Design of Application-specific Instruction Set Processors with Asynchronous Methodology for Embedded Digital Signal Processing Applications

KWOK Yan-lun Andy

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Electronic Engineering

November 2004

ABSTRACT

Abstract of thesis entitled:

Design of Application-specific Instruction Set Processors with Asynchronous Methodology for Embedded Digital Signal Processing Applications

Submitted by KWOK Yan-lun Andy

for the degree of Master of Philosophy in Electronic Engineering

at The Chinese University of Hong Kong in November 2004

This thesis presents a new design methodology of application-specific instruction set processors (ASIPs) using asynchronous design methodology. ASIPs are today's enabling technology for tackling increasingly complex embedded systems together with tightening time-to-market constraints. It combines the high design productivity of the software approach and the high performance of the hardware approach, which brings us programmable devices with dedicated hardware features for real-time applications. A major obstacle of ASIP design is the larger design space compared with pure hardware or pure software implementations. This makes it hard for the designers to search for large amounts of architecture alternatives in order to find an optimal implementation within a competitive design timeframe.

The platform-based design methodology using asynchronous technology is developed. A highly extensible and flexible platform is designed as the heart. Using asynchronous interfaces, components can be added to the platform rapidly to expand its functionalities without affecting the global timing. The platform can be effectively optimized for particular applications.

The proposed design methodology is proven to be effective in the case studies. It shows that the base platform can be scaled up easily to dramatically speed up different kinds of kernels, reaching the performance level of some advanced parallel DSPs. The benchmark of rotation CORDIC algorithm even illustrates further performance gain by using asynchronous design methodology for seamless cooperation between two different clock domains.

摘要

日益進步的半導體技術，使計算機系統得以向多樣化和高整合進化。面對越來越複雜的應用以及縮短開發周期的需求，使用專用指令集處理器（ＡＳＩＰ）是最佳的選擇。專用指令集處理器結合專用集成電路（ＡＳＩＣ）的卓越運算能力及可編程架構的高靈活性，使系統設計師能在短時間內完成產品上市。但是設計專用指令集處理器本身是一個極難解的問題。當中涉及複雜的軟硬體協調，使設計師無從入手，更遑論嘗試各種不同的軟硬體組合以求得到最合適的設計。

有鑒於此，本論文提出一個利用異步設計技術和以開發平台爲基礎的方案。利用異步設計的高度模組化和其溝通界面，不同的模組可迅速組裝在一個可延展的平台上，面向目標應用對平台進行優化。建基於平台的方案爲專用指令集處理器設計提供一個切入點，並有效將設計流程簡化。設計師因而可在有限的時間內評價不同配置的效能。

本論文所提出的方案在不同的實作中能有效面向不同的數字訊號處理核心（ＤＳＰ Ｋｅｒｎｅｌ）進行優化，並帶來顯著的效能增益。其效能甚至可媲美超長指令數字處理器（ＶＬＩＷ ＤＳＰ）。此結果令人鼓舞，亦證實本方案爲可行並有效。

ACKNOWLEDGEMENTS

Gratitude to The Almighty, without whose blessing nothing can be achieved.

Special thanks to my supervisor Professor Choy Chiu-Sing, who has been a constant source of inspiration for me, and has given me invaluable support, encouragement and guidance. My appreciation also goes to Professor Chan Cheong-Fat and Professor Pun Kong-Pang for their constructive comments on the work.

I would also like to thank Mr Yeung Wing-Yee, who maintained our design environment and helped me with most of the technical problems. I will not forget the students in our research group, especially Cheng Wang-Tung, Shen Jun-Hua, Yu Chun-Pon, Xu Ke, Leung Pak-Keung, Chan Chi-Hong and Chan Wing-Kin, whose energy and diversity have made for many stimulating interactions. It has been a genuine pleasure to work with them.

My heartfelt thanks to my parents for their continuing support of all kinds, and to auntie Mavis and my friend Chan Wan-Man for their endless encouragement.

Kwok Yan Lun

2004

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES AND EXAMPLES

1. Introduction

1.1. Motivation

Moore's law 0 has driven the development of technology in the silicon industry for the past three decades. The silicon gate count continues to grow; and the transistor size continues to be scaled down with the regular pace that Moore predicted. Continuing growth in silicon capability is rapidly magnifying the functionality of digital circuits, leading us into a new era of diversified applications on embedded digital devices.

However, Moore's law is found to be increasingly irrelevant. While the silicon gate count still continues to grow as Moore predicted, hardware designers find difficulties in using all the gates efficiently and effectively with the leading edge technology. The improvement of electronic design aided (EDA) tools and engineering skills cannot keep track of the growth in the capability and complexity of digital circuits.

This gap of design productivity leads to great tension in achieving time-to-market, and therefore it becomes a serious burden for realizing advanced competitive designs.

As time-to-market is primarily important, application-specific integrated circuits (ASIC) are no longer the best option although they deliver excellent performance in terms of speed, power and silicon efficiency. Shifting from hardware to software can take advantages in this scenario. Increasing the software proportion of a design improves the design productivity due both to the simple and predictable software design flow and to the high degree of reusability of the software library together with the programmable device itself. It can be expected that programmable devices, especially general-purpose processors, continue to benefit from Moore's law scaling. However, general-purpose processors show limits in meeting stringent requirements of embedded real-time applications. Compared to ASICs, general-purpose processors consume more power and offer lower performance. For this reason, the new type of programmable devices, so-called application-specific instruction-set processors (ASIP) are being developed to fill the gap between performance and time-to-market of purely hardware and purely software solutions as shown in Figure 1.1.

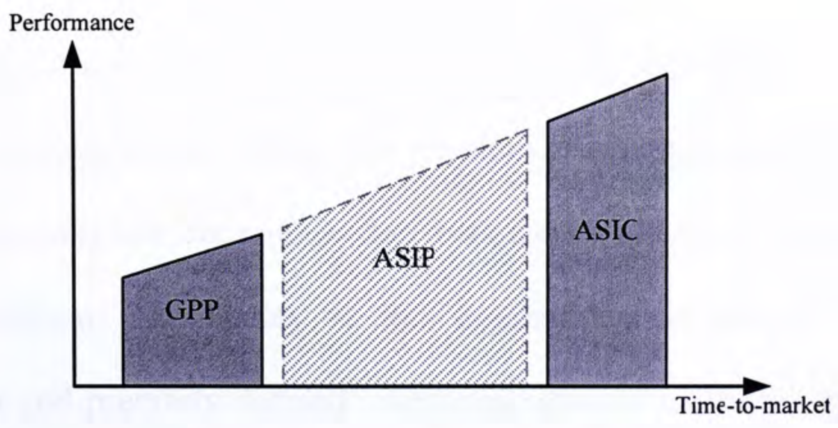


Figure 1.1: ASIP bridges the performance and time-to-market gap

The philosophy of ASIPs is to put to good use the optimized user-defined instruction set and datapath to gain higher performance of computation for certain target applications. From the software designers' point of view, ASIPs offer efficient macros to replace heavily loaded subroutines in software implementation. From the hardware designers' point of view, ASIPs simplify the hardware algorithms by realizing some tedious but uncritical modules in software. The advantages of this arrangement are threefold : 1) High degree of flexibility allows late design change to keep track of the evolving standards. 2) High degree of reusability facilitates rapid retargeting. 3) Tailor-made optimized features breakthrough the performance bottleneck of general-purpose processors.

Design of an efficient ASIP is not a straightforward task. On the one hand, it requires precise judgment to balance the performance and the flexibility. On the other hand, it eliminates over-design and wastage of silicon. This multi-dimensional optimization problem widely covers three areas: hardware logic design; computer architecture design; and application software design, which makes it hard for the designers to search for large amounts of architecture alternatives in order to identify an optimal implementation within a competitive design timeframe.

Another eye-catching technology, asynchronous design methodology [2][3][4] may provide a new opportunity to tackle this problem. Asynchronous design style uses handshake to accomplish communication between modules in order to solve the clock skew problem. The beauty of this communication mechanism inherently provides robust and precisely defined interfaces between modules. By having these interfaces, the specifications of the modules can be precisely defined, and can be independent of any global timing reference. The design task of a large system is greatly simplified to smaller tasks – design of component modules and verification of their interfaces. Undoubtedly, this is a good practice for designing ASIP efficiently. In addition, this module-based design facilitates design reuse, which means it increases the degree of retargetability of the designed ASIP. By exploring suitable design space for asynchronous technology, it is possible to solve the complex optimization problem.

1.2. Objective and Approach

The objective of this research is to study the way to apply asynchronous technology to ASIP design, and to provide an effective design methodology to optimize an asynchronous ASIP to meet real-time requirements of the target application.

Our work focuses on the optimization of the datapath with its associated instruction-set in order to fulfil the timing criteria for embedded digital signal processing (DSP) applications. Power efficiency is the second concern but is not a major factor in optimization. The other issues about architecture design are also addressed. However, the rest of the ASIP design issues, such as application analysis and software generation, are not covered.

As optimization of the datapath is closely related to the characteristics of the target application and some parameters of the overall architecture, such as memory bandwidth, depth of pipeline stages and degree of computational concurrency, the following approach is taken to achieve our objective:

1. To find an architecture that can maximize the design space of the datapath;
2. To find a parameterized extensible architecture that can take advantage of asynchronous technology;
3. To find a methodology to design an optimized datapath based on the characteristics of the target application.

The extensible architecture is the centre of our proposed methodology. Further datapath optimization essentially depends on that architecture. For the sake of ease of notation, this extensible architecture is called a "platform" for the rest of the thesis.

1.3. Thesis Organization

The remainder of the thesis is organized as follows:

Chapter 2: Related Work. This chapter gives an outline of the related work in ASIP design methodology and some remarkable achievements in asynchronous technology research.

Chapter 3: Asynchronous Technology. This chapter briefly describes the design style of asynchronous circuits. Afterwards, there is a discussion on the best option for our platform and also on its implementation.

Chapter 4: Platform-based ASIP Design Methodology. This chapter provides a complete description of the proposed design methodology. The exploration of the design space of the datapath, the architecture of the platform and the overall design flow are addressed.

Chapter 5: Design of ASIP Platform. This chapter presents the microarchitecture of the platform. It includes functional description of each module, their working mechanism and the design consideration.

Chapter 6: Case Studies. To prove this design methodology, case studies were conducted. A detailed description of the case studies is available.

Chapter 7: Conclusion. This chapter summarizes the overall research work. Perspectives of future work are pointed out.

2. Related Work

2.1. Coverage

Presently, there is no publication covering the design methodology that explores advantages of using asynchronous technology in ASIP design. Publications related to asynchronous ASIP design can be separated into two areas: ASIP design methodologies and asynchronous processor methodologies. In order to focus on our objective, the summary of ASIP design methodologies describes only the works that have an explicit treatment on hardware implementation, and the resulting hardware should be in the class of instruction set architecture. The design methodologies mainly on compilation technique, software generation and synthesis of dedicated hardware accelerators are not covered.

For the area of asynchronous design methodologies, we summarize some remarkable asynchronous processor designs in industry and in academia. A detailed discussion of the asynchronous design methodology is given in Chapter 3.

2.2. ASIP Design Methodologies

Existing ASIP design environments can be classified into two approaches. Some design environments are based on predefined processor platforms and provide different architectural options for customization. Other environments provide architecture description languages for the designers to describe their target processor architectures.

Xtensa of Tensilica [5][6], R.E.A.L of Philips[7], ARCtangent-A5 of ARC [8] and Jazz DSP processor of Improv Systems [9] are the commercial design environments using predefined processor platforms approach.

Xtensa of Tensilica is a configurable, extensible and synthesizable RISC (reduced instruction set computer) processor with load store architecture. Its base architecture has a compact 16- and 24-bit instruction set comprising of 80 instructions. The configurable parameters include the choice of 32 or 64 general-purpose 32-bit registers, the size of cache, the write buffer size, and the availability of designer defined instruction execution unit. Designers can define the mnemonic, the encoding, and the semantics of single cycle instructions using TIE language. In addition, the development environment includes ANSI C/C++ compiler, linker, assembler, debugger, code profiler, and instruction set simulator.

The R.E.A.L of Philips is customizable DSP having two independent 16x16 bit multipliers, four parallel 16-bit ALUs which can be combined into two 40-bit ALUs (including eight overflow bits each), and a number of parallel shifters and saturators in base architecture. Besides a standard 16- and 32-bit instruction set, there are additional Application Specific Instructions (ASIs), which allow the full parallelism of the DSP to be exploited. The ASI concept allows up to 256 VLIW instructions in a 96-bit width look-up table inside the R.E.A.L. DSP. These are triggered by a special class of 16 bit instructions, stored in the normal programme memory. The ASI look-up table can be a RAM (for prototype chips), ROM, a synthesized netlist, or a combination of these. If the ASI table is implemented in RAM, then its contents can be modified using the JTAG port, or under DSP programme control by writing to dedicated registers within the DSP.

The ARCtangent-A5 of ARC is a four-stage 32-bit RISC processor that can be configured and extended to match the application requirements. Designers can customize the processor in two ways: configuration and extension. Configuration is the ability to change existing features of the processor, such as the main-memory and auxiliary-bus widths; the size and organization of the instruction and data caches; or the size of local memory and DSP XY memory. Extension is the ability to add entirely new features to the processor such as a 32x32-multiply instruction, a USB peripheral and user-defined application-specific extensions. The resulting core is generated to HDL code together with synthesis scripts, simulation make-files, documentation and an automated test environment.

The Jazz DSP processor of Improv Systems is a configurable VLIW processor for their proprietary Programmable System Architecture (PSA). Improv employs this architecture that can scale from a single, uniquely configured Jazz DSP processor core, to a system level platform implementation that consists of many of these uniquely configured Jazz processors in an interconnected structure defined by shared memory maps between the processors. Each processor instance can be customized by custom RTL blocks and instructions to create a designer-defined DSP core. The Jazz PSA Composer Tool Suite provides designers with automatically generated synthesizable HDL code and a full set of software design tools including the debugger, simulator and profiler.

Other design environments using architecture description languages include the design environment of Target Compiler Technologies [10], LISA Processor Design Platform [11][12], MetaCore [13] and PEAS-III [14].

The design environment of Target Compiler Technologies is based on the processor modelling language nML. nML offers designers the abstraction level for describing a processor architecture and instruction set (ISA), which serves as an input to the various tools. nML captures the specification of the processor's instruction set, together with sufficient structural information to enable efficient compilation. Processor designers can describe alternative instruction-set architectures in nML. The support-tools for corresponding architecture are automatically available. Once the architecture has been optimized in nML, the control logics of processor description can be translated automatically into an HDL model. This HDL description can be synthesized with commercially available synthesis tools, for ASIC or FPGA implementation.

The LISA Processor Design Platform (LPDP) tool-suite is based on the machine description LISA. Starting from architecture descriptions in the LISA language, software development tools can be generated including HLL C-compiler, assembler, linker, simulator, and debugger front end. LISA is a language which aims at the formal description of programmable architectures, their peripherals, and external interfaces. The language elements of LISA enable the description of different aspects of processor architectures like behaviour, instruction set coding and syntax. The language LISA and its generic machine model can produce bit- and cycle/phase-accurate models of systems that consist of programmable architectures and peripheral hardware components. Moreover, synthesizable HDL (VHDL, Verilog, SystemC) code of the target processor can be generated and processed by the standard synthesis tools.

MetaCore is a DSP-oriented ASIP development system that can generate efficient ASIP using benchmark-driven design methodology. The heart of the MetaCore system is a predefined micro-architecture. The design style of the predefined micro-architecture is parameterized and pipelined. The architectural parameters include register file size, bus width, address space of each memory, and bit width of functional blocks. The specification of the target ASIP in the MetaCore system is described using the structural specification language MSL and behavioural specification language MBL. MSL is used to specify the data path structure of the target micro-architecture, while MBL is used to specify the architectural parameters and the behaviour of instructions for the target ASIP. The MSL description consists of declarations of hardware resources such as busses, latches, multiplexer, functional units, and interconnections among the hardware resources. A synthesis tool called SMART is used to translate the given processor specification into the corresponding HDL code of the target ASIP equipped with the user-defined application-specific instructions.

PEAS-III is an architectural level processor design environment based on a micro-operation description of instructions. In the environment, designers model the target processor with the following five items: 1) Architecture parameters such as the number of pipeline stages, the number of delayed branch slots; 2) Declarations of resources to be included in processor (e.g. ALUs, registers); 3) Instruction format definitions which include interrupt conditions and the number of execution cycles of interrupt conditions and the number of execution cycles of interrupt; 5) Micro-operation descriptions of instructions and interrupts. PEAS-III synthesizes the datapath and the control logic of the processor, and generates a simulation model and synthesizable VHDL descriptions of the processor.

2.3. Asynchronous Technology on Processors

Over the past few years, industry and academia have put much effort on asynchronous circuit technology. Their achievements can be concluded by many advanced and sophisticated asynchronous processors. Table 2.1: The summary of asynchronous processors designed by industry and academia gives a summary of asynchronous processors designed by industry and academia.

Table 2.1: The summary of asynchronous processors designed by industry and academia

Organization and Reference	Description	Achievements
University of Manchester, [15][16][17]	**The Amulet Series.** A series of asynchronous ARM processors using self-timed micropipelined VLSI implementation.	Successfully delivered the asynchronous processors for commercial use.
Philips Electronics, [18]	**Asynchronous 80C51.**	Four times lower power than a power-optimized synchronous version. Significant reduction of EM emissions.
Intel Corporation, [2]	**1) Asynchronous Pentium !!!; 2) Incorporate clockless elements in Pentium 4**	The asynchronous Pentium !!! processor is three times faster and consumes half the power of synchronous counterpart.
Sharp Corporation, [19]	**DDMP Signal Processor.** A self-timed data driven multi-media processor aimed at digital television receivers and other applications	Operating at a speed of 8600 Million Operations per Second and with power consumption of less than 1 watt.
Caltech, [20]	**Asynchronous MIPS R3000.** Using asynchronous circuits to implement a deep, fine-grained pipelined MIPS processor	R3000 exhibits significantly improved MIPS/watt performance over the synchronous version when scaled to account for different processes and voltages
Asynchronous Digital Design Pasadena, Calif, [21]	**1) Vortex processor; 2) MiniMIPS processor.**	MiniMIPS processor is twice as fast as all other designs using the 0.6 micron process in addition to 30 percent less power consumption.
Tokyo Institute of Technology and Tokyo University, [22][23]	**TITAC2.** A full-featured 32-bit architecture.	Using delay-scaling techniques to improve performance by taking real circuit delays into account, rather than conservatively assuming unbounded gate delays

2.4. Summary

Industry and academia provided convincing demonstrations on asynchronous processors. Their works showed the feasibility and potential performance gain of using asynchronous circuits in processor design. The published ASIP design methodologies do not pay special attention to asynchronous technology. All of them are focused on synchronous designs. None of them explicitly explores the power of asynchronous circuits in ASIP design.

The differentiation of our work against the pervious works is that we consider asynchronous technology as a factor in optimization. We do not consider asynchronous ASIPs as the straightforward translation of synchronous ASIPs. We also focus on the delay insensitive nature of asynchronous circuits in order to explore the opportunity to enhance the design reusability in ASIP design.

3. Asynchronous Design Methodology

3.1. Overview

Asynchronous circuits are fundamentally different from its well-known counterpart – synchronous circuits. The operation of asynchronous circuits does not rely on global clock signal as that of synchronous, but on local handshake signals. The handshake signals are basically the control signals in the communication between modules. Different styles of asynchronous circuit implementation may have different handshake protocols [24], for instance, two-phase protocol and four-phase protocol. In essence, all handshake protocols are the composition of request states and acknowledgement states.

An abstract interface of asynchronous circuits is shown in Figure 3.1. This interface has three channels for communications, request, acknowledgement and data channel between two modules. For simplicity, one module is defined as sender which is a data provider and another one defined as receiver, a data consumer. When the sender is ready to send data, a request state is asserted. A request signal is sent through the request channel to inform the receiver. (This signal can be transition sensitive, level sensitive or in binary encoded form. This is also true for acknowledgement signal). After receiving this signal, the receiver starts to process the data and sends back an acknowledgement signal when it finishes its work. Then the sender prepares another set of data for next transfer. This data transfer mechanism can safely avoid hazards. This concept is much clearer on asynchronous pipeline, which is discussed in the section on Micropipelines on page 17.

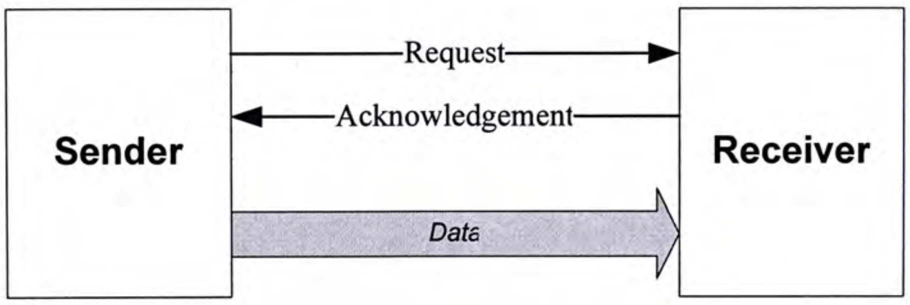


Figure 3.1: An abstract interface of asynchronous circuits

Compared to synchronous circuits, asynchronous circuits have no common or discrete reference time for all modules. There is only local reference time between two communicating modules. This was previously considered a disadvantage, because this violates the beauty of synchronous circuits – all components reference to common and discrete time defined by clock, which is believed to greatly simplify the design work. However, when the clock skew problem becomes significant due to process scaling, asynchronous circuits beat its counterpart in this arena. Besides, according to [25] asynchronous technology offers opportunities in the following areas:

1. High performance;
2. Low power consumption;
3. Low noise and low EMI emission;
4. A good match with heterogeneous system timing.

For system integration, asynchronous technology is undeniably a better option than synchronous. Its handshake-based communication mechanism provides a reliable environment for reuse of pre-designed, pre-verified, pre-characterized IP blocks. The freedom of using IP blocks with different specifications offers the highest potential for improving design productivity.

3.2. Asynchronous Design Style

This section is an introduction to different asynchronous design styles. Based on different sizes of communication blocks, three styles are presented – micropipelines, fine-grain pipelining and globally-asynchronous-locally-synchronous design.

3.2.1. Micropipelines

Micropipeline was first introduced in Ivan Sutherlands' Turing Award lecture [26]. Sutherlands designed micropipelines as an asynchronous alternative to synchronous pipelines. From the definition of micropipelines, this is a simple form of event-driven elastic pipeline that contains simple circuitry in each pipeline stage.

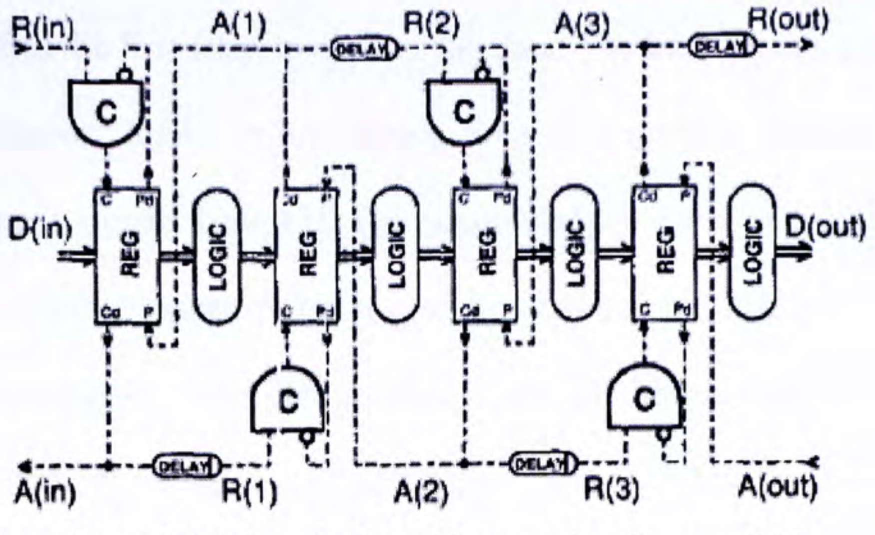


Figure 3.2: Micropipeline with processing. (Source: [15])

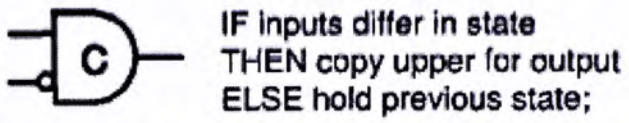

C
IF inputs differ in state
THEN copy upper for output
ELSE hold previous state;
Figure 3.3: Muller C-element with inverter. (Source: [15])

Figure 3.2 is a typical structure of micropipelines. This circuit operates in two-phase handshake protocol which is based on the signal-transition conceptual framework proposed in [26]. To fit to signal-transition signalling control system, capture and pass latches are used as storage elements. The inputs *C* and *P* govern the capture and pass function of the latch, and the outputs *Cd* and *Pd* represent "capture done" and "pass done" respectively. When there is a transition occurring at signal *C*, data will be captured and held in the latch. On the other hand, the latch looks transparent while a transition is present at signal *P*.

The basic operation of the micropipeline can be easily explained using the events of request and acknowledgement signal. Assuming that all the wires are initially set at zero and all latches are initially transparent, when there is a transition in the request input, then output of the first C-element will be changed from zero to one. This transition notifies the first latch to capture the data. The latch passes the captured data to the computation logic, at the same time it asserts a pair of request and acknowledgement signals from *Cd*. The acknowledgement signal is sent back to the data source while the request signal is sent through a delay line to the second stage. The delay line matching the computation logic to the computation is completed before the arrival of the request signal. Meanwhile, the first C-element blocks the request from the data source and waits for an acknowledgement from the second stage. After receiving the request, the second latch captures the data and sends back an acknowledgement, and then the first latch is allowed to capture data again. This operation is repeated when the next request signal arrives, and the data propagates along the pipeline to the output.

Micropipelines have a simple and effective structure. It is easy to implement and easy to achieve high throughput. Also, the latches moderate the flow of data through the pipeline, and can be used to filter out hazards. Thus, any logic structure can be used in the logic blocks, including the straightforward translation of synchronous pipelines.

Presently, there are different derivatives of micropipelines. Some designs give up using capture and pass latches but use simple latches with four-phase latch control (Figure 3.4). Some designs involve self-timed logic, which makes the pipeline even more elastic (Figure 3.5).

