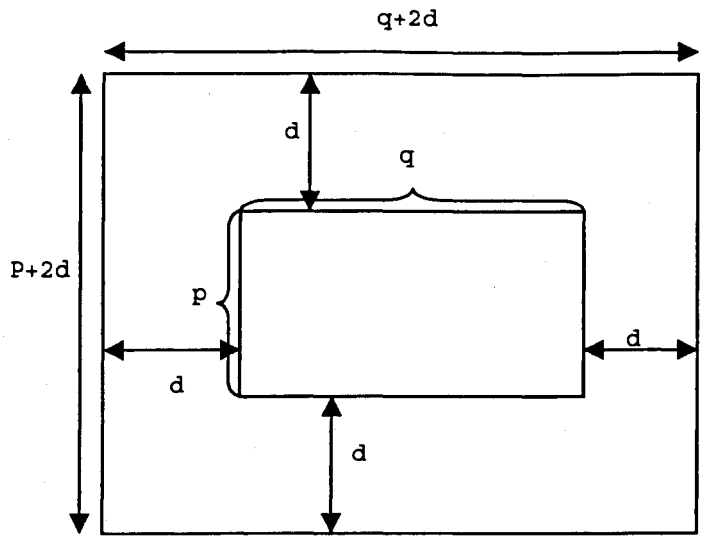


Figure 5.16: Search window and correlation windows

Block matching belongs to image matching and can be viewed from a wider perspective. In many image processing tasks, we need to examine two images or two portions of images on pixel-by-pixel basis. These two image regions can be selected either from a spatial image sequence (from two frames taken at the same time with two different sensors aiming at the same object), or from a temporal image sequences (from two frames taken at two different moments by the same sensor). The purpose is to determine the similarity between the two images or two portions of images. Therefore, the similarity measure is a key element in the matching process. However, instead of finding the maximum similarity, an equivalent but yet more computationally efficient way of block matching is to find the minimum dissimilarity or matching error. The dissimilarity D(s,t) (sometimes referred to as the error, distortion or distance) between two images t_n and t_{n-1} is defined as:

$$D(s,t)= \frac{1}{lm}\sum_{j=1}^{p}\sum_{k=1}^{q} M(f_n(j,k), f_{n-1}(j+s,k+t)) \qquad (5.1)$$

where M(u,v) is a metric that measures the dissimilarity between u and v.

The D(s,t) is also referred as the matching criterion or the D values. Mean square error (MSE) is most commonly used matching criterion where dissimilarity metric M(u,v) is defined as

$$M(u,v) = (u-v)^2 \qquad (5.2)$$

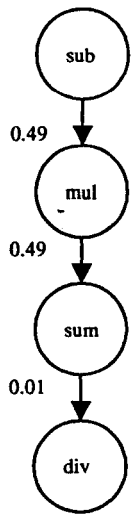
The searching strategy is another important issue to deal with in block matching. Figure 5.16 shows a search window, a correlation window and their sizes. In searching for the best match, the correlation window is moved to each candidate position within the search window. In this way, there are a total of (2d+1) x (2d+1) positions that need to be examined. The minimum dissimilarity gives the best match. Apparently, this full search procedure is brute force in nature. While the full search delivers good accuracy in searching for the best match, a large amount of computation is involved. We choose a full search method for implementation to increase the complexity of the system to make the problem more challenging. In the next section, we are familiarizing the readers to the Rapid Prototyping Platform (RPP) we employed for implementing the partitioned block matching system.

5.6.2 Specification

The most computationally intensive part of block matching is to calculate the matching criterion function as presented in Equation 5.1. After calculating the matching criterion of an image, a match is found and motion vector is calculated. The image size and the search space of our system is 256 x 256. Instead of reducing the search space to near by micro block of previous image as it is usually done to reduce the

computation, we have implemented a full search method for the whole 256 x 256 image. Micro block size is selected at 8 x 8 to increase the computation complexity. To find a matching 8 x 8 micro block, there are (256 - 8) rows and (256 - 8) columns in the image that requires 961 comparison operations. Furthermore, each comparison operation calculates the matching criterion function that requires 64 additions, subtractions, multiplications and one division operation. However, to take the advantage of burst transfer mode, each operation (addition, subtraction etc.) is performed for the whole image instead of per comparison. The bus speed is 20 MHz and the maximum bus transfer rate for an AMBA (AHB) system is 128 bits in burst mode. This means that instead of sending only 2048 bits (8 x 8 x 32 bits) per block matching with 961 separate bus transactions. There will be 961 block matching comparison operation for 256 x 256 image, we can send 1968128 (2048 x 961) bits in one bus transaction as a burst mode. This method has significantly reduced the communication overhead between different functional blocks.

Each image pixel represents a gray scale value from 0 to 255. The granularity level of block matching is chosen as operation level of subtraction, square, summation and division to calculate the matching criterion function. Each block has been created in both software and hardware to create library information for the partitioning algorithm (Figure 5.17). The execution time of each block is also measured using the hardware/software timer module developed in house. This tool allows execution time measurements of hardware and software components for the RPP system components. The hardware/software timer is an essential part of the partitioning process for an accurate simulation.



	SW	HW	Area (gates)
Sub	4.133s	0.5s	100
Square	6.1s	1.1s	500
Sum	4.133s	0.5s	100
Div	2.34	0.9s	700

Figure 5.17: Initial Block matching solution with library info

5.6.3 Software Simulation

The simulation results of block matching algorithm with various constraints are recorded and drawn in Figure 5.18. Each node in the graph represents an improvement to system execution time with the addition of one more hardware components. This simulated result shows that with the current granularity level and by employing hardware-software library of Figure 5.17, the system performance is within 18.23s → 5.284s as shown in Figure 5.18. However, if the system specification requires better performance (faster than 5.284s), a different granularity level must be selected with its corresponding hardware-software library. The change in hardware/software library will allow partitioning algorithm to explore different level of solution space. If the system specification is bounded in the region of 18.23s → 5.284, the hardware-software partitioned results generated by DADGP partitioning algorithm will give the estimated performance when implemented.

To verify the results of simulated solution of partitioning, we have implemented the hardware software partitioned system using RPP and measured the actual execution time of the system and compared it to the simulated results. The measurements of the two domains are similar that proves the accuracy of simulated results and verifies the partitioned hardware-software system as a valid solution. A more detailed comparison of the experiments is presented in a later section.

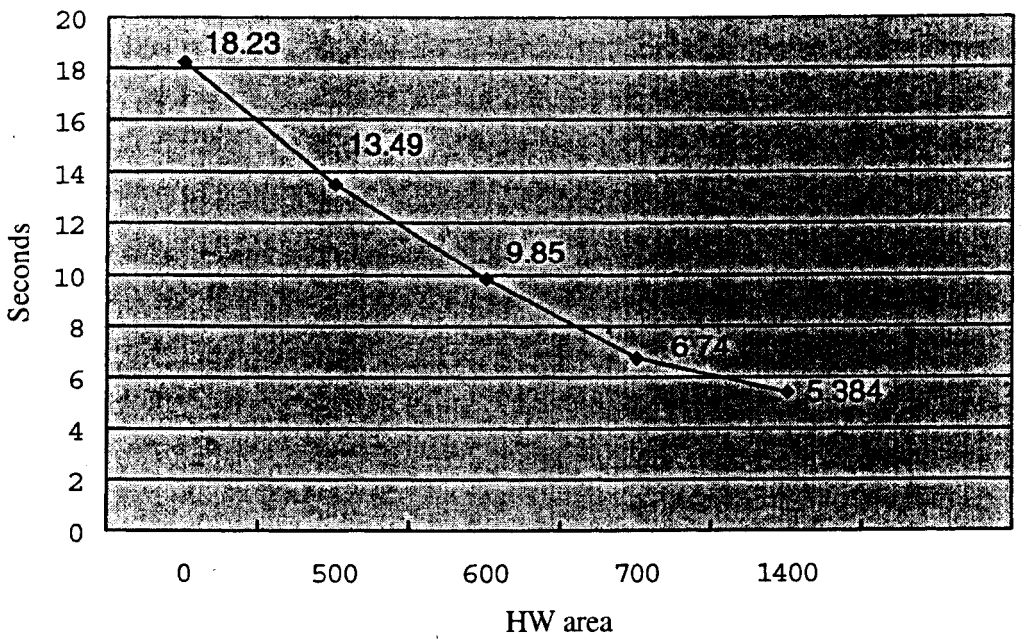


Figure 5.18: Simulated Performance Improvement Curve Block Matching

5.6.4 Overall Architecture

The overall system architecture of a partitioning solution is presented here for simplicity because the system architecture of other solutions are similar. The system

is implemented using RPP as incorporation of extra hardware modules is equivalent to loading the hardware component to the FPGA AHB bus system as shown in Figure 5.19. System uses the AHB (AMBA) bus system to connect ARM7 processor to FPGA. Therefore, all the sub components or the hardware blocks implemented in FPGA requires AHB bridge connection to communicate with other devices.

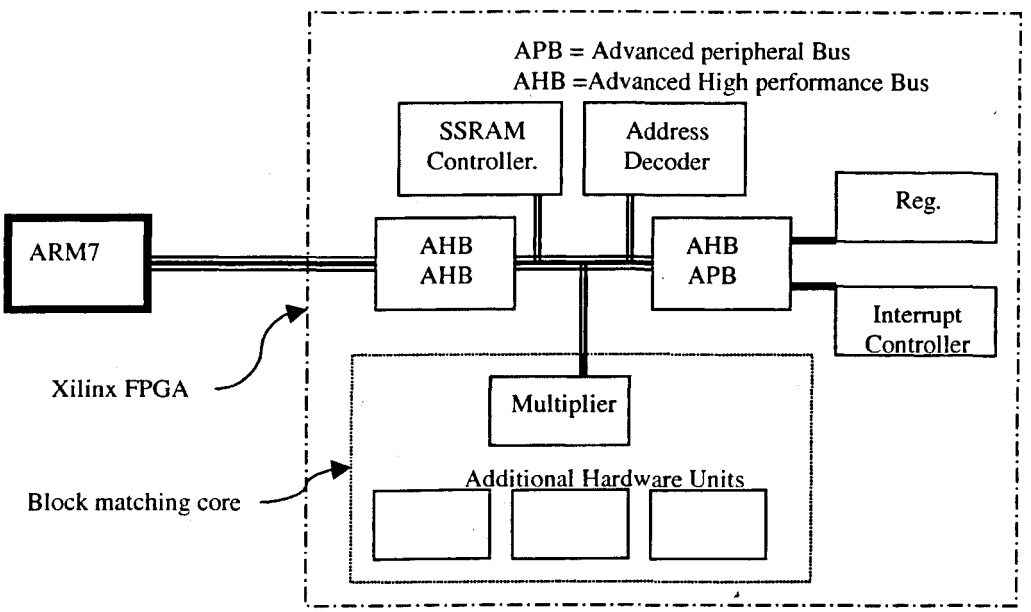


Figure 5.19: Overall system implementation

The simulated partitioning suggests that the very first solution to Block Matching DADGP shown in Figure 5.17 is to map the square operation to dedicated hardware. This partitioning result gives the most performance gain of 2.9 as compared to previous solution. The multiplier implemented for this operation is shown in Figure 5.20 that is a parallel 32 bit multiplier. The parallel 32-bit multiplier is an unsigned multiplier using a carry save adder structure. A partial schematic of the multiplier is also shown in

Figure 5.21. This multiplier takes two 32 bit numbers and multiply them in parallel to generate a 64-bits result.

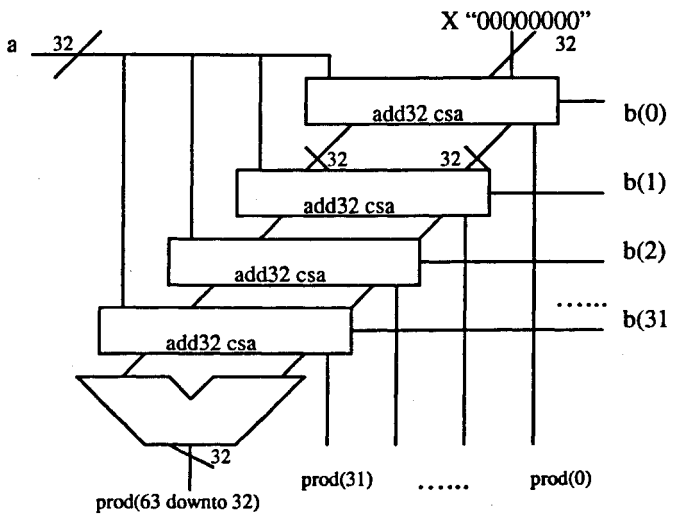


Figure 5.20: 32 bit parallel multiplier

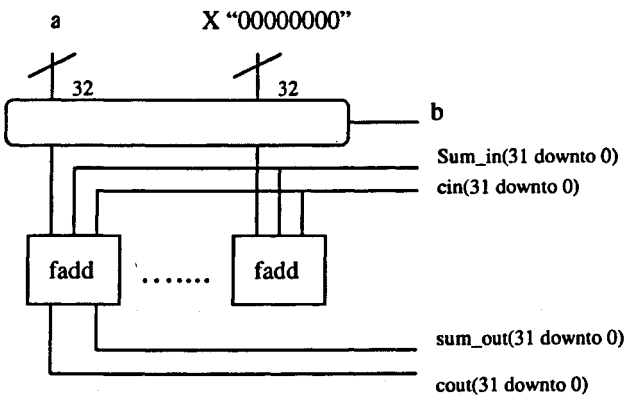


Figure 5.21: 32 bit carry save adder

5.6.5 Simulation vs. Actual Implementation

The implementation of mapping square function to the hardware unit has shown similar results to software simulation. The implemented system takes 13.7133 seconds to complete, where as the software simulation estimated 13.49 seconds for completion. The results indicate a margin of error of 1.66 %. The rest of the implementation vs. the software simulation comparison is shown in Table 5.2. The results show that the margin of error of all the partitioned solutions is within 2%.

The difference between the simulated hardware area and the actual hardware area indicated a very small margin of error as shown in Table 5.3. The main sources of errors are due to ignoring the routing and interconnections area between the components. Furthermore, as more hardware is added to the system, the margin of error increases. These results show that a different scheme of estimating the hardware area is required to accurately model the addition of multiple hardware units.