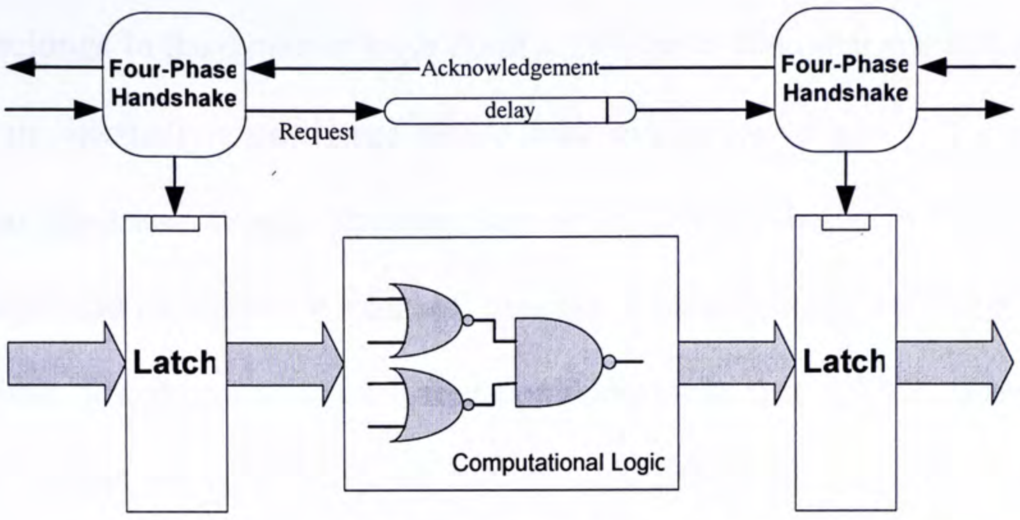


Figure 3.4: Micropipeline for simple logics

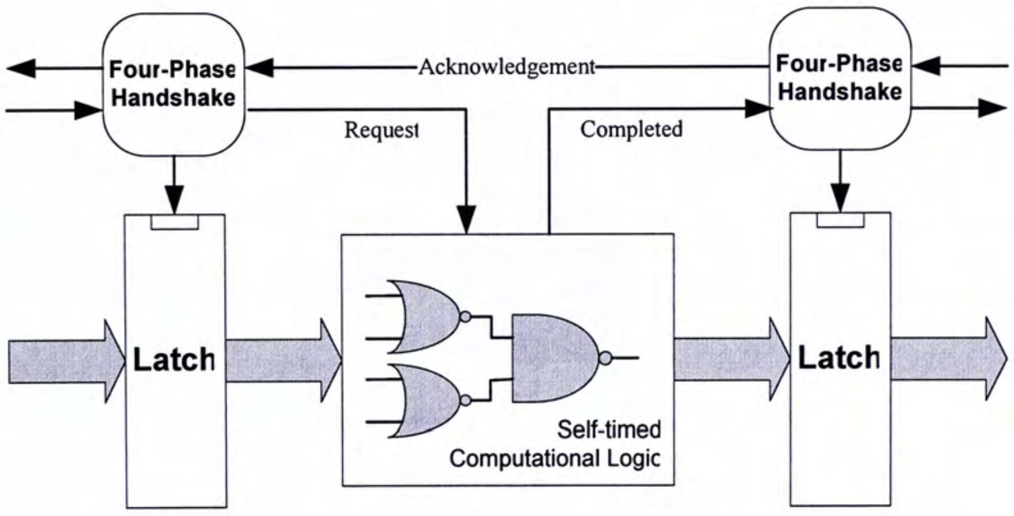


Figure 3.5: Micropipeline for self-timed logics

3.2.2. Fine-grain Pipelining

A number of design styles targets higher performance by using much smaller communicating blocks. These styles decompose the design into a fine-grain pipeline. In some aggressive approaches, the critical path of each pipeline stage is limited to a few logic gates [27]. In order to have ultimately high throughput, this kind of design styles adopt a latch-free structure in fine-grain pipelines [28][29], as the capture and pass latch is too slow compared to the computation logic. Differential cascode voltage switch logic (DCVSL) [30] is the spot of this structure.

DCVSL belongs to the dynamic logic family. Similar to the other members, DCVSL operates in alternative precharge phase and evaluation phase [31], but it has differential input and output. The structure of DCVSL is shown in Figure 3.7. It is symmetrical and comprise a pair of domino logics (Figure 3.6). An attractive characteristic for using it in latch-free applications is that DCVSL can hold the evaluated output whereas the input data is changed [32]. Thus, DCVSL can be understood as a combination of the logic and storage elements that are preferred in fine-grain pipelines.

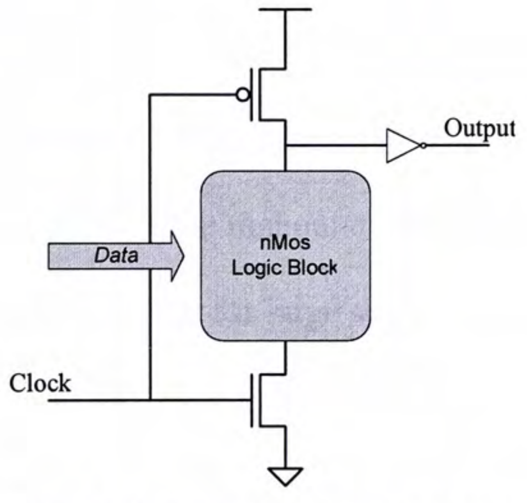


Figure 3.6: Conventional domino logic

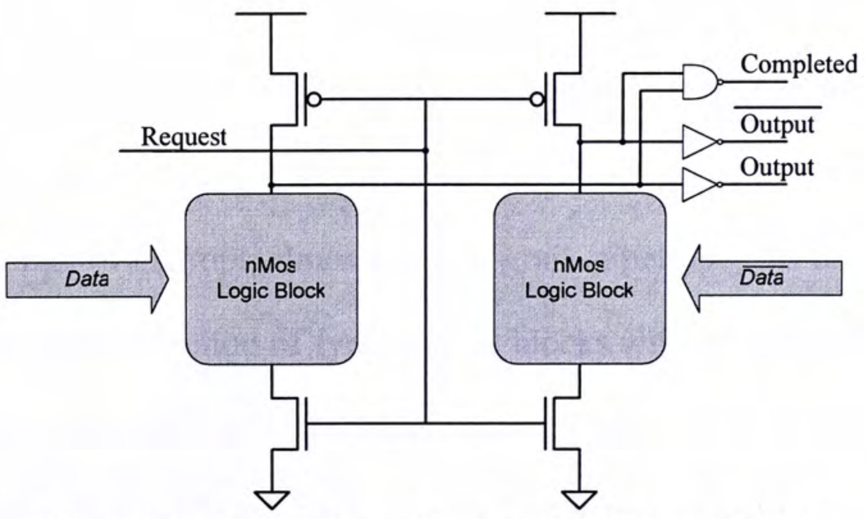


Figure 3.7: Differential cascode voltage switch logic (DCVSL)

The operation of DCVSL is similar to conventional dynamic logic. When the request is low, the DCVSL shifts to precharge phase. At this moment the two upper pMOS are turned on and make the two outputs low. When the request is high, then it is in evaluation phase. Either one of the nMOS logic trees is turned on to change the output to high. A differential output is obtained when the evaluation is completed. This operation mechanism is inherently an incomplete handshake protocol, which can provide a foundation to simplify the handshake logic. On the other hand, the logic can indicate the completion of computation by the differential output. No surplus timing margin is needed in contrast to the worst-case delay line used in micropipelines.Thus, higher performance can be expected.

Fine-grain pipelining is excellent for high speed applications [33][34][35]. However, designing dynamic logic requires more manual effort and incurs much longer design cycles. This design style is not suitable for large scale designs.

3.2.3. Globally-Asynchronous Locally-Synchronous (GALS) Design

GALS uses largest communication blocks compared to the other two design styles. Its asynchronous communication scheme targets on coarse grained block level whose size can be as large as a finite state machine or an IP. The scope of GALS is also different from that of micropipelines and fine-grain pipelining. Its design philosophy focuses on the interconnection of synchronous blocks with asynchronous technology. This approach partitions the large synchronous system into smaller synchronous blocks and interconnects them with asynchronous handshake protocol. Similar to other asynchronous design styles, the communication among blocks is referenced to local handshake signals, therefore the synchronization can be spread among the system effectively and reliably.

Synchronous-Asynchronous Cooperation

In order to carry out asynchronous global communication with others, all synchronous modules are wrapped by an asynchronous interface. This asynchronous wrapper is potentially capable to communicate with purely asynchronous modules as long as they share the same protocol. This is a low cost way to establish synchronous-asynchronous cooperation in a system (Figure 3.8). GALS can bridge synchronous and asynchronous technology together to form a heterogeneous system that is free to make good use of synchronous and asynchronous IPs.

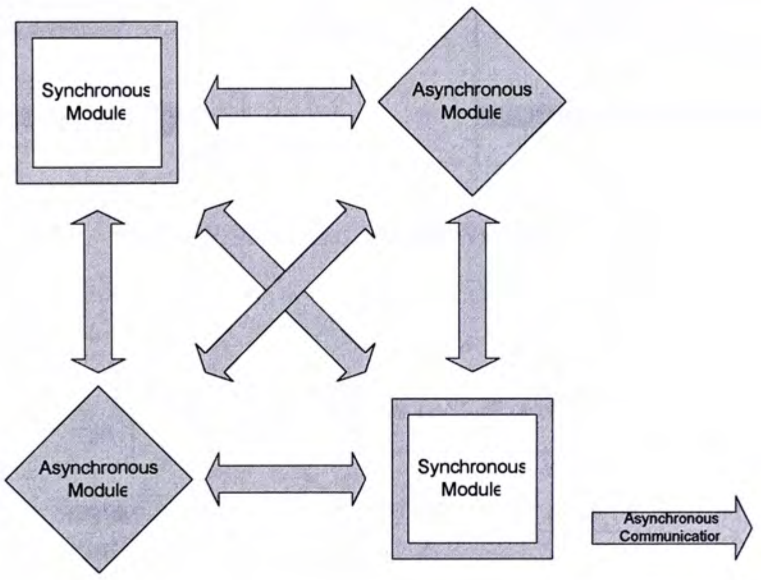


Figure 3.8: Globally asynchronous communication between modules.

Asynchronous Wrappers

The structure of an asynchronous wrapper is illustrated in Figure 3.9. The asynchronous wrapper surrounds a synchronous module aiming to provide a completely asynchronous external interface. All input and output ports of the module are managed by separate port controller. When data enters or leaves the module, the controller bundles the data with handshake signals to ensure its validity in the whole transfer process. Additionally, the asynchronous wrapper provides a local clock signal for the synchronous module. This clock signal is independent from outside modules in order to fully encapsulate the synchronous module. On the other hand, this clock is generated as stretchable. If incoming data arrives too close to a sampling clock edge, either the clock edge or the data transfer gets shifted to a later point in time in order to avoid being metastable (Figure 3.10).

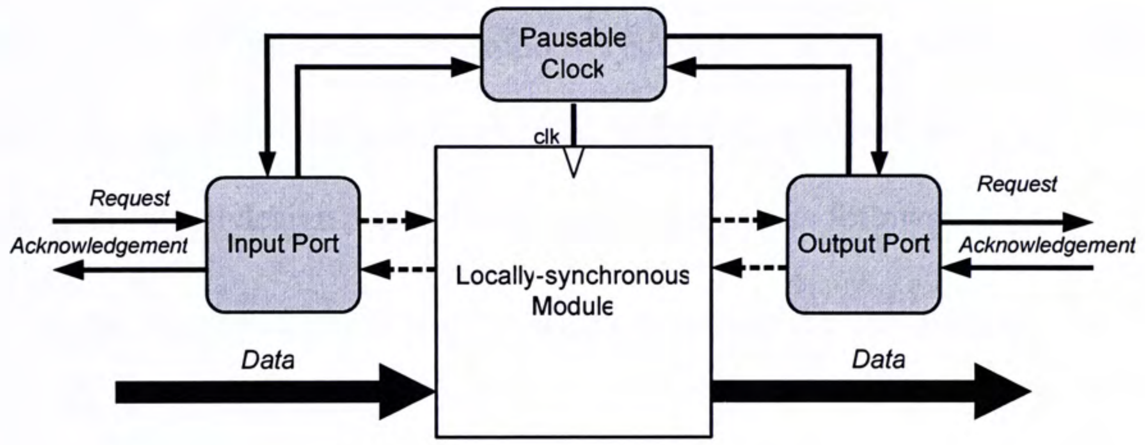


Figure 3.9: Asynchronous Wrapper

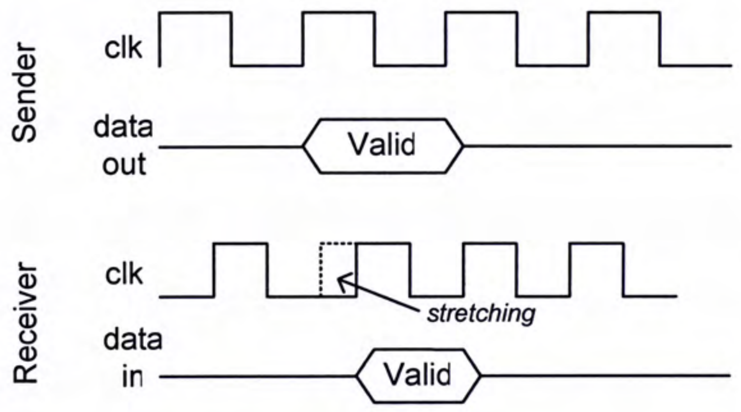


Figure 3.10: Pausible clock while stretching (Other control signals are not shown)

Design Methodology of GALS Systems

While asynchronous design technology promises to solve the clock skew problem and favours reuse of IPs, hardware designers are not willing to migrate completely from synchronous to asynchronous in short. The reason is that the design of asynchronous circuits needs special design methodology that has no or very little support from commercial EDA tools. Without dedicated EDA tools, designers have to work out an asynchronous circuit in semi-custom or full-custom manner.

Using GALS is an easier entry point to the asynchronous world. The design methodology of GALS is an extension of the familiar synchronous design methodology. It partitions the synchronous system into optimal size of synchronous modules and redefines the communication among these modules to asynchronous manner. The overall design methodology is summarized as follows:

1. At the beginning, the hierarchical description of the synchronous system has to be accomplished.
2. According to the structure of hierarchy, a trail partitioning is performed by separating the modules on the first level of hierarchy. If the size of module violates the system specification, that module may be further partitioned into its inner hierarchy or merged with other modules.
3. In the communication refinementstage, each module has to be characterized by its operating frequency, the expected throughput and the nature (push or pull mode) of its ports. By considering the requirement of each module and the communication requirement between two modules, suitable asynchronous wrappers can be identified.
4. The synchronous modules are synthesized and partitioned in floor planning.
5. Finally, the design undergoes evaluation. If the design cannot meet the timing constraints, there are two paths to go. One is to adjust the clock periods of some modules by adding delay in the layout. If the result is too bad or the

clock period of each module is already fine tuned, the design has to be re-partitioned and its communication redefined.

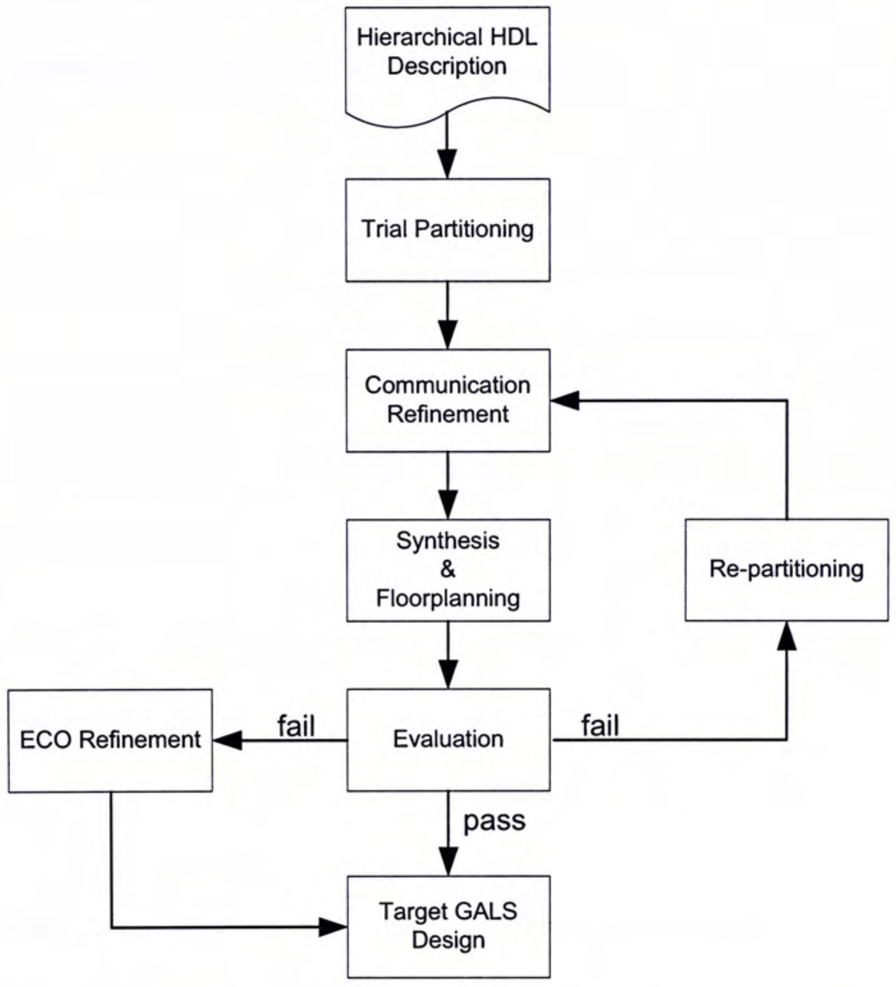


Figure 3.11: The GALS design methodology (modified from [36])

Hardware designers can maintain synchronous design methodology to implement the computation and control parts of the whole GALS system, and need to pay more attention to and manual effort on the asynchronous wrappers only. As time-to-market and design efficiency are the number one considerations, the GALS design style is the best among micropipelines and fine-grain pipelining.

3.3. Advantages of GALS in ASIP Design

Design of ASIP is not only an arena of performance, but also is an arena of time-to-market and design efficiency. To take this into account, our ASIP platform is designed with the GALS design style. There are three points to support our choice.

3.3.1. Reuse of Synchronous and Asynchronous IP

Design reuse can greatly improve time-to-market. Designers are now seriously exploring opportunity for reusing IPs to compose a system. The GALS design is the pioneer in this area. It has the freedom to use the mixture of synchronous IPs and asynchronous IPs The exploration space of the GALS design in IP reuse is much wider than other design styles.

3.3.2. Fine Tuning of Performance and Power Consumption

Using multiple frequency and voltage in a system is recognized to be an aggressive power saving and performance tuning strategy [36]. In GALS systems, all modules are perfectly encapsulated. All modules are isolated from one another, and do not reference to a correlated clocking system. Their communication is controlled by reliable handshakes, and therefore GALS systems are adaptive to change of timing. Using GALS, designers are empowered to use fine grained frequency and voltage scaling, even a dynamic one to compose the target system. The design space of power efficient ASIP can be further widened.

3.3.3. Synthesis-based Design Flow

The push factor for using asynchronous is that designers have to work in transistor level or standard cell level to some extent. For GALS, this adverse factor no longer exists. Muttersbach reported a set of almost synthesizable asynchronous wrappers in [37]. Only one cell has to be designed at layout level. Designers are allowed to use behavioural model or register transfer level (RTL) model to describe GALS designs.

3.4. Design of GALS Asynchronous Wrapper

To realize a GALS processor, a set of input- and output-port controllers for asynchronous wrappers is designed based on [37]. The input port module is also reused in our design. Different from Muttersbach's design, our wrappers are fully synthesizable.

3.4.1. Handshake Protocol

For the asynchronous communication channel, the four-phase handshake protocol is selected. The timing diagram of the protocol is illustrated in Figure 3.12. In four-phase protocol, valid data is accompanied by a pair of request and acknowledgement signals. When the data is ready for the receiver, the send sets the request signal to high. The data is guaranteed to be valid until the request is dropped. After getting the request signal, the receiver takes the data and sends back an acknowledgement signal. Then the sender can set the request to low and process another set of data.

Four-phase handshake protocol is level sensitive. It can interface with memory naturally and control the latches effectively. Compared to two-phase one, four-phase protocol is more robust because the data is wrapped by the request signal. The invalid data can also be indicated by the low request signal.

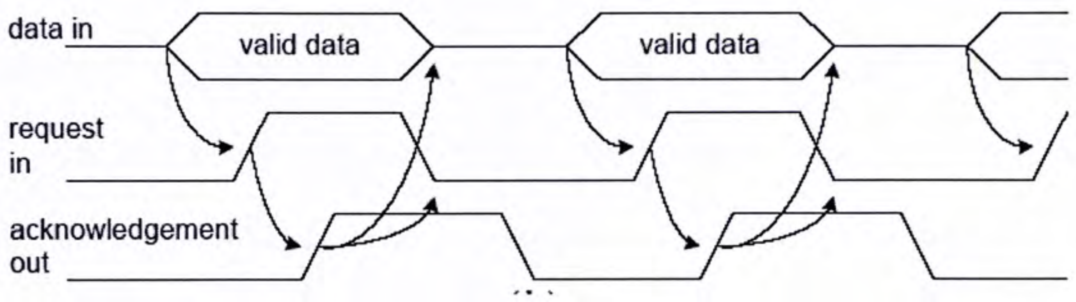


Figure 3.12: Timing diagram of the four-phase handshake protocol

3.4.2. Pausible Clock Generator

The pausible clock generator is an important component in asynchronous wrappers. The module either establishes or is requested for synchronization with another module, and the period of the clock is stretched to match the clock of another one. The port controllers are entitled to govern the stretch of the clock by sending a stretch signal to stop the clock. As the clock keeps oscillating, it is possible for the stretch signal to get too close to the clock edge leading to the state of being metastable. A mutual exclusion (ME) element is used in the pausible clock to decide which one can take over the control.

The structure of the pausible clock is shown in Figure 3.13. A ring oscillator is used instead of crystal oscillators or PLL in order to be able to have full control of the clock generation. To provide a control interface and to resolve the competition of the clock and the stretch signal, an ME is inserted to the inverter chain of the ring oscillator. Figure 3.14 shows the structure of ME. This element serves the request signals on a first-come-first-served basis. Only the first coming request signal can invoke the corresponding grant. If the two signals arrive concurrently, the ME selects one to pass arbitrarily. The two grant signals are guaranteed to be mutually exclusive. In the operation of the plausible clock generator, the ME is normally transparent to the ring oscillator. Once the stretch signal wins the control, the ME blocks the inverter chain and lowers the clock signal. At the same time, the stopped signal is asserted. The ring oscillator can be recovered unless the stretch signal is released.

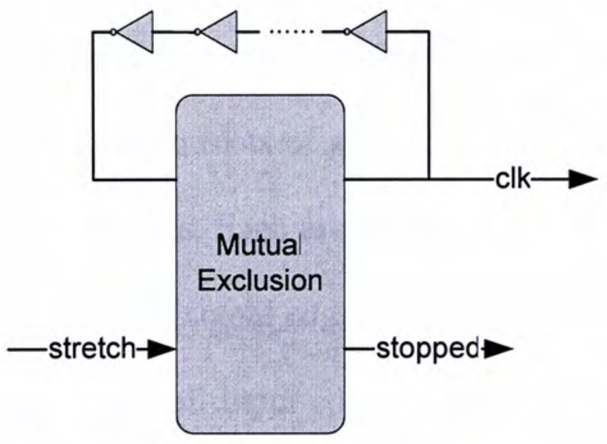


Figure 3.13: Pausible Clock Generator

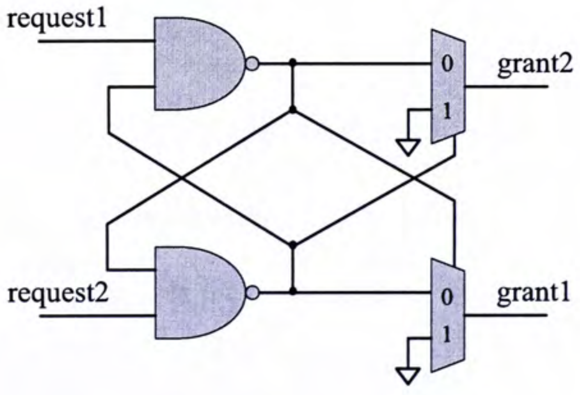


Figure 3.14: Mutual Exclusion

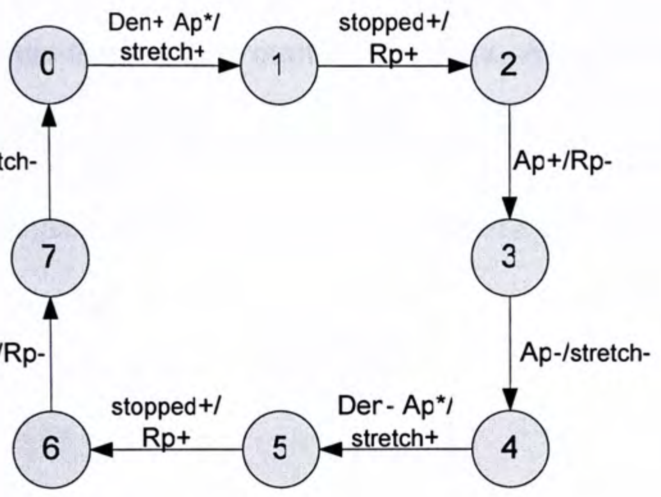
3.4.3. Port Controllers

Another component in asynchronous wrapper is port controllers. The function of data port controllers is to handle the handshake protocol and to control the local pausible clock. As the port controllers operate in the absence of the clock, they are designed as asynchronous finite state machines (AFSMs). Unlike synchronous finite state machines (FSM), an AFSM has the potential problem of output hazard for multiple input changes. To solve the hazard problem, our controllers are captured by the extended burst-mode specification [38]. This kind of AFSM can be triggered by input bursts – transition signalling, therefore, signals from the synchronous module can trigger the port controller in every cycle.

The extended burst-mode specifications of our port controllers are depicted by Figure 3.15 and Figure 3.16. An extended burst-mode asynchronous finite state machine is specified by a state diagram which consists of a finite number of states, and a set of directed arcs connecting pairs of states. Each arc indicates the transition between two states and is labelled with two sets of signal edges comprising the input burst and the output burst. In a given state, when all input edges appear, the machine generates a set of output changes and moves to a new state.

For the output port controller, *Den* is the enable signal for the start of handshake. A transition of *Den* from low to high triggers the AFSM to enter state 1 from state 0 and lift *sketch* to high. Then the AFSM moves to state 2 when there is a positive transition of *stopped*. Walking through the AFSM state by state, the handshake sequence is accomplished in state 4. The AFSM waits for the negative transition of *Den* this time. The rest of the states repeat works of state1 to 3.

To implement port controller, the specifications are translated to 3D machines and are synthesized with the method mentioned in [38]. A detailed description of the synthesis can be found in appendix A. The results of the synthesis are available in Figure 3.15 and Figure 3.16.

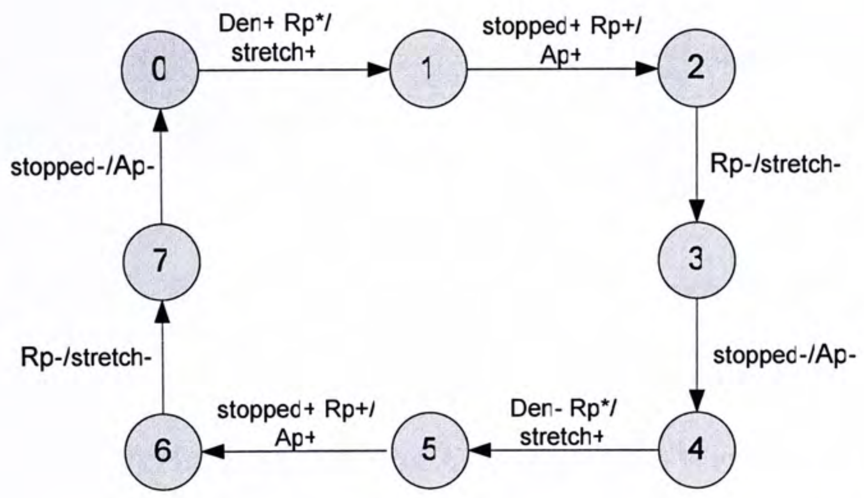


$$\text{stretch} = \overline{Z0}*Der + R*Ap + Z0*\overline{Der}$$

$$Rp = Rp*Ai*\overline{Ap} + \overline{Z0}*Ri*\overline{Rp}*Der*Ai + Z0*Ri*\overline{Rp}*\overline{Der}*Ai$$

$$Z0 = Z0*Der + Z0*\overline{Ap} + Z0*\overline{Rp} + Rp*Der*Ap$$

Figure 3.15: The extended burst-mode specification and the logical implementation of the output port controller



$$\text{stretch} = Rp*\text{stretch} + \overline{Der}*Z0 + Der*\overline{Ap}*\overline{Z0}$$

$$Ap = Rp*\text{stopped} + Ap*\text{stopped}$$

$$Z0 = \overline{Rp}*sZ0 + \overline{Ai}*Z0 + Der*\overline{Rp}*Ap$$

Figure 3.16: The extended burst-mode specification and the logical implementation of the input port controller

A one-way asynchronous communication channel between two modules is configured as in Figure 3.17. The input port controller and output port controller play different roles in the data transfer mechanism. Output port controller is the one to establish the communication channel. When the sender module activates the output port controller with *Den+*, the local clock is stretched. An event of *Rp+* is sent to the receiver module immediately after the clock is stopped. If the input port controller is already activated, it stretches the clock signal in the presence of *Rp+*, and replies with an *Ap+* as soon as the local clock is stopped. Simultaneously, the Ap+ commands the latch to capture the data. After detecting the *Ap+*, the *Rp+* is released to *Rp-*. The receiver feedbacks with an *Ap-* and recovers the local clock to sample the data captured in the latch. And the sender can recover its clock eventually.