Table 5.2: Execution time comparison result

Iteration	Software simulation	RPP Actual measurements
1st run	18.23s	18.94s
2nd run	13.49s	13.7133s
3rd run	9.85s	10.55s
4th run	6.74s	7.21s
5th run	5.384s	5.88s

Table 5.3: Hardware area comparison result

Iteration	Software simulation area	RPP Actual area measurements
1st run	0	N/A
2nd run	500	512
3rd run	600	634
4th run	700	755
5th run	1400	1490

5.7 SOBEL Edge Detection Implementation

5.7.1 Introduction to SOBEL edge detection

The Sobel operator performs a 2-D spatial gradient measurement on an image. Typically this is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The Sobel edge detector uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction and the other estimating the gradient in the y-direction. A convolution mask is usually much smaller than the actual image. As a result, the mask is convolved and sled over the image, manipulating a square of pixels at a time. The actual Sobel masks are shown in Figure 5.22.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 5.22: SOBEL masks

The magnitude of the gradient is then calculated using the formula:

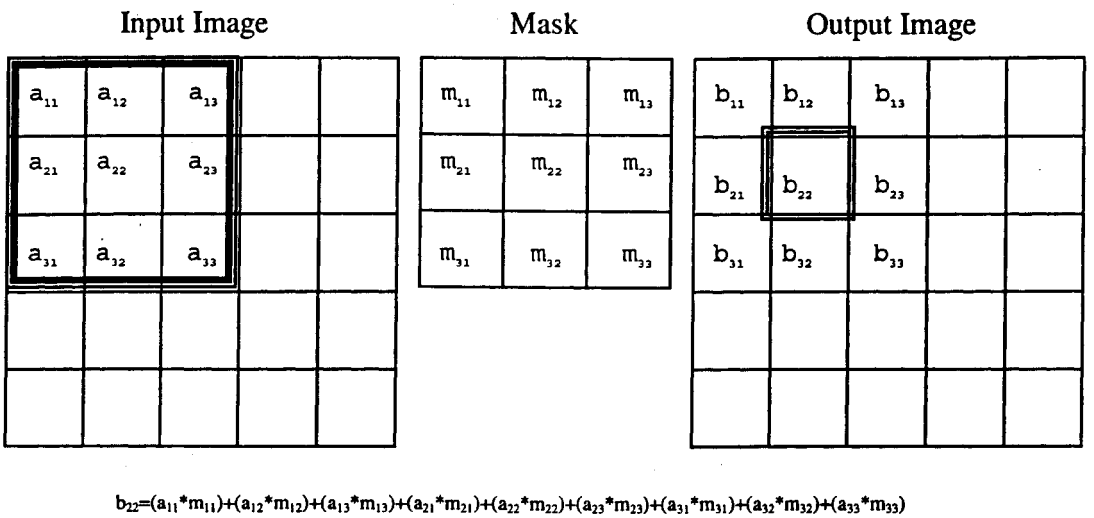
$$|G| = \sqrt{Gx^2 + Gy^2} \qquad (5.3)$$

An approximate magnitude can be calculated using:

$$|G| = |Gx| + |Gy| \qquad (5.4)$$

The mask is convolved over an area of the input image and then shifts one pixel to the right and continues to the right until it reaches the end of a row. It then starts at the beginning of the next row. The example in Figure 5.23 shows the mask being convolved over the top left portion of the input image represented by the thick black box. The formula shows how a particular pixel in the output image would be calculated. The center of the mask is placed over the pixel you are manipulating in the image. It is important to notice that pixels in the first and last rows, as well as the first and last columns cannot be manipulated by a 3x3 mask. This is because when placing

the center of the mask over a pixel in the first row (for example), the mask will be outside the image boundaries.



$$b_{22}=(a_{11}*m_{11})+(a_{12}*m_{12})+(a_{13}*m_{13})+(a_{21}*m_{21})+(a_{22}*m_{22})+(a_{23}*m_{23})+(a_{31}*m_{31})+(a_{32}*m_{32})+(a_{33}*m_{33})$$

Figure 5.23: SOBEL example

The G_X mask highlights the edges in the horizontal direction while the G_Y mask highlights the edges in the vertical direction. After taking the magnitude of both, the resulting output detects edges in both directions. A SOBEL edge detection algorithm is given below.

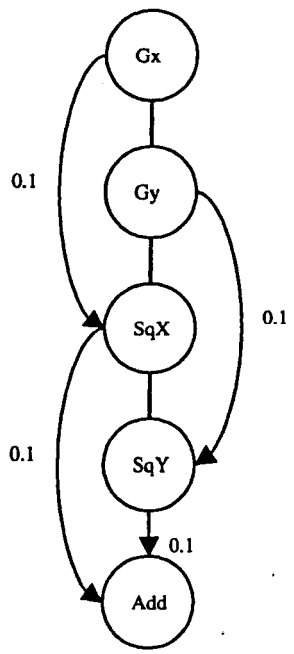
Sobel Edge Detection Algorithm

```
#define ROWS 512
#define COLS 512
#define Threshold 100
main()
{
    unsigned char image_in[ROWS][COLS];
    unsigned char image_out[ROWS][COLS];
    int r, c; /* row and column array counters */
    int pixel; /* temporary value of pixel */
   for (r=0; r<ROWS; r++) /*initialize output image
array*/
          for (c=0; c<COLS; c++)
                image_out[r][c] = 0;
/*filter the image and store result in output array */
     for (r=1; r<ROWS-1; r++)
        for (c=1; c<COLS-1; c++) {
        /* Apply Sobel operator. */
          pixel = image_in[r-1][c+1]-image_in[r-1][c-1]
          + 2*image_in[r][c+1] - 2*image_in[r][c-1]
         + image_in[r+1][c+1] - image_in[r+1][c-1];
        /* Normalize and take absolute value */
          pixel = abs(pixel/4); /* Check magnitude */
        if (pixel > Threshold)
          pixel= 255; /*EDGE_VALUE;*/
        /* Store in output array */
        image_out[r][c] = (unsigned char) pixel;
        }
}
```

5.7.2 Specification

The input to SOBEL edge detector system is 256 x 256 gray scale images. The most computationally intensive part of SOBEL edge detection is to calculate the gradients in x and y direction by using 3x3 convolution masks. The system's granularity level is chosen as operation level of square (SqX, SqY) and summation (add). However, for the calculation of gradient in x and y directions, two smaller operations have been encapsulated (multiplication and addition, Gx, Gy). Dividing the system in this fashion has allowed identical amount of data transfer for all blocks (256 x 256 x 32 bits). Therefore, the communication time between any blocks remains constant.

Each block has been created in both software and hardware to create the library information for our partitioning algorithm as shown in (Figure 5.24). The interesting part of this system compare to block matching system is that, its initial DADGP graph contains precedence dependence edges. This means that with the environment, the partitioning algorithm will be able to explore parallelism that will result in modification of the DADGP representation as shown in the next section.



	SW	HW	Area (gates)
Gx/Gy	9.4s	1.4s	1200
SqX/SqY	5.2s	0.9s	500
Add	3.88s	0.3s	100

Figure 5.24: Initial SOBEL solution with library info

5.7.3 Software simulation

The simulation results of SOBEL edge detection is drawn in Figure 5.25. Each node in the graph represents an improvement in overall execution time by the addition of one more hardware component. The simulated result shows that with the current granularity level and hardware/software library, the system performance is

within range 33.8s → 2.8s. However, a different granularity level must be selected for a faster system. One possible solution of improving the system is to combine two or more functional units into one hardware unit to improve the execution time and to reduce inter PE communication. The change in granularity level and hardware/software library will allow DADGP-based partitioning to explore different local minimum solution.

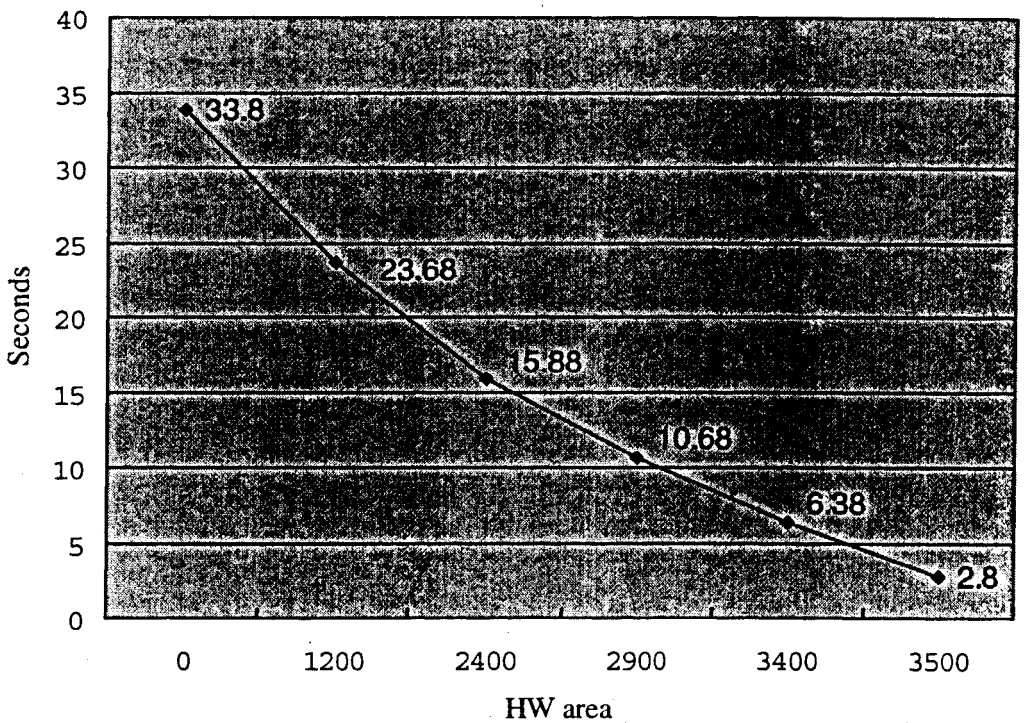


Figure 5.25: Simulated Performance Improvement Curve (SOBEL)

Figure 5.26 shows how the initial DADGP structure changes by the proposed DADGP based partitioning. The dotted circle node represents a hardware mapped modules, while dotted line enclosing several nodes represent an LD Path. This simulation is executed by allowing the partitioning algorithm to explore the solution

space based on a large hardware constraint. In this way, the partitioning algorithm will make the best move according to the execution time. It is observed that the Longest Delay path changes as DADGP changes and the improvement ratio of the solution decreases. This shows that the DADGP partitioning algorithm always make the best performance improvement moves. However, if the hardware area constraint is considered such that the best performance improvement can not be chosen due to hardware area violation, then the next best performance improving solution is chosen without violating the hardware area constraint. For example, in Figure 5.26 (a), the best performance improvement move is to choose Gx node as hardware, however if the hardware constraints is less than 1200 gates, the partitioning algorithm chooses the next best move by mapping SqX node to hardware. Therefore, depending on the hardware area constraint, the partitioning algorithm suggests various sub optimal solutions.

Similar to block matching, to verify the result of the simulation result as shown in Figure 5.25, we have implemented the partitioned system using RPP and measured the execution time of the system and compared it to the software simulated results. The measurements of the two domains are very close to each other which prove the accuracy and validity of simulated results. A more detailed comparison of this experiment is presented in the next section.

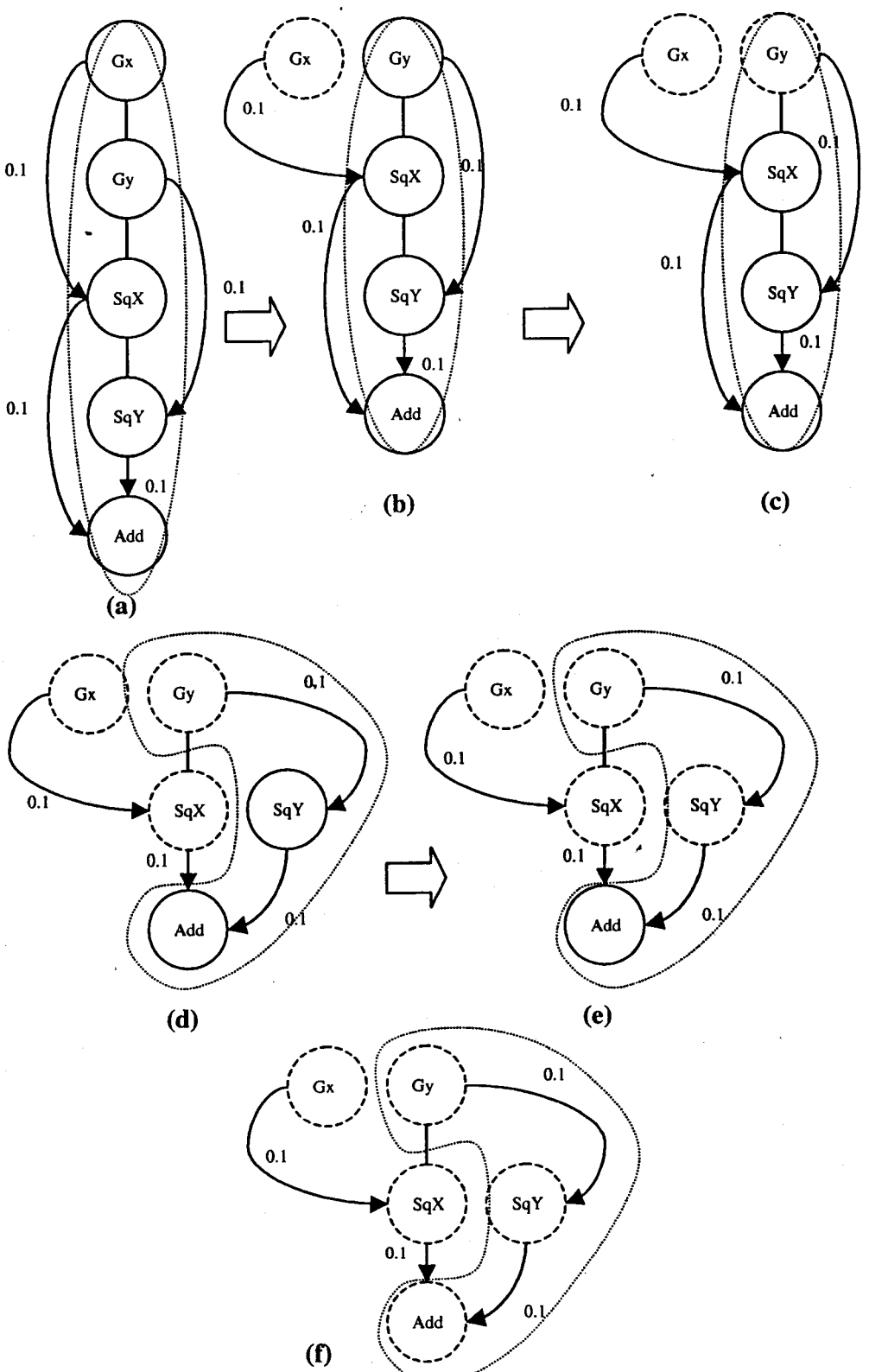


Figure 5.26: SOBEL DADGP solution set

5.7.4 Simulation vs. Actual Implementation

All of the solution space explored by the software simulation and hardware implementation are shown in Tables 5.4 and 5.5. The measured results of implementation as well as the software simulation comparison are presented in Table 5.4. The results of simulated vs. actual measurements show small margin of errors, indicating that the software simulation accurately models the hardware-software interactions. The difference between the simulated hardware area and the actual hardware area showed some margin of errors (Table 5.5). As mentioned previously, the main source of error is due to area required for routing and interconnections between hardware components that has not been taken into account by our algorithm.

Table 5.4: Execution time comparison result

Iteration	Software simulation	Actual measurements
1st run	33.08s	33.88s
2nd run	23.68s	24.05s
3rd run	15.88s	16.22s
4th run	10.68s	10.96s
5th run	6.38s	6.82s
6th run	2.8s	2.92s

Table 5.5: Hardware area comparison result

Iteration	Software simulation area	Actual area measurements
1st run	0	N/A
2nd run	1200	1215
3rd run	2400	2455
4th run	2900	2997
5th run	3400	3525
6th run	3500	3700

Chapter 6
Conclusions and Future Work

6.1 Summary and Conclusions

We have introduced the concept of hardware-software co-design methodology. Various hardware software portioning methods are presented and compared to the DADGP-based system partitioning algorithm. The thesis presents a full design flow from specification to implementation; using C/C++ language as system specification, DADGP based partitioning algorithm as design simulation tool, and partitioned system implementation on a rapid prototyping platform consisting of ARM7 CPU and Xilinx FPGA.

Directed Acyclic data Dependence Graph with Precedence (DADGP) is an extension of DAG. DADGP-based partitioning algorithm can also work with DAG by converting DAG to DADGP. This characteristic has allowed us to compare the performance of DADGP-based partitioning with other partitioning methods that use DAG as an input graph. The results demonstrate superior performance of DADGP as compared to GDL and Simulated Annealing methods in terms of simulation time and quality of the partitioned solutions. The DADGP partitioning algorithm not only produces an optimal partitioned solution for a given initial graph, but it also gives other partitioned solution between initial and optimal solution. This characteristic is very important because various sub optimal solutions give more choices to the designer in terms of system cost and performance gain.

We demonstrated the verification of our DADGP partitioning technique. Two computationally intensive algorithms namely Block Matching and SOBEL edge detection have been designed and implemented using the DADGP design flow. The results indicate that the performance gain for software simulated solution is very close to the actual system performance measured for both applications. However, the hardware area estimation measurement is not accurate as our partitioning method does not consider the interconnection hardware area between multiple hardware units. Overall, the DADGP partitioning algorithm showed promising results. The incorporation of DADGP partitioning algorithm, C/C++ profiling, and rapid prototyping to the hardware-software co-design methodology has significantly reduced the design complexity of embedded systems.

6.2 Future Work

Followings are some of the directions of future research on hardware software partitioning using DADGP algorithm:

- More accurate measure of estimating hardware area and its interconnection is required.
- A more diverse and dynamic sets of hardware and software library need to be developed. This improvement will allow DADGP partitioning algorithm to generate profound hardware software partitioning.
- An automated approach to granularity selection is necessary to explore broader solution space. Currently, the granularity level of system is selected manually by the designer from experience.