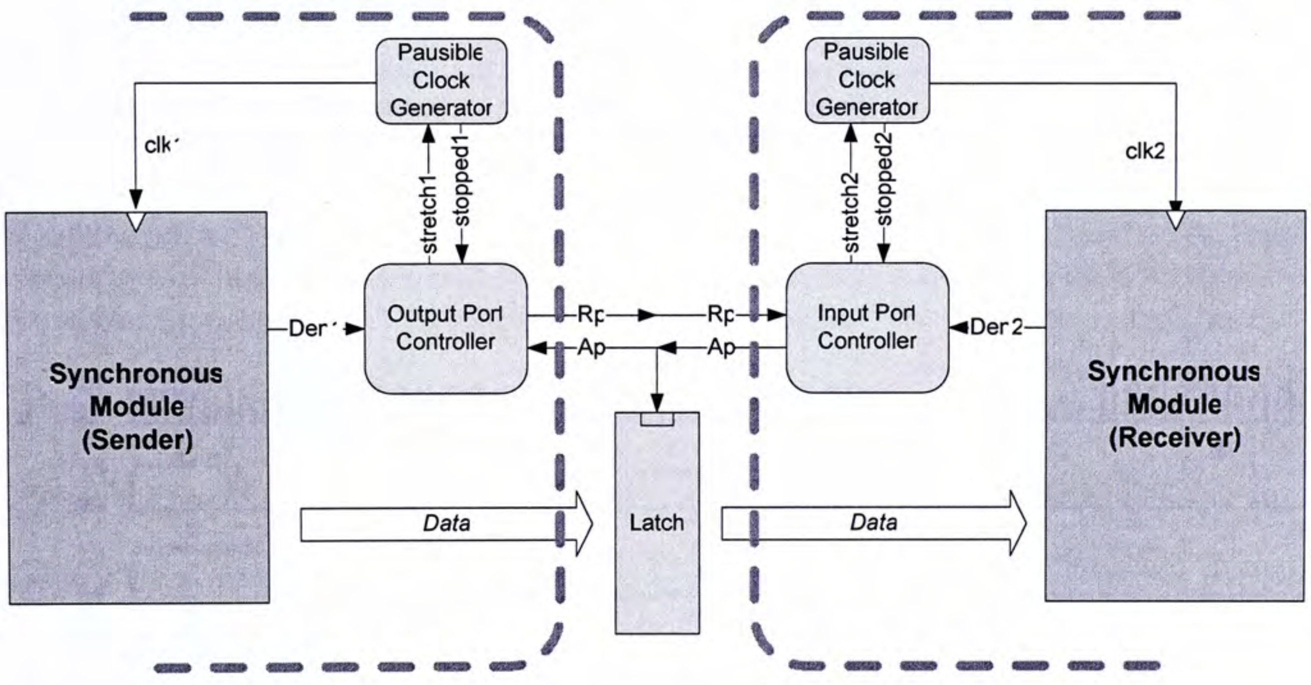


Figure 3.17: The configuration of an asynchronous communication interface

3.4.4. Performance of the Asynchronous Wrapper

To evaluate our design, spice level simulations are performed using AMS 0.35um CMOS technology. The unit under simulation is configured as in Figure 3.17. The sender module runs with nominal frequency of 540MHz and the receiver with 680MHz. The waveform in Figure 3.18 shows the simulated behaviour of the communication channel. From the waveform, the transfer of data takes about 3.1ns on average. This value is the time difference between the first positive clock edge that initiates the data transfer and the sampling clock edge of the receiver, which is defined as the communication overhead of the channel.

There are also some simulations of the mutual exclusion element and the pausible clock generator. The simulation results are summarized in Table 3.1.

Table 3.1: Simulation results of the asynchronous wrapper

Component	Parameter	Value
Asynchronous Interface	Communication Overhead	*3.1ns*
Mutual Exclusion	Latency	*0.45ns*
	Response Time	*0.15ns*
Pausible Clock Generator	Maximum Clock Frequency	*1.7GHz*

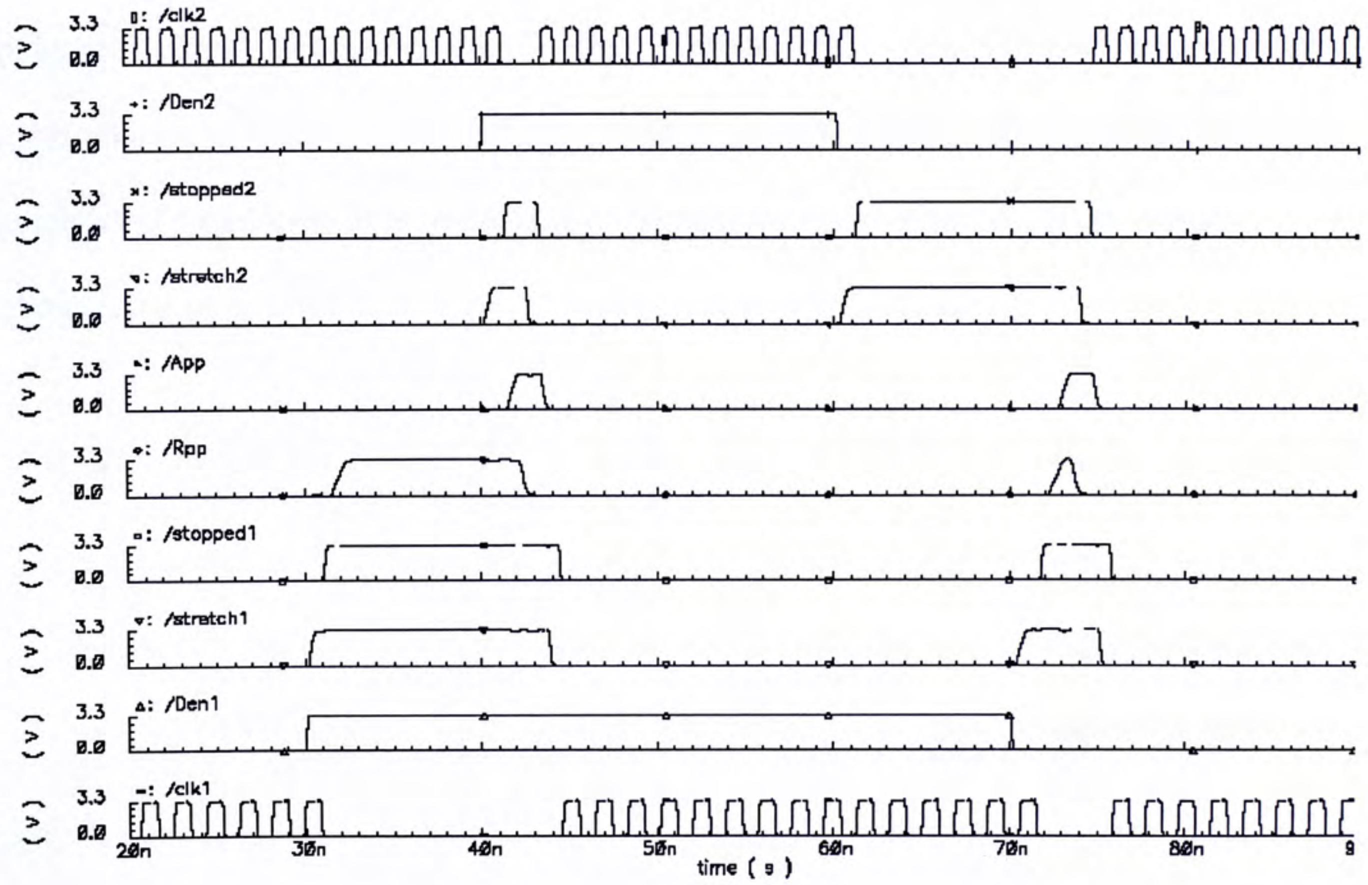


Figure 3.18: The simulated waveform of the communication channel

3.5. Summary

This chapter describes three asynchronous design styles: micropipelines, fine-grain pipelining and GALS design. The GALS design style has been chosen for our platform for three reasons: 1) It is heterogeneous and supports the mixture of synchronous and asynchronous IPs; 2) It widens the design space by allowing the designers to fine-tune the voltage and the frequency of each module. 3) It has a synthesis-based design flow.

The design of the asynchronous wrapper based on the work of [37] has been presented at the end. We have designed a set of fully synthesizable components – pausible clock generator, input- and output- port controllers. The design methodology of each component and their behaviour have been discussed. From the spice-level simulation, it is found that the communication overhead of our wrapper is about 3.1ns on average.

4. PLATFORM BASED ASIP DESIGN METHODOLOGY

4.1. Platform Based Approach

The asynchronous design style is excellent for system integration. Local handshake interfaces allow seamless communication of modules with heterogeneous timing. Modules can be put together in an ad-hoc manner on the ground of sharing common handshake protocol. In ASIP design, it is obvious that some architectural parameters, especially the datapath, have to be changed iteratively in the optimization cycle. To take full advantage of the asynchronous design style, the target processor can be realized by adding modules to expand and customize the functionality of the base processor. This approach has several advantages:

1. The complexity of optimization, software generation can be lowered.
2. Accurate application profiling is possible.
3. The real-time performance of the whole system can be evaluated at an early

stage.

4. The base processor can be reused from design to design.
5. The design cycle can be shortened substantially.
6. The target processor is capable of being modified/upgraded in order to keep track of the evolving application needs.

The effectiveness and efficiency of this approach largely depends on the base processor. The design methodology used in this research is also based on a base processor. We call the used base processor a 'platform'.

4.1.1. The Definition of Our Platform

Our platform is a base processor environment that provides sufficient facilities for developing the target processor. It is a semi-finished product with general functions for the target application domain. It provides maximum freedom for application-specific customization. For this research, the platform is an extensible architecture that targets on embedded DSP applications. It supports rapid assembly and modification among synchronous and asynchronous modules. The target processor can be built on top of it.

4.1.2. The Definition of the Platform Based Design

The platform based design is a design methodology based on the foundation provided by the platform. Design begins in the middle of the whole process. The design philosophy is to scale up the datapath of the platform and to customize its architectural parameters to meet the real time requirements of the target application.

It is a straightforward way to design a complex system.

4.2. Platform Architecture

The platform is the centre of our design methodology. Its characteristics outline the functionality and the performance of the target processor. The quality of the platform is determined by three factors: 1) the design space of its datapath; 2) the customization options; 3) the coverage of target application domain. These three factors decide how much performance can be improved in the optimization and how tight the functionality can be coupled to the target application.

In this section, we derive the architecture of our platform from the nature of DSP algorithms. By investigating the DSP applications, the maximum desired design space of the datapath and the elements needed for performance enhancement can be identified. The extensible architecture of the platform is tailored for carrying these features in order to give the largest room for optimization.

4.2.1. The Nature of DSP Algorithms

A generic DSP system, as shown in Figure 4.1, consists of one or more input signals being processed by a digital circuitry to produce an output with the desired characteristics. The characteristics of the system can be described by mathematical models which is the transfer function *H(z)* in the z-transform domain. Although a complete system can perform very complex functions, the majority of signal processing operation can be broken down into a combination of the primitive mathematical operations listed in Table 4.1.

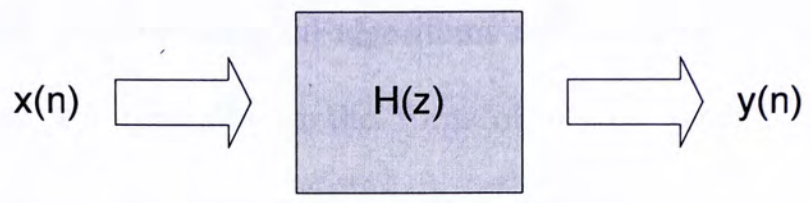


Figure 4.1: A digital signal processing system

Table 4.1: DSP primitive mathematical operations

Finite Impulse Response (FIR) Filter	$y(n)=\sum_{k=0}^{N-1} a_k x(n-k)$
Infinite Impulse Response (IIR) All-Pole Filter	$y(n)=\sum_{k=1}^{N-1} a_k y(n-k)+x(n)$
General Filter	$y(n)=\sum_{k=0}^{N-1} a_k x(n-k)+\sum_{l=1}^{M-1} b_l y(n-l)$
Cross-Correlation	$C_{xy}(m)=\frac{1}{N}\sum_{n=0}^{N-1} x(n)y(n+m)$
Discrete Fourier Transform	$X(n)=\sum_{k=0}^{N-1} x(n)e^{-j2\pi k\frac{n}{N}}$
Autocorrelation	$C_{xy}(m)=\frac{1}{N}\sum_{i=0}^{N-1-m} x(i)x(i+m)$

DSP has been widely used in many areas, such as speech synthesis and recognition, computer vision systems, control systems and digital communications. Many different kinds of algorithms have been devised for different applications. But most of the algorithms share some common characteristics, providing us with priori knowledge to make use of. We outline those that are closely related to the performance of a DSP application.

Computation-Intensive Kernels

Kernels are pieces of computational algorithms that make up the heart of the DSP application. They are typically in the form of nested short loops that involve intensive computation. The kernels often occupy the largest share of the computation power, thus affecting the peak performance of the application.

Strong Data Locality

DSP applications tend to access data in a relatively small block of memory. Large displacement of data address is infrequent. On the other hand, the data access pattern is regular. For filters, correlations and most of the matrix operations, the data is accessed in circular addressing with constant steps (Figure 4.2a). More sophisticated bit-reversed addition pattern (Figure 4.2b) is used for butterfly-like transformations

K	address
0	0
1	3
2	6
3	9
4	12
5	15
6	2
7	5
8	8

(a)

K	address	
0	000	= 0
1	100	= 4
2	010	= 2
3	110	= 6
4	001	= 1
5	101	= 5
6	011	= 3
7	111	= 7
8	000	= 0

Add 1 to MSB in each iteration

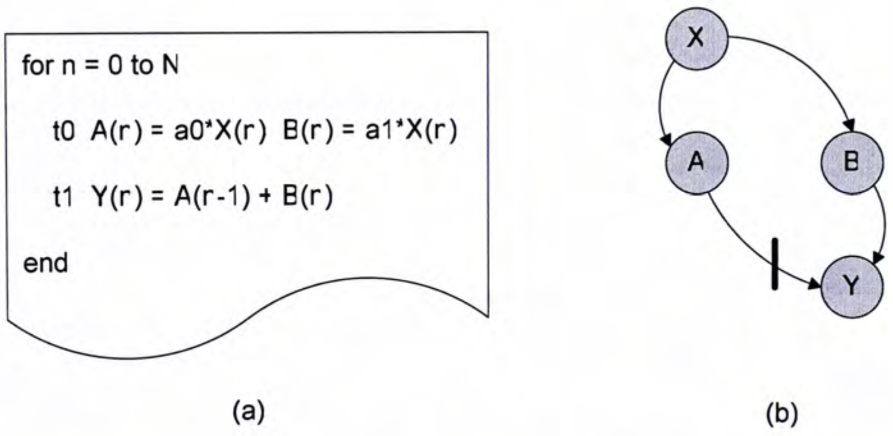
The carry propagates from MSB to LSB

(b)

Figure 4.2: (a) Circular addressing of size 16 and step 3 at iteration K;
(b) Bit-reversed addressing of 3 bits in size at iteration K

Explicit and Implicit Instruction Level Parallelism

DSP applications possess a high degree of instruction level parallelism (ILP), especially in the kernels. For simplicity without losing generality, parallelism exploration of a two-tap FIR filter is demonstrated as an example. The mathematical formulation of a two-tap FIR filter is $y(n) = \sum_{k=0}^{1} a_k x(n-k)$ and the algorithm and the corresponding data flow graph (DFG) are depicted by Figure 4.3. From the algorithm, it can be found that the two multiplications can operate in parallel. This is the explicit parallelism of this algorithm. However, the parallelism is not limited to this level. There is implicit parallelism that can be explored by doing transformation on the algorithm.



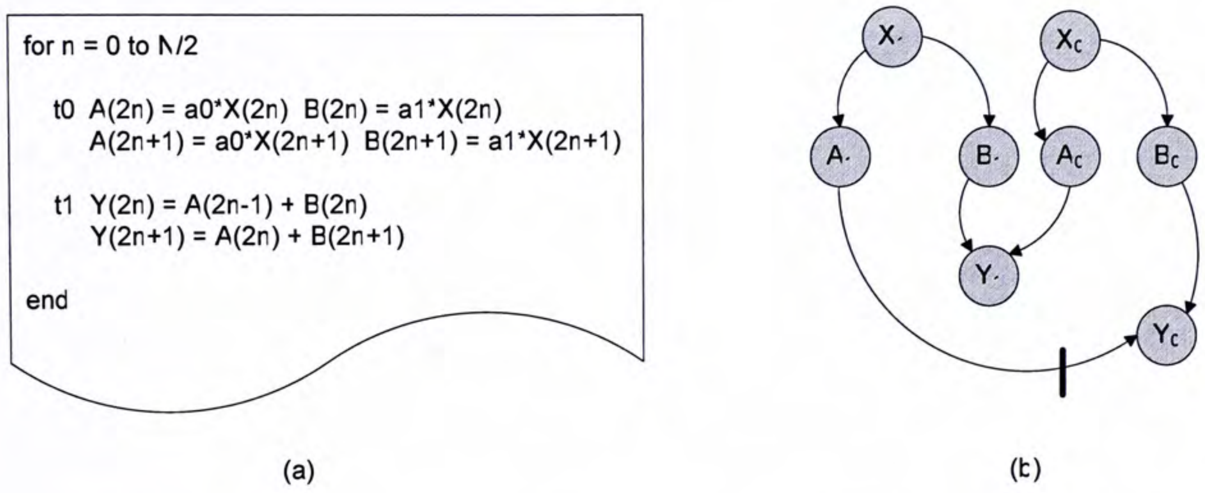
```
for n = 0 to N
    t0  A(r) = a0*X(r)  B(r) = a1*X(r)
    t1  Y(r) = A(r-1) + B(r)
end
```

Figure 4.3: (a) The algorithm of a two-tap FIR filter; (b) The corresponding DFG. The vertical bar represents a delay element

One of the transformation techniques is unfolding. The summation series of the two-tap FIR can be split into two summation series:

$$y(n) = \sum_{k=0}^{1} a_k x(n-k) = \begin{cases} y(2n) = a_0 x(2n) + a_1 x(2n-1) \\ y(2n+1) = a_0 x(2n+1) + a_1 x(2n) \end{cases}$$

Each of them is represented by a new DFG. Due to the dependency of the two summation series, the two DFG can be merged as in Figure 4.4b. From this new complete DFG, there are four parallel operations discovered (A_0, A_1, B_0, B_1) which are double the original one. The unfolded algorithm is shown in Figure 4.4a. This demonstration only unfolds the algorithm to two levels. For a higher degree of parallelism, the algorithm can be further unfolded.



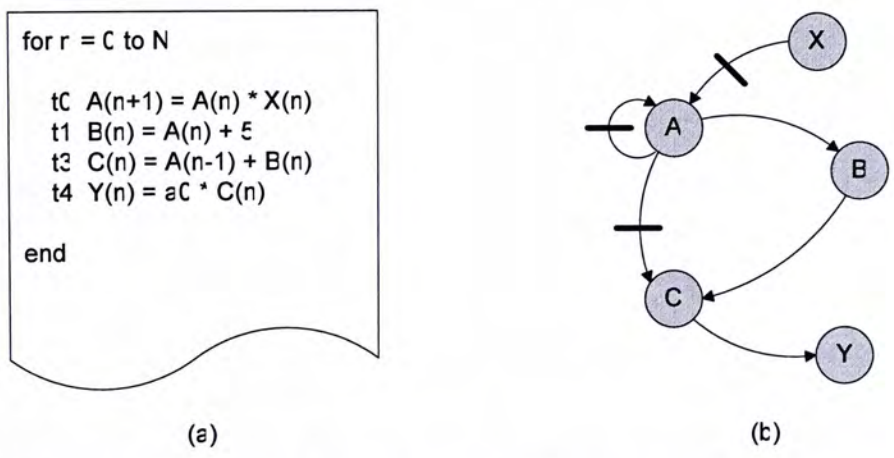
```
for n = 0 to N/2

   t0  A(2n) = a0*X(2n)  B(2n) = a1*X(2n)
       A(2n+1) = a0*X(2n+1)  B(2n+1) = a1*X(2n+1)

   t1  Y(2n) = A(2n-1) + B(2n)
       Y(2n+1) = A(2n) + B(2n+1)

end
```

Figure 4.4: (a) The unfolded algorithm of a two-tap FIR filter; (b) The corresponding DFG



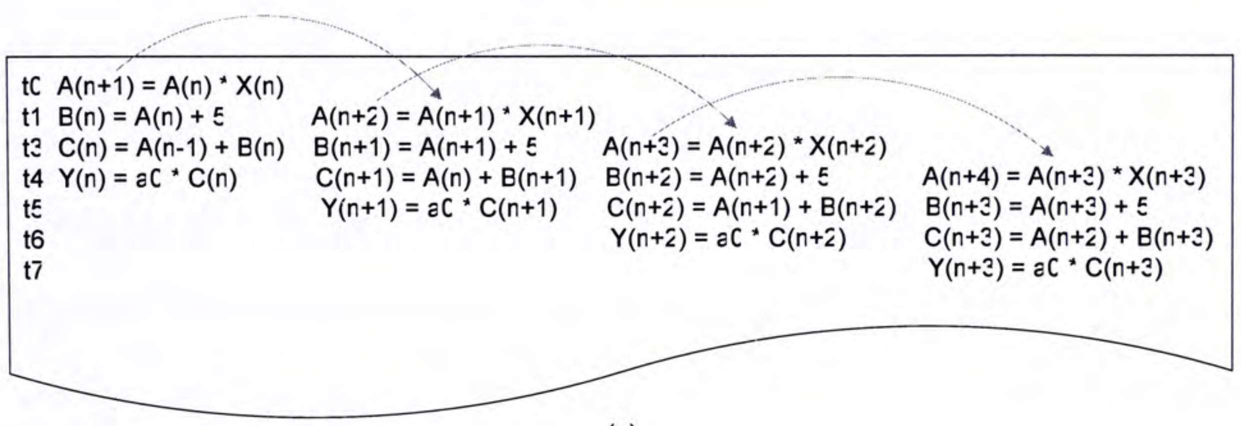
```
for n = 0 to N

   t0  A(n+1) = A(n) * X(n)
   t1  B(n) = A(n) + 5
   t3  C(n) = A(n-1) + B(n)
   t4  Y(n) = a0 * C(n)

end
```

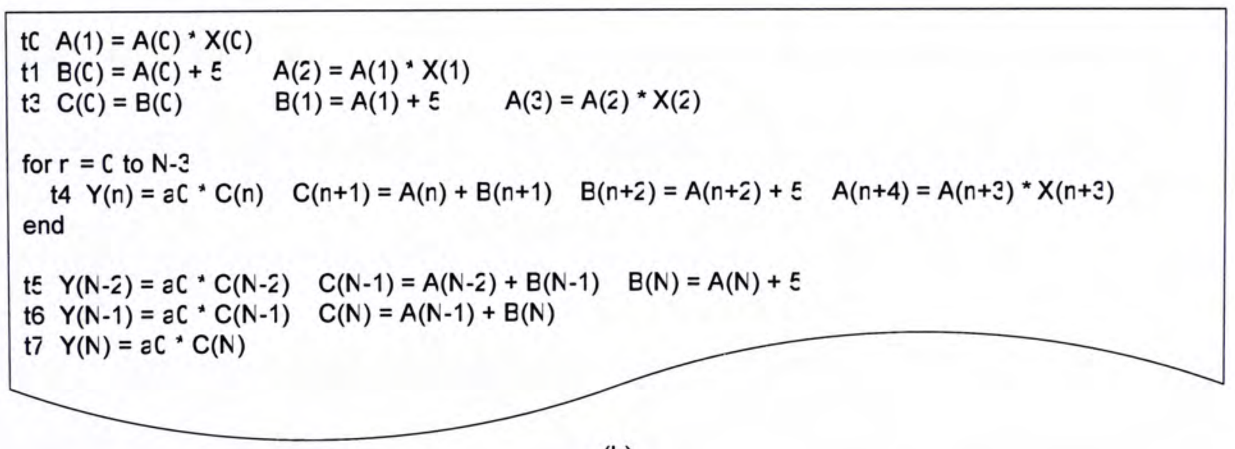
Figure 4.5: (a) The arbitrary algorithm with heavy data dependence; (b) The corresponding DFG

Some algorithms carry heavy data dependence in which some synchronization is necessary between operations of various iterations. An arbitrary algorithm of this kind is shown in Figure 4.5a and its DFG in Figure 4.5b as an example. The data dependence between one iteration and the successive iteration prohibits parallelism as that in the two-tap FIR filter. Another transformation technique, software pipelining can unveil some potential parallelism in such cases. The idea behind software pipelining is that the body of a loop can be reformed so that one iteration of the loop can start before previous iterations finish executing. The reformed body of the algorithm is shown in Figure 4.6a. The arrows indicate the data dependence of two operations. Figure 4.6b shows the result of the transformation. The maximum level of parallelism gains to four.



```
t0 A(n+1) = A(n) * X(n)
t1 B(n) = A(n) + 5          A(n+2) = A(n+1) * X(n+1)
t2 C(n) = A(n-1) + B(n)     B(n+1) = A(n+1) + 5        A(n+3) = A(n+2) * X(n+2)
t4 Y(n) = a0 * C(n)         C(n+1) = A(n) + B(n+1)     B(n+2) = A(n+2) + 5        A(n+4) = A(n+3) * X(n+3)
t5                          Y(n+1) = a0 * C(n+1)       C(n+2) = A(n+1) + B(n+2)   B(n+3) = A(n+3) + 5
t6                                                     Y(n+2) = a0 * C(n+2)       C(n+3) = A(n+2) + B(n+3)
t7                                                                                Y(n+3) = a0 * C(n+3)
```

(a)



```
t0 A(1) = A(0) * X(0)
t1 B(0) = A(0) + 5       A(2) = A(1) * X(1)
t2 C(0) = B(0)           B(1) = A(1) + 5        A(3) = A(2) * X(2)

for n = 0 to N-3
   t4 Y(n) = a0 * C(n)   C(n+1) = A(n) + B(n+1)   B(n+2) = A(n+2) + 5   A(n+4) = A(n+3) * X(n+3)
end

t5 Y(N-2) = a0 * C(N-2)   C(N-1) = A(N-2) + B(N-1)   B(N) = A(N) + 5
t6 Y(N-1) = a0 * C(N-1)   C(N) = A(N-1) + B(N)
t7 Y(N) = a0 * C(N)
```

(b)

Figure 4.6: (a) The reformed body of the arbitrary algorithm (b) The software pipelined arbitrary algorithm

Data Parallelism

In some DSP applications, the data sets have an extremely high degree of internal parallelism, which means that all the elements of the data sets can be processed simultaneously. Matrix is an example. Matrix itself is a large data set that consists of an array of data. There is no crucial relationship among the data. Only the spatial relationship of each data is important. Computer vision and image processing use this kind of data representation. Their computation is on images consisting of a large array of pixels.

Because of the weak interdependence, each data or together with its close neighbours can be mapped to an individual operation. The whole data set can therefore be manipulated by a large number of individual operations that can be executed simultaneously. This is illustrated by the edge detection of image processing. In Figure 4.7 the input data set is an eight by eight grey scale image. Each data element represents the light intensity. To do edge detection, a Sobel edge detector is applied to each element to produce a weight that links to the possibility of the presence of an edge. The Sobel edge detector is also shown in Figure 4.7 in a matrix form. The mathematical formulation of this operation is $O_{n,m} = \sum_{k=0}^{2}\sum_{l=0}^{2} I_{n+k-1,m+l-1} S_{k,l}$ for $I_{n,m}, S_{n,m}, O_{n,m}$ are the elements of input matrix, Sobel edge detector and output matrix respectively in row n and column m. As there is no linkage between any two inputs of an edge detector, it is possible to apply maximum sixty-four edge detectors on the image simultaneously. From a macro view, once a parallel data entity is defined, it can be processed as an individual.

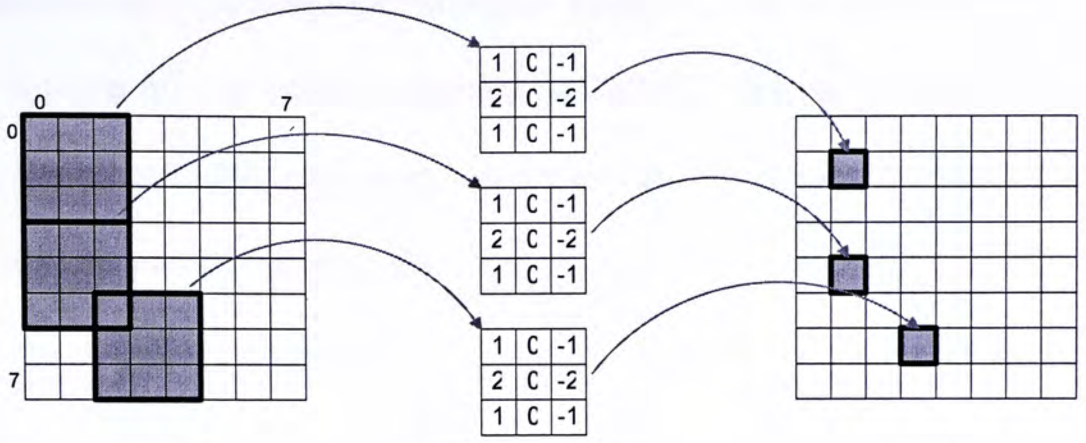
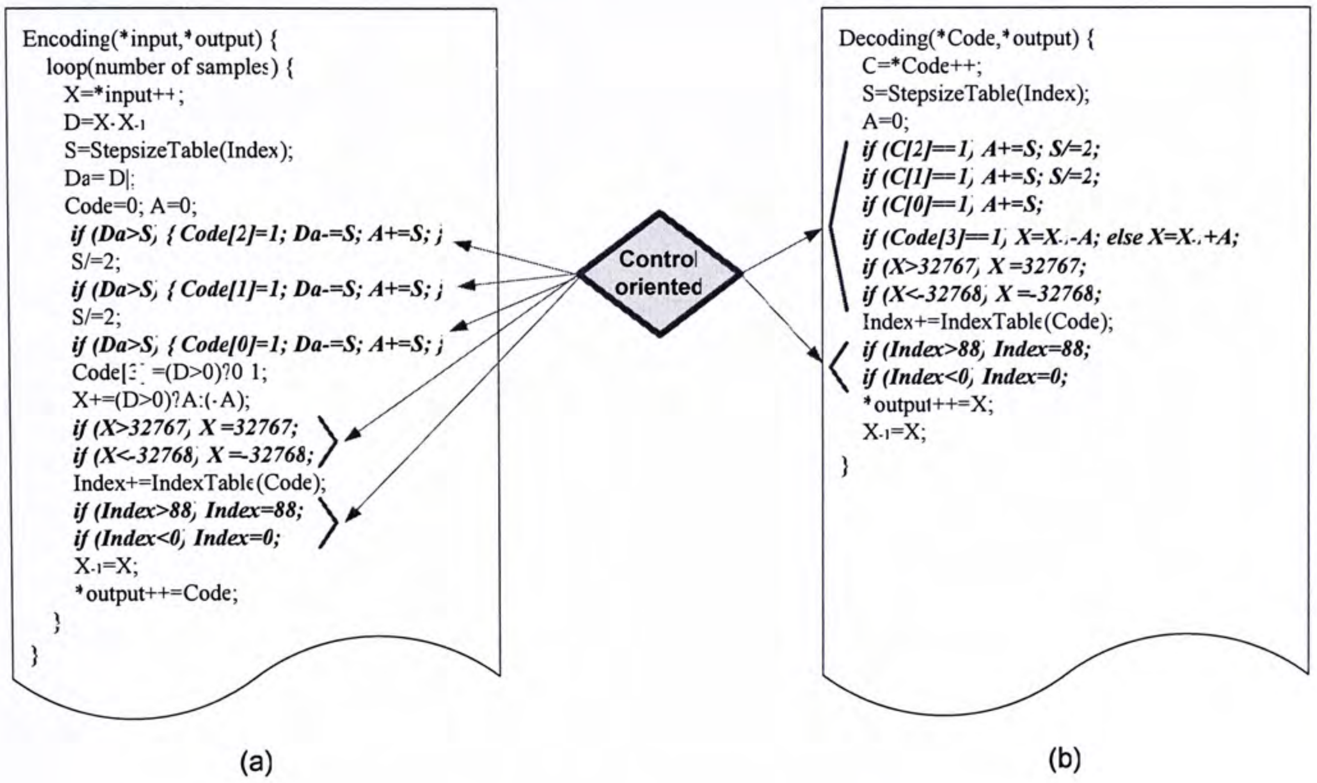


Figure 4.7: Applying Sobel edge detectors on an 8x8 image

Mixed Control and Data Dominated

Some DSP applications do not have the high degree of parallelism as aforementioned. The algorithms are a mixture of data operations and control operations. The control operations manage the flow of the algorithms and govern their runtime behaviour dynamically. The sequence of data operations cannot be known until the related control operations have made the decision. This kind of dependence restricts the exploration of parallelism across control operations. If the DSP application has a large portion of control operations, its parallelism cannot be too high. Adaptive differential pulse code modulation (ADPCM), coordinate rotation digital computer (CORDIC) and Viterbi decoding belong to this type.

The pseudocode of ADPCM is shown in Figure 4.8. The lines highlighted in bold and italics are control operations. In the encoding algorithm, the control operations separate the data operations into pieces. In the first highlighted line, there are three data operations under the influence of a control operation. This control operation depends on the value of *Da* which varies with runtime. Once the condition of this control operation is fulfilled, the values of *Code[2]*, *Da* and *A* are modified. This presents a potential dependency among *Da*, *Code[2]* and *A*. Similar situations also occur in the other highlighted lines. As the control operations introduce more data dependency, the parallelism is limited.



```
Encoding(*input,*output) {
  loop(number of samples) {
    X=*input++;
    D=X-X-1
    S=StepsizeTable(Index);
    Da=|D|;
    Code=0; A=0;
     if (Da>S) { Code[2]=1; Da-=S; A+=S; }
     S/=2;
     if (Da>S) { Code[1]=1; Da-=S; A+=S; }
     S/=2;
     if (Da>S) { Code[0]=1; Da-=S; A+=S; }
     Code[3]=(D>0)?0:1;
     X+=(D>0)?A:(-A);
     if (X>32767) X=32767;
     if (X<-32768) X=-32768;
     Index+=IndexTable(Code);
     if (Index>88) Index=88;
     if (Index<0) Index=0;
     X-1=X;
     *output++=Code;
  }
}
```

```
Decoding(*Code,*output) {
  C=*Code++;
  S=StepsizeTable(Index);
  A=0;
  if (C[2]==1) A+=S; S/=2;
  if (C[1]==1) A+=S; S/=2;
  if (C[0]==1) A+=S;
  if (Code[3]==1) X=X-1-A; else X=X-1+A;
  if (X>32767) X=32767;
  if (X<-32768) X=-32768;
  Index+=IndexTable(Code);
  if (Index>88) Index=88;
  if (Index<0) Index=0;
  *output++=X;
  X-1=X;
}
```

Figure 4.8: The pseudocode of ADPCM (a) encoding and (b) decoding (Source: [39])

4.2.2. Design Space of Datapath Optimization

According to the aforementioned nature of DSP applications, instruction level parallelism and data parallelism are the two directions for improving computational performance. Definitely, the room for performance improvement is related to the characteristics of the algorithms in use, but the hardware can also limit the degree of parallelism explored. To facilitate parallelism exploration, the processor must 1) have a parallel datapath with a sufficient number of functional units; 2) supply sufficient operands to the functional units at the same time. The layout of the desired datapath is shown in Figure 4.9. It is assumed that each functional unit takes at most two operands for computation, which is aligned to most primitive mathematical operations. The figure shows two configurations of the datapath. The upper one is a typical parallel datapath. The lower one is especially for an efficient implementation of software pipelining. The designers are responsible for deciding the number of parallel functional units and their content.

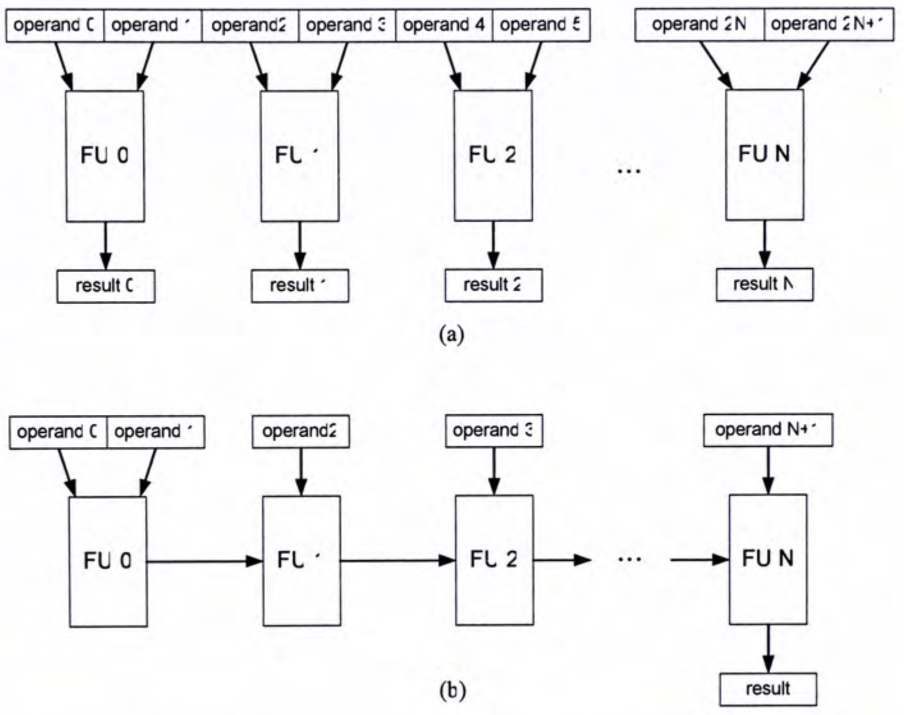


Figure 4.9: The layout of the parallel datapath (a) for ordinary parallel operations (b) for software pipelining

For the algorithms dominated by mixed control and data operations, exploiting parallelism cannot be effective for performance enhancement. The reason is that control operations establish extra dependency to the algorithm, limiting parallel executions. In this case, dedicated hardware can be used to accelerate the algorithm by resolving the dependency. Reviewing the encoding pseudocode of ADPCM in Figure 4.8a, we can find two control operations: *if(X>32767) X=32767*; and *if(X<-32768) X=-32768;* which are actually performing saturation arithmetic. By performing conditional decision in hardware, these operations can be reformulated to a sequential complex instruction. The simplified datapath is portrayed by Figure 4.10.

Introducing dedicate hardware to tackle sequential tasks is very effective because the dependency is resolved inside the hardware instead of across several instructions. However, designers have to take extra care in the timing of the hardware accelerator. Its critical path should be less than that of the processor, or it should be divided into multiple cycles. Otherwise the overall performance of the processor will deteriorate.

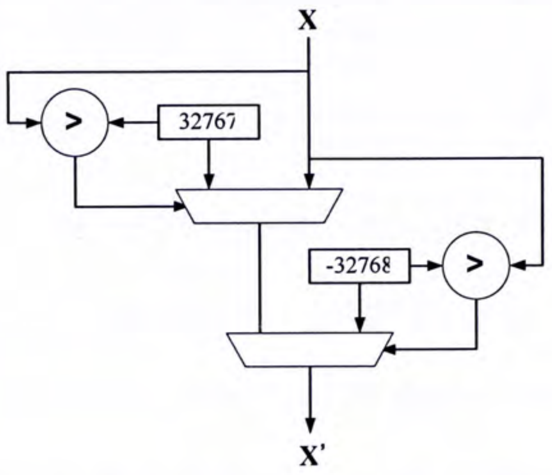


Figure 4.10: The datapath of saturation arithmetic

To sum up, the optimization of the datapath can be broken down into two aspects as depicted by Figure 4.11. The first aspect exploits the parallelism of the algorithm to enable simultaneous execution of multiple operations. Depending on the pattern of the involved data dependency, the datapath can be configured to parallel-data-parallel-output manner as in Figure 4.9a or software pipelining as in Figure 4.9b. The second aspect uses dedicated hardware accelerator which is tightly coupled to the datapath. The hardware accelerator performs multiple operations in sequence to realize a powerful complex instruction as in Figure 4.10. The length of the operation sequence has to be matched with the timing of the processor, therefore some accelerators have to work in multiple cycles.

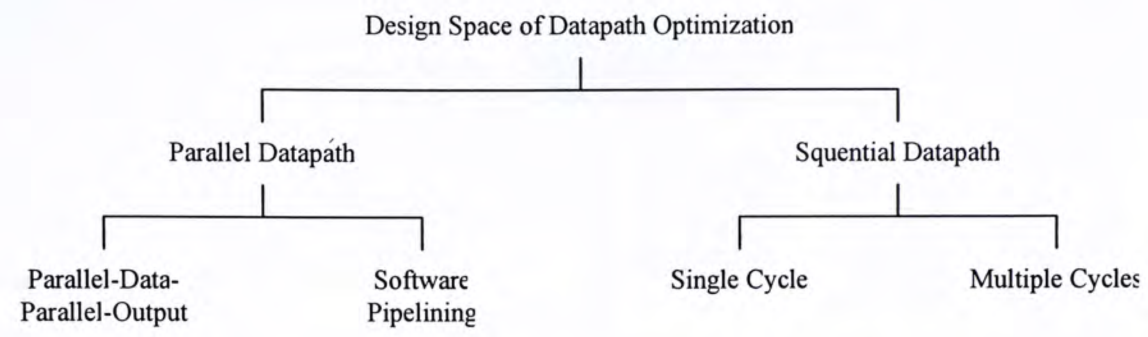


Figure 4.11: The design space of datapath optimization

4.2.3. Proposed Architecture

Our platform is a parameterized extensible processor, which supports aggressive optimization of the datapath. The proposed architecture is illustrated in Figure 4.12. There are four kinds of modules shown in the figure. The base modules are the essential parts of the platform. They sustain the basic functions, for example memory manipulation, flow control and basic data processing. The extensible modules are also a member of the foundation of the platform. In addition they are parameterized for customization. The optional modules are add-on for datapath optimization. They are the main engines for parallel and complex instructions. The modules shown in yellow are massive storage for the programme and the data. They can be read only memory (ROM) or random access memory (RAM) locating on or off the chip. A brief introduction to the functionality of the modules is given in the following paragraphs. After that, there is an in-depth discussion on the strategy of realizing an optimized datapath. The detailed description of the microarchitecture of the platform is presented in Chapter 5.

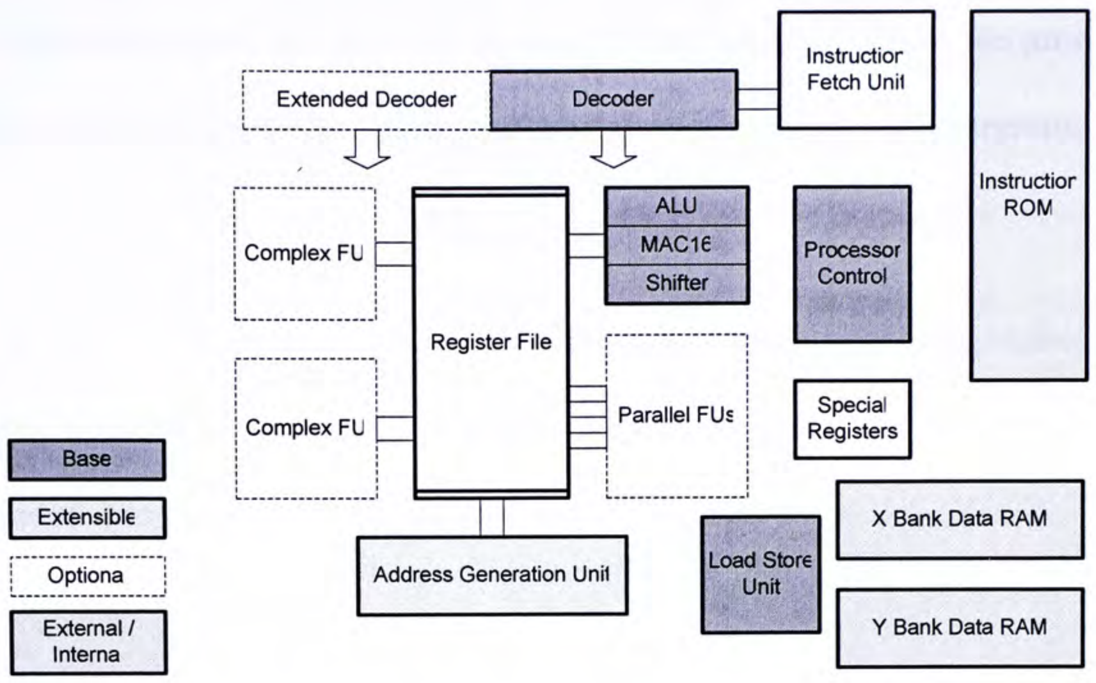


Figure 4.12: The architecture overview of the platform

The instruction fetch unit is used to access the programme stored in the instruction memory. It manages the programme counter to control the flow of the programme. For the kernel-like loops of DSP algorithms, a dedicated hardware is used to handle address displacement and loop count checking in order to alleviate the workload of the datapath for address calculation.

The function of the decoder is to interpret the fetched instructions and translate them into control signals and operands for other modules. It is also responsible for invoking the optional decoder and the datapath if they are available.

The processor control is used to control the branching and exception states of the processor. While a branch is taken or an exception is present, the processor control informs the instruction fetch unit to modify the content of the programme counter.

The special registers store the status flags and the configuration parameters for the address generation unit and the load store unit. The configuration of the processor is centralized and unified by modifying the special registers with one instruction.

The load store unit serves as an interface for the data memory and the register file. It is the only channel for data memory access. All data located in the data memory have to be loaded into the register file before being processed.

The address generation unit provides addresses for fetching operands from and storing results to the register file. It supports varies addressing mode in order to get use of the data locality of DSP algorithms.

The base datapath is designed for both data and control domain operations. There is arithmetic logic unit (ALU) for Boolean operations, bitwise manipulation and simple addition and subtraction. For data processing, a multiply accumulation unit (MAC) and a barrel shifter are available.

The register file is the source of the operands and the destination of the results for the datapath. It is designed to be multi-port in order to provide sufficient data bandwidth for the datapath.

4.2.4. The Strategy of Realizing an Optimized Datapath

Datapath optimization is the key to the customization of instruction set architecture of the platform. In the course of the optimization, application-specific functions are added to the datapath to accelerate the algorithms to meet the real-time requirement of the application. The optimized datapath can be specialized for parallel instructions and complex sequential instructions. Unlike simple basic instructions, these two aspects of instructions induce several implementation challenges. For parallel instructions, there are two major challenges: 1) supplying a sufficient number of instructions; 2) supplying a sufficient number of operands. For complex instructions, the major challenge is the coordination of timing of the hardware accelerators and the platform.

Supplying Instructions to the Parallel Datapath

The concept of exploiting instruction level parallelism is to shorten the execution time of the given task by executing multiple instructions at the same time. This is practical only if multiple instructions can arrive at the parallel datapath at the same time to command their operations. In the general propose processors domain, superscalar and VLIW are the two kinds of architectures designed for taking advantage of instruction level parallelism.

The instruction fetching mechanism of superscalar processors is similar to the processors without parallel datapath, which fetches instructions sequentially. To keep the parallel datapath busy, the fetched instructions are rescheduled and packed into parallel. The parallel instructions are dispatched to the functional units for parallel executions. VLIW architectures are a straightforward solution for the instruction fetching issue. The instructions of VLIW processors are extraordinary long in width, and can include all the commands for each functional unit in a single instruction. This method can be analogy, packing several short instructions in parallel to form a long instruction. Hence, all the functional units can be commanded at once for one instruction fetch.

However, these two solutions are prohibitive for embedded applications. The rationales are: 1) superscalar processors tend to have significantly complex hardware for exploring parallelism in runtime; 2) VLIW processors are extremely costly on instruction width -- they can be in the order of hundreds to thousands of bits. These drawbacks lead to unaffordable power consumption and silicon area.

For embedded DSP applications, we have another solution. According to the nature of DSP applications, some of the kernels are the critical path of the DSP application. The performance of the DSP application can be improved dramatically by accelerating the kernels which are a small portion of codes in the application. Therefore we presume that allowing a small number of predefined patterns of parallel instructions for the kernels can achieve significant performance improvement. Based on this presumption, a parallel instructions compression scheme is used. Some of the selected patterns of the parallel instructions are subjected to compression by means of using a lookup table. These patterns are reduced to the enumerated index of the lookup table, and hence can fit into a fixed width instruction. The compressed patterns are stored in the lookup table in the form of opcodes, so that no further decoding is needed.

The idea is shown in Figure 4.13. Once the instruction decoder receives a compressed instruction from the instruction fetch unit, it enables the instruction decompressor and passes the instruction to it. The compressed instructions contain the index of the patterns of the parallel instructions. The decompressor fetches the corresponding parallel instructions according to the index and dispatches them to the datapath. Since the compressed instructions are independent of the number of parallel instructions, we can expect minimal impact when scaling up the parallelism of the datapath.

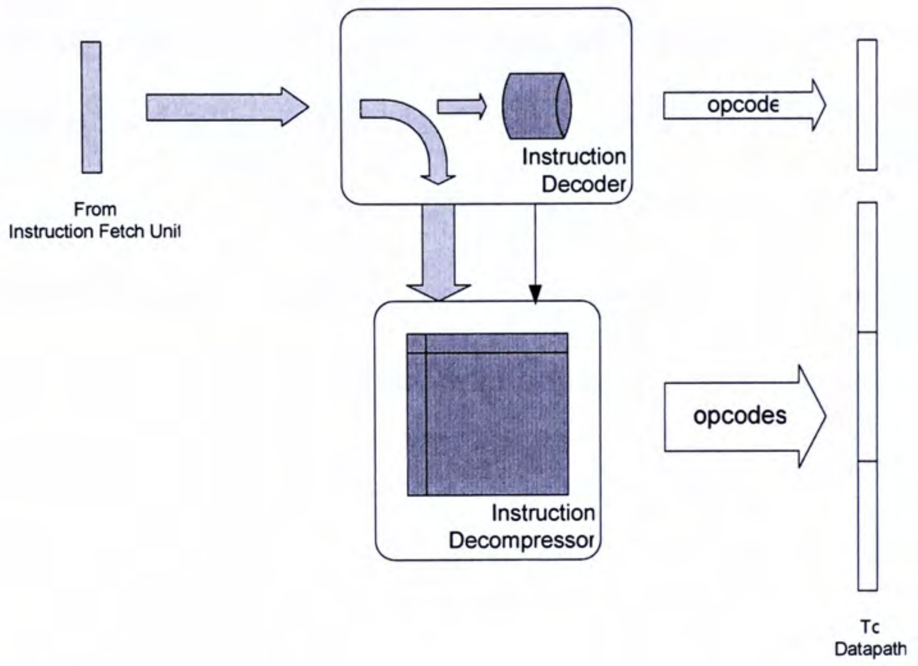


Figure 4.13: The idea of instruction decompression

Supplying Operands to the Parallel Datapath

For the similar argument of supplying parallel instructions, a smooth supply of operands is also crucial for parallel processing. Multiple operands have to be sent to the datapath simultaneously in order to execute the parallel datapath. This challenge is similar to that of supplying instructions to datapath, but the solution used for parallel instructions is not suitable for operands. Unlike the patterns of parallel instructions, the address of operands is not confined to a small space. Encoding all the possible addresses needs a lookup table with enormous entries. Consequently the compression rate cannot be high, but great silicon overhead is incurred due to the large lookup table.

As the addresses of operands are too difficult to encode, the solution makes use of the strong data locality of DSP applications. The kernels of DSP applications typically have regular operand access patterns. These patterns are classified into different addressing modes. Automation of operand fetch is feasible when the addressing mode is known. To exploit this, each parallel functional unit has a hardware engine for fetching operands. These hardware engines are configurable by instructions. After the configuration, the engines fetch operands automatically according to the selected addressing mode. Therefore, it is not necessary to include the addresses of the operands in the instructions. Only extra instructions are required for configuring the operand fetch engines. As the number of operands required is decoupled from the width of the instructions, it is again considered as having minimal impact when scaling up the parallelism of the datapath, especially on the space of the instruction encoding.

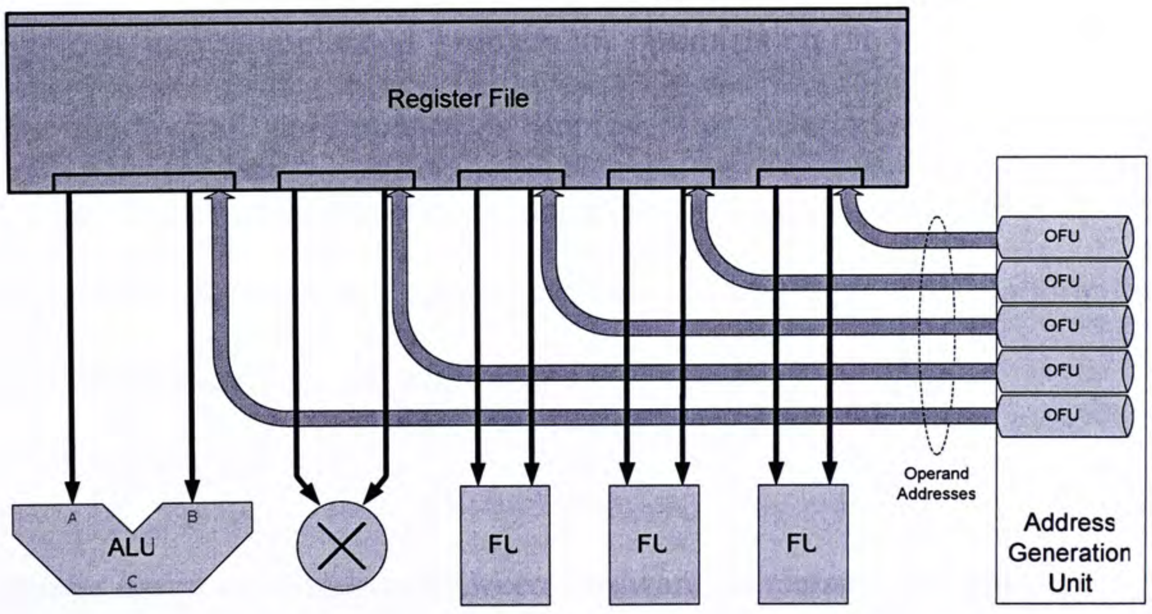


Figure 4.14: Operand fetch units (OFU) are part of the address generation unit. They are used to fetch operands for the functional units.