- An automated hardware software synthesis system is required to fully automate the design of hardware software systems. Currently, when the partitions are decided, the system is manually integrated with the required glue logic to connect hardware and software components. Similar gluing systems are available from some FPGA vendors.
- Finally, a uniform design environment is necessary to simplify the use of DADGP-based partitioning solution.

References

1) Lee Garber and David Sims, "In pursuit of Hardware-Software Codesign", IEEE Computer, Vol.31, No.6, pp. 12-14, June 1998.
2) M. Eisenring, L. Thiele and E. Zitzler, "Conflicting Criteria in Embedded System Design", IEEE Design & Test of Computers, Vol. 17, No.2, pp. 51-59, April-June, 2000.
3) Stan Y. Liao, "Towards a New Standard for System-level Design", in Proc. of the Eighth Int. Workshop on Hardware/Software Codesign, pp.2-6, 2000.
4) W. Hardt, "An automated approach to HW/SW-codesign [Hardware/software partitioning]", IEE Colloquium, on partitioning in Hardware Software Codesign pp. 4/1-4/11,Feb 1995.
5) M. D. Edwards and J. Forrest, "Hardware/software partitioning for performance enhancement", in Proc. of IEE Colloquium on Partitioning in Hardware Software Codesign, pp. 2/1-2/5, February 1995.
6) D.R. Sadler, D.W. Lloyd and I.E. Jelly, "Object-based Hardware-Software Co-design", in Proc. of the IEEE 15 Annual Int. Computers and Communications pp. 282-288, March, 1996.
7) SystemC Inc. http://www.systemc.com
8) R. B. Ramakrishna and M. S. Schlansker, "Embedded Computer Architecture and Automation", IEEE Computer, Vol.34, No.4, pp. 75-83, April 2001.
9) F. Slomka, M. Dorfel, R. Munzenberger and R. Hofmann, "Hardware/Software Codesign and Rapid Prototyping of Embedded Systems", IEEE Design & Test of Computers, Vol. 17, No. 2, pp. 28-38, April-June 2000.
10) M. Jin and G.N. Khan, "Heterogeneous haradware-software system partitioning using extended directed acyclic graph", Parallel and distributed computing systems, in Proc. of the ISCA 16 Int. Conf. pp. 181-187, Reno Aug. 2003.
11) G.D. Micheli and R.K. Gupta, "Hardware/Software Co-design", in Proc. of the IEEE, Vol. 85, No.3, pp. 349-365, March 1997.
12) P. Eles, Z. Peng, K. Kuchcinski and A. Doboli. "System Level Hardware/Software Partitioning based on Simulated Annealing and Tabu Search.", Design Automation for Embedded Systems, Vol.2, No.1, pp. 5-32, January 1997.
13) Mentor Graphics Corp. Seamless Co-verification. http://www.mentorgraphics.com/seamless
14) Synopsys Inc. Eaglei tools. http://www.synopsys.com/eagle

15) R.P. Kurshan, Automata-Theoretic Verification of Coordinating Processes. Princeton, NJ, Princeton Univ. Press, 1994.
16) Rolf Ernst, "Codesign of Embedded Systems: Status and Trends", IEEE Design & Test of Computers, Vol.15, No.2, pp. 45-53, April-June.
17) K. Pramataris, G. Lykakis and G. Stassinopoulos, "Hardware/Software co-simulation methodology based on two alternative approaches", in Proc. of the 6th IEEE Int. Conf. on, Vol.1, pp. 63-66,1999.
18) G.C. Sih and E.A. Lee, "A compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", IEEE Trans. Parallel and Distributed Systems, Vol. 4, No.2, pp. 175-187, February 1993.
19) F. Schirrmeister and A.S. Vincentelli, "Virtual Component Co-design – Applying function architecture co-design to automotive applications", Vehicle Electronics, in Proc. of the IEEE Int, Conf., pp. 221-226, Sept. 2001.
20) M. Potkonjak and W. Wolf, "Cost optimization in ASIC implementation of periodic hard-real time systems using behavioral synthesis techniques," Computer-Aided Design, in Proc. of the IEEE Int Conf. ICCAD, pp 446-451, Nov. 1995.
21) M.L. Lopez Vallejo, C.Carreras, J.C. Lopez, and L. Sanchez, "Coarse Grain Partitioning for Hardware-Software Co-design". In Proc. of the 22nd Euromicro Conf., pp 161-167, Sep 1996.
22) A N. Ngoc, M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIP's with least gate counts," in Proc. of the 33rd DAC, pp. 527-532, June 1996.
23) Saha, D. Mitra, and R.S. Basu, "Hardware software partitioning using genetic algorithm" in Proc of the Tenth Int. Conf. VLSI Design, pp. 155-160, 1997.
24) E.A. Lee and A. Kalavade "The extended partitioning problem: hardware/software mapping and implementation-bin selection", in Proc. of the Sixth IEEE Int. Workshop on Rapid System Prototyping, pp. 12-18, 1995.
25) H. Ondghiri, B. Kaminska and J. Rajski, "A hardware/software partitioning technique with hierarchical design space exploration", in Proc. of the IEEE Conf. on Custom Integrated Circuits, pp. 95-98, 1997.
26) N. Togawa, T. Sakurai, M. Yanaglsawa and T. Ohtsuki, "A hardware/software partitioning algorithm for processor cores of digital signal processing", in Proc. of the Asia and South Pacific Design Automation Conf., the ASP-DAC '99, Vol. 1, pp.335-338, 1999.

27) F. Vahid. "Modifying Min-Cut for Hardware and Software Functional Partitioning". In Proc. of the Workshop on HW/SW Co-Design CODES/CASHE'97, pp. 43, Mar 1997.

28) R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers". IEEE Design & Test of Computers, pp 64-75, Dec. 1993.

29) M.L. Lopez-Vallejo, J Grajal and J.C. Lopez, "Constraint-driven system partitioning", in Proc. of the Design, Auto and Test in Europe Conf., pp. 411-416, 2000,

30) D.F. Wong, H.W. Leong and C.L. Lin, *Simulated Annealing for VLSI Design.* Norwell: Kluwer Academic Publishers, 1988.

31) T. Cormen, C. Leirserson and R. Rivest, *Introduction to Algorithms.* Cambridge: MIT Press, 1990.

32) J.R. Jain and A.K.Jain, "Displacement measurement and its application in interframe image coding", IEEE Trans. Commun., Vol. 29,No.12, pp.1799-1808, Dec. 1981.

Appendix A: Block Matching Implementation Code

Software Components

```
/*
 * bmTools.h
 * DEFINITION :
 * Definitions common to all of the block
matcher implementaitons
 */
#ifndef BM_TOOLS_H
#define BM_TOOLS_H
#include "dpramSemaphore.h"
#include "cveUtils.h"
/* Pointers to the different important
areas in memory */
#define PATTERN1     ((unsigned
long*)0x80001000) /* Pattern bank 1 */
#define IMAGE1       ((unsigned
long*)0x80001100) /* Image bank 1   */
#define RESULT       ((unsigned
long*)0x80003000) /* Result area    */
#define PATTERN2     ((unsigned
long*)0x80002000) /* Pattern bank 2 */
#define IMAGE2       ((unsigned
long*)0x80002100) /* Image bank 2   */
#define MBR          ((unsigned
long*)0X80000004) /* Right mailbox  */
#define MBL          ((unsigned
long*)0X80000000) /* Left mailbox   */
#define INT1         (volatile unsigned
long*)0x20000000 /* Interrupt Pin 1 */
#define INT2         (volatile unsigned
long*)0x20000001
#define INT3         (volatile unsigned
long*)0x20000002
#define INT4         (volatile unsigned
long*)0x20000003
#define INTMASK1     (volatile unsigned
long*)0x20000004 /* Interupt Mask */
#define INTMASK2     (volatile unsigned
long*)0x20000004
#define INTMASK3     (volatile unsigned
long*)0x20000004
#define INTMASK4     (volatile unsigned
long*)0x20000004
#define SEM_BASE     ((unsigned
long*)0xE0000000) /* Semaphore area */
#define CVE_SCREEN   ((unsigned
char*)0xC0000000) /* Seamless "Console" */
#define EVENT1       ((unsigned
long*)0x40000000) /* Event generator */
/* Pattern and image sizes */
#define BM_PATTERN_WIDTH  8
#define BM_IMAGE_WIDTH  16

/* Map the processor and coprocessor
mailboxes to left and right */
/* respectively
*/
#define PROC_MB  MBL
#define COPRO_MB MBR

/* Definition of the pattern structure */
typedef struct pattern_type
  {
  unsigned long xPosition;  /* x Position
of pattern in image */
  unsigned long yPosition;  /* y Position
of pattern in image */
  unsigned long bitmap;     /* bit map of
1 scan line of the pattern */
  } pattern;

/* Extern declarations for modules
including this file */
extern pattern bank1Pattern[3];
extern pattern bank2Pattern[3];

/* Function prototypes */
void init(void);
void write_pattern(pattern pat, unsigned
long* image, unsigned long* block);
int check_result(pattern pat);

#endif

/* bmTools.C
 * BM TOOLS block matcher test functions
library
 * DEFINITION :
 * These functions are common to all of
the block matcher implementations.*/
#include "bmTools.h"
/* Test data receptacles */
pattern bank1Pattern[3];
pattern bank2Pattern[3];
/*****************************************
 write_pattern()
 This function is a quick and easy way to
generate test data. It
 copies a pattern into a memory bank and
the generates an image
 with the given pattern at the coordinates
given by the pat parameter
 INPUT:
  pattern pat :  A struct containing the
```

```
coords and the bitmap of the pattern in
the image
unsigned long* image : Pointer to memory
for the image containing the pattern
unsigned long* block : Pointer to memory
that will contain the desired pattern
OUTPUT:
   none
*****************************************/
void write_pattern(pattern pat, unsigned
long* image, unsigned long* block)
  {
  int x,y;

  /* Initialize the image */
  for (y=0;y<16;y++)
    {
    for (x=0;x<16;x++)
      {
      image[16*y+x]=0;
      }
    }

  /* Copy the pattern into BLOCK and at
the desired position in */
  /* the image
*/
  for (y=0;y<8;y++)
    {
    for (x=0;x<8;x++)
      {
      block[8*y+x]=pat.bitmap;
      image[16*(y + pat.yPosition) +
x+pat.xPosition] = pat.bitmap;
      }
    }

  }

/*****************************************
 check_result()
 Verifies if the coprocaeesor found the
pattern at the right place.
 INPUT:
pattern pat: stuct containing the
coordinates in the image that the pattern
was written to.
 OUTPUT:
true (nonzero) if the coprocessor returned
the right coordinates.false otherwise
*****************************************/
int check_result(pattern pat)
  {
  int position;

  position =
16*pat.yPosition+pat.xPosition;

  return (RESULT[0] == (unsigned
long)position);
  }

/*****************************************
 init()
 Generates a set of test data.
 INPUT:
   none
 OUTPUT:
   none
*****************************************/
void init(void)
  {
  bank1Pattern[0].xPosition = 8;
  bank1Pattern[0].yPosition = 8;
  bank1Pattern[0].bitmap =    0xAAAAAAAA;

  bank1Pattern[1].xPosition = 2;
  bank1Pattern[1].yPosition = 1;
  bank1Pattern[1].bitmap =    0xBBBBBBBB;

  bank1Pattern[2].xPosition = 8;
  bank1Pattern[2].yPosition = 9;
  bank1Pattern[2].bitmap =    0xCCCCCCCC;

  bank2Pattern[0].xPosition = 0;
  bank2Pattern[0].yPosition = 0;
  bank2Pattern[0].bitmap =    0xDDDDDDDD;

  bank2Pattern[1].xPosition = 8;
  bank2Pattern[1].yPosition = 9;
  bank2Pattern[1].bitmap =    0xEEEEEEEE;

  bank2Pattern[2].xPosition = 2;
  bank2Pattern[2].yPosition = 2;
  bank2Pattern[2].bitmap =    0xABCDABCD;
  }

/* bm_int.C
 *  -- Block Matcher with Synchronization
Via Interrupts --
 * DEFINITION :
 *  This program writes images in memory
and notifies the
 *  coprocessor by generating a direct int.
to INTREG
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#include "bmTools.h"

#define IMAGE1_DONE 0x01
#define IMAGE2_DONE 0x02

// Function prototypes
extern "C"{
  void irq_handlerFunc(void);
}

int imagebank;
int image1loop;
int image2loop;
int ready1;
int ready2;

/****************************************\
 void main()
 Need we say more?
 INPUT
   none
 OUTPUT
   none
\****************************************/
int main(void)
  {
  /* Indicate that we can write to both
banks */
  ready1 = 1;
  ready2 = 1;

  /* some init */
  imagebank = 1; // we start in image bank
#1
  image1loop = 0;
  image2loop = 0;
  init();

  /* Welcome message -- The output window
sould pop to display this
     message */
  out_string("INTERRUPTIONS -  EPM CIRCUS
DEMO 2001\n\n");

  /* setting interrupt #1 mask */
  *INTMASK1 = 1;

  // embedded softwares have no limit!
that's infinite baby!
  for(;;)
    {

    while(ready1 == 0); /* wait until
memory bank 1 is free */
    /* Write the pattern, then send "mail"
to the coprocessor */
    write_pattern(bank1Pattern[image1loop],
IMAGE1, PATTERN1);
    ready1 = 0;
    *EVENT1 = 1;
     while(ready2 == 0); /* wait until
memory bank 2 is free */
    /* Write the pattern, then send "mail"
to the coprocessor */
    write_pattern(bank2Pattern[image2loop],
IMAGE2, PATTERN2);
    ready2 = 0;
    *EVENT1 = 2;
    }
  }

/****************************************/
 void irq_handlerFunc()
 Function called by the low-level
interrupt handler.
 INPUT
   none
 OUTPUT
   none
\****************************************/
void irq_handlerFunc()
  {
    unsigned long reg;

    if (reg= (*(INT1)) == 1) // look for
int #1
      {
        *(INT1) = reg; // clear int by
writing it back

        switch (imagebank)
          {
          case IMAGE1_DONE:

if(check_result(bank1Pattern[image1loop]))
              out_string("1: Image
found\n");
            else
              out_string("1: Image not
found\n");

            image1loop++;
            if (image1loop >= 3)
image1loop = 0;

            ready1 = 1;
            imagebank++; // switch to next
image bank
```

```
            break;

          case IMAGE2_DONE:

if(check_result(bank2Pattern[image2loop]))
              out_string("2: Image
found\n");
            else
              out_string("2: Image not
found\n");
            image2loop++;
            if (image2loop >= 3)
image2loop = 0;
            ready2 = 1;
            imagebank--; // switch to next
image bank
            break;

          default:
            out_string("Invalid
interruption!!\n");
            break;
          }
      }
    else
      {
        out_string("Invalid
interruption!!\n");
      }

}
```