Coordinating the Timing of the Hardware Accelerators and the Platform

The timing of the hardware accelerators has to be carefully designed – otherwise the overall performance of the DSP application will be adversely affected. When introducing a hardware accelerator to the datapath of the platform, there are three possible situations. 1) The execution period of the hardware accelerator within the timing budget for the datapath. The hardware accelerator can be synchronized with the platform safely without affecting the overall timing. 2) The execution period of the hardware accelerator is slightly over the timing budget. In this case, the designers have to decide whether to operate the platform at a lower frequency or to break the hardware accelerator into multiple cycles. 3) The execution period of the hardware accelerator is several times that of the timing budget. The hardware accelerator has to operate in multiple cycles.

Lowering the operating frequency of the platform and breaking the accelerator into multiple cycles are at different costs. The former one affects the overall timing leading to a very complicated problem in optimization. It is hard to determine whether the overall performance is improved or deteriorated after adding the accelerator. The latter one needs a sophisticated processor controller to handle the synchronization between the accelerator and the platform. The platform has to reserve more resources for the accelerators.

To achieve easier coordination between hardware accelerator and the platform, the GALS design style is used. Using asynchronous communication, the timing of the accelerator can affect the platform only in synchronization. Both accelerators and platform can run at its maximum speed without adverse effect on the overall timing. For multiple cycle complex instruction, GALS handshake serves as a local controller. It is unnecessary to have a dedicated processor controller for synchronization. This is significantly worthwhile for the algorithm because its execution time depends on the input data. In such a case, the accelerator can have average case instead of worst case performance. On the other hand, the processor only needs to treat it as a single cycle instruction. No additional polling mechanism is needed to detect the completion.

<u>Cost Function of Using Application Specific Instructions</u>

The presented strategy obviously allows for effective and efficient implementation of the optimized datapath, but inevitably some means of costs are also established. Therefore, we work out the cost functions of using parallel instructions and complex instructions in order to provide a merit for optimization.

First of all, the gain and the overhead of using an application specific instruction are defined as follow. The setup time is the effort used to setup the operation of a particular instruction. For example, setup the configuration registers for address generation.

Definition: $$Gain = \frac{\text{original execution time } (T_{original}) \text{ - optimized execution time } (T_{optimized})}{\text{original execution time } (T_{original})}$$

Definition: $$Overhead = \frac{\text{setup time } (T_{setup})}{\text{original execution time } (T_{original})}$$

The definition of the cost function is the ratio of the overhead of introducing new instructions to the gain. This function indicates whether the overhead or the gain has a dominating effect. If the cost function approaches zero, the gain outweighs the overhead and therefore this new instruction is apparently at no cost. The overhead cancels out the gain when the cost function is equal to one.

Definition: $Cost\ Function = \dfrac{Overhead}{Gain}$

Before formulating the cost function, some terms have to be specified.

T_{plat} : Operating period of the platform
T_{acc} : Operating period of the hardware accelerator
F_{plat} : Operating frequency of the platform
F_{acc} : Operating frequency of the hardware accelerator
T_{comm} : Time for handshake communication

N_{loop} : Number of iteration of a loop
N_{para} : Number of instructions executed in parallel
N_{cycle} : Number of cycle to complete the task
$I_{original}$: Number of executed instructions
I_{para} : Number of executed instructions involved when using parallel instructions
$I_{complex}$: Number of executed instructions involved when using complex instructions
I_{conf} : Number of instructions involved to configure an operand fetch unit

The cost function of using parallel instructions can be elaborated as follows:

$$\begin{aligned}
Cost\ Function &= \frac{Overhead}{Gain} \\
&= \frac{T_{setup}}{T_{original} - T_{optimized}} \\
&= \frac{I_{conf} N_{para} T_{plat}}{N_{loop} I_{original} T_{exec} - N_{loop} I_{para} T_{plat}} \\
&= \frac{I_{conf} N_{para}}{N_{loop} N_{para} - N_{loop}} \quad for\ I_{para} = 1,\ I_{original} = N_{para} \\
&= \frac{I_{conf}}{N_{loop}(1 - \frac{1}{N_{para}})}
\end{aligned}$$

As I_{conf} is a constant, fully exploring parallelism in long loops is most beneficial.

The cost function of using complex instructions can be elaborated as follows:

$$\begin{aligned}
Cost\ Function &= \frac{Overhead}{Gain} \\
&= \frac{T_{setup}}{T_{original} - T_{optimized}} \\
&= \frac{T_{comm}}{I_{original} T_{plat} - N_{cycle} I_{complex} T_{acc}} \\
&= \frac{T_{comm}}{\frac{I_{original}}{F_{plat}} - \frac{N_{cycle}}{F_{acc}}} \quad for\ I_{complex} = 1
\end{aligned}$$

T_{comm} and F_{plat} are the architectural parameters that are treated as constant in the optimization. The cost function indicates that including more instructions in the complex instruction is advantageous providing that the operating frequency and the number of cycles remain unchanged.

4.2.5. Pipeline Organization

The platform is a typical pipelined processor with five stages. The organization of the pipeline is illustrated in Figure 4.14. The first stage is instruction fetch (IF). In this stage, the instruction fetch unit provides instructions addressed to the instruction memory and fetches the corresponding instructions into the processor. Then the processor moves to decode stage (DEC). The fetched instruction is decoded into commands and operands. Meanwhile, the address generation unit calculates for operand address calculation. The third stage is read stage (RD). The major task is to read out the operands from the register file. At the same time, the load store unit also accesses the register file for preparing store operation. Some instruction decoding works related to datapath is finished in this stage. The fourth stage is execution stage (EX). All the data processing and Boolean manipulation tasks are performed there. The load store unit accesses the data memory in this stage. The last stage is writeback (WB). The processed data is written back to the register file. On the other hand, the load store unit puts the loaded data into the register file.

The normal flow of pipeline operation involves five stages. However, for some instructions that do not execute the datapath, the execution stage is skipped in order to lower the latency. In this case, the read and write stages are combined. The normal flow and the shortened flow are shown in Figure 4.16.

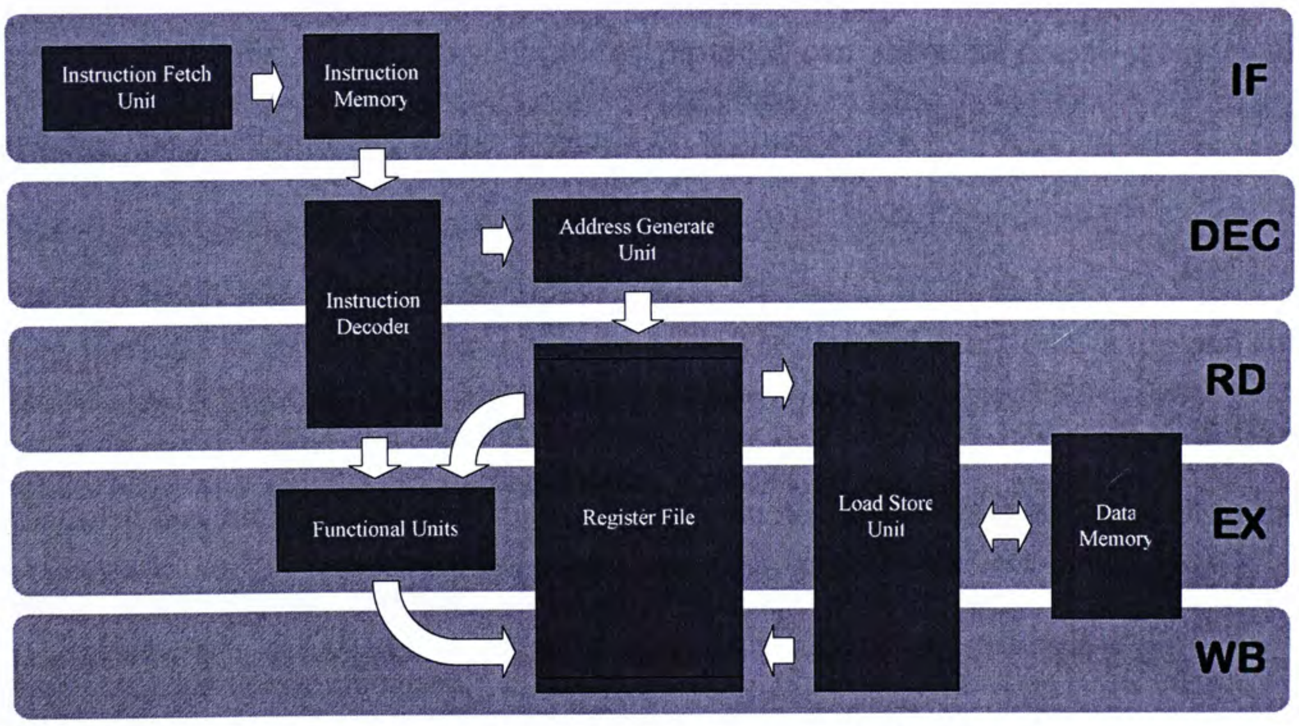


Figure 4.15: The pipeline organization of the platform

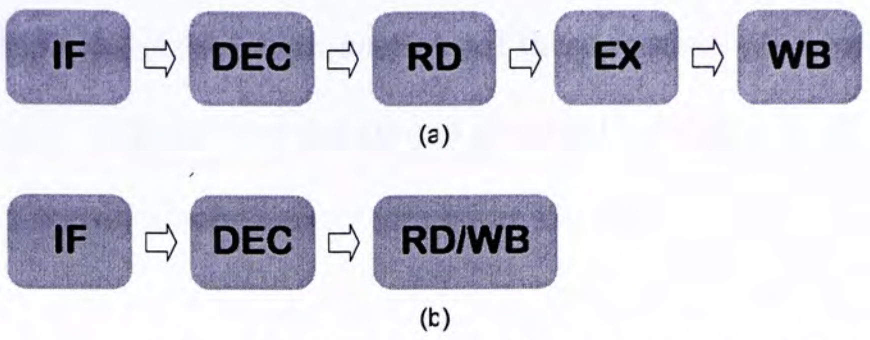


Figure 4.16: (a) The normal flow of pipeline operation; (b) The shortened flow with stage skipping.

4.2.6. GALS Partitioning

GALS design style is one of the key features of the platform. To design a GALS system, partitioning the system into synchronous modules is crucial. There are four considerations for partitioning the platform: 1) The synchronous modules should be large enough to hide the communication cost; 2) The modules that are subjected to change in the optimization should be modulized by asynchronous wrappers, so that the overall timing cannot be affected; 3) There should be some interfaces for using asynchronous IPs; 4) The asynchronous protocol can serve as coarse-grain clock gating. The asynchronous wrapper should be applied to power hungry modules.

The platform is partitioned as in Figure 4.17. The parallel functional units and the complex functional units are put into separated asynchronous wrappers. The base platform that includes the base modules and base functional units are grouped in the same synchronous module. All external components are interfaced with asynchronous communication.

The parallel functional units and the complex functional units are the key roles in the optimization. They change dramatically by means of structure and timing. Therefore, the best practice is to modulize them perfectly with asynchronous wrappers. In addition, due to the sophisticated algorithms of these modules, they are usually huge and power hungry. As these modules are not involved in other general instructions, it is better to wrap them asynchronously for power saving.

In contrast, the whole base platform is put into a single asynchronous wrapper. The reason is that there are only minor changes in the platform compared to the parallel and complex functional units. In addition, most of the instructions (all general instructions) can be accomplished within the base platform. Further decomposing the platform into finer partitions cannot take advantage of the timing encapsulation, but introduces unnecessary communication overhead.

This partitioning establishes asynchronous communications between optimized datapath (parallel and complex functional units) and the base platform. This enables us to use asynchronous IPs in aggressive datapath optimization without altering the operation mechanism of the whole system.

To cope with external components with different speeds, asynchronous communication is used in the external interfaces. In our case, the instruction memory and the data memory are surrounded by asynchronous wrappers, so that the target processor can work with memory at slow speed. On the other hand, the processor can operate at full throttle when there is no memory access.

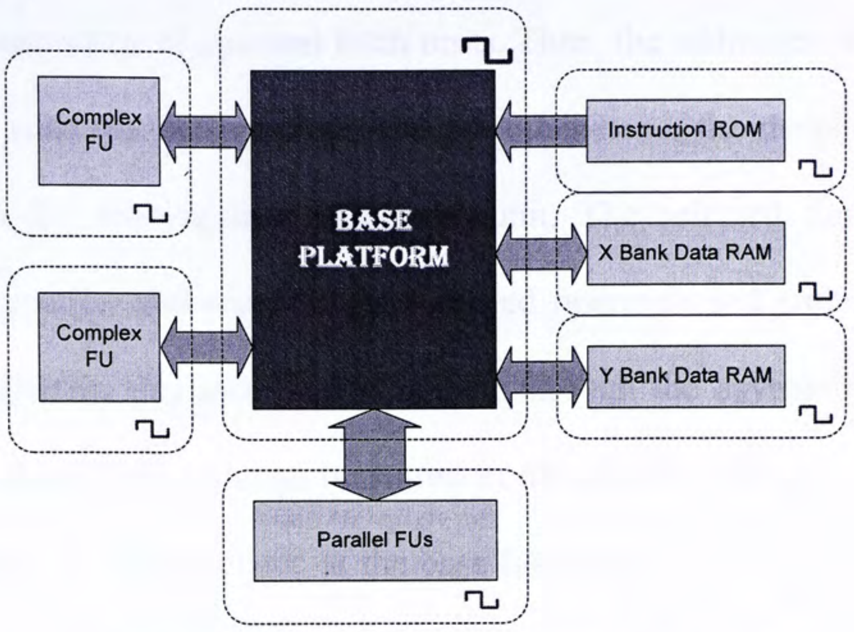


Figure 4.17: The GALS partitioning of the platform

4.2.7. Operation Mechanism

To summarize the platform architecture and the techniques we used, a demonstration of the operation of an optimized platform is given. The demonstration is focused on the operation mechanism of 1) simple and complex instructions; 2) parallel instructions; 3) software pipelined parallel instructions.

<u>Simple and Complex Instructions</u>

Figure 4.18 shows the sequence of operations of performing simple and complex instructions. First of all, the instruction fetch unit provides the instruction address to the instruction memory and fetches the corresponding instruction. The fetched instruction is classified and decoded into control signals for driving the other parts of the processor and constant values. The address generation unit calculates the operand addresses according to the predefined addressing mode unless the locations of the operands are specified by immediate values in the instruction. As the base functional units and the complex functional units are not designed for operating simultaneously,

they share the same pair of operand fetch units. Then, the addresses are passed to the register file to read out the operands. On the other hand, the decoder prepares the control signals for the required functional unit. The selected functional unit is enabled for execution and receives the required operands and control signals. For complex instructions, this information is sent through the asynchronous interface. After the execution, the result can be stored in the internal storage of the functional unit, for example the accumulator of the base functional unit, or written back to the register file. The address for the writeback is also prepared by the address generation unit or the immediate address from the instruction.

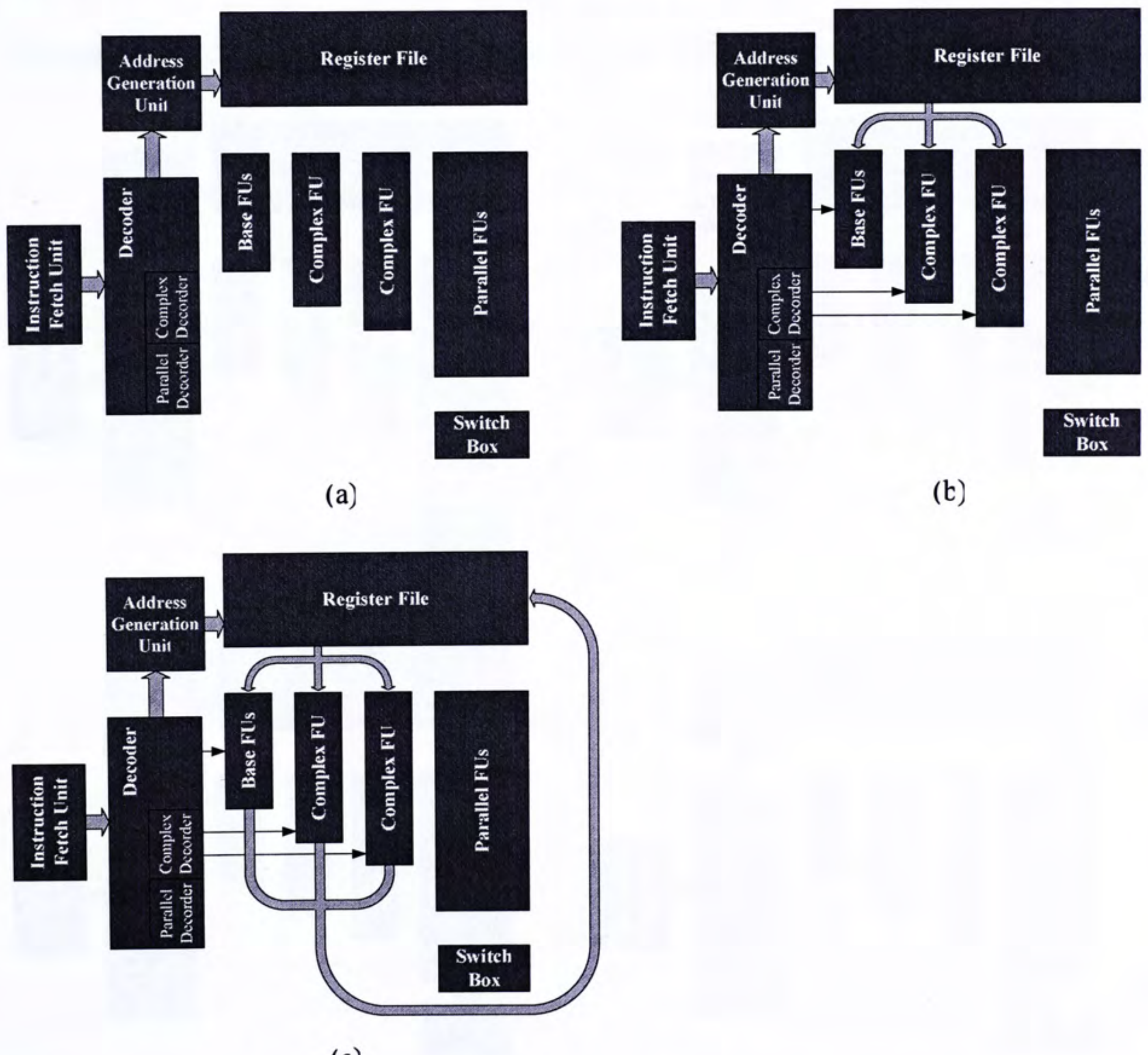


Figure 4.18: The sequence of the data and control flow in the operation mechanism of simple instructions and complex instructions

Parallel Instructions

Figure 4.19 shows the operations of parallel instructions. Similar to that of simple and complex instructions, the procedures of fetching instructions are the same. However, that of decoding instructions is different. As the parallel instructions are in compressed form, the instruction decoder dispatches these instructions to the instruction decompressor for further interpretation. The address generation unit prepares operand addresses for each parallel functional unit. And the control signals are sent to the functional units involved in the execution. Then the operations of execution and writeback are the same as those of complex instructions. The only difference is that all the results can be written back to the register file in parallel.

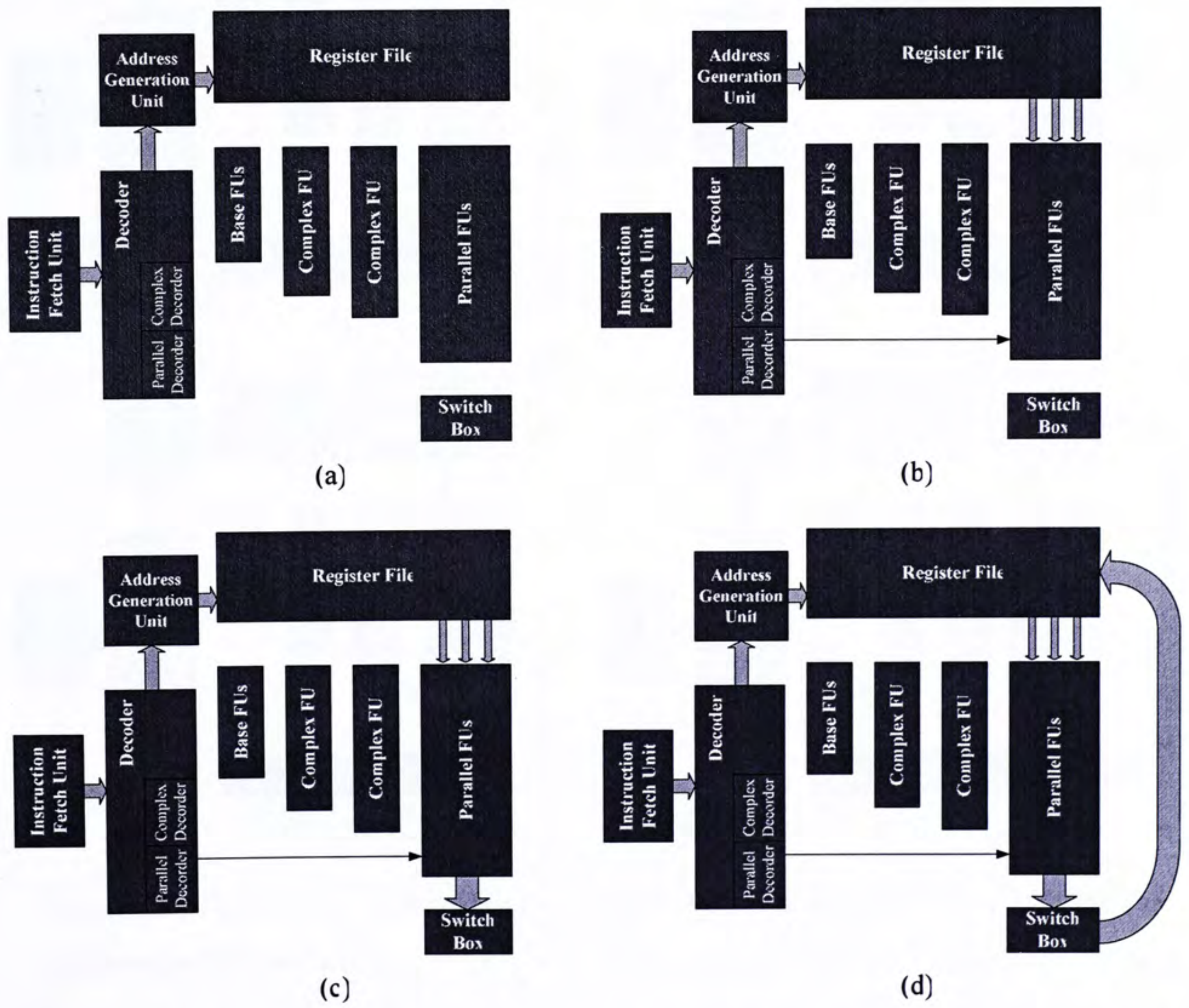


Figure 4.19: The sequence of the data and control flow in the operation mechanism of parallel instructions

Software Pipelined Parallel Instructions

The operation of software pipelined parallel instructions is more or less the same as that of the parallel instructions. To enable software pipelining, the results from the functional units have to be sent to another functional unit as operand right after the execution. The required configuration is shown in Figure 4.9b. In the platform, a switch box is used for the connections among the parallel functional units. The results are routed back as operands (Figure 4.20d). The final result that comes out from the pipeline is written back to the register file.

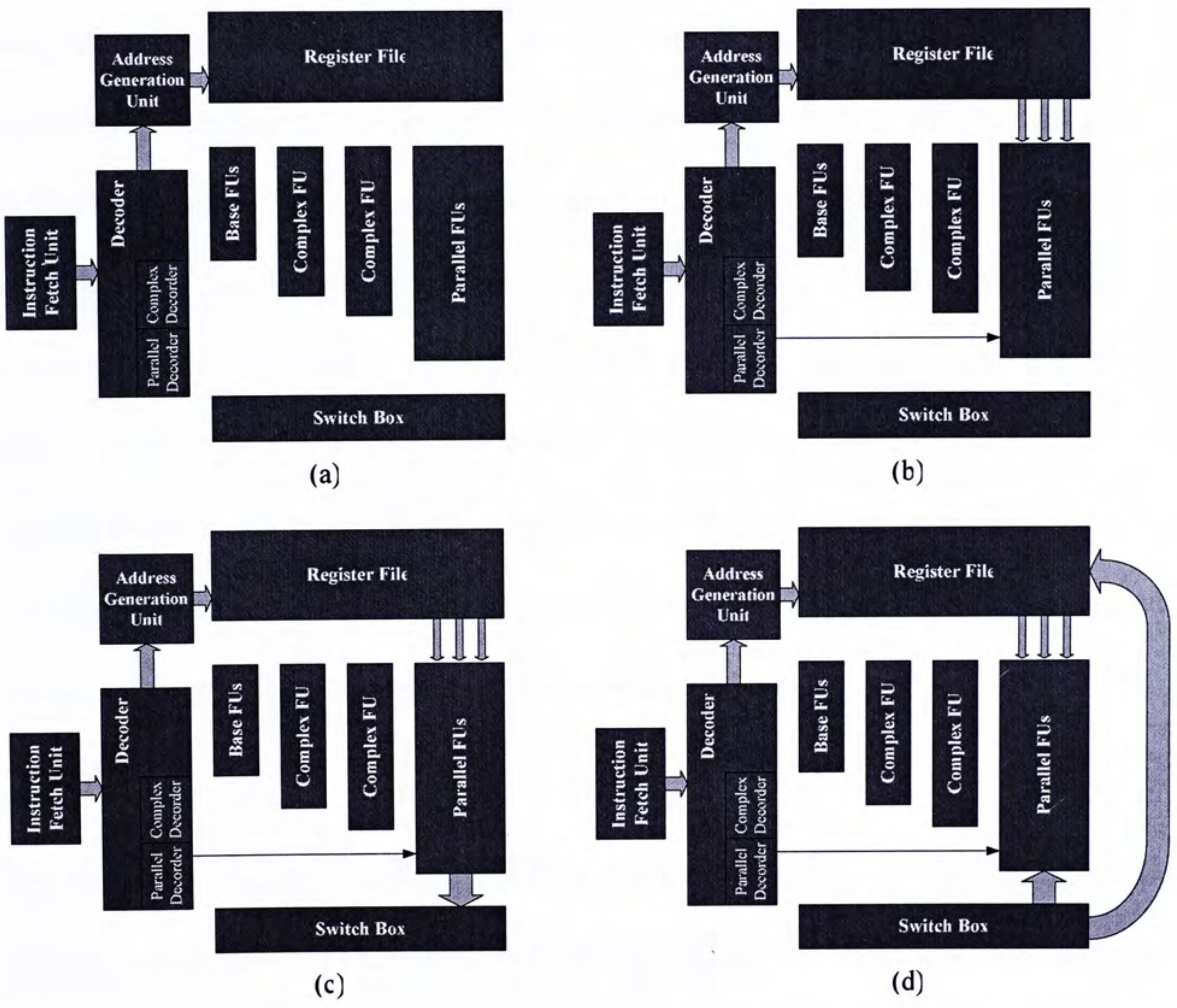


Figure 4.20: The sequence of the data and control flow in the operation mechanism of software pipelined parallel instructions

4.3. Overall Design Flow

To enrich the proposed design methodology, we round it up with a suggested overall design flow. As we do not cover the application analysis and software generation in our research, the design flow is conceptual but the suggestions are believed to be practical. Refinement by the experts in these areas is recommended.

Jain [40] studied various ASIP design methodologies and found that typically there are five main steps following the design of ASIP (Figure 4.21). The five steps are: 1) application analysis – to get the desirable characteristics which can guide the hardware synthesis as well as instruction set generation; 2) architecture design space exploration – to find out the possible architecture for a specific application with minimum hardware cost; 3) instruction set generation – to generate an instruction set for that particular application and for the architecture selected; 4) code synthesis – to synthesize code for the particular application or for a set of applications; 5) hardware synthesis – to synthesize the selected architecture and the hardware modules corresponding to the instruction set architecture.

The suggested design flow is more or less similar to the typical one. As the proposed design methodology starts with a platform, those five steps are restructured and realized as in Figure 4.22. First of all, we describe the application in high level language (HLL). The description is mapped to the base instruction set (Appendix B) for profiling. In application profiling, an instruction set simulator is used to collect the static and dynamic characteristics of the application. The simulator uses some sets of carefully selected realistic data as test vectors, and relaxes the size of the register file in order to investigate the data locality. In the simulation, the number of executions of each line of code, the memory access, the locality of data and the statistics of programme flow are recorded. Loop kernels with a significant number of iterations can be clearly identified.

Based on the results of the profiling, we can customize the architecture parameters of the platform. Basically, there are four parameters: 1) the size of general register files; 2) the number of special registers; 3) the size of the programme counter stack; 4) the size of the data memory; 5) the size of instruction memory; 6) the width of the data. Intuitively, the size of the data and instruction memories required and the width of the data can be found according to the memory access and the application specification respectively. The size of the general register file depends on the locality of data and the programme counter stack on statistics of programme flow, especially the numbers about zero-overhead loops and function calls. The number of special registers is related to the number of configuration registers needed for the optimized datapath, hence this parameter can only be known after datapath optimization.

The loop kernels discovered in application profiling are treated in datapath optimization. Parallel instructions and complex instructions are introduced to accelerate the loop kernels with heavy duty. In the course of optimization, each of the discovered loop kernels is investigated to identify its potential parallelism and the available hardware accelerator for it. A gain value and the corresponding cost are calculated for each attempt of acceleration. The instruction that has the highest gain but its cost does not exceed the pre-defined threshold is selected.

After the optimization, the new instruction set is evaluated in architectural profiling. This simulation takes the customized architectural parameters and the new instruction set architecture into account. If the optimized platform meets the real-time requirement of the target application, the new functional units will be realized in HDL and inserted to the platform to scale up its datapath. The corresponding software is implemented by compiling the HLL description of the application with the new instruction set. The failed design has to go back for further optimization.

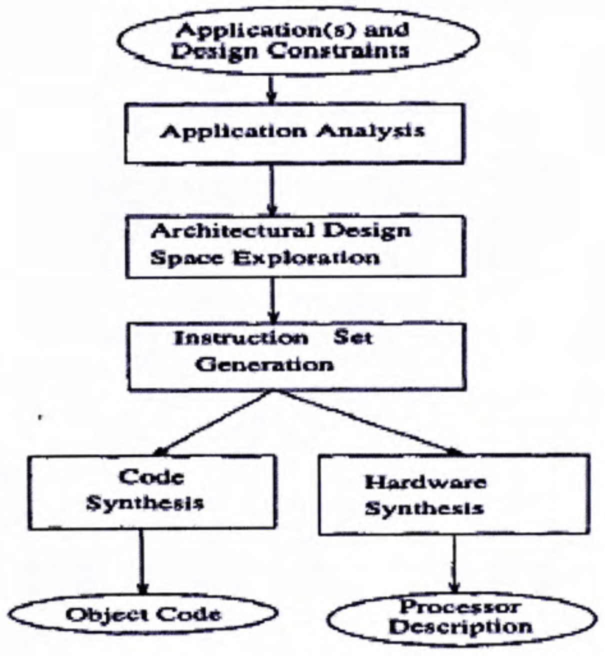

Application(s) and Design Constraints
Application Analysis
Architectural Design Space Exploration
Instruction Set Generation
Code Synthesis
Hardware Synthesis
Object Code
Processor Description

Figure 4.21: Typical ASIP design flow (source: [40])

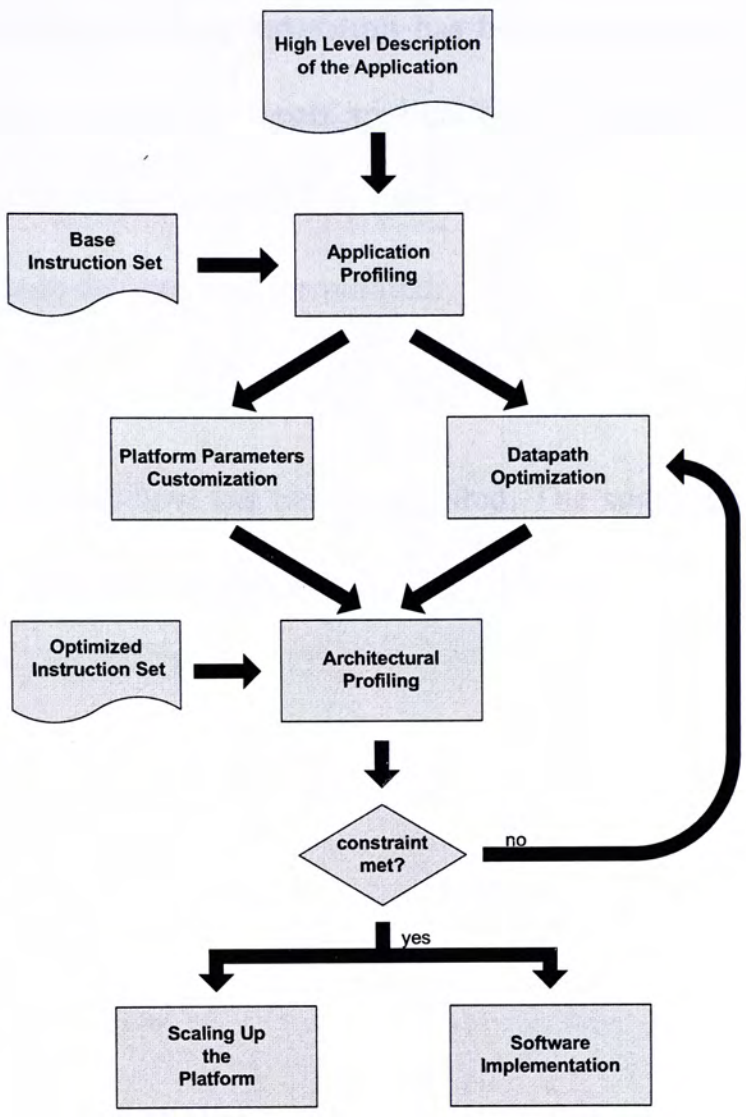


Figure 4.22: Platform-based design flow

4.4. Summary

This chapter describes the platform-based design methodology of ASIP. This methodology starts with a pre-defined platform. The design philosophy is to scale up the datapath of the platform and to customize its architectural parameters to meet the real time requirements of the target application. An efficient and effective platform for embedded DSP application has been derived from the five highlighted natures of DSP applications.

The design space of datapath optimization has been investigated. There are mainly two approaches to accelerate the target application: 1) introduce parallel instructions 2) introduce complex instructions. The gain and the cost functions for these two approaches have been defined and formulated.

Lastly, an overall design flow has been suggested. The suggestions are not formally discussed, but are believed to be practical. They are ready for further refinement and development.

5. Design of the ASIP Platform

5.1. Design Goal

The system level architecture of the platform has been presented in Chapter 4. In this chapter, we discuss the micro-architecure of the platform. The design goal of the platform is to maximize the degree of reusability by 1) minimizing the extent of modification needed for the base platform; 2) minimizing the impact on timing and the power consumption when changing the architectural parameters. For the sake of explanation, the architecture is divided into instruction fetch, instruction decode, datapath and register file system as in Figure 5.1.

It was decided to implement the described platform on silicon for verification. Due to cost considerations, the implementation of the platform is with comparatively smaller size of instruction memory, data memory and register file. But this decision does not affect the functionality, the architecture and the instruction encoding of the platform. The selected architectural parameters are listed in Table 5.1.

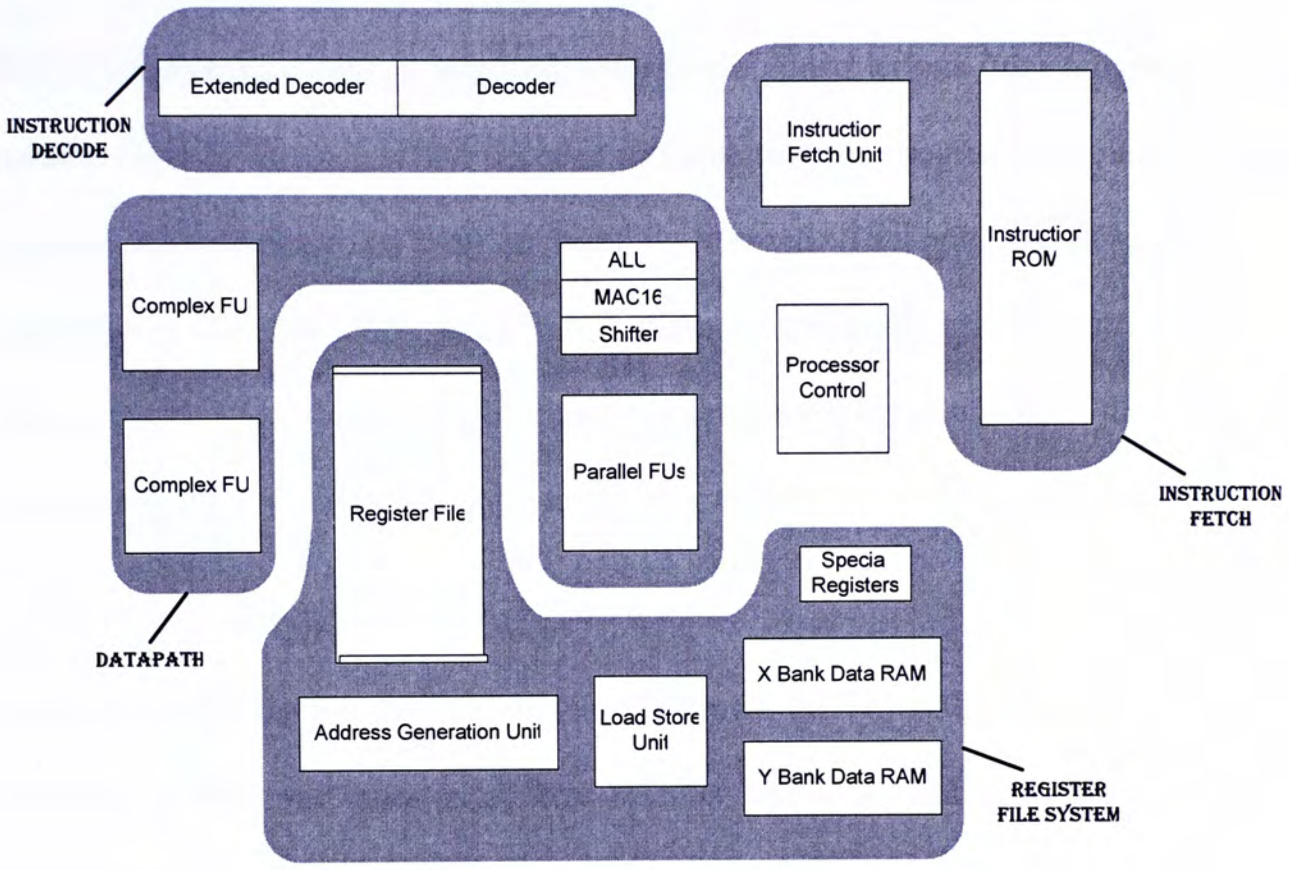


Figure 5.1: The organization of the platform architecture

Table 5.1: The architectural parameters of the platform for verification

Instruction Addressing	16 bits
Instruction Width	24 bits
Data Addressing	2 x 16 bits
Data Width	16 bits
Register File	2 x 64 x 16 bits

5.2. Instruction Fetch

5.2.1. Instruction fetch unit

The instruction fetch unit is responsible for reading instructions from programme memory, passing them to the instruction decoder and updating the programme counter. It begins to operate autonomously as soon as reset is released. The only factor complicating the operation of the instruction fetch unit is the need to handle branch instructions. When a branch is executed, the fetch unit must stop fetching instructions from the current stream and change the programme counter to the new value.

The structure of the instruction fetch unit is shown in Figure 5.2. It consists of three major modules: programme counter, address selector and loop, and subroutine controller. The programme counter is a register that stores the current position of the programme. This stored value is used as the instruction address for fetching instruction. On the other hand, this value is also passed to the instruction decoder as a reference for branching and other programme flow control activities. The programme counter updates its content autonomously with the new address provided by the address selector. The address selector controls the content of the programme counter. It calculates the new address and also collects the addresses from instruction decoder and the controller for loops and subroutines. Depending on the status of the processor and the requests from those two modules, the address selector supplies the appropriate address to update the programme counter. In this way, branches, loops and subroutine calls can be realized.

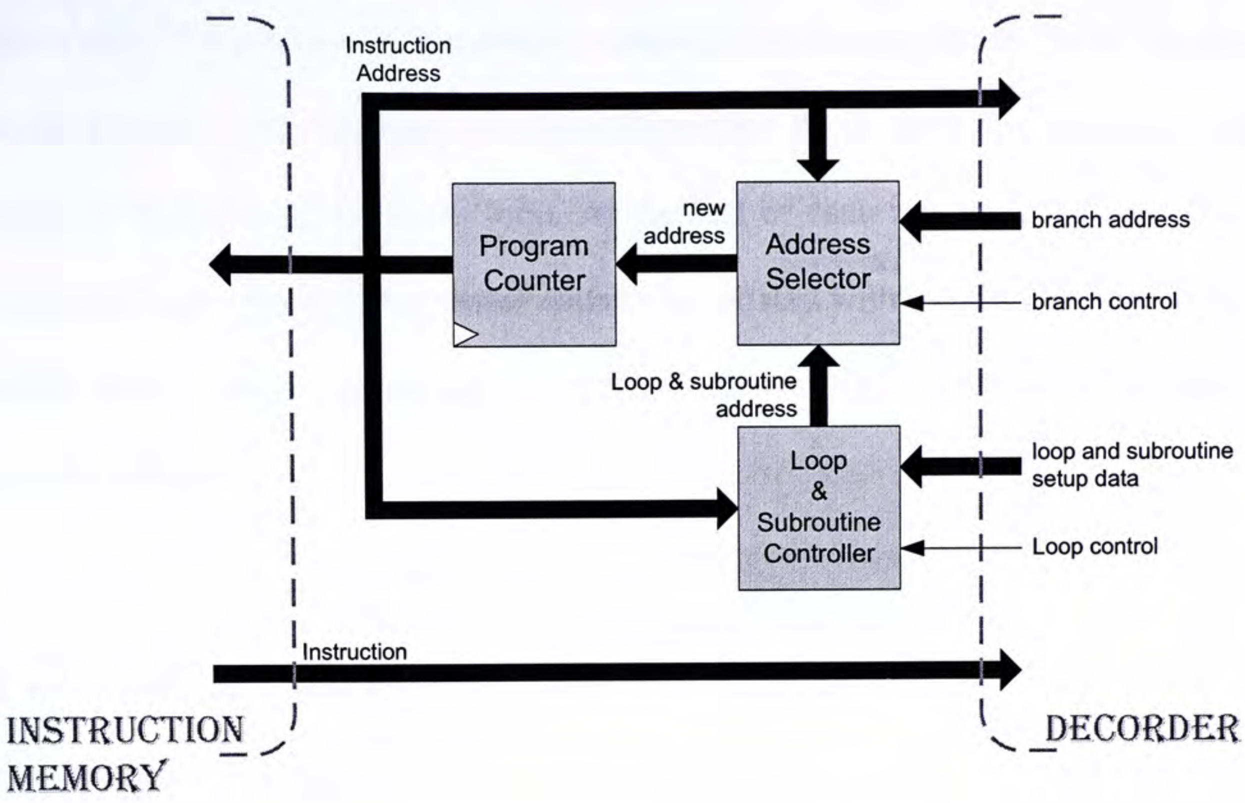


Figure 5.2: The structure of instruction fetch unit

5.2.2. Zero-overhead loops and Subroutines

Loops and subroutine calls are complicated tasks for instruction fetching. A dedicated controller is used to maintain the current status of a loop or a subroutine and to handle the address calculation. The structure of the controller is shown in Figure 5.3. The internal control logic interacts with the request from instruction decoder and, in addition, controls the operation of the stack and the loop counter. The status of the loop or subroutine is temporarily stored in the stack. The content of the stack is shown in Figure 5.4. The loops and subroutine calls are treated in a unified way in order to save resources and to be flexible to applications with different behaviours. The first one bit field tags the entry with an identity. The second field indicates the start or the return position of a loop or a subroutine respectively. The third field indicates the number of lines of instructions covered by a loop. The fourth field states the number of iterations left. The last two fields are not for subroutine calls. When a zero-overhead loop is set up, the current status and the setup data of the loop (start address and size) are pushed into the stack and the loop tag is set to one. The number of iterations is stored in the loop counter. Based on the setup data, the control logic can figure out the end position of the loop and the current relative position of the programme in the loop. At the end of each iteration, the loop counter is decreased by one, and the programme counter is updated with the start address of the loop. Once the loop counter reaches zero, the stored loop status pops up and the previous status can be maintained.

For subroutine calls, the setup procedure is more or less the same with zero-overhead loops. When a subroutine call is set up, the current status and the return address of the subroutine are push into the stack. This time the tag is set to zero. The programme counter then jumps to the position of the subroutine. Unlike the zero-overhead loops, there is no explicit end of the subroutine unless a return instruction is present. The return instruction can force the controller to restore to its pervious status and also the programme counter to the previous address stored in the stack. For zero-overhead loops, this return instruction is also effective. It is used to break the loop and to make the programme continue at the end of the loop.

As the loops and subroutine calls are treated similarly in the stack, it is possible to have nested levels of mixture of zero-overhead loops and subroutines. The total number of levels depends on the number of entries of the stack. Hence the size of the stack should be considered in application analysis in order to match the behaviour of the target application.

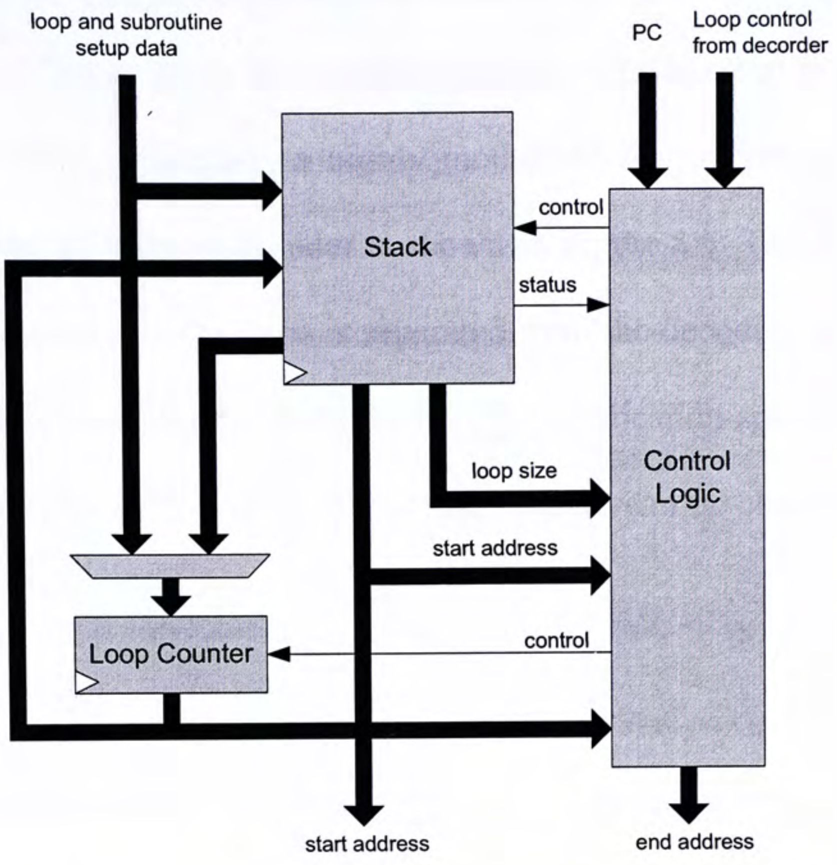


Figure 5.3: The structure of the loop and subroutine controller

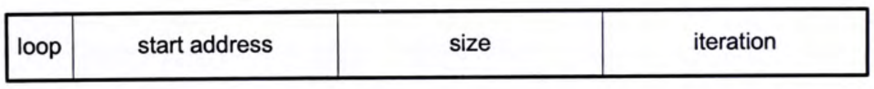


Figure 5.4: The content of the stack

5.3. Instruction Decode

5.3.1. Instruction decoder

The instruction decoder is responsible for translating the fetched instruction into useful information for different parts of the processor. Basically, it has three tasks to do: 1) identify the fetched instruction; 2) interpret the encoded part of the instruction and generate the corresponding control signals and opcodes; 3) dispatch the decoded information to the corresponding modules. In ASIP design, different application-specific instructions are introduced to accommodate different application's needs. Inevitably, the instruction decoder needs to be redesigned frequently, which is not favoured in design reuse. To meet our design goal, the changing part must be isolated in order to minimize the modification effort. Therefore, a highly modulized instruction decoder is designed. The structure of the instruction decoder is shown in Figure 5.5. The decoding of parallel instructions and complex instructions is separated from the decoding of base instructions. Two internal decoder modules are dedicated to these application-specific instructions. The contents of the parallel and complex instruction decoder can be changed without altering the others.

Secondly, the whole decoder is divided into two levels in order to match the pipeline organization. It is natural that the instructions involving execution of datapath is assigned to the second level which is closer to the datapath. This partitioning has two advantages: 1) numbers of pipeline registers can be saved because the data that has passed along the pipeline stages is the encoded part of the instruction instead of the massive control signals and opcodes; 2) The unused modules can be turned off efficiently. Table 5.2 lists out the modules that are activated in the execution of different classes of instructions. It shows that the second level can be disabled when the processor is doing flow control, configuration and memory manipulation. Moreover, the first level can activate one of the modules in second level only, as the base instructions, parallel instructions and complex instructions are orthogonal.

To make good use of this partitioning, the class of the fetched instruction has to be identified as soon as possible. Hence, the instruction encoding is first based on the classification of the instructions, then the number of bits needed for the arguments, so that the fetched instruction can be classified within the first few bits.

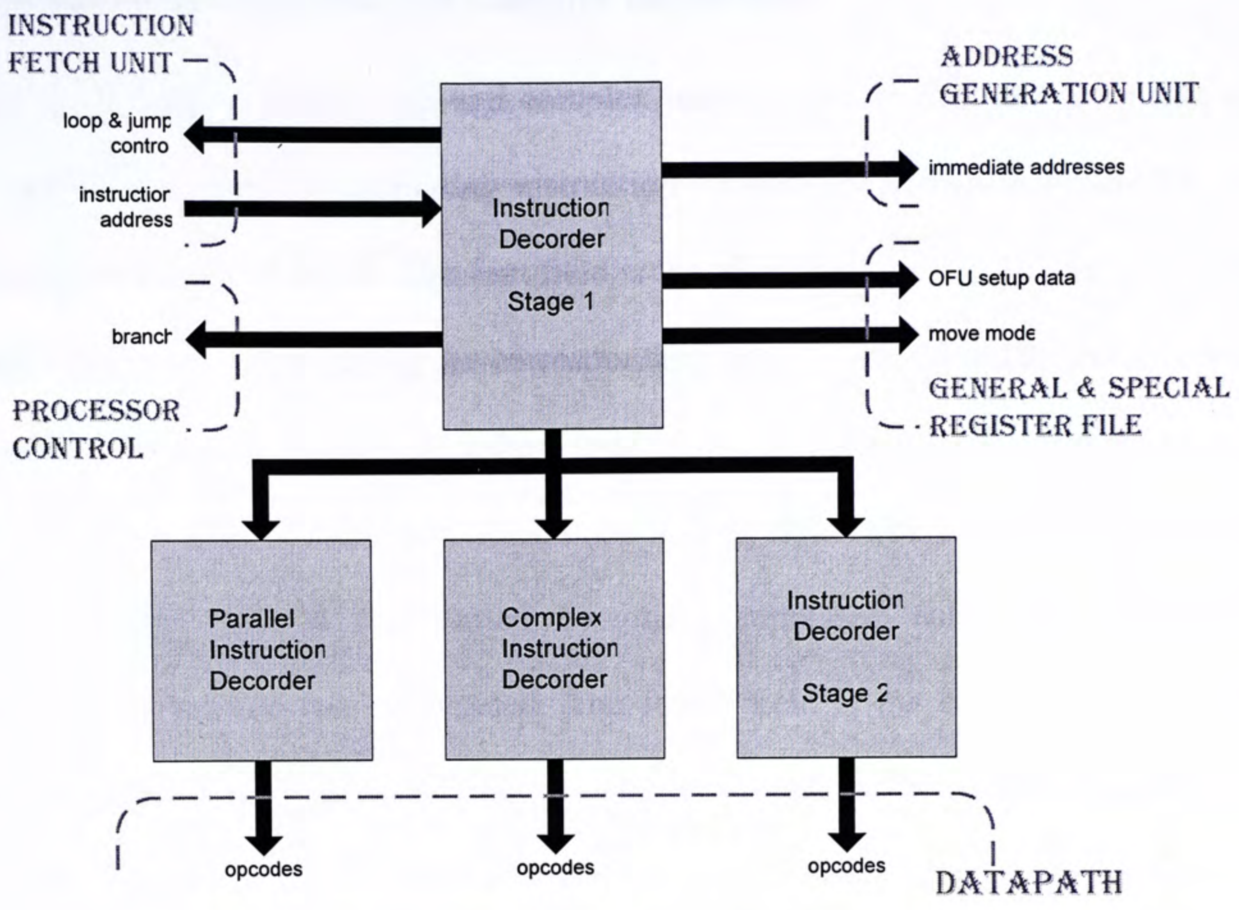


Figure 5.5: The structure of the instruction decoder

Table 5.2: The activity list of executing different classes of instructions

Classification	Instruction Fetch Unit	Processor Control	Address Generation Unit	General Register File	Special Register File	Datapath
Data Processing			☑	☑		☑
Bit Manipulation			☑	☑		☑
Boolean Operation			☑	☑		☑
Flow Control	☑	☑				
Configuration					☑	
Memory Manipulation			☑	☑		

5.3.2. The Encoding of Parallel and Complex Instructions

The encoding of parallel instructions and complex instructions are different from that of the base instructions. The corresponding instruction formats are shown in Figure 5.6. A parallel instruction has five fields. The first field is the identification tag. The second field is an enumerated index for recalling the corresponding opcodes stored in the lookup table of the parallel instruction decoder. The third field is the enable flags for each functional unit. It is possible to disable some functional units in the parallel operations. Some similar parallel instructions can be represented by one compressed entity, and therefore redundancy of the storage can be avoided. The fourth field is the enable flags for the writeback operation of each functional unit. The last two fields are the enable flags for the load store unit. One is for the X bank and the other is for Y bank. Besides using load and store instructions, the load store unit can be activated by the parallel instructions as a side effect. The load store unit can work autonomously if this flag is set. In this way, the load store operations can be decoupled from the operation of the processor, and their latency can be shadowed.

The format of the complex instructions is more complicated than that of parallel instructions. A complex instruction has eight fields. The first field is also an identification tag. The second field states the target hardware accelerator of this instruction. The third field is used for configuring the accelerator. The meaning of this field is defined by the functionality of the accelerator. The fourth and the fifth fields are enable flags for the operand fetch units. If they are not set, the immediate addresses provided in the last two fields are used. The sixth and seventh fields are bank select bit to indicate the bank of the provided immediate addresses.