VHDL Components

```
--Add32
-- add32.vhdl    entity add32 and
behavioral architecture

library IEEE;
use IEEE.std_logic_1164.all;
entity add32 is
  port(a    : in  std_logic_vector(31
downto 0);
       b    : in  std_logic_vector(31
downto 0);
       cin  : in  std_logic;
       sum  : out std_logic_vector(31
downto 0);
       cout : out std_logic);
end entity add32;

library IEEE;
use IEEE.std_logic_arith.all;  -- defines
"+" on unsigned
architecture behavior of add32 is
  signal temp : std_logic_vector(32 downto
0);
  signal vcin : std_logic_vector(32 downto
0) := X"00000000"&'0';
  signal va   : std_logic_vector(32 downto
0) := X"00000000"&'0';
  signal vb   : std_logic_vector(32 downto
0) := X"00000000"&'0';
  -- 33 bits (32 downto 0) needed to
compute cout
begin  -- circuits of add32
  vcin(0) <= cin;
  va(31 downto 0) <= a;
  vb(31 downto 0) <= b;
  temp <= unsigned(va) + unsigned(vb) +
unsigned(vcin);
  cout <= temp(32) after 10 ps;
  sum  <= temp(31 downto 0) after 10 ps;
end architecture behavior;  -- of add32
--Divider
-- div_ser.vhdl  division implemented as
serial adds (one 32 bit adder)
--               needs component  add32
-- non restoring division  (remainder may
need correction - in this case
--                          add divisor,
because remainder not same sign
--                          as dividend.)

entity div_ser is  -- test bench for
divide serial
end div_ser ;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

architecture schematic of div_ser is
  subtype word is std_logic_vector(31
downto 0);
  --    85 / 7 = 12 with remainder 1
(FFFFFFFA + 00000007 = 00000001)
  signal md      : word := x"00000007";  -
- multiplier or divisor
  signal hi      : word := x"00000000";  -
- top of dividend (final remainder)
  signal lo      : word := x"00000055";  -
- bottom of dividend
  signal cout    : std_logic;            -
- adder carry out
  signal divs    : word := x"00000000";  -
- adder sum
  signal diva    : word := x"00000000";  -
- shifted dividend
  signal divb    : word := x"00000000";  -
- multiplexor output
  signal quo     : std_logic := '0';     -
- quotient bit
  signal sub_add : std_logic := '1';     -
- subtract first (also cin)
  signal clk     : std_logic := '0';     -
- system clock
  signal divenb  : std_logic := '1';     -
- divide enable
  signal divclk  : std_logic := '0';     -
- run division
  signal cntr    : std_logic_vector(5
downto 0) := "000000";  -- counter
begin  -- schematic
  clk <= not clk after 5 ns;    -- 10 ns
period
  cntr <=
unsigned(cntr)+unsigned'("000001") when
clk'event and clk='1';
          -- cntr statement is equivalent
to six bit adder and clocked register

  divenb <= '0' when cntr="100001";  --
stop divide
  divclk <= clk and divenb after 50 ps;

  -- divider structure, not a component!
```

```
  diva    <= hi(30 downto 0) & lo(31)
after 50 ps; -- shift
  divb    <= not md when sub_add='1' else
md after 50 ps; -- subtract or add
  adder:entity WORK.add32 port map(diva,
divb, sub_add, divs, cout);

  quo     <= not divs(31) after 50 ps; --
quotient bit

  hi      <= divs  when divclk'event and
divclk='1';
  lo      <= lo(30 downto 0) & quo when
divclk'event and divclk='1';
  sub_add <= quo                   when
divclk'event and divclk='1';

  printout: postponed process(clk)  --
just to see values
  variable my_line : LINE;  -- not part of
working circuit
begin
  if clk='0' then  -- quiet time, falling
clock
    if cntr="000000" then
       write(my_line,
string'("divisor="));
       write(my_line, md);
       writeline(output, my_line);
    end if;
    write(my_line, string'("at count "));
    write(my_line, cntr);
    write(my_line, string'("  diva="));
    hwrite(my_line, diva);
    write(my_line, string'("  divb="));
    hwrite(my_line, divb);
    write(my_line, string'("  hi="));
    hwrite(my_line, hi);
    write(my_line, string'("  lo="));
    hwrite(my_line, lo);
    write(my_line, string'("  quo="));
    write(my_line, quo);
    writeline(output, my_line);
    end if;
end process printout;
end schematic;

--Square function
-- mul32c.vhdl parallel multiply 32 bit x
32 bit to get 64 bit unsigned product
-- uses add32 component and fadd component,

library IEEE;
use IEEE.std_logic_1164.all;

entity add32csa is  -- one stage of carry
save adder for multiplier
  port(
    b       : in  std_logic;
-- a multiplier bit
    a       : in  std_logic_vector(31
downto 0);  -- multiplicand
    sum_in  : in  std_logic_vector(31
downto 0);  -- sums from previous stage
    cin     : in  std_logic_vector(31
downto 0);  -- carrys from previous stage
    sum_out : out std_logic_vector(31
downto 0);  -- sums to next stage
    cout    : out std_logic_vector(31
downto 0)); -- carrys to next stage
end add32csa;

architecture circuits of add32csa is
  signal zero : std_logic_vector(31 downto
0) := X"00000000";
  signal aa : std_logic_vector(31 downto
0) := X"00000000";
  component fadd    -- duplicates entity
port
    port(a    : in  std_logic;
         b    : in  std_logic;
         cin  : in  std_logic;
         s    : out std_logic;
         cout : out std_logic);
  end component fadd;
begin  -- circuits of add32csa
  aa <= a when b='1' else zero after 1 ns;
  stage: for I in 0 to 31 generate
    sta: fadd port map(aa(I), sum_in(I),
cin(I) , sum_out(I), cout(I));
  end generate stage;
end architecture circuits; -- of add32csa

library IEEE;
use IEEE.std_logic_1164.all;

entity mul32c is  -- 32 x 32 = 64 bit
unsigned product multiplier
  port(a    : in  std_logic_vector(31
downto 0);  -- multiplicand
       b    : in  std_logic_vector(31
downto 0);  -- multiplier
       prod : out std_logic_vector(63
downto 0)); -- product
end mul32c;

architecture circuits of mul32c is
  signal zero : std_logic_vector(31 downto
```

```
0) := X"00000000";
  signal nc1  : std_logic;
  type arr32 is array(0 to 31) of
std_logic_vector(31 downto 0);
  signal s    : arr32; -- partial sums
  signal c    : arr32; -- partial carries
  signal ss   : arr32; -- shifted sums

  component add32csa is  -- duplicate
entity port
    port(b:in  std_logic;
       a:in  std_logic_vector(31 downto
0); sum_in:in std_logic_vector(31 downto
0);
    cin:in  std_logic_vector(31 downto 0);
sum_out:out std_logic_vector(31 downto 0);
  cout:out std_logic_vector(31 downto 0));
  end component add32csa;
  component add32 -- duplicate entity port
  port(a:in std_logic_vector(31 downto 0);
       b:in  std_logic_vector(31 downto
0);
       cin  : in  std_logic;
   sum:out std_logic_vector(31 downto 0);
   cout : out std_logic);
  end component add32;
begin  -- circuits of mul32c
  st0: add32csa port map(b(0), a, zero ,
zero, s(0), c(0));  -- CSA stage
  ss(0) <= '0'&s(0)(31 downto 1) after 1
ns;
  prod(0) <= s(0)(0) after 1 ns;
  stage: for I in 1 to 31 generate
    st: add32csa port map(b(I), a, ss(I-
1) , c(I-1), s(I), c(I));  -- CSA stage
    ss(I) <= '0'&s(I)(31 downto 1) after 1
ns;
    prod(I) <= s(I)(0) after 1 ns;
  end generate stage;
  add: add32 port map(ss(31), c(31), '0' ,
prod(63 downto 32), nc1);  -- adder
end architecture circuits; -- of mul32c

--Memory FSM

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY memory_signal_fsm IS
   PORT(
     a_unit    : IN     std_logic_vector
(31 DOWNTO 0) ;
      clk       : IN     std_logic  ;
      dout_unit : IN     std_logic_vector
(31 DOWNTO 0) ;
      nrd_unit  : IN     std_logic  ;
      nreset    : IN     std_logic  ;
      nwe_unit  : IN     std_logic  ;
      a_mem     : OUT    std_logic_vector
(31 downto 0) ;
      din_unit  : OUT    std_logic_vector
(31 DOWNTO 0) ;
      nack_mem  : OUT    std_logic  ;
      ncs_mem   : OUT    std_logic  ;
      nrd_mem   : OUT    std_logic  ;
      nwe_mem   : OUT    std_logic  ;
      d_mem     : INOUT  std_logic_vector
(31 DOWNTO 0)
   );

-- Declarations

END memory_signal_fsm ;

LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ARCHITECTURE memory_signal_fsm OF
memory_signal_fsm IS

   -- Architecture Declarations
   TYPE STATE_TYPE IS (
      idle,
      wr_propagate,
      rd_propagate,
      rd_ack,
      wr_hold,
      wr_ack
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF
memory_signal_fsm : ARCHITECTURE IS
"current_state" ;

   -- Declare current and next state
signals
   SIGNAL current_state : STATE_TYPE ;
   SIGNAL next_state : STATE_TYPE ;

BEGIN
 --------------------------------
clocked : PROCESS(
      clk,
      nreset
```

```
   )
   -------------------------------
   BEGIN
      IF (nreset = '0') THEN
         current_state <= idle;
         -- Reset Values
      ELSIF (clk'EVENT AND clk = '1') THEN
         current_state <= next_state;
         -- Default Assignment To
Internals
      END IF;
   END PROCESS clocked;
nextstate: PROCESS (
current_state, nrd_unit,nwe_unit)
-------------------------------------
BEGIN
      CASE current_state IS
      WHEN idle =>
         IF (nwe_unit = '0') THEN
            next_state <= wr_propagate;
         ELSIF (nrd_unit = '0') THEN
            next_state <= rd_propagate;
         ELSE
            next_state <= idle;
         END IF;
      WHEN wr_propagate =>
            next_state <= wr_hold;
      WHEN rd_propagate =>
            next_state <= rd_ack;
      WHEN rd_ack =>
            next_state <= idle;
      WHEN wr_hold =>
            next_state <= wr_ack;
      WHEN wr_ack =>
            next_state <= idle;
      WHEN OTHERS =>
         next_state <= idle;
      END CASE;

   END PROCESS nextstate;

   -------------------------------
   output : PROCESS (
      a_unit,
      current_state,
      d_mem,
      dout_unit
   )
   -------------------------------
   BEGIN
      -- Default Assignment
      a_mem <= (others => 'Z');
      nack_mem <= '1';
      ncs_mem <= '1';
      nrd_mem <= '1';
      nwe_mem <= '1';
      -- Default Assignment To Internals

      -- State Actions
      CASE current_state IS
      WHEN idle =>
         d_mem <= (others => 'Z');
         din_unit <= (others => 'Z');
         nwe_mem <= '1';
         nrd_mem <= '1';
         ncs_mem <= '1';
         a_mem <= (others => 'Z');
         nack_mem <= '1';
      WHEN wr_propagate =>
         if (current_state'event) then
           d_mem <= dout_unit;
         end if;

         din_unit <= (others => 'Z');
         nwe_mem <= '0';
         nrd_mem <= '1';
         ncs_mem <= '0';
         a_mem <= a_unit;
         nack_mem <= '1';
      WHEN rd_propagate =>
         d_mem <= (others => 'Z');
         din_unit <= (others => 'Z');
         nwe_mem <= '1';
         nrd_mem <= '0';
         ncs_mem <= '0';
         a_mem <= a_unit;
         nack_mem <= '0';
      WHEN rd_ack =>
         if (current_state'event) then
           din_unit <= d_mem;
         end if;

         d_mem <= (others => 'Z');
         nwe_mem <= '1';
         nrd_mem <= '1';
         ncs_mem <= '1';
         a_mem <= (others => 'Z');
         nack_mem <= '1';
      WHEN wr_hold =>
         if (current_state'event) then
           d_mem <= dout_unit;
         end if;

         din_unit <= (others => 'Z');
         nwe_mem <= '1';
         nrd_mem <= '1';
         ncs_mem <= '0';
         a_mem <= a_unit;
         nack_mem <= '0';
```

```
        WHEN wr_ack =>
           d_mem <= (others => 'Z');
           din_unit <= (others => 'Z');
           nwe_mem <= '1';
           nrd_mem <= '1';
           ncs_mem <= '1';
           a_mem <= (others => 'Z');
           nack_mem <= '1';
        WHEN OTHERS =>
           NULL;
        END CASE;

    END PROCESS output;

    -- Concurrent Statements

END memory_signal_fsm;


--DataPath

library ieee;
USE ieee.numeric_std.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_1164.all;


ENTITY DataPath IS
    PORT(
        clk             : IN
std_logic  ;
        done            : IN
std_logic  ;
        nlocal_rst      : IN
std_logic  ;
        nstart          : IN
std_logic  ;
        address         : OUT
std_logic_vector (31 downto 0) ;
        index           : OUT
ieee.numeric_std.unsigned (7 DOWNTO 0) ;
        nread           : OUT
std_logic  ;
        nwrite          : OUT
std_logic  ;
        match           : OUT
std_logic  ;
        loadpix         : in
std_logic  ;
        getpix          : in
std_logic  ;
        data_out: OUT    std_logic_vector
(31 downto 0) ;
        pattern_loaded  : IN
std_logic  ;
        image_loaded    : IN
std_logic  ;
        data_in         : IN
std_logic_vector (31 downto 0);
        doneinit        : out
std_logic
    );

-- Declarations

END DataPath ;

architecture datapath of DataPath is
    constant pattern_address :
integer   := 16#001000#;
    constant image_address :
integer     := 16#001100#;
    constant pattern_address2 :
integer  := 16#002000#;
    constant image_address2 : integer
         := 16#002100#;
    constant result_address :
integer    := 16#003000#;

    signal pattern: std_logic_vector(0 to
2047); -- pattern strip
    signal image: std_logic_vector(0 to
3839);  -- image strip

BEGIN

datapath_main_process: process(clk, nstart,
nlocal_rst, loadpix, getpix,
pattern_loaded, image_loaded, done)
    variable internal_index :
ieee.numeric_std.unsigned (7 downto 0);
    variable pattern_address_toconvert :
integer := pattern_address;
    variable image_address_toconvert :
integer := image_address;
    variable compare : boolean := true;  -
- true tells we may proceed compare
    variable imageset : integer := 1;  --
which set is currently used to compare
image

begin
    ------------------------------
    -- PROCESSING RESET or START
    ------------------------------
    if (nlocal_rst = '0') then
        match <= '0';
        nread <= '1';
```

```
        nwrite <= '1';
        internal_index := "00000000";
        index  <= internal_index;
        address <= (others => 'Z');
        data_out <= (others => 'Z');
        doneinit <= '0';
        compare := true;
    end if;

    if (nstart'event and nstart = '0')
then
-- reset common ports
        internal_index := "00000000";
        data_out <= (others => 'Z');
        index  <= internal_index;
        match  <= '0';
        nread  <= '1';
        nwrite <= '1';
        compare := true;

        -- set the memory image bank, we
        start with 1st image, then 2nd, -
        -and 1st .
        if (imageset = 0) then
            imageset := 1;
        else
            imageset := 0;
        end if;
        if (imageset = 0) then
            image_address_toconvert :=
image_address;
            pattern_address_toconvert :=
pattern_address;
        else
            image_address_toconvert :=
image_address2;
            pattern_address_toconvert :=
pattern_address2;
        end if;
    end if;

    ------------------------------
    -- PROLOGUE WHEN RESETING index TO 0
    ------------------------------   if
((pattern_loaded'event and pattern_loaded
= '1') or (image_loaded'event and
image_loaded = '1')) then
        internal_index := "00000000";
        index<=internal_index;
    end if;
    ------------------------------
    -- COMPARING IMAGE TO PATTERN WHEN IN
COMPARE STATE
    ------------------------------    if
(image_loaded = '1') then

    -- when image is found, terminate
        if (compare = true and
            (image(0 to 255) & image(512
to 767) & image(1024 to 1279) &
            image(1536 to 1791) &
image(2048 to 2303) & image(2560 to 2815)
&
            image(3072  to 3327) &
image(3584 to 3839)) = pattern ) then

            match <= '1';
        -- pattern found in image!
            nread <= '1';
        -- don't read anymore
            compare := false;
        -- don't compare anymore

            -- write position to memory
according to imageset
            -- function will extend the
sign to a negative value since
          -- data_out is a logic vector.
We keep the 8 LSB.
            if (imageset = 0) then
                data_out <=
"00000000000000000000000011111111" and
conv_std_logic_vector((image_address_tocon
vert-image_address)/4-120,32);
            else
                data_out <=
"00000000000000000000000011111111" and
conv_std_logic_vector((image_address_tocon
vert-image_address2)/4-120,32);
            end if;
        -- no match, we load next pixel
and disable compare
        else
            if (internal_index <
"10000111") then
                compare := false;
        -- compare on next pixel only
            else --if (internal_index >=
135) then
                compare := true;
            end if;
        end if;
    end if;

    ------------------------------
    -- PROCESSING MAIN LOOP WHEN LOADING
PIXEL
    ------------------------------
```

```
    if ((loadpix = '1' and loadpix'event)
or (getpix='1' and getpix'event) or
(image_loaded = '1' and
image_loaded'event)) then

        --------------------------
        -- PRELOADING PATTERN
        --------------------------
        if (pattern_loaded = '0') then
-- generate next pixel and address to load
next pixel
        if (loadpix = '1') then
-- ask interface to read pixel
address <=
conv_std_logic_vector(pattern_address_toco
nvert,32);
        nread <= '0';
pattern_address_toconvert :=
pattern_address_toconvert + 4;
    end if;
-- get pixel on the line
        if (getpix = '1') then
        nread <= '1';
        -- shift image
        pattern <= data_in & pattern(0 to
2015);
        internal_index := internal_index
+ 1;
        index <= internal_index;
        -- update index
            end if;
        --------------------------
        -- PRELOADING IMAGE
        --------------------------
        elsif (image_loaded = '0' and
pattern_loaded = '1') then
        if (loadpix = '1') then
        address <=
conv_std_logic_vector(image_address_toconv
ert,32);
        nread <= '0';
        image_address_toconvert :=
image_address_toconvert + 4;
        end if;

        if (getpix = '1') then
        nread <= '1';
          -- get information on the line
        image <= data_in & image(0 to
3807);  -- shift image

        internal_index := internal_index
+ 1;
        index <= internal_index ;
        end if;

        --------------------------
        -- LOADING PIXEL AND SHIFT
        --------------------------
        elsif (image_loaded = '1') then

            if (loadpix = '1') then

        -- load next pixel from image
                address <=
conv_std_logic_vector(image_address_toconv
ert,32);
                nread <= '0';

        image_address_toconvert :=
image_address_toconvert + 4;
            end if;
            if (getpix = '1') then
                nread <= '1';
                image <= data_in &
image(0 to 3807);  -- this shifts image
        internal_index := internal_index
+ 1;
        index <= internal_index;    --
        update index
        compare := true;

        end if;

end if;
    ------------------------------
    -- DONE HAS BEEN DETECTED AND WE
TERMINATE
    ------------------------------
    elsif (done = '1' and done'event) then
        address <=
conv_std_logic_vector(result_address,32);
        nwrite <= '0';
    elsif (done = '0' and done'event) then
        nwrite <= '1';
    end if;
end process datapath_main_process;
end datapath;
--Controller

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY controller IS
   PORT(
      clk           : IN     std_logic  ;
```

```
      doneinit       : IN     std_logic  ;
      index          : IN     unsigned (7
DOWNTO 0) ;
      match          : IN     std_logic  ;
      nack_mem       : IN     std_logic  ;
      nlocal_rst     : IN     std_logic  ;
      nstart         : IN     std_logic  ;
      done           : OUT    std_logic  ;
      getpix         : OUT    std_logic  ;
      image_loaded   : OUT    std_logic  ;
      loadpix        : OUT    std_logic  ;
      pattern_loaded : OUT    std_logic
   );

-- Declarations

END controller ;

LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ARCHITECTURE fsm OF controller IS

   -- Architecture Declarations
   TYPE STATE_TYPE IS (
      pp2,
      pi2,
      pi1,
      pp1,
      preload_image,
      preload_pattern,
      init,
      idle,
      compare,
      save_result,
      c1,
      c2
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm :
ARCHITECTURE IS "current_state" ;


   -- Declare current and next state
signals
   SIGNAL current_state : STATE_TYPE ;
   SIGNAL next_state : STATE_TYPE ;

BEGIN

      ------------------------------
      clocked : PROCESS(
         clk,
         nlocal_rst
      )
      ------------------------------
      BEGIN
         IF (nlocal_rst = '0') THEN
            current_state <= idle;
            -- Reset Values
         ELSIF (clk'EVENT AND clk = '1') THEN
            current_state <= next_state;
            -- Default Assignment To
Internals
         END IF;
      END PROCESS clocked;

      ------------------------------
      nextstate : PROCESS (
         current_state,
         index,
         match,
         nack_mem,
         nstart
      )
      ------------------------------
      BEGIN
         CASE current_state IS
         WHEN pp2 =>
               next_state <= preload_pattern;
         WHEN pi2 =>
               next_state <= preload_image;
         WHEN pi1 =>
            IF (nack_mem = '0') THEN
               next_state <= pi2;
            ELSE
               next_state <= pi1;
            END IF;
         WHEN pp1 =>
            IF (nack_mem = '0') THEN
               next_state <= pp2;
            ELSE
               next_state <= pp1;
            END IF;
         WHEN preload_image =>
            IF (index >= 120) THEN
               next_state <= compare;
            ELSIF (index < 120) THEN
               next_state <= pi1;
            ELSE
               next_state <= preload_image;
            END IF;
         WHEN preload_pattern =>
```

```
            IF (index >= 64) THEN
               next_state <= preload_image;
            ELSIF (index < 64) THEN
               next_state <= pp1;
            END IF;
         WHEN init =>
               next_state <= preload_pattern;
         WHEN idle =>
            IF (nstart = '0') THEN
               next_state <= init;
            ELSE
               next_state <= idle;
            END IF;
         WHEN compare =>
            IF ((index > 255-120) OR match =
'1') THEN
               next_state <= save_result;
            ELSIF (index < 256-120) THEN
               next_state <= c1;
            ELSE
               next_state <= compare;
            END IF;
         WHEN save_result =>
            IF (nack_mem = '0') THEN
               next_state <= idle;
            ELSE
               next_state <= save_result;
            END IF;
         WHEN c1 =>
            IF (nack_mem = '0') THEN
               next_state <= c2;
            ELSE
               next_state <= c1;
            END IF;
         WHEN c2 =>
               next_state <= compare;
         WHEN OTHERS =>
            next_state <= idle;
      END CASE;

   END PROCESS nextstate;

   -------------------------------
   output : PROCESS (
      current_state
   )
   -------------------------------
   BEGIN
      -- Default Assignment
      done <= '0';
      getpix <= '0';
      image_loaded <= '0';
      loadpix <= '0';
      pattern_loaded <= '0';
      -- Default Assignment To Internals
      -- State Actions
      CASE current_state IS
      WHEN pp2 =>
         getpix <= '1';
         loadpix <= '0';
      WHEN pi2 =>
         getpix <= '1';
         loadpix <= '0';
         pattern_loaded <= '1';
      WHEN pi1 =>
         loadpix <= '1';
         pattern_loaded <= '1';
      WHEN pp1 =>
         loadpix <= '1';
      WHEN preload_image =>
         image_loaded <= '0';
         pattern_loaded <= '1';
         loadpix <= '0';
         getpix <= '0';
      WHEN preload_pattern =>
         getpix <= '0';
         loadpix <= '0';
         pattern_loaded <= '0';
         image_loaded <= '0';
      WHEN idle =>
         done <= '0';
         image_loaded <= '0';
         pattern_loaded<= '0';
         loadpix <= '0';
         getpix <= '0';
      WHEN compare =>
         image_loaded <= '1';
         pattern_loaded <= '1';
         loadpix <= '0';
         getpix <= '0';
      WHEN save_result =>
         done <= '1';
         image_loaded <= '1';
         pattern_loaded <= '1';
      WHEN c1 =>
         loadpix <= '1';
         image_loaded <= '1';
         pattern_loaded <= '1';
      WHEN c2 =>
         getpix <= '1';
         loadpix <= '0';
         image_loaded <= '1';
         pattern_loaded <= '1';
      WHEN OTHERS =>
         NULL;
      END CASE;
   END PROCESS output;
   -- Concurrent Statements
```

```
END fsm;

--Block matching coprocessor
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY co_processeur IS
   PORT(
      clk         : IN     std_logic  ;
      data_in     : IN     std_logic_vector
(31 DOWNTO 0) ;
      nack_mem    : IN     std_logic  ;
      nlocal_rst  : IN     std_logic  ;
      nstart      : IN     std_logic  ;
      address     : OUT    std_logic_vector
(31 DOWNTO 0) ;
      data_out    : OUT    std_logic_vector
(31 downto 0) ;
      nread       : OUT    std_logic  ;
      nwrite      : OUT    std_logic
   );
-- Declarations
END co_processeur ;

LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

LIBRARY copro;
ARCHITECTURE struct OF co_processeur IS

   -- Architecture declarations
   -- Internal signal declarations
   SIGNAL done           : std_logic;
   SIGNAL doneinit       : std_logic;
   SIGNAL getpix         : std_logic;
   SIGNAL image_loaded   : std_logic;
   SIGNAL index          : unsigned(7
DOWNTO 0);
   SIGNAL loadpix        : std_logic;
   SIGNAL match          : std_logic;
   SIGNAL pattern_loaded : std_logic;

   -- Component Declarations
   COMPONENT controller
   PORT (
      clk            : IN     std_logic ;
      doneinit       : IN     std_logic ;
      index          : IN     unsigned (7
DOWNTO 0);
      match          : IN     std_logic ;
      nack_mem       : IN     std_logic ;
      nlocal_rst     : IN     std_logic ;
      nstart         : IN     std_logic ;
      done           : OUT    std_logic ;
      getpix         : OUT    std_logic ;
      image_loaded   : OUT    std_logic ;
      loadpix        : OUT    std_logic ;
      pattern_loaded : OUT    std_logic
   );
   END COMPONENT;
   COMPONENT datapath
   PORT (
      index          : OUT    unsigned (7
DOWNTO 0);
      done           : IN     std_logic ;
      match          : OUT    std_logic ;
      image_loaded   : IN     std_logic ;
      pattern_loaded : IN     std_logic ;
      getpix         : IN     std_logic ;
      loadpix        : IN     std_logic ;
      clk            : IN     std_logic ;
      address        : OUT
std_logic_vector (31 DOWNTO 0);
      nread          : OUT    std_logic ;
      data_out       : OUT
std_logic_vector (31 downto 0);
      nwrite         : OUT    std_logic ;
      data_in        : IN
std_logic_vector (31 DOWNTO 0);
      nstart         : IN     std_logic ;
      nlocal_rst     : IN     std_logic ;
      doneinit       : OUT    std_logic
   );
   END COMPONENT;
   -- Optional embedded configurations
   -- pragma synthesis_off
   FOR ALL : controller USE ENTITY
copro.controller;
   FOR ALL : datapath USE ENTITY
copro.datapath;
   -- pragma synthesis_on

BEGIN
   -- Instance port mappings.
   I2 : controller
      PORT MAP (
         clk            => clk,
         doneinit       => doneinit,
         index          => index,
         match          => match,
         nack_mem       => nack_mem,
         nlocal_rst     => nlocal_rst,
         nstart         => nstart,
         done           => done,
         getpix         => getpix,
         image_loaded   => image_loaded,
```

```
            loadpix         => loadpix,
            pattern_loaded => pattern_loaded
         );
      I1 : datapath
         PORT MAP (
            index           => index,
            done            => done,
            match           => match,
            image_loaded    => image_loaded,
            pattern_loaded => pattern_loaded,
            getpix          => getpix,
            loadpix         => loadpix,
            clk             => clk,
            address         => address,
            nread           => nread,
            data_out        => data_out,
            nwrite          => nwrite,
            data_in         => data_in,
            nstart          => nstart,
            nlocal_rst      => nlocal_rst,
            doneinit        => doneinit
         );

END struct;

--Memory interface
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY bm_mem_interface_int IS
   PORT(
      a_bm       : IN     std_logic_vector
(31 DOWNTO 0) ;
      clk        : IN     std_logic  ;
      dout_bm    : IN     std_logic_vector
(31 DOWNTO 0) ;
      event_sig  : IN     std_logic  ;
      nrd_bm     : IN     std_logic  ;
      nreset     : IN     std_logic  ;
      nwe_bm     : IN     std_logic  ;
      a_dpram    : OUT    std_logic_vector
(31 downto 0) ;
      din_bm     : OUT    std_logic_vector
(31 DOWNTO 0) ;
      irq        : OUT    std_logic  ;
      nack_mem   : OUT    std_logic  ;
      ncs_dpram  : OUT    std_logic  ;
      nrd_dpram  : OUT    std_logic  ;
      nstart_bm  : OUT    std_logic  ;
      nwe_dpram  : OUT    std_logic  ;
      d_dpram    : INOUT  std_logic_vector
(31 DOWNTO 0)
   );
```

```
-- Declarations
END bm_mem_interface_int ;
LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

LIBRARY copro;
ARCHITECTURE struct OF
bm_mem_interface_int IS
-- Architecture declarations
-- Non hierarchical state machine
declarations
TYPE MACHINE3_STATE_TYPE IS (
      idle,
      go,
      wait_done,
      send_irq
   );

-- Declare current and next state signals
SIGNAL machine3_current_state :
MACHINE3_STATE_TYPE ;
SIGNAL machine3_next_state :
MACHINE3_STATE_TYPE ;

   -- Internal signal declarations
   SIGNAL count       : integer;
   SIGNAL nack_mem_bm : std_logic;
   SIGNAL nrst_cnt    : std_logic;

   -- Component Declarations
   COMPONENT memory_signal_fsm
   PORT (
      a_unit     : IN     std_logic_vector
(31 DOWNTO 0);
      clk        : IN     std_logic ;
      dout_unit  : IN     std_logic_vector
(31 DOWNTO 0);
      nrd_unit   : IN     std_logic ;
      nreset     : IN     std_logic ;
      nwe_unit   : IN     std_logic ;
      a_mem      : OUT    std_logic_vector
(31 downto 0);
      din_unit   : OUT    std_logic_vector
(31 DOWNTO 0);
      nack_mem   : OUT    std_logic ;
      ncs_mem    : OUT    std_logic ;
      nrd_mem    : OUT    std_logic ;
      nwe_mem    : OUT    std_logic ;
      d_mem      : INOUT  std_logic_vector
(31 DOWNTO 0)
   );
   END COMPONENT;
```

```
   -- Optional embedded configurations
   -- pragma synthesis_off
   FOR ALL : memory_signal_fsm USE ENTITY
copro.memory_signal_fsm;
   -- pragma synthesis_on

BEGIN
   -- Architecture concurrent statements
   -- HDL Embedded Block 3 comm_int