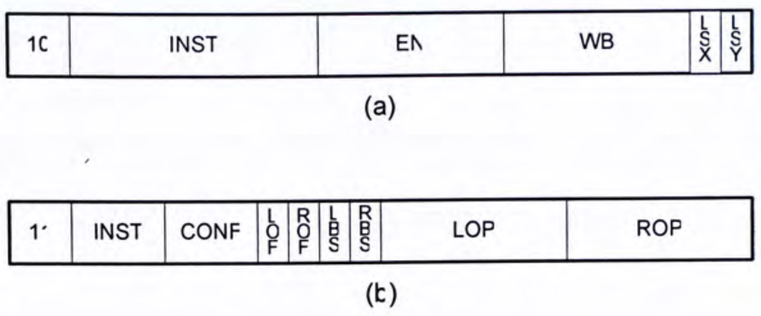


Figure 5.6: The instruction format of (a) parallel instructions and (b) complex instructions

5.4. Datapath

5.4.1. Base Functional Units

The base datapath is designed for general DSP application. Similar to other general digital signal processors, the number representation is two's complement and the heart of this datapath is a multiplier-and-accumulator (MAC). The structure of the base datapath is shown in Figure 5.7. In the centre is a 16×16 40-bit MAC. It is made of 3:2 compressors in Wallace tree configuration, an adder and a 40-bit register for accumulation. The most significant eight bits are guard bits for avoiding overflow. The adder in the MAC is also responsible for addition and subtraction operations.

For shift operation, a barrel shifter is used in the datapath. It can shift the accumulator by at most 32 bits to left and to right in arithmetic manner or in logical manner. The shift distance can be defined by the immediate value from the instruction and the value store in the register file. In addition, this shifter is also used to implement normalization operation. After the exponent instruction, the exponent of the current accumulated value is stored into a special register. This stored value is used as the shift distance of the shift operation when executing the normalization instruction.

There are also other modifiers for the accumulator: 1) logical unit for bitwise logic manipulation including AND, OR, XOR and NOT; 2) absolute unit for working out the absolute value of the accumulator; 3) negation unit for converting the accumulator to its opposite sign. The modified value is stored back to the accumulator.

Besides data processing, there is a comparison unit for comparing the two values in the register file or comparing the accumulator with one stored value in the register file. The comparison unit can report six conditions: 1) equal; 2) not equal; 3) greater than; 4) less than; 5) greater than or equal; 6) less than or equal. The result is stored as conditional flags in special register.

A complete instruction set description is shown in Appendix B.

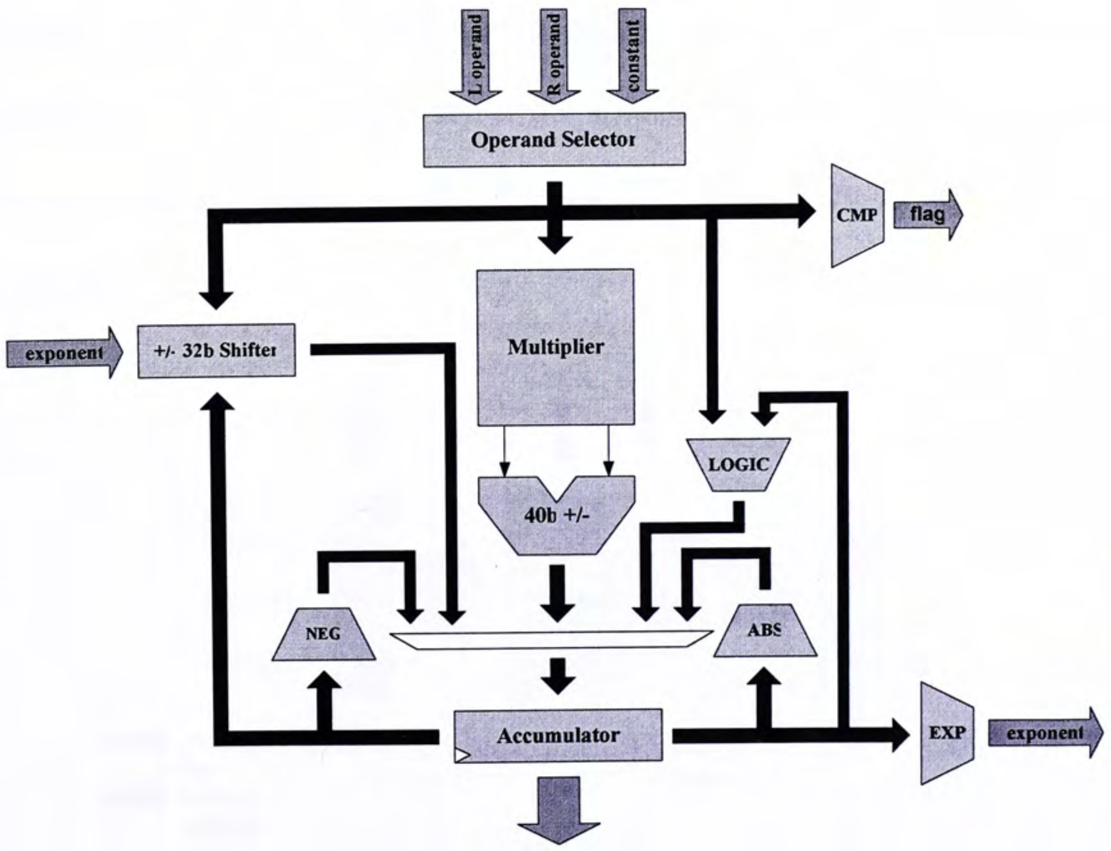


Figure 5.7: The structure of the base datapath

5.4.2. Functional Unit Wrapper Interface

For the application specific datapath, the platform is designed to offer maximum freedom to the hardware designers to design the extended functionality of the datapath. Designers can design synchronous and asynchronous, parallel and complex, single-cycle and multi-cycle functional units to enrich the datapath. For seamless integration of designer-defined functional units and the platform, a standard wrapper interface for each functional unit is defined. The wrapper interface is illustrated in Figure 5.8. Basically, the interface defines four parameters for controlling the source of operands, the working mode of the functional unit and the destination of the result. It also defines an optional internal register for temporary storage.

Parameter selOp is used to select the operands among the data from the register file, from the internal register and from the internal register of other parallel functional units (notated as SPin). Parameter selFunc is used to define the working mode of a multi-functional functional unit. Parameter option is used to define the configurable options. Parameter selOut is used to select the source of output: 1) result from functional unit or 2) from internal register; and the destination of the output: 1) to internal register or 2) to register file.

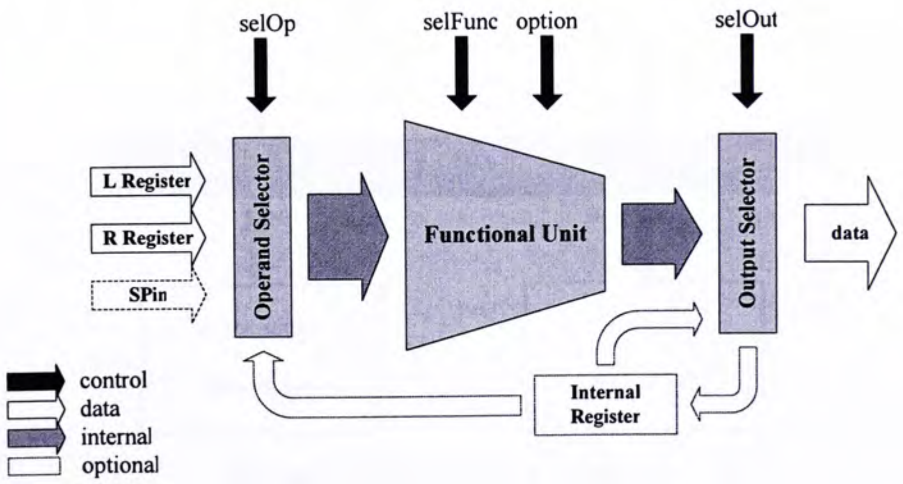


Figure 5.8: The abstract view of the functional unit wrapper

5.5. Register File Systems

5.5.1. Memory Hierarchy

In common with other current DSPs, the platform uses a dual Harvard architecture where one programme memory and two separate data memories (labelled X and Y) are used. This avoids conflicts between programme and data fetches, and many DSP operations map naturally onto dual memory space. For example, the data and the filter coefficients can be stored separately in X and Y memories. This is also true for storing two individual streams of data for convolution or cross correlation.

As most of the DSP algorithms illustrate strong locality of data reference, a large register file is preferred in order to allow high degree of data reuse. The memory hierarchy is shown in Figure 5.9. The register file is partitioned into X and Y banks for matching the organization of the data memory. To interface with the data memories, a load store unit is used. It is designated for reading from and writing to the data memories in bulk.

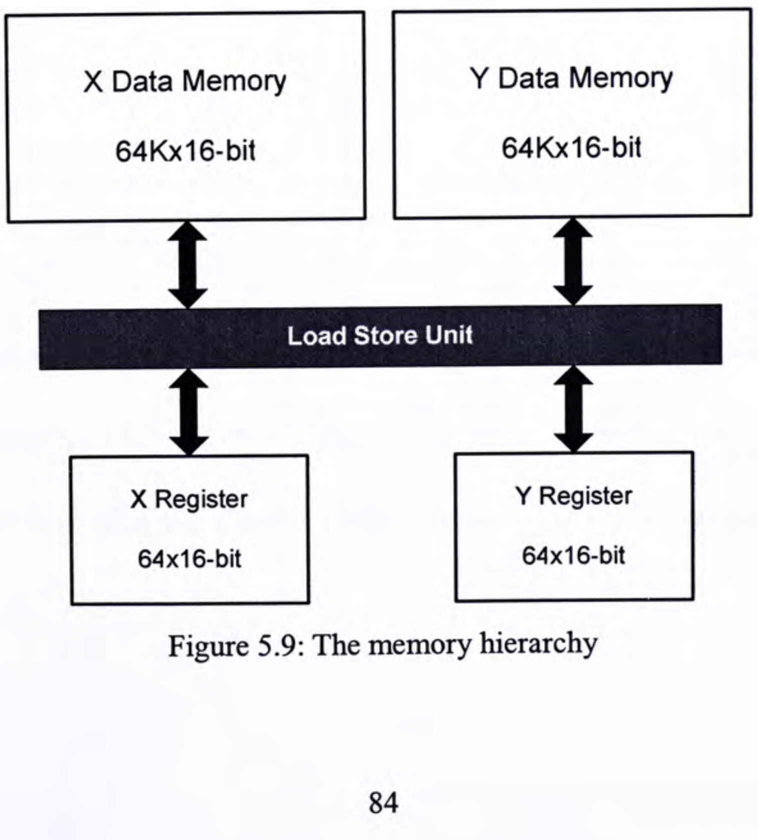


Figure 5.9: The memory hierarchy

5.5.2. Register File Organization

To supply enough operands to the parallel datapath without introducing any conflict, the register file is designed to be multi-ported. However, when the number of ports grows, the performance of the register file deteriorates in terms of delay, area and power consumption. Rixer presented the impact to the performance of a register file with increasing number of arithmetic units in [47]. He showed that for N arithmetic units, the area of the register file grows as N^3, the delay as $N^{3/2}$, and the power consumption as N^3. The main reason is that more arithmetic units need more ports for parallel execution, which implies exponential growth in the complexity of the address decoder and the interconnection between the arithmetic units and the register file. Inevitably, this can have a great impact on the platform when scaling up the parallel datapth.

Partitioning register file into multiple banks was reported to be an effective solution to slowing down the performance deterioration in [44][45][46][47]. The design philosophy of multi-banked register files is to distribute the ports to different register banks, so that the number of ports per bank can be reduced. This method can alleviate the complexity of the interconnection, but the drawback is that each port is confined to access the corresponding bank only. The primary challenge of this scheme is to avoid the number of simultaneous accesses to any bank exceeding the available ports on each bank. In other design, a recovery stage is inserted after the read stage in order to resolve the conflict. Our design is based on the observation that the data that needs to be accessed simultaneously can be uniformly assigned to different banks for many DSP algorithms. In other words, data conflict can be omitted if the data is carefully assigned to suitable banks.

Data Memory Access Pattern

A study of the data access pattern of DSP algorithm is carried out. Convolution and correlation are the subjects of the study. The rationales are: 1) they are the most fundamental in DSP, most of the algorithms are derived from and based on it; 2) they show lots of parallelism, which is suitable for studying parallel data access pattern; 3) they involve arithmetic operation (multiply) and access to local temporary storage (accumulation), which is sophisticated enough for modelling more complex algorithms. On the other hand, they are simple enough for analysis.

To analyze convolution and correlation in a unified view, they are combined mathematically into a general form, $y(n)=\sum_{k=0}^{N-1} A(k)B(n \pm k)$ where $A(n)$ and $B(n)$ are the two digitized input signals. $y(n)$ is the resultant signal. N is the frame length of $A(n)$ or $B(n)$. The data access pattern for exploiting data parallelism is tabulated in Table 5.3. It is assumed that $N = 16$ and there are four functional units in the parallel datapath. In the table, the bold items are the arithmetic operations. The operation **mul** represents a multiplication; operation **mac** is a multiplication-and-accumulation; operation **add** is an addition. The four functional units have their own accumulator for temporary storage which are notated in *acc*. The items in italics are the operands needed for the corresponding operation. And *t* indicates a particular time instant.

Observing the data dependency of the access pattern in Table 5.3, it is possible to further partition each register bank into four blocks (conventional term is bank, but to get rid of misleading to X and Y banks, we call them 'blocks'). The first block contains data with index 4n; the second one contains those with 4n+1; the third one contains those with 4n+2;

the last one contains those with 4n+3. In this arrangement, each functional unit possesses a block of X bank and a block of Y bank. The functional units only need to access data from the local blocks, and there is no need to access data across other blocks. As a result, each block is only required to provide one read port for the functional unit. The corresponding structure of the register file is shown in Figure 5.10. *A*(*n*) and *B*(*n*) are supposed to be stored in X and Y banks separately. This structure illustrates a way to assemble an eight read ports register file with eight register blocks that with one local read port on each block.

Table 5.3: The data access pattern for exploiting data parallelism

t	FU0				FU1				FU2				FU3			
0	**mul**	A_0	B_n		**mul**	A_1	$B_{n\pm1}$		**mul**	A_2	$B_{n\pm2}$		**mul**	A_3	$B_{n\pm3}$	
1	**mac**	A_4	$B_{n\pm4}$	acc_0	**mac**	A_5	$B_{n\pm5}$	acc_1	**mac**	A_6	$B_{n\pm6}$	acc_2	**mac**	A_7	$B_{n\pm7}$	acc_3
2	**mac**	A_8	$B_{n\pm8}$	acc_0	**mac**	A_9	$B_{n\pm9}$	acc_1	**mac**	A_{10}	$B_{n\pm10}$	acc_2	**mac**	A_{11}	$B_{n\pm11}$	acc_3
3	**mac**	A_{12}	$B_{n\pm12}$	acc_0	**mac**	A_{13}	$B_{n\pm13}$	acc_1	**mac**	A_{14}	$B_{n\pm14}$	acc_2	**mac**	A_{15}	$B_{n\pm15}$	acc_3
4	**add**	acc_0	acc_1						**add**	acc_2	acc_3					
5	**add**	acc_0	acc_2													

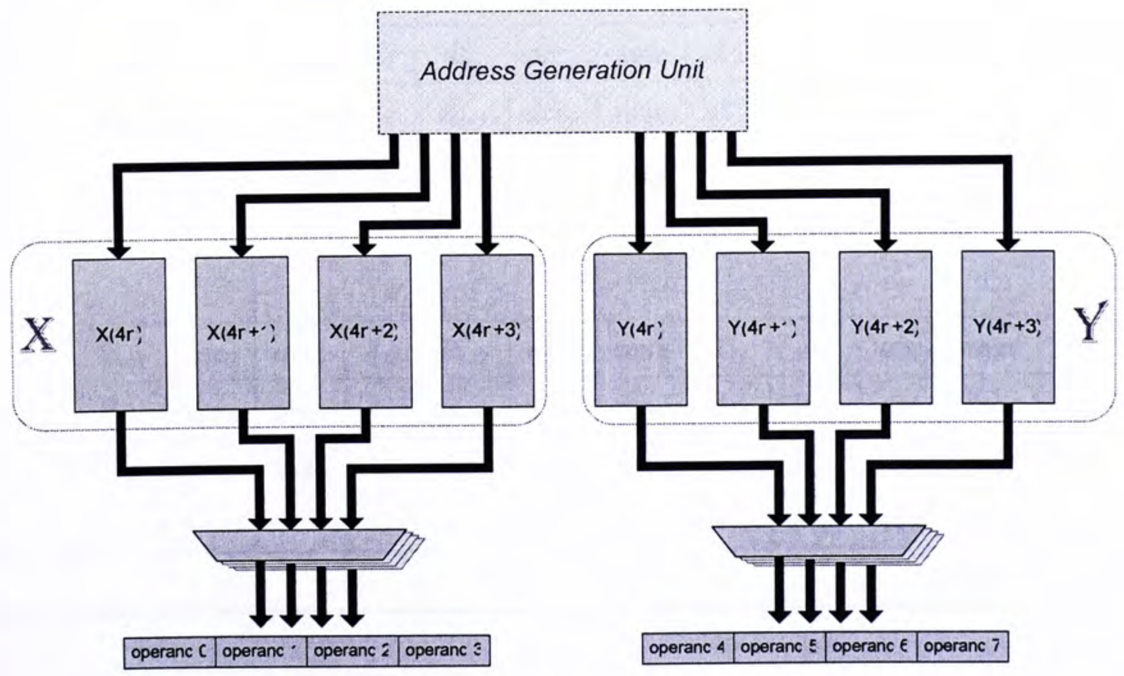


Figure 5.10: The structure of register file for exploiting data parallelism

Software pipelining is another method to exploit parallelism. Its data access pattern is tabulated in Table 5.4. Similar to the previous case, we can organize a register bank to four blocks without data conflict for only one read port per block. However, it is impossible to assign two dedicated blocks for each functional unit. According to the access pattern, the functional units need to access different blocks in a rotating manner. For example, one functional unit accesses in the sequence of the first block, then successively the second, the third and the fourth. Afterwards, it accesses back to the first block and repeats the sequence. Therefore, rotators are used for dispatching the address to the correct block. The corresponding structure of the register file is shown in Figure 5.11.

Table 5.4: The data access pattern for applying software pipelining

t	FU0				FU1				FU2				FU3			
0	**mul**	A_0	B_n													
1	**mul**	A_0	$B_{n\pm1}$		**mac**	A_1	B_n	acc_0								
2	**mul**	A_0	$B_{n\pm2}$		**mac**	A_1	$B_{n\pm1}$	acc_0	**mac**	A_2	B_n	acc_1				
3	**mul**	A_0	$B_{n\pm3}$		**mac**	A_1	$B_{n\pm2}$	acc_0	**mac**	A_2	$B_{n\pm1}$	acc_1	**mac**	A_3	B_n	acc_2
4	**mac**	A_4	B_n	acc_3	**mac**	A_1	$B_{n\pm3}$	acc_0	**mac**	A_2	$B_{n\pm2}$	acc_1	**mac**	A_3	$B_{n\pm1}$	acc_2
5	**mac**	A_4	$B_{n\pm1}$	acc_3	**mac**	A_5	B_n	acc_0	**mac**	A_2	$B_{n\pm3}$	acc_1	**mac**	A_3	$B_{n\pm2}$	acc_2
6	**mac**	A_4	$B_{n\pm2}$	acc_3	**mac**	A_5	$B_{n\pm1}$	acc_0	**mac**	A_6	B_n	acc_1	**mac**	A_3	$B_{n\pm3}$	acc_2
7	**mac**	A_4	$B_{n\pm3}$	acc_3	**mac**	A_5	$B_{n\pm2}$	acc_0	**mac**	A_6	$B_{n\pm1}$	acc_1	**mac**	A_7	$B_{n\pm3}$	acc_2
⋮			⋮				⋮				⋮				⋮	
14	**mac**	A_{12}	$B_{n\pm2}$	acc_3	**mac**	A_{13}	$B_{n\pm1}$	acc_0	**mac**	A_{14}	B_n	acc_1	**mac**	A_{11}	$B_{n\pm3}$	acc_2
15	**mac**	A_{12}	$B_{n\pm3}$	acc_3	**mac**	A_{13}	$B_{n\pm2}$	acc_0	**mac**	A_{14}	$B_{n\pm1}$	acc_1	**mac**	A_{15}	B_n	acc_2
16					**mac**	A_{13}	$B_{n\pm3}$	acc_0	**mac**	A_{14}	$B_{n\pm2}$	acc_1	**mac**	A_{15}	$B_{n\pm1}$	acc_2
17									**mac**	A_{14}	$B_{n\pm3}$	acc_1	**mac**	A_{15}	$B_{n\pm2}$	acc_2
18													**mac**	A_{15}	$B_{n\pm3}$	acc_2

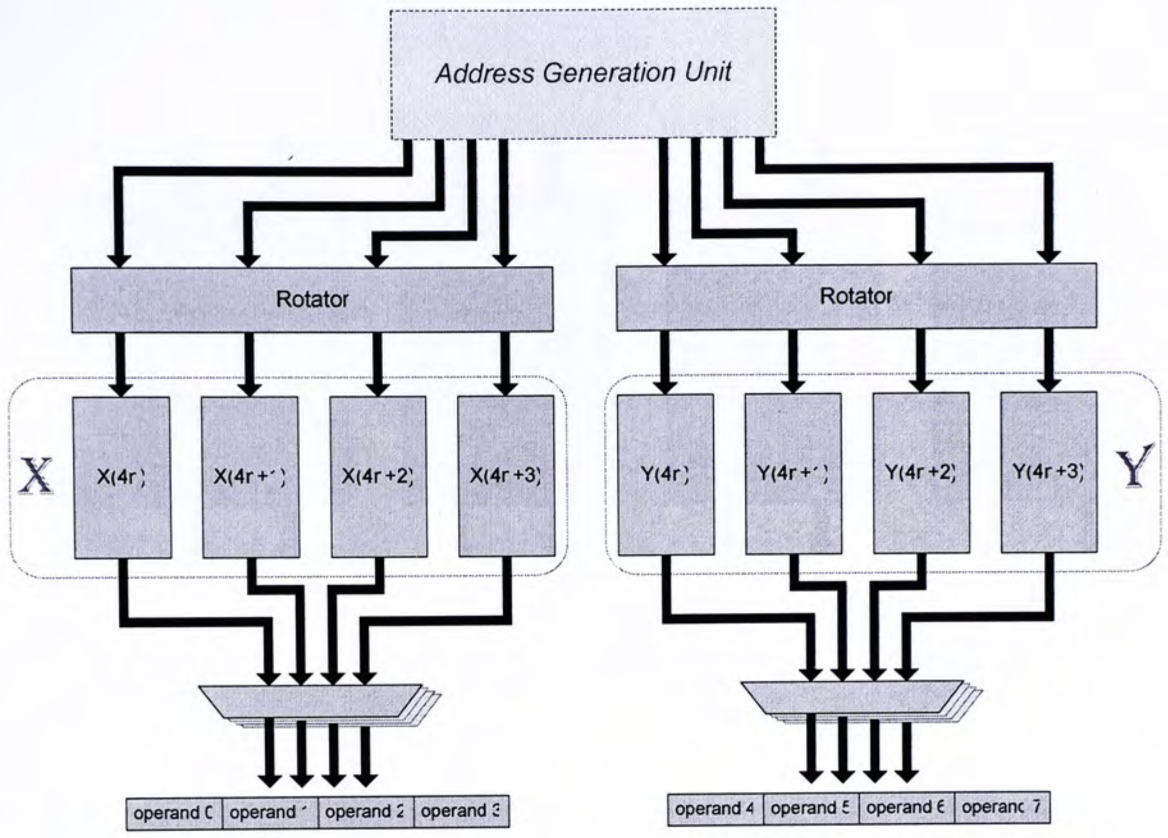


Figure 5.11:The structure of register file for applying software pipelining

<u>The Organization of the Register File</u>

Considering that $A(n)$ and $B(n)$ can be referred to the same signal for some algorithms, such as autocorrelation, all the data need to be fetched from the same register bank. The required number of ports per bank is double of the previous analysis. In order to provide enough ports, a pair of X and Y blocks with the same index are combined and organized in register block with two local read ports. The organization of these register blocks are shown in Figure 5.12. The first half of the blocks is assigned to the X bank and the other half is assigned to the Y bank. The most significant bit of the address of the blocks is used to identify the two banks. This arrangement has two advantages: 1) it unifies the ports for X and Y banks, thus allowing reading across X and Y banks freely; 2) it keeps the two banks separate in the programming model without requiring complex logics for address mapping.

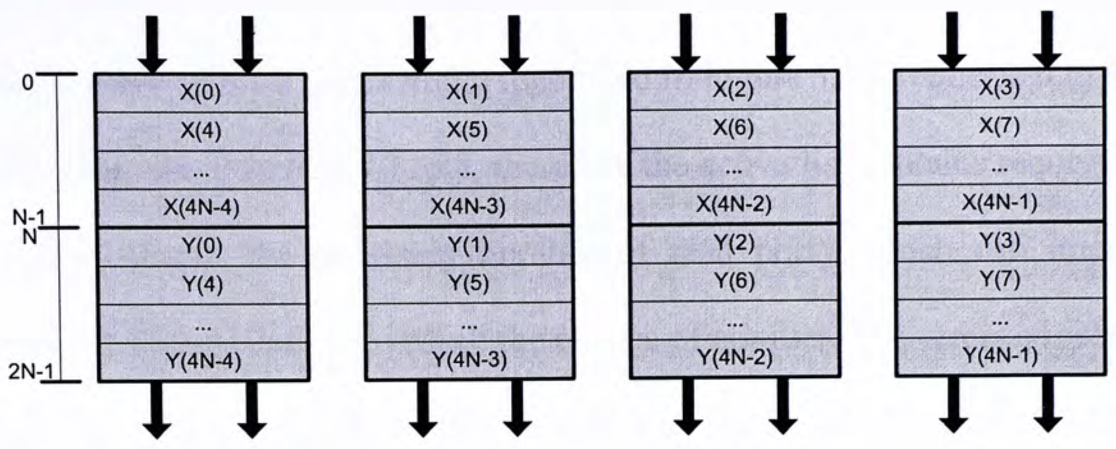


Figure 5.12: Organize two read port register blocks to X bank and Y bank

For writing back the results, it is necessary to have four global write ports for four functional units. Based on the finding in [44] by Tseng, we decide to use two local write ports in a block to realize the required number of global ports. Tseng reports that the incidence of writeback conflicts can be reduced by having two local write ports per block. On the other hand, the block with two read two write ports is comparable to that with two read one write in term of delay, area and power consumption. In our case, there are totally eight local write ports among the blocks. If the data can be allocated carefully, it is believed that the chance of conflict is very low.

Working Mechanism

The mechanism of reading data from the register file is shown in Figure 5.13. For a parallel datapath with four functional units, there are totally 10 read requests at most. These comprise eight operands for the functional units and two data for the load store unit. The requests have two fields to indicate the location of the data: 1) the bank selection and

2) the reference address. Before supplying this information to the register file, the requests that refer to the same location are screened out. Only one of the duplicated requests can be included in the active list. In the example illustrated in Figure 5.13, requests 0 to 2 refer to the same location, also 4 to 6 and 7 to 8, therefore the active list contains requests 0, 3, 4, 7 and 9. This reduces the necessary number of read ports, which can improve the reusability of the fetched data and reduce the chance of conflict. The screened requests are recovered in the latter stage. The addresses in the active list are translated to block selections and local addresses to fit our register organization. The block selection indicates the block containing the requested data. It is the residual of dividing the address by the number of functional units. The local address is the physical location within a block. It is the quotient of dividing the address by the number of functional units. Then the translated addresses are scanned sequentially and distributed to the first read port of the blocks. The second scan is in reverse order and for the second read port of the blocks. The distribution scheme is on a first-come-first-served basis, which means if one request occupies a particular port, the scanning for that port stops immediately. The reason is based on the assumption that there are always at most two read requests for the same block. Therefore, all the undistributed requests can be caught by the second scan. When the data are fetched from the register blocks, they are all tagged with its address in the active list. Based on the address tags, the multiplexers select the correct data and send them to the correct operand slot.

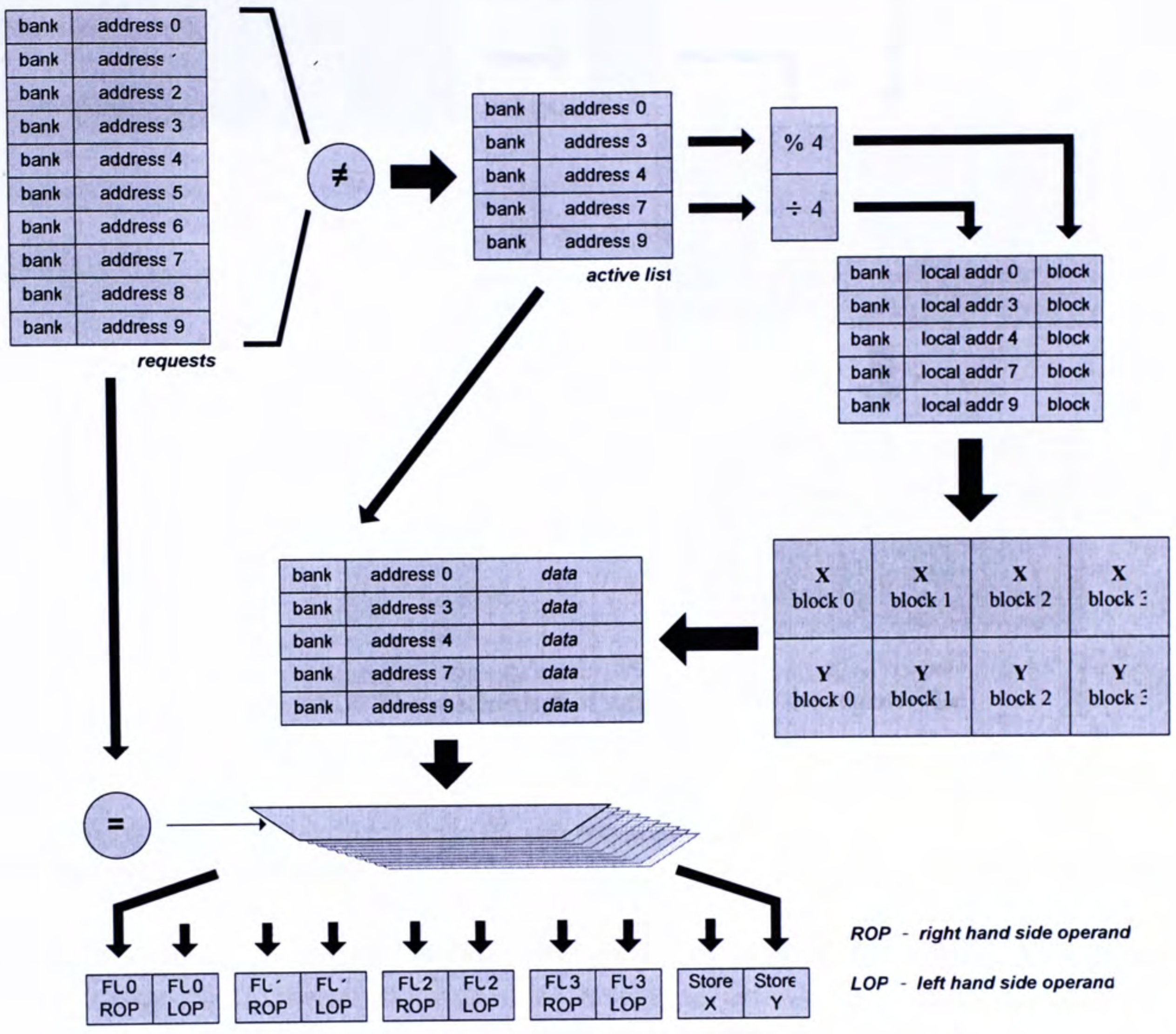


Figure 5.13: The mechanism of reading data from the register file

For writing data to the register file, the mechanism is at most the same, as shown in Figure 5.14. There are at most six write requests, four of them for the write back of the four functional units and the other two for the load operation of the load store unit. The requests are the bundle of bank selections, reference addresses and the data. Similarly, the addresses are translated to block selections and local addresses, and then supplied to the register blocks together with the corresponding data to accomplish the write operation.

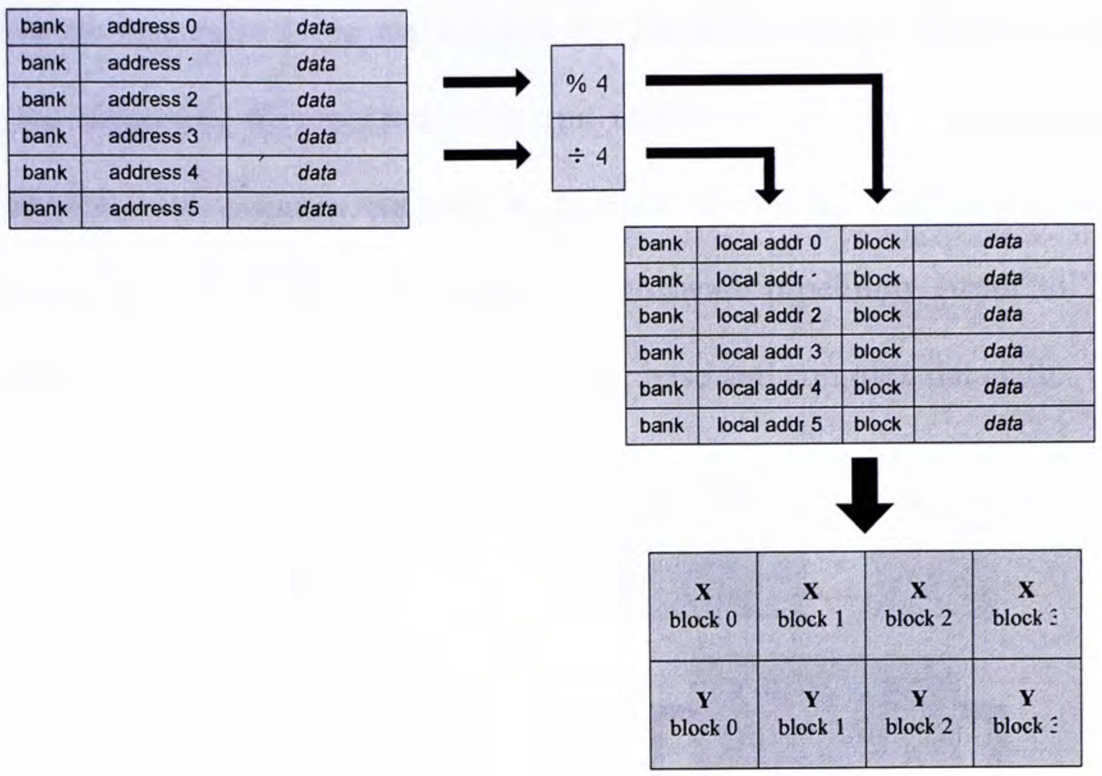


Figure 5.14: The mechanism of writing data to the register file

5.5.3. Address Generation

Address generation is another important task for smooth data supply. An autonomous generation can avoid interrupting the data processing task by address calculation. A dedicated component, address generation unit is responsible for handling the automation of address generation.

Address generation unit comprises hardware engines for address calculation. There are two kinds of address generation engines, one is for base datapath and complex datapath, the other one is for parallel datapath. The base and complex datapath share one engine, but the parallel datapath has one engine per functional unit. The organizations of the engines are shown in Figure 5.15. Both have 1) an operand fetch unit for calculating the addresses of the right and left hand side operands of a functional unit; 2) a writeback unit for the addresses of the result; 3) some registers for keeping the current address and the configuration parameters. These registers belong to the class of special registers and are organized as in Appendix C. In the engines for parallel datapath, there is an additional controller for triggering the operand fetch unit and writeback unit. The interval register specifies the interval between triggers in number of cycles. The controllable trigger pattern is used to realize the access pattern of software pipelining (refer to Table 5.4), hence the controller is omitted from the engine for base and complex datapath.

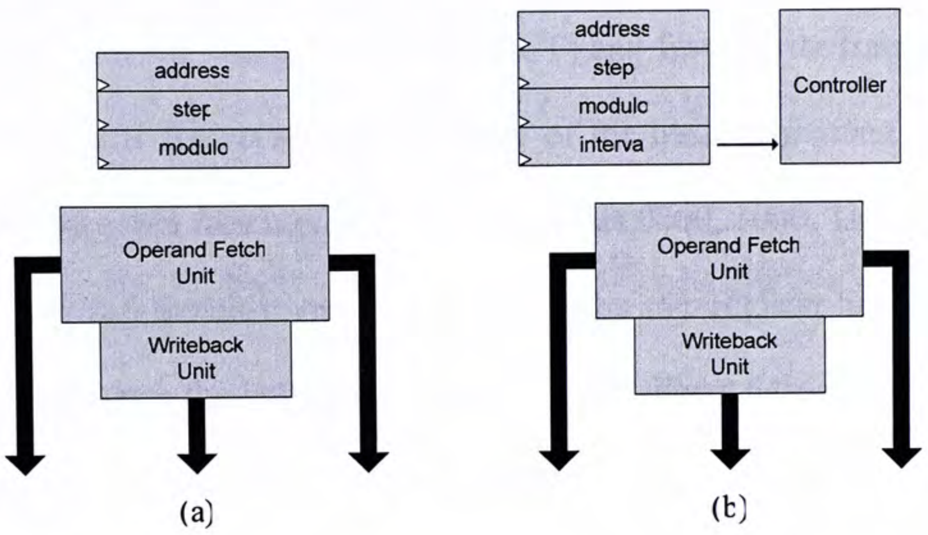


Figure 5.15: The organization of address generation engine for (a) base and complex datapath and (b) parallel datapath

Addressing Modes

The operand fetch unit and the writeback unit support three basic addressing modes: 1) increment, increasing the current address by one; 2) decrement, decreasing the current address by one; 3) displacement, increasing or decreasing the current address by the value specified in step register. There are also two complicated addressing modes: 1) circular, and 2) bit-reversed.

Circular addressing repetitively accesses a fixed size of block of data, in the way that accesses of data continue at the beginning of the block once the end of the block is passed. To set up the circular addressing on a block of size R, the *size* register has to be set to R.

The data block must be aligned to N-bit boundary, which means N least significant bits (LSBs) of the start address must be zero. The upper bound of the block size R has to be 2^N. For example, block with 32 entries must start at an address whose five LSBs are zero.

Bit-reversed addressing is used for butterfly-like data movement in some transformation algorithms, such as discrete cosine transform (DCT) and fast fourier transform (FFT). Its access pattern to data is the reverse of bit order of the basic increment addressing. For example, the sequence of a four bits address in binary is 0000, 1000, 1100, 0010, 1010, …, 0111 and 1111. To setup the bit-reversed addressing, the *step* register has to be set to zero. The *size* register indicates the half size of the transform. For a 64-point FFT, the value in the *size* register must be 32. The criterion for setting up start address is similar to that of circular addressing, which must be aligned to N-bit boundary. In addition, the size of the transformation must be less than or equal to 2^N.

Address Generation Datapath

The structure of the datapath for address generation used in the operand fetch unit and the writeback unit is shown in Figure 5.16. First of all, the mechanism of generating circular addressing is explained. Before doing any operation to the current address, a mask is applied in order to isolate the address range that is going to be modified. For example, when having circular addressing at $(010000)_b$ with block size of eight, the most significant three bits are masked and kept untouched. The rest of the bits are dispatched to the following datapath. The actual operation of circular addressing is $address' = (address + step)\%size$. If the value of step is always less than that of size, then $address' = \begin{cases} address + step \\ address + step - size \end{cases} \text{for} \begin{array}{c} \text{address} + \text{step} < \text{size} \\ \text{size} \le \text{address} + \text{step} \end{array}$. This expression can be implemented as a datapath with a two-input adder, a three-input adder and a multiplexer. The three-input adder is composed of a carry save adder (CSA) and a carry propagate adder (CPA). For stepping down, the operation is $address' = \begin{cases} address - step \\ address - step + size \end{cases} \text{for} \begin{matrix} 0 \leq \text{address} - \text{step} \\ \text{address} + \text{step} < 0 \end{matrix}$. As the address space is unsigned integer, we use overflowing effect to realise down-stepping with the same datapath. It is found that when address passes across the block boundary, the address is actually stepping down to get back to the data block. Therefore, we assume that $address' = address - step$ can be realized by $address' = address + step' - size$, if $step' = size - step$. When we substitute $step' = size - step$ into $address' = \begin{cases} address + step' \\ address + step' - size \end{cases} \text{for} \begin{matrix} \text{address} + \text{step}' < \text{size} \\ \text{size} \leq \text{address} + \text{step}' \end{matrix}$, we can obtain the expression for the operation of down-stepping. It is decided that such conversion work will be left for the assembler and hidden from programmers.

For bit-reversed addressing, we need an adder with reversed carry chain. The CPA in the three-input adder is reused and modified to fit the need. The structure of the modified CPA is shown in Figure 5.17. The carry chain of the modified CPA is designed to be bidirectional. The direction of propagation is selected by the multiplexer of each full adder. To activate bit-reversed addressing mode, the step and the size have to be set to zero and to the half size of the transform respectively. As the size register always presents a one at the MSB of the unmasked bits, we can simply add the current address and the size with a reversed adder to obtain bit-reversed addressing. An example of the generation of 16-bit bit-reversed address is illustrated in Figure 5.18.

For both addressing modes, the calculated address is eventually merged with the masked bits to form a complete address. This new address replaces the old one in the *address* register and is ready for the next cycle of address generation.

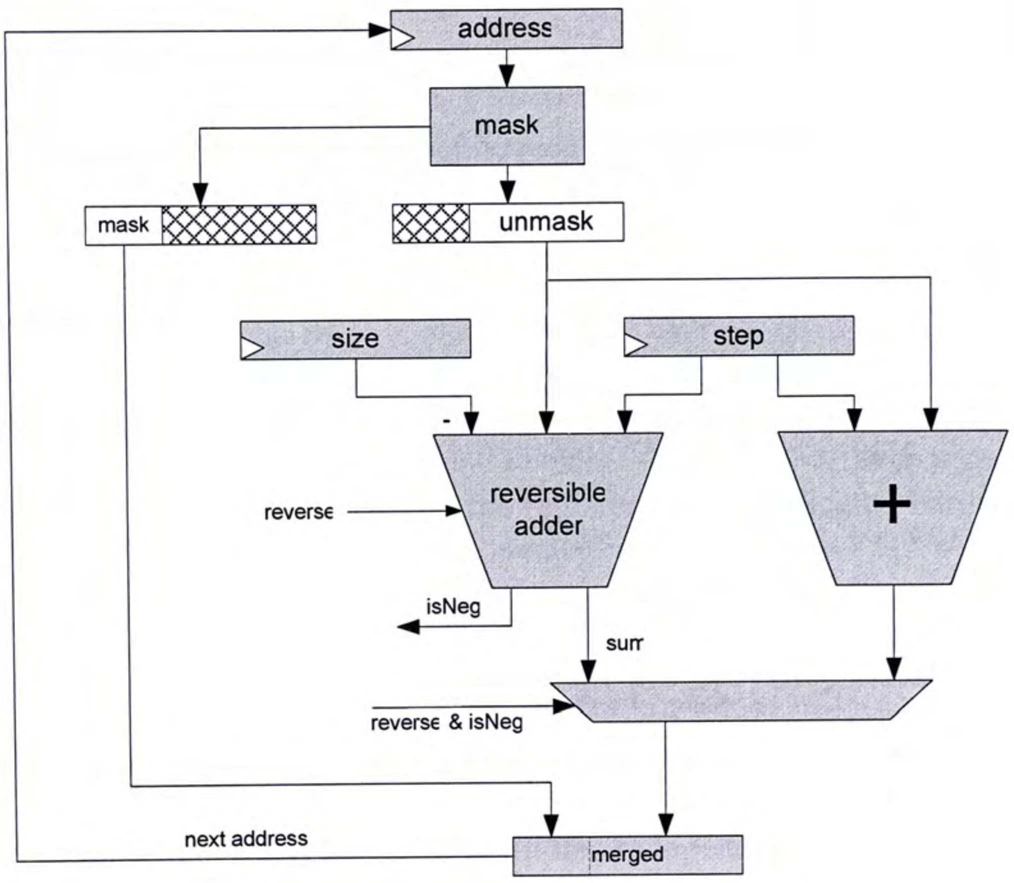


Figure 5.16: The datapath for address generation

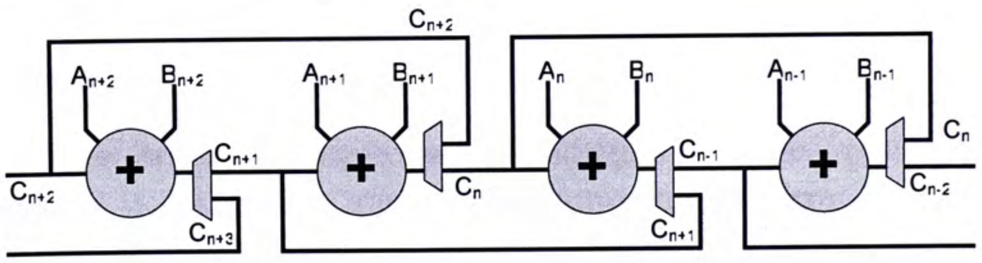


Figure 5.17: The modified carry propagate adder

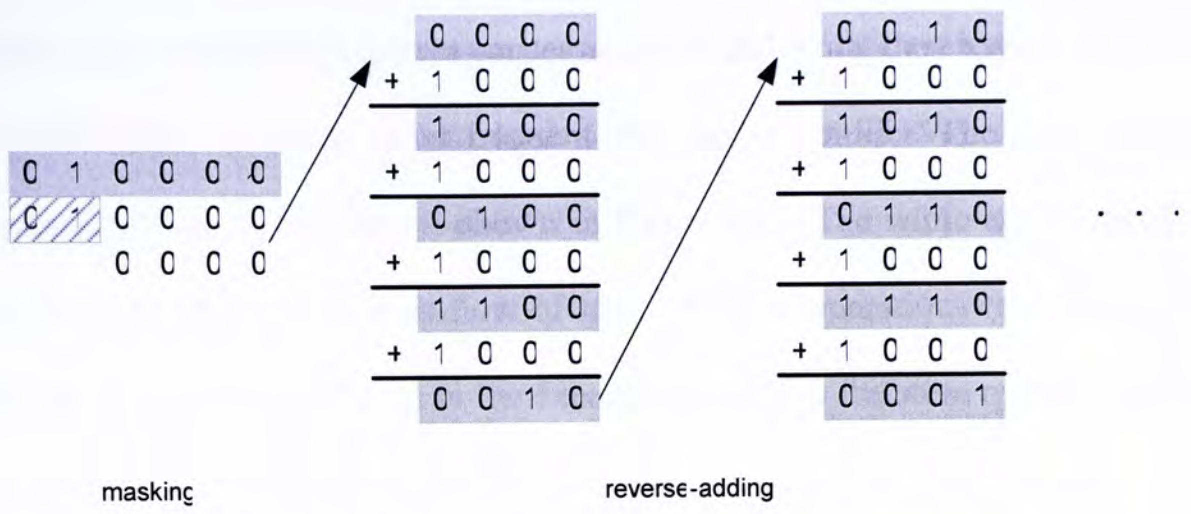


Figure 5.18: An example of the generation of 16-bit bit-reversed address

5.5.4. Load and Store

Load and store are the two operations for accessing the data memory, and they are the only bridge between the data memory and the register file. A dedicated hardware, the load store unit is responsible for the tasks in transferring data from data memory to register file and vice versa. The duties of the load store unit are providing addresses, generating control signals and dispatching the fetched data. To accommodate the X-Y bank organization of the memory, the load store unit has independent address datapaths for the two banks. For each bank, there are one address datapath for memory, one for reading from register file and one for writing to register file. The address datapaths are the same as those in address generation unit and the special registers attached to the address datapath are also organized as in Appendix C.

The load store unit is designed to transfer data between data memory and register file in bulk. Its operation is decoupled from the datapath, therefore they can work in parallel unless there are conflicts. To cooperate with the address generation unit, the load store unit has lower priority in gaining access to the register file. It is restricted to using only the unused read ports and write ports after the port assignment to the address generation unit.

As each load or store operation involves successive read and write, they have to be fit well into the pipeline organization to avoid frequent stall due to conflict. The partitioning of load and store operation in pipeline is shown in Figure 5.19. The white and black broad arrows represent the data and address flow of store and load operations respectively. We can see that the load operation begins in the execution stage but the store operation begins in the read stage. The main reason for this arrangement is to group the access to data memory in one pipeline stage, so that there can be no conflict when a load operation is followed by a store operation immediately and vice versa. For the programmers' point of view, the access to register file is aligned to other data processing operations, which is convenient to predict and control the time of data arrival at and from register file, especially when doing load store operation together with parallel instructions.

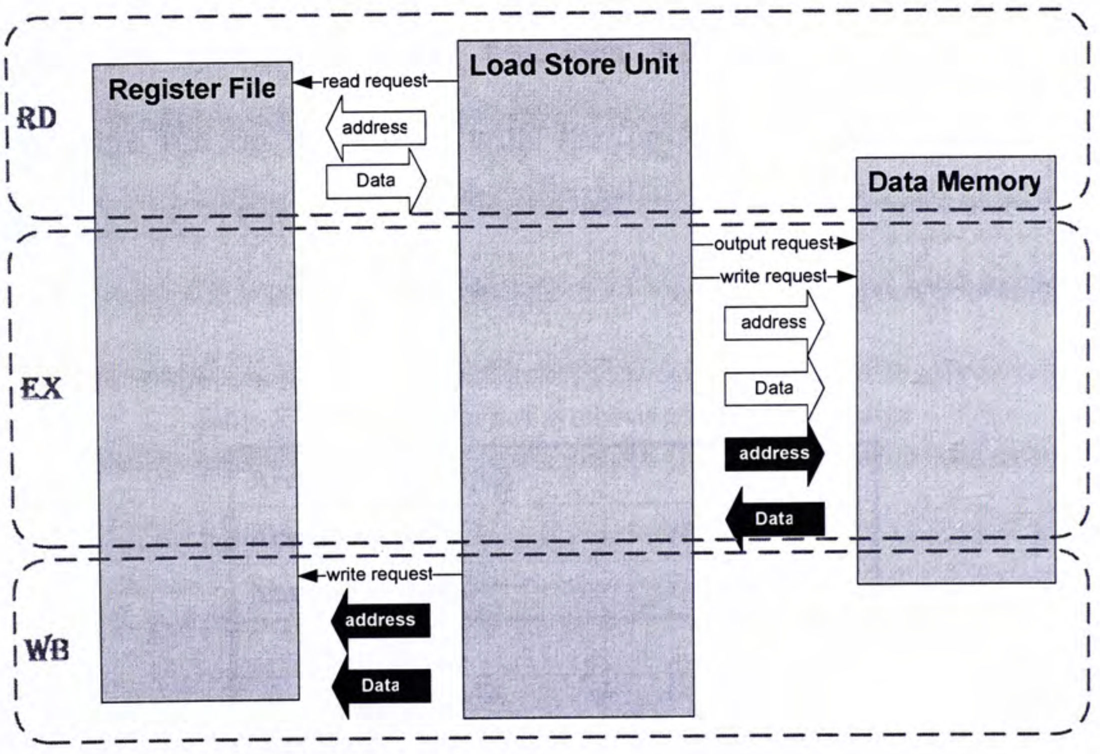


Figure 5.19: The partitioning of load and store operation in pipeline

5.6. Design Verification

The base platform was implemented for verification. The focus of the verification is on the timing and the functionality of the micro-architecture of the base platform, therefore the GALS interfaces were removed in order to obtain the exact timing and to avoid non-determinism due to asynchronous interfaces.

The design was modelled in synthesizable Verilog and synthesized in *Design Compiler*. The physical design was performed in *Silicon Ensemble* with 0.35μm 4 metal 2 poly technology. The results of synthesis and physical design are summarized in Table 5.5. The silicon-ready design was under two tests: static timing analysis (STA) and dynamic simulation. The former one was done using *PrimeTime*. For the latter test, some small programmes were written and their functions are listed in Table 5.6. These test programmes were run on the design with *Verilog-XL*. The simulation environment is introduced in the latter section.

Table 5.5: The summary of synthesis and physical design

Area (NAND2 equ.)	*34K*
Area (μm×μm)	*2600×2600*
Number of Pads	*120*
- Clock	*1*
- Power	*8*
- Address	*48*
- Data	*56*
- Control	*7*

Table 5.6: Test programmes for the base platform

Program	Function
flow0	Test of jump and conditional branching
flow1	Test of zero-overhead loop and subroutine call
load	Test of loading to register file and accumulator
store	Test of storing from register file and accumulator
arith	Test all arithmetic instructions
address	Test of different address generation mode
FIR16	16 taps FIR filter

Dynamic Simulation Environment

Unlike ASIC design, microprocessor-based architecture involves apparently infinite states which can be defined by software. It is impossible to test it heuristically with a long list of test vectors. Having real programmes for verification is more sensible. A hardware and software co-simulation technique is deployed to fit the need. The co-simulation environment is illustrated in Figure 5.20. Conventionally, a simulation only involves the device under test (DUT) and the testbench. The testbench is responsible for supplying control signals, such as clock and reset, and test vectors. Based on the testbench, the simulator calculates the response of the DUT, and records the results as waveforms in database. In this approach, the DUT takes a passive role. The whole simulation is steered by the testbench, which is difficult to simulate the branching and subroutine calls activities of a processor. In the co-simulation approach, the testbench is responsible for the control signals only. The test vectors are stored separately in programme memory model and data memory model as test programme and test data respectively. The memory model emulates the physical memories. Based on the addresses supplied by the DUT, the memory models return the corresponding test vectors (instructions and data) to the DUT, hence, the DUT can steer the simulation, which is useful for simulating branching and subroutine calls. To prepare the programme memory model, the test programme is first compiled to machine codes, then the codes are included in the memory model template as seen in Figure 5.21. Similarly, the data vectors are also included in another memory model instance for preparing the data memory model.

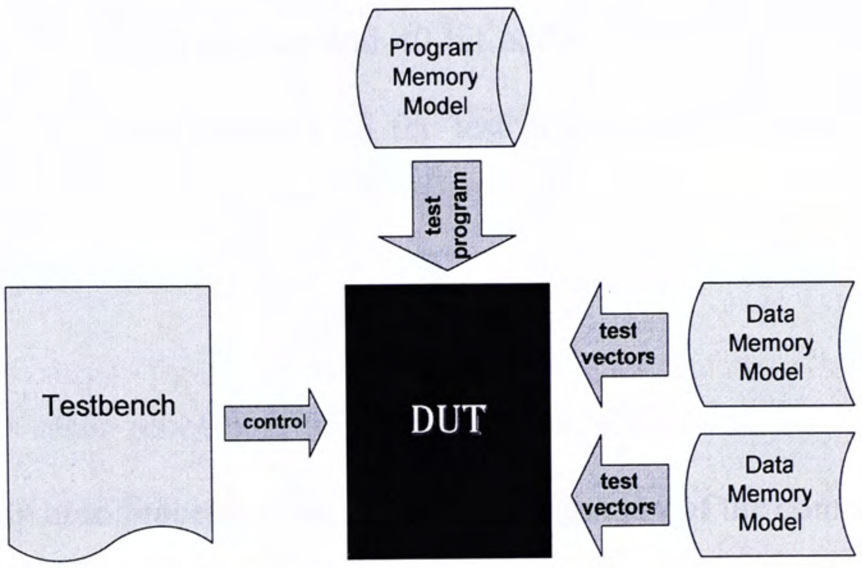