   -- Non hierarchical state machine
   ------------------------------
   machine3_clocked : PROCESS(
      clk,
      nreset
   )
   ------------------------------
   BEGIN
      IF (nreset = '0') THEN
         machine3_current_state <= idle;
         -- Reset Values
      ELSIF (clk'EVENT AND clk = '1') THEN
         machine3_current_state <=
machine3_next_state;
         -- Default Assignment To
Internals

      END IF;

   END PROCESS machine3_clocked;

   ------------------------------
   machine3_nextstate : PROCESS (
      count,
      event_sig,
      machine3_current_state,
      nack_mem_bm,
      nwe_bm
   )
   ------------------------------
   BEGIN
      CASE machine3_current_state IS
      WHEN idle =>
         IF (event_sig = '1') THEN
            machine3_next_state <= go;
         ELSE
            machine3_next_state <= idle;
         END IF;
      WHEN go =>
         IF (nwe_bm = '0') THEN
            machine3_next_state <=
wait_done;
         ELSE
            machine3_next_state <= go;
         END IF;
      WHEN wait_done =>
         IF (nack_mem_bm = '0') THEN
            machine3_next_state <=
send_irq;
         ELSE
            machine3_next_state <=
wait_done;
         END IF;
      WHEN send_irq =>
         IF (count = 4) THEN
            machine3_next_state <= idle;
         ELSE
            machine3_next_state <=
send_irq;
         END IF;
      WHEN OTHERS =>
         machine3_next_state <= idle;
      END CASE;

   END PROCESS machine3_nextstate;

   ------------------------------
   machine3_output : PROCESS (
      machine3_current_state
   )
   ------------------------------
   BEGIN
      -- Default Assignment
      irq <= '0';
      nrst_cnt <= '0';
      nstart_bm <= '1';
      -- Default Assignment To Internals

      -- State Actions
      CASE machine3_current_state IS
      WHEN idle =>
         nstart_bm <= '1';
         irq <= '0';
         nrst_cnt <= '0';
      WHEN go =>
         nstart_bm <= '0';
         irq <= '0';
         nrst_cnt <= '0';
      WHEN wait_done =>
         nstart_bm <= '1';
         irq <= '0';
         nrst_cnt <= '0';
      WHEN send_irq =>
         nstart_bm <= '1';
         irq <= '1';
         nrst_cnt <= '1';
      WHEN OTHERS =>
         NULL;
```

```
    END CASE;

  END PROCESS machine3_output;

  -- Concurrent Statements
  -- HDL Embedded Text Block 4 chgname
  -- chgname 3
  nack_mem <= nack_mem_bm;

  -- HDL Embedded Text Block 5 counter
  -- counter 3
  process (nrst_cnt, clk)
  begin
    if (nreset = '0') then
      count <= 0;
    elsif(nrst_cnt = '0') then
      count <= 0;
    elsif (clk'event and clk = '1') then
      if (count = 200) then
        count <= 0;
      else
        count <= count + 1;
      end if;
    end if;
  end process;

  -- Instance port mappings.
  bm_msfsm : memory_signal_fsm
    PORT MAP (
      a_unit    => a_bm,
      clk       => clk,
      dout_unit => dout_bm,
      nrd_unit  => nrd_bm,
      nreset    => nreset,
      nwe_unit  => nwe_bm,
      a_mem     => a_dpram,
      din_unit  => din_bm,
      nack_mem  => nack_mem_bm,
      ncs_mem   => ncs_dpram,
      nrd_mem   => nrd_dpram,
      nwe_mem   => nwe_dpram,
      d_mem     => d_dpram
    );

END struct;

--Interrupt Main function
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY main_int IS
-- Declarations
END main_int ;

LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

LIBRARY basicarm;
LIBRARY copro;

ARCHITECTURE struct OF main_int IS

  -- Architecture declarations

  -- Internal signal declarations
  SIGNAL Intr_data_out :
std_logic_vector(31 DOWNTO 0);
  SIGNAL a             :
std_logic_vector(31 DOWNTO 0);
  SIGNAL a_dpram       :
std_logic_vector(31 downto 0);
  SIGNAL address       :
std_logic_vector(31 DOWNTO 0);
  SIGNAL be            :
std_logic_vector(3 DOWNTO 0);
  SIGNAL busyl         : std_logic;
  SIGNAL busyr         : std_logic;
  SIGNAL clk           : std_logic;
  SIGNAL copro_0_add   :
std_logic_vector(17 DOWNTO 0);
  SIGNAL copro_0_d     :
std_logic_vector(31 DOWNTO 0);
  SIGNAL cs            : std_logic;
  SIGNAL data_out      :
std_logic_vector(31 DOWNTO 0);
  SIGNAL din           :
std_logic_vector(31 DOWNTO 0);
  SIGNAL din_bm        :
std_logic_vector(31 DOWNTO 0);
  SIGNAL dout          :
std_logic_vector(31 downto 0);
  SIGNAL dout_bm       :
std_logic_vector(31 DOWNTO 0);
  SIGNAL dpram_0_cs    : std_logic;
  SIGNAL dpram_0_d     :
std_logic_vector(31 DOWNTO 0);
  SIGNAL ev_b          : std_logic;
  SIGNAL ev_c          : std_logic;
  SIGNAL ev_d          : std_logic;
  SIGNAL event_cs      : std_logic;
  SIGNAL event_sig     : std_logic;
  SIGNAL irq           : std_logic;
  SIGNAL mas           :
std_logic_vector(1 DOWNTO 0);
  SIGNAL nWAIT         : std_logic;
```

```
    SIGNAL nack_mem_bm    : std_logic;
    SIGNAL ncopro_0_be    :
std_logic_vector(3 DOWNTO 0);
    SIGNAL ncopro_0_cs    : std_logic;
    SIGNAL ncopro_0_oe    : std_logic;
    SIGNAL ncopro_0_we    : std_logic;
    SIGNAL nirq           : std_logic;
    SIGNAL nmreq          : std_logic;
    SIGNAL nread          : std_logic;
    SIGNAL nrst           : std_logic;
    SIGNAL nrw            : std_logic;
    SIGNAL nscreen_cs     : std_logic;
    SIGNAL nsram_cs       : std_logic;
    SIGNAL nstart_bm      : std_logic;
    SIGNAL nwe_bm         : std_logic;
    SIGNAL oe             : std_logic;
    SIGNAL screen_d       :
std_logic_vector(31 DOWNTO 0);
    SIGNAL seml           : std_logic;
    SIGNAL semr           : std_logic;
    SIGNAL sram_d         :
std_logic_vector(31 DOWNTO 0);
    SIGNAL we             : std_logic;

    -- Component Declarations
    COMPONENT ARM_CORE
    PORT (
       din    : IN     std_logic_vector (31
downto 0);
       mclk   : IN     std_logic ;
       nWAIT  : IN     std_logic ;
       nirq   : IN     std_logic ;
       nreset : IN     std_logic ;
       a      : OUT    std_logic_vector (31
downto 0);
       dout   : OUT    std_logic_vector (31
downto 0);
       mas    : OUT    std_logic_vector (1
downto 0);
       nmreq  : OUT    std_logic ;
       nrw    : OUT    std_logic
    );
    END COMPONENT;
    COMPONENT CONTROLLER_MEM
    PORT (
       Intr_data_in  : IN
std_logic_vector (31 DOWNTO 0);
       a             : IN
std_logic_vector (31 DOWNTO 0);
       clk           : IN     std_logic ;
       dout          : IN
std_logic_vector (31 DOWNTO 0);
       mas           : IN
std_logic_vector (1 downto 0);
       nmreq         : IN     std_logic ;
       nrw           : IN     std_logic ;
       reset         : IN     std_logic ;
       Intr_data_out : OUT
std_logic_vector (31 DOWNTO 0);
       be            : OUT
std_logic_vector (3 downto 0);
       din           : OUT
std_logic_vector (31 DOWNTO 0);
       nWAIT         : OUT    std_logic ;
       ndpram_0_cs   : OUT    std_logic ;
       nevent_cs     : OUT    std_logic ;
       nintr_cs      : OUT    std_logic ;
       nscreen_cs    : OUT    std_logic ;
       nsem_cs       : OUT    std_logic ;
       nsram_cs      : OUT    std_logic ;
       oe            : OUT    std_logic ;
       screen_d      : OUT
std_logic_vector (31 DOWNTO 0);
       we            : OUT    std_logic ;
       dpram_0       : INOUT
std_logic_vector (31 DOWNTO 0);
       sram_d        : INOUT
std_logic_vector (31 DOWNTO 0)
    );
    END COMPONENT;
    COMPONENT DPRAM_MEM
    PORT (
       copro_add : IN     std_logic_vector
(17 DOWNTO 0);
       dpram_add : IN     std_logic_vector
(31 DOWNTO 0);
       ncopro_be : IN     std_logic_vector
(3 DOWNTO 0);
       ncopro_cs : IN     std_logic ;
       ncopro_oe : IN     std_logic ;
       ncopro_we : IN     std_logic ;
       ndpram_be : IN     std_logic_vector
(3 DOWNTO 0);
       ndpram_cs : IN     std_logic ;
       ndpram_oe : IN     std_logic ;
       ndpram_we : IN     std_logic ;
       seml      : IN     std_logic ;
       semr      : IN     std_logic ;
       intl      : OUT    std_logic ;
       intr      : OUT    std_logic ;
       busyl     : INOUT  std_logic ;
       busyr     : INOUT  std_logic ;
       copro_d   : INOUT  std_logic_vector
(31 downto 0);
       d         : INOUT  std_logic_vector
(31 downto 0)
    );
    END COMPONENT;
```

```
   COMPONENT EVENT_DEC
   PORT (
      event_add : IN     std_logic_vector
(31 DOWNTO 0);
      event_cs  : IN     std_logic ;
      event_we  : IN     std_logic ;
      event_a   : OUT    std_logic ;
      event_b   : OUT    std_logic ;
      event_c   : OUT    std_logic ;
      event_d   : OUT    std_logic
   );
   END COMPONENT;
   COMPONENT INTERRUPT
   PORT (
      a         : IN     std_logic_vector
(31 DOWNTO 0);
      clk       : IN     std_logic ;
      cs        : IN     std_logic ;
      data_in   : IN     std_logic_vector
(31 DOWNTO 0);
      ev_a      : IN     std_logic ;
      ev_b      : IN     std_logic ;
      ev_c      : IN     std_logic ;
      ev_d      : IN     std_logic ;
      nrst      : IN     std_logic ;
      oe        : IN     std_logic ;
      we        : IN     std_logic ;
      data_out  : OUT    std_logic_vector
(31 DOWNTO 0);
      nirq      : OUT    std_logic
   );
   END COMPONENT;
   COMPONENT SCREEN_MEM
   PORT (
      data_screen : IN
std_logic_vector (31 DOWNTO 0);
      nscreen_cs  : IN     std_logic ;
      nscreen_we  : IN     std_logic
   );
   END COMPONENT;
   COMPONENT SRAM_MEM
   PORT (
      nsram_be : IN     std_logic_vector
(3 DOWNTO 0);
      nsram_cs : IN     std_logic ;
      nsram_oe : IN     std_logic ;
      nsram_we : IN     std_logic ;
      sram_add : IN     std_logic_vector
(31 DOWNTO 0);
      d        : INOUT  std_logic_vector
(31 downto 0)
   );
   END COMPONENT;
   COMPONENT bm_mem_interface_int
   PORT (
      a_bm      : IN     std_logic_vector
(31 DOWNTO 0);
      clk       : IN     std_logic ;
      dout_bm   : IN     std_logic_vector
(31 DOWNTO 0);
      event_sig : IN     std_logic ;
      nrd_bm    : IN     std_logic ;
      nreset    : IN     std_logic ;
      nwe_bm    : IN     std_logic ;
      a_dpram   : OUT    std_logic_vector
(31 downto 0);
      din_bm    : OUT    std_logic_vector
(31 DOWNTO 0);
      irq       : OUT    std_logic ;
      nack_mem  : OUT    std_logic ;
      ncs_dpram : OUT    std_logic ;
      nrd_dpram : OUT    std_logic ;
      nstart_bm : OUT    std_logic ;
      nwe_dpram : OUT    std_logic ;
      d_dpram   : INOUT  std_logic_vector
(31 DOWNTO 0)
   );
   END COMPONENT;
   COMPONENT clock_generator
   PORT (
      clk  : OUT    std_logic ;
      nrst : OUT    std_logic
   );
   END COMPONENT;
   COMPONENT co_processeur
   PORT (
      clk       : IN     std_logic ;
      data_in   : IN     std_logic_vector
(31 DOWNTO 0);
      nack_mem  : IN     std_logic ;
      nlocal_rst : IN    std_logic ;
      nstart    : IN     std_logic ;
      address   : OUT    std_logic_vector
(31 DOWNTO 0);
      data_out  : OUT    std_logic_vector
(31 downto 0);
      nread     : OUT    std_logic ;
      nwrite    : OUT    std_logic
   );
   END COMPONENT;

   -- Optional embedded configurations
   -- pragma synthesis_off
   FOR ALL : ARM_CORE USE ENTITY
basicarm.ARM_CORE;
   FOR ALL : CONTROLLER_MEM USE ENTITY
basicarm.CONTROLLER_MEM;
   FOR ALL : DPRAM_MEM USE ENTITY
```

```
basicarm.DPRAM_MEM;
   FOR ALL : EVENT_DEC USE ENTITY
basicarm.EVENT_DEC;
   FOR ALL : INTERRUPT USE ENTITY
basicarm.INTERRUPT;
   FOR ALL : SCREEN_MEM USE ENTITY
basicarm.SCREEN_MEM;
   FOR ALL : SRAM_MEM USE ENTITY
basicarm.SRAM_MEM;
   FOR ALL : bm_mem_interface_int USE
ENTITY copro.bm_mem_interface_int;
   FOR ALL : clock_generator USE ENTITY
basicarm.clock_generator;
   FOR ALL : co_processeur USE ENTITY
copro.co_processeur;
   -- pragma synthesis_on

BEGIN
   -- Architecture concurrent statements
   -- HDL Embedded Text Block 2 addrcnv1
   -- eb1 1
   copro_0_add <= a_dpram(17 downto 0);
   ncopro_0_be <= "0000";

   -- HDL Embedded Text Block 4 eb3
   -- eb2 3
   semr <= '1';
   seml <= '1';

   -- Instance port mappings.
   I2 : ARM_CORE
      PORT MAP (
         din    => din,
         mclk   => clk,
         nWAIT  => nWAIT,
         nirq   => nirq,
         nreset => nrst,
         a      => a,
         dout   => dout,
         mas    => mas,
         nmreq  => nmreq,
         nrw    => nrw
      );
   I5 : CONTROLLER_MEM
      PORT MAP (
         Intr_data_in  => data_out,
         a             => a,
         clk           => clk,
         dout          => dout,
         mas           => mas,
         nmreq         => nmreq,
         nrw           => nrw,
         reset         => nrst,
         Intr_data_out => Intr_data_out,
         be            => be,
         din           => din,
         nWAIT         => nWAIT,
         ndpram_0_cs   => dpram_0_cs,
         nevent_cs     => event_cs,
         nintr_cs      => cs,
         nscreen_cs    => nscreen_cs,
         nsem_cs       => OPEN,
         nsram_cs      => nsram_cs,
         oe            => oe,
         screen_d      => screen_d,
         we            => we,
         dpram_0       => dpram_0_d,
         sram_d        => sram_d
      );
   DPRAM_1 : DPRAM_MEM
      PORT MAP (
         copro_add => copro_0_add,
         dpram_add => a,
         ncopro_be => ncopro_0_be,
         ncopro_cs => ncopro_0_cs,
         ncopro_oe => ncopro_0_oe,
         ncopro_we => ncopro_0_we,
         ndpram_be => be,
         ndpram_cs => dpram_0_cs,
         ndpram_oe => oe,
         ndpram_we => we,
         seml      => seml,
         semr      => semr,
         intl      => OPEN,
         intr      => OPEN,
         busyl     => busyl,
         busyr     => busyr,
         copro_d   => copro_0_d,
         d         => dpram_0_d
      );
   I7 : EVENT_DEC
      PORT MAP (
         event_add => a,
         event_cs  => event_cs,
         event_we  => we,
         event_a   => event_sig,
         event_b   => OPEN,
         event_c   => OPEN,
         event_d   => OPEN
      );
   I6 : INTERRUPT
      PORT MAP (
         a       => a,
         clk     => clk,
         cs      => cs,
         data_in => Intr_data_out,
         ev_a    => irq,
         ev_b    => ev_b,
```

```
      ev_c     => ev_c,
      ev_d     => ev_d,
      nrst     => nrst,
      oe       => oe,
      we       => we,
      data_out => data_out,
      nirq     => nirq
   );
I4 : SCREEN_MEM
   PORT MAP (
      data_screen => screen_d,
      nscreen_cs  => nscreen_cs,
      nscreen_we  => we
   );
I3 : SRAM_MEM
   PORT MAP (
      nsram_be => be,
      nsram_cs => nsram_cs,
      nsram_oe => oe,
      nsram_we => we,
      sram_add => a,
      d        => sram_d
   );
I0 : bm_mem_interface_int
   PORT MAP (
      a_bm      => address,
      clk       => clk,
      dout_bm   => dout_bm,
      event_sig => event_sig,
      nrd_bm    => nread,
      nreset    => nrst,
      nwe_bm    => nwe_bm,
      a_dpram   => a_dpram,
      din_bm    => din_bm,
      irq       => irq,
      nack_mem  => nack_mem_bm,
      ncs_dpram => ncopro_0_cs,
      nrd_dpram => ncopro_0_oe,
      nstart_bm => nstart_bm,
      nwe_dpram => ncopro_0_we,
      d_dpram   => copro_0_d
   );
I1 : clock_generator
   PORT MAP (
      clk  => clk,
      nrst => nrst
   );
I9 : co_processeur
   PORT MAP (
      clk        => clk,
      data_in    => din_bm,
      nack_mem   => nack_mem_bm,
      nlocal_rst => nrst,
      nstart     => nstart_bm,
      address    => address,
      data_out   => dout_bm,
      nread      => nread,
      nwrite     => nwe_bm
   );

END struct;
```

Appendix B: SOBEL Edge Detection Implementation Code

Software Components

```
#ifndef INCLUDE_IMAGE_H
#define INCLUDE_IMAGE_H

#include <stdio.h>

#define max(a, b) (((a)>(b))?(a):(b))
#define min(a, b) (((a)<(b))?(a):(b))

const double PI = 3.1415926535;

typedef int ImageDatum;
#define dataToDouble(a) (double(a)/255.0)
#define doubleToData(b)
(ImageDatum(b*255.0))

union Pixel {
        struct RGB_Pixel {
                ImageDatum r;
                ImageDatum g;
                ImageDatum b;
        } rgb;

        struct HSI_Pixel {
                ImageDatum h;
                ImageDatum s;
                ImageDatum i;
        } hsi;
};
enum ImageMode {
        Mode_RGB,
        Mode_HSI
};

enum ImageChannel {
        ch_Red,
        ch_Blue,
        ch_Green,
        ch_Hue,
        ch_Saturation,
        ch_Intensity
};

class ImageDisplayer;

class Image {
 public:
        Image(int, int);
        Image(char*);
        Image(Image&);
        ~Image();
        bool load(char*);
        bool save(char*);
        void display(char*);
        void annotate(char*, char*);
        ImageDatum getValue(ImageChannel,
int, int);
        void setValue(ImageChannel, int,
int, ImageDatum);
        bool isBlack(int, int);
        bool isWhite(int, int);
        ImageDatum getGrey(int, int);
        void settoBlack(int, int);
        void settoWhite(int, int);
        void setGrey(int, int,
ImageDatum);
        int height();
        int width();
        void RGBtoHSI();
        void swap(Image* that);

        // for internal use - not for the
faint of heart
        void write_to_fp(FILE*);
        void read_from_fp(FILE*);

        static bool showGUI;
        static long memReq;

 protected:
        void init(int, int);
        int h, w;
        ImageMode mode;
        Pixel* p;
        Pixel* getPixel(int, int);

        void openGUI();
        void loadRGB(char*,int,int);
        void saveRGB(char*);
        static FILE* guiRead;
        static FILE* guiWrite;
};

#endif

/*********************************
***  image.cpp
***    simple image manipulation
functions:
***       loading, saving, reading and
chaging intensity values
***
\*********************************/
```

```
#include "image.h"

#include <assert.h>
#include <iostream.h>
#include <fstream.h>
#include <unistd.h>     // fork()
#include <math.h>       // atan() sqrt()
#include <stdio.h>      // sscanf()
#include <stdlib.h>     // system()
#include <sys/types.h>  // socketpair()
#include <sys/socket.h> // socketpair()
#include <unistd.h>     // fcntl()
#include <fcntl.h>      // fcntl()

// QT Stuff
#include <qapp.h>
#include <qwidget.h>
#include <qpainter.h>
#include <qsocketnotifier.h>
#include <qmessagebox.h>
#include <qlistbox.h>
#include <qpixmap.h>
#include <qlabel.h>

const int lbWidth=200;
const int pad=5;
const int maxPanes=100;

static long Image::memReq = 0;
// ImageDisplayer and ImageGallery should
only be used from within the Image
// class.  Client coders should probably
just call Image::display.
class ImageDisplayer : public QFrame {
        Q_OBJECT;
 public:
        ImageDisplayer(QWidget* parent,
Image *image) : QFrame(parent)

        { construct_ImageDisplayer(image)
; };
        ~ImageDisplayer();

 protected:
        void
construct_ImageDisplayer(Image *image);
        virtual void
resizeEvent(QResizeEvent* );
        QPixmap* pm;
        QLabel* lbl;
};

class ImageGallery : public QWidget {
        Q_OBJECT;
 public:
        ImageGallery(int);
        void addPane(QFrame*, char*);

 protected:
        int fd;
        QListBox* lb;

        int maxHeight;
        int maxWidth;

        QWidget* panes[maxPanes];
        virtual void
resizeEvent(QResizeEvent*);

 protected slots:

        void dataReceived(int);
        void chooseImage(int);
};

//******************************/

Image::Image(int h_in, int w_in) {
        init(h_in, w_in);
}

Image::Image(char*fileName) {
        // constructs an image and loads
the specified file
        load(fileName);
}

Image::Image(Image& i) {
        init(i.width(), i.height());
        mode = i.mode;
        memcpy(p, i.p,
width()*height()*sizeof(Pixel));
}

void Image::init(int w_in, int h_in) {
        h = h_in;
        w = w_in;
        p = new Pixel[h*w];
        mode = Mode_RGB;
        memReq += h*w;
}

bool Image::showGUI = true;
Image::~Image() {
        if(p) delete[] p;
        p = 0;
        mode = Mode_RGB;
        memReq += h*w;
```

```
}
bool Image::load(char* fileName) {
        int x, y;
        int w_in, h_in;
        unsigned char c;
        Pixel *cp;

        if(!strstr(fileName, ".rgb")) {
                // not an rgb file --
convert it
                char *p;
                FILE* fp;
                char cmd[128];

                // get the dimensions
                p = tmpnam(0);
                sprintf(cmd, "imdim %s
> %s", fileName, p);
                if(system(cmd)) {
                        // a non-zero
return value from imdim
                        // bad news.
                        // TODO: deal
with this.
                }

                fp = fopen(p, "r");
                if(fscanf(fp, "%d %d",
&w_in, &h_in) != 2) {
                        // couldn't scan
two numbers from the file
                        // this is bad
news.
                        // TODO: deal
with this
                }
                fclose(fp);
                unlink(p);

                // convert the file
                p = tmpnam(0);
                sprintf(cmd, "convert %s
rgb:%s", fileName, p);

                if(system(cmd)) {
                        // TODO: Deal
with error
                }

                // load the file
                init(w_in, h_in);
                loadRGB(p, w, h);
                unlink(p);
        } else {
                // already rgb -- no
conversion needed.
                char *p;
                p = fileName;

                // extract the image's
dimensions from the filename
                while(*p && *p != '-')
p++; p++;
                sscanf(p, "%dx%d", &w_in,
&h_in);

                loadRGB(fileName, w_in,
h_in);

        }
        return true;
}
bool Image::save(char *fileName) {
        char* p;

        if(!strstr(fileName, ".rgb")) {
                // not an rgb file

                // create temporary file
in rgb format
                p = tmpnam(0);
                saveRGB(p);

                char cmd[100];
                sprintf(cmd, "convert -
size %dx%d rgb:%s %s", w, h, p, fileName);
                if(system(cmd)) {
                        // TODO: Deal
with error
                }

                unlink(p);
        } else {
                // rgb file
                saveRGB(fileName);
        }
        return true;
}

void Image::loadRGB(char* fileName, int
w_in, int h_in) {
        init(w_in, h_in);
        ifstream in(fileName);

        int x, y;
        Pixel *cp;
        unsigned char c;
```

```
        for(y=0;y<h;y++) {
                for(x=0;x<w;x++) {
                        if(!in) return;
                        cp = getPixel(x,
y);
in.get(c); cp->rgb.r = min(255, c);
in.get(c); cp->rgb.g = min(255, c);
in.get(c); cp->rgb.b = min(255, c);
                }
        }
}
void Image::saveRGB(char* fileName) {
        int x, y;
        Pixel *cp;
        ofstream out(fileName);

        for(y=0;y<h;y++) {
                for(x=0;x<w;x++) {
                        cp = getPixel(x,
y);
                        out << (unsigned
char) cp->rgb.r;
                        out << (unsigned
char) cp->rgb.g;
                        out << (unsigned
char) cp->rgb.b;
                }
        }
}
Pixel* Image::getPixel(int x, int y) {
  if((x<w)&&(y<h)) //Sorry J, I just
couldn't let this go. -A. :)
    return(p + x + y*w);
  else
    return(NULL);
}

ImageDatum Image::getValue(ImageChannel ch,
int x, int y) {
        Pixel *cp = getPixel(x, y);

        switch(ch) {
                case ch_Red  : return cp-
>rgb.r;
                case ch_Green: return cp-
>rgb.g;
                case ch_Blue : return cp-
>rgb.b;

                case ch_Hue        :
return cp->hsi.h;
                case ch_Saturation :
return cp->hsi.s;
                case ch_Intensity  :
return cp->hsi.i;
        }

        //should never get here
        return 0;
}

void Image::setValue(ImageChannel ch, int
x, int y, ImageDatum i) {
        Pixel *cp = getPixel(x, y);

        switch(ch) {
                case ch_Red  : cp->rgb.r
= i; break;
                case ch_Green: cp->rgb.g
= i; break;
                case ch_Blue : cp->rgb.b
= i; break;

                case ch_Hue        : cp-
>hsi.h = i; break;
                case ch_Saturation : cp-
>hsi.s = i; break;
                case ch_Intensity  : cp-
>hsi.i = i; break;
        }
}

bool Image::isBlack(int x, int y) {
  Pixel *cp = getPixel(x,y);
  return((cp->rgb.r==0)
      &&(cp->rgb.g==0)
      &&(cp->rgb.b==0)); //||(cp-
>hsi.Intensity==0));
}

bool Image::isWhite(int x, int y) {
  Pixel *cp = getPixel(x,y);
  return((cp->rgb.r==255)
      &&(cp->rgb.g==255)
      &&(cp->rgb.b==255));
}

ImageDatum Image::getGrey(int x, int y) {
  int sum = 0;
  sum += getValue(ch_Red,x,y);
  sum += getValue(ch_Blue,x,y);
  sum += getValue(ch_Green,x,y);
  return(sum /= 3);
}

void Image::settoBlack(int x, int y) {
  Pixel *cp = getPixel(x,y);
```

```
  cp->rgb.r=0;
  cp->rgb.g=0;
  cp->rgb.b=0;
}

void Image::settoWhite(int x, int y) {
  Pixel *cp = getPixel(x,y);
  cp->rgb.r=255;
  cp->rgb.g=255;
  cp->rgb.b=255;
}

void Image::setGrey(int x, int y,
ImageDatum grey) {
  Pixel *cp = getPixel(x,y);
  cp->rgb.r = grey;
  cp->rgb.g = grey;
  cp->rgb.b = grey;
}

int Image::height() {
        return h;
}

int Image::width() {
        return w;
}

void Image::swap(Image* that) {
        assert(this->height() == that-
>height());
        assert(this->width() == that-
>width());

        Pixel *t;
        t = this->p;
        this->p = that->p;
        that->p = t;
}

void Image::RGBtoHSI() {
        Pixel *cp;
        mode = Mode_HSI;

        double rt3 = sqrt(3);
        int x, y;
        for(x=0;x<w;x++)
                for(y=0;y<h;y++) {
                cp = getPixel(x, y);
                double r =
dataToDouble(cp->rgb.r);
                double g =
dataToDouble(cp->rgb.g);
                double b =
dataToDouble(cp->rgb.b);
                double e = max(g, b);
                double f = max(g, b);

                cp->hsi.h =
doubleToData(PI/2 - atan((2*r-e-f)/rt3*(e-
f)) / (2*PI));
                cp->hsi.s =
doubleToData(1 - min(r, min(g, b)));
                cp->hsi.i =
doubleToData((r+g+b)/3.0);
                //cout << cp->hsi.h << "
" << cp->hsi.s << " " << cp->hsi.i <<
endl;
                //cout << "(" << w << " "
<< h << ")" << endl;
                }
}
void
ImageDisplayer::construct_ImageDisplayer(I
mage *i) {
        // start with the right size
        setGeometry(x(), y(), i->width(),
i->height());
        // draw the image on and internal
canvas
        pm = new QPixmap(i->width(), i-
>height());
        QPainter* p = new QPainter;
        p->begin(pm);
        int x, y;

        for(y=0;y<i->height();y++)
                for(x=0;x<i->width();x++)
{
                        p->setPen(QColor(

        max(0, min(255, i-
>getValue(ch_Red,   x, y))),

        max(0, min(255, i-
>getValue(ch_Green, x, y))),

        max(0, min(255, i-
>getValue(ch_Blue,  x, y)))
                                        ));
                p->drawPoint(x, y);
                }
        p->end();
// create a label to show the pixmap
        lbl = new QLabel(this, "");
        lbl->setPixmap(*pm);
}
```

```
ImageDisplayer::~ImageDisplayer() {
        if(pm) delete[] pm;
}

void
ImageDisplayer::resizeEvent(QResizeEvent*
) {
        lbl->setGeometry(0,0,width(),
height());
}

FILE* Image::guiWrite = 0;
FILE* Image::guiRead = 0;

void Image::openGUI() {
        // Create child process, if
needed.
        if(!guiWrite) {
                int fd[2];

                socketpair(AF_UNIX,
SOCK_STREAM, 0, fd);
                if(fork()) {
// in parent process
// open a file stream for writing to the
gui window.
        guiWrite = fdopen(fd[0], "w");
        guiRead = fdopen(fd[0], "r");
                } else {
// in child process
// create and display the ImageGallery
window.

        int n = 0; char **c = 0;
        QApplication* app = new
QApplication(n, c);
        ImageGallery *ig = new
ImageGallery(fd[1]);

        ig->show();

// when the window is closed, end this
process
        app->setMainWidget(ig);
        exit(app->exec());
                }
        }
}

void Image::display(char* caption) {
        if(!showGUI) return;
        // make sure a window is opened
        openGUI();
        // send the image
        fprintf(guiWrite, "i%d %d %s|", w,
h, caption);
        write_to_fp(guiWrite);
        fflush(guiWrite);

        // wait for a confirmation
        char dev_null;
        fscanf(guiRead, "%c", &dev_null);
}

void Image::annotate(char* caption, char*
rtf) {
        if(!showGUI) return;

        // make sure a window is opened
        openGUI();

        // send the text
        fprintf(guiWrite, "t %d %d\n",
strlen(caption), strlen(rtf));
        fwrite(caption, sizeof(char),
strlen(caption)+1, guiWrite);
        fwrite(rtf, sizeof(char),
strlen(rtf)+1, guiWrite);
        fflush(guiWrite);

        // wait for a confirmation
        char dev_null;
        fscanf(guiRead, "%c", &dev_null);
}

void Image::write_to_fp(FILE* fp) {
        fwrite(p, sizeof(Pixel), h*w,
fp);
}

void Image::read_from_fp(FILE* fp) {
        fread(p, sizeof(Pixel), h*w, fp);
}

ImageGallery::ImageGallery(int fd_in) {
        // remember the handle to the
socket to from which to read
        fd = fd_in;

        //This was in the example code.
I'm not quite sure what it's for. -O'K
        //  fcntl(fd, O_NONBLOCK);

        // Arrange to be notified when
new data arrives.
        QSocketNotifier *sn = new
QSocketNotifier(fd, QSocketNotifier::Read,
```

```
this);
        QObject::connect(sn,
SIGNAL(activated(int)), this,
SLOT(dataReceived(int)));
        // Create a list box show the
available images.
        lb = new QListBox(this);
        lb->show();
        connect(lb,
SIGNAL(highlighted(int)), this,
SLOT(chooseImage(int)));

        // Start with no images
        maxHeight = 0;
        maxWidth  = 0;

}

void ImageGallery::addPane(QFrame* w,
char* caption) {
        // add it the internal and
visible lists
        lb->insertItem(caption);
        panes[lb->count()-1] = w;

        // place and stylize the new
widget
        w->move(lbWidth+pad, pad);
        //w->setFrameStyle(QFrame::Box |
QFrame::Sunken);

        // manage the size of our window
        maxHeight = max(maxHeight, w-
>height());
        setMinimumHeight(maxHeight+pad+pa
d);

        maxWidth = max(maxWidth, w-
>width());
        setMinimumWidth(maxWidth+lbWidth+
pad+pad);


        // TODO: Look inot why this
doesn't work.
        //   special case: initial size
        /*
        if(lb->count() == 1)
                setGeometry(x(), y(),
maxWidth+lbWidth+pad+pad,
maxHeight+pad+pad);
        */
}
```

```
void ImageGallery::dataReceived(int
socket) {
        char caption[80];
        int h, w;
        Image* img;
        QFrame* id;
        setCursor(waitCursor);
        FILE* fpRead  = fdopen(socket,
"r");
        FILE* fpWrite = fdopen(socket,
"w");

// determine what kind of pane this is
        char t = fgetc(fpRead);

        if(t == 'i') {
// get the image's name, height and width
fscanf(fpRead, "%d %d %[^|]s|", &w, &h,
caption);
fgetc(fpRead);  // not sure why we need to
do this - O'K

img = new Image(w, h);

// read the image data directly into
memory
img->read_from_fp(fpRead);

// create a widget to display the image
                id = new
ImageDisplayer(this, img);
        } else if (t == 't') {
                char rtf[4096];
                int capLen, rtfLen;

                fscanf(fpRead, "%d %d",
&capLen, &rtfLen);
                fgetc(fpRead);

                fread(caption,
sizeof(char), capLen+1, fpRead);
                fread(rtf, sizeof(char),
rtfLen+1, fpRead);

                QLabel* l = new
QLabel(rtf, this);
                l-
>setAlignment(AlignTop);
                l-
>setMinimumWidth(lbWidth+200);
                l->setMinimumHeight(300);
                id = l;

        } else
```

```
                return;
        // add it to the list of images
        addPane(id, caption);

        fprintf(fpWrite, "%c", t);
        fflush(fpWrite);
        setCursor(arrowCursor);
}
void
ImageGallery::resizeEvent(QResizeEvent*) {
        lb->setFixedWidth(lbWidth);
        lb->move(0, 0);
        lb->setGeometry(0, pad, lbWidth,
height()-2*pad);
}
void ImageGallery::chooseImage(int index)
{
        unsigned i;
        unsigned dex = unsigned(index);
        for(i=0;i<lb->count();i++)
                (i==dex) ?
                (panes[i]->show())
                :(panes[i]->hide());
}

#include "image.h"
#include <iostream.h>

const int kx[][3] =
        { { -1, -2, -1},
          {  0, 0,  0},
          {  1, 2,  1}};

const int ky[][3] =
        { { -1, 0,  1},
          { -2, 0,  2},
          { -1, 0,  1}};

int convolve(Image *img, int x, int y, int
k[][3]) {
        int xx, yy, r=0;
        for(xx=-1;xx<=1;xx++)
                for(yy=-1;yy<=1;yy++)
                        r+= img-
>getGrey(x+xx,y+yy) * k[xx+1][yy+1];

        return r;
}
int main(int argc, char** argv) {
        int thresh;

        if(argc != 3) {
                cerr << "performs Sobel
edge detection" << endl;
                cerr << "   usage: " <<
argv[0] << " <threshold> <filename>" <<
endl;
                exit(1);
        }
        thresh = atoi(argv[1]);
        Image* src  = new Image(argv[2]);
        Image* dest = new Image(src-
>width(), src->height());
        src->display("original");
        int x, y, sx, sy;
        for(x=0;x<src->width();x++)
        for(y=0;y<src->height();y++)
        src->setGrey(x, y,
        (src->getValue(ch_Red, x, y)
        + src->getValue(ch_Green, x, y)
        + src->getValue(ch_Blue, x,
y))/3);

        for(x=1;x<src->width()-1;x++)
        for(y=1;y<src->height()-1;y++) {
        sx = convolve(src, x, y, kx);

        sy = convolve(src, x, y, ky);

        if(sqrt(sx*sx+sy*sy)>thresh)
                dest->settoBlack(x, y);
                else
                dest->settoWhite(x, y);
        }
        dest->display("after Sobel edge
detection");
        dest->save("done.rgb");
}
#include "image.cpp.moc"
```

VHDL Components

```
--Gx, Gy calculater (matrix multiplier)
--
*****************************************/
-- 3x3matrix.v - 3X3 Matrix Multiply
-- Implementation using basic equations
--
--                   Author: Matthew
--                   Date:   Aug 2, 2003
-- Other modules instanced in this design:
--
--          MULT18X18
--

--BRIEF DESCRIPTION
--This code describes using a technique
called time multiplexing to
--leverage a fast hardware multiply in a
relatively slow operation,
--thereby increasing the efficiency of the
implementation.
--The operation being shown is a 3X3
matrix of constants times a 3
--component vector. The equations look
like:
--KA1 * A + KA2 * A + KA3 * A = X
--KB1 * A + KB2 * A + KB3 * A = X
--KC1 * A + KC2 * A + KC3 * A = X

--The hardware to accomplish this task
consists of a multiplier fed by 3
--input registers and an accumulator to
compute the three terms in each
--line above.
--DETAILED DESCRIPTION:
--The multiplier output is fed into the
adder A input. It takes 3 clk
--cycles for the first valid mutiplier
output reach the adder input A. The
--B input of the adder can be a zero or
the adder's accumulating register.
--By selecting a zero on the B input the
adder just passes the input A
--through to the accumulating register. By
selecting the accumulating
--register, the contents of the previous
add can be added to the output of
--the multiplier.
--The repeating flow for the accumlating
register will be for the 1st clk,
--the mux output is '0', so we always pass
the first argument through to
--the accumulator register. For the 2nd
and 3rd clks, the accumulator
--register is fed back and added to the
output of the multiply. This is
--made possible by using the cntr3 outputs
as the select lines.
--The following text describes the
condition of the internal nodes after
--consecutive clocks.  The state of the
nodes assumes the clock has
--occured and data is stable.
--clock    multiplier      adder
adder
--number   output          output
output
--         register
register
-----------------------------------------
---
--rst      X               X
0
--1        X               X
0
--2        X               X
0
--3        KA1*A           KA1*A
0
--4        KB1*B           KA1*A+KB1*B
KA1*A
--5        KC1*C
KA1*A+KB1*B+KC1*C   KA1*A+KB1*B
--6        KA2*A           KA2*A
KA1*A+KB1*B+KC1    answer 1
--7        KB2*B           KA2*A+KB2*B
KA2*A
--8        KC2*C
KA2*A+KB2*B+KC2*C   KA2*A+KB2*B
--9        KA3*A           KA3*A
KA2*A+KB2*B+KC2*C  answer 2
--10       KB3*B           KA3*A+KB3*B
KA3*A
--11       KC3*C
KA3*A+KB3*B+KC3*C   KA3*A+KB3*B
--12       next KA1*A      next KA1*A
KA3*A+KB3*B+KC3*C  answer 3
--*/
--
/*****************************************
******************************/
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

--library virtex;
```

```
--use virtex.components.all;
--library synplify;
--use synplify.attributes.all;
library unisims_ver; -- include this for
modelsim simulation
-- when using mult18x18
entity matrix3x3 is
     port ( A, B, C: in
std_logic_vector(11 downto 0);
    CLK, RST: in std_logic;
    CWEL: in std_logic_vector(1 downto 0);
    KA, KB, KC: in std_logic_vector(9
downto 0);
    X, Y, Z: out std_logic_vector(11
downto 0));
end matrix3x3 ;
architecture model of matrix3x3 is

signal A_reg, B_reg, C_reg:
std_logic_vector (11 downto 0);
signal A_reg1, B_reg1, C_reg1:
std_logic_vector (11 downto 0);
signal CWEL_reg,i_wait: std_logic_vector(1
downto 0);
signal cnt9_wait: std_logic_vector (2
downto 0);
signal ain, bin: std_logic_vector (17
downto 0);
signal KA1, KB1, KC1, KA2, KB2, KC2, KA3,
KB3, KC3: std_logic_vector (9 downto 0);
signal data_mux: std_logic_vector (11
downto 0);
signal coeff_mux: std_logic_vector (9
downto 0);
signal cntr9, cntr9_out : std_logic_vector
(3 downto 0);
signal P1_reg,adder_mux,sum:
std_logic_vector (35 downto 0);
signal P1, P2, P3: std_logic_vector (35
downto 0);
signal cntr3 : std_logic_vector (1 downto
0);
signal j : integer range 0 to 7;
signal indexi : integer range 0 to 8;
signal i : integer range 0 to 3;

component MULT18X18
port(
A,B: in std_logic_vector (17 downto 0);
P: out std_logic_vector (35 downto 0));
end component;

begin

--/* ----------DATA INPUT SECTION------*/
--/* In the 3:1 data mux. To match the
pipeline of the
--Data inputs with the coeeficient inputs,
the data values are registered first
--At the input of the 3:1 mux and then
again at the output of the 3:1 mux.
--To make sure that the inputs don't
change in the middle of a set of vector
--summation, the inputs are held constant
for 9 clks. This will ensure that
--the input values seen by the 3x3 vector
is the same for the first set of
--answers. */

--/* cntr3 to count 0-1-2-3-1-2-3 */
process(CLK,RST) begin
  if (RST='1') then
      cntr3 <= "00";
  elsif (rising_edge (CLK)) then
      if(cntr3 = "11") then
            cntr3 <= "01";
      else cntr3 <= cntr3 + 1;
          end if;
  end if;
end process;

--/* inputs registered twice to match the
pipe length of the coefficients */

process(CLK,RST)
   begin
   if (RST = '1') then
          A_reg1 <= (others=>'0');
B_reg1 <= (others=>'0'); C_reg1 <=
(others=>'0');
   elsif (rising_edge (clk)) then
      if (j = 0) then
         A_reg1 <= A; B_reg1 <= B; C_reg1
<= C;
      end if;
   end if;
end process;

process(CLK,RST)
   begin
   if (RST = '1') then
          A_reg <= (others=>'0'); B_reg
<= (others=>'0'); C_reg <= (others=>'0');
   elsif (rising_edge (clk)) then
      A_reg <= A_reg1; B_reg <= B_reg1;
C_reg <= C_reg1;
   end if;
end process;
```

```
process(CLK,RST)
   begin
   if (RST = '1') then
        j <= 0 ;
   elsif (rising_edge (clk)) then
      if ( j < 8) then j <= j + 1;
        else j <= 0;
      end if;
   end if;
end process;

--/* ----------MODE SELECT SECTION-----*/
--/* mode select should be constant for 3
clk cycles  to complete one set .
--of coefficients. So modeselect is
updated every 3rd clk */
--/* cntr3 used to hold CWEL constant for
3 clocks. */
process(CLK,RST)
   begin
   if (RST = '1') then
          CWEL_reg <= "00";  i <= 0;
   elsif (rising_edge (clk)) then
      CWEL_reg <= CWEL;
   end if;
   if ( i < 4) then i <= i + 1;
        else i <= 0;
   end if;
   --end if;
end process;

--/* coefficient register update. The
register shd hold the
--value for 3 clks to get the right output.
*/
process (clk,rst)
  begin
   if (rst = '1') then
      KA1 <= "0000000000"; KB1 <=
"0000000000"; KC1 <= "0000000000";
      KA2 <= "0000000000"; KB2 <=
"0000000000"; KC2 <= "0000000000";
      KA3 <= "0000000000"; KB3 <=
"0000000000"; KC3 <= "0000000000";
   elsif (rising_edge (clk)) then
      case CWEL_reg is
          when "01" =>  KA1 <= KA; KB1 <=
KB; KC1 <= KC;
          when "10" =>  KA2 <= KA; KB2 <=
KB; KC2 <= KC;
          when "11" =>  KA3 <= KA; KB3 <=
KB; KC3 <= KC;
          when others => null;
      end case;
   end if;
  end process;

--/* ----------COEFFIECIENT MUX SECTION-*/
--/*cntr9 to count 0-1-2-3-4-5-6-7-8-9-1
*/

process(CLK,RST) begin
  if (RST='1') then
      cntr9 <= "0000";
  elsif (rising_edge (CLK)) then
      if (cntr9 = "1001") then
            cntr9 <= "0001";
      else cntr9 <= cntr9 + 1;
          end if;
  end if ;
  end process;

process (clk,rst)
  begin
   if (rst = '1') then
      coeff_mux <= "0000000000"; data_mux
<= (others => '0');
   elsif (rising_edge (clk)) then
      case indexi is
          when 0 => coeff_mux <= KA1;
data_mux <= A_reg;
          when 1 => coeff_mux <= KB1;
data_mux <= B_reg;
          when 2 => coeff_mux <= KC1;
data_mux <= C_reg;
          when 3 => coeff_mux <= KA2;
data_mux <= A_reg;
          when 4 => coeff_mux <= KB2;
data_mux <= B_reg;
          when 5 => coeff_mux <= KC2;
data_mux <= C_reg;
          when 6 => coeff_mux <= KA3;
data_mux <= A_reg;
          when 7 => coeff_mux <= KB3;
data_mux <= B_reg;
          when 8 => coeff_mux <= KC3;
data_mux <= C_reg;
          when others => null;
      end case;
   end if;
  end process;

process(CLK,RST) begin
  if (RST='1') then
      i_wait <= "01";
  elsif (rising_edge (CLK)) then
      if (i_wait > "00") then
```

```
            i_wait <= i_wait - '1';
        else i_wait <= i_wait;
            end if;
  end if ;
  end process;

process(CLK,RST) begin
  if (RST='1') then
      indexi <= 8;
  elsif (rising_edge (CLK)) then
      if (i_wait = "00") then
        if (indexi = 8) then
                  indexi <= 0;
        else indexi <= indexi + 1;
        end if;
      end if;
  end if ;
  end process;

--/* ----------MULTIPLIER SECTION-------*/

--/* 9x pumped multiplier; P3 registered
twice to match the pipelining
--   of the first adder  */
ain <= "00000000" & coeff_mux; bin <=
"000000" & data_mux;
MULT1: MULT18X18 port map( A => ain, B =>
bin, P => P1);

-- registering multiplier outputs --
process(CLK,RST)
   begin
   if (RST = '1') then
      P1_reg <= (others => '0');
   elsif (rising_edge (clk)) then
      P1_reg <= P1;
   end if;
   end process;

--/* ----------ADDER SECTION-------------
*/

--/* Adder mux. Inputs a '0' every 3rd clk
*/

process (cntr3(1) , cntr3(0) , sum)
begin
if (cntr3 = "01") then
adder_mux <= (others => '0');
else adder_mux <= sum;
end if;
end process;

-- Final adder -

process(CLK,RST)
   begin
   if (RST = '1') then
      sum <= (others => '0');
   elsif (rising_edge (clk)) then
   sum <= P1_reg + adder_mux ;
   end if;
   end process;

--/* ----------OUTPUT SECTION-------------
*/

--/* At the output of the adder, the first
valid X values appears at the 6th clk
--after reset. After this, at every 3rd
clk, a valid output values are obtained
for
--Y ,Z, X, Y, Z and so on.  This function
is realised using a enable cntr. The cntr
--after reset, counts upto 3 at which
point another output counter is enabled.
The
--output of the enable counter holds its
value of 3 as long as it is not reset. */

--/* output cntr starts after 4 clk to
match the initial pipe
--delays of inputs/coeeficients */

process(CLK,RST)
   begin
   if (RST = '1') then
 cnt9_wait <= "100";
   elsif (rising_edge (clk)) then
      if (cnt9_wait > "000") then
          cnt9_wait <= cnt9_wait - '1';
      else cnt9_wait <= cnt9_wait;
      end if;
   end if;
   end process;
--/*cntr9_out to count 0-1-2-3-4-5-6-7-8-
9-1-2- */

process(CLK,RST)
   begin
   if (RST = '1') then
 cntr9_out <= "0000";
   elsif (rising_edge (clk)) then
      if (cnt9_wait = "000") then
         if (cntr9_out = "1001") then
             cntr9_out <= "0001";
         else cntr9_out <= cntr9_out + 1;
         end if;
      end if;
```

```
   end if;
end process;

--/* adder output assigned to X,Y and Z */
process (clk,rst)
  begin
   if (rst = '1') then
        X <= "000000000000"; Y <=
"000000000000"; Z <= "000000000000";
   elsif (rising_edge (clk)) then
      case cntr9_out is
          --when "0001" => X <= X; Y <=
Y; Z <= Z;
          --when "0010" => X <= X; Y <=
Y; Z <= Z;
          when "0011" => X <= sum(11
downto 0); --Y <= Y; Z <= Z;
          --when "0100" => X <= X; Y <=
Y; Z <= Z;
          --when "0101" => X <= X; Y <=
Y; Z <= Z;
          when "0110" =>  Y <= sum(11
downto 0); --X <= X; Z <= Z;
          --when "0111" => X <= X; Y <=
Y; Z <= Z;
          --when "1000" => X <= X; Y <=
Y; Z <= Z;
          when "1001" =>  Z <= sum(11
downto 0); --X <= X; Y <= Y;
          when others => null;
      end case;
   end if;
end process;
end model;
```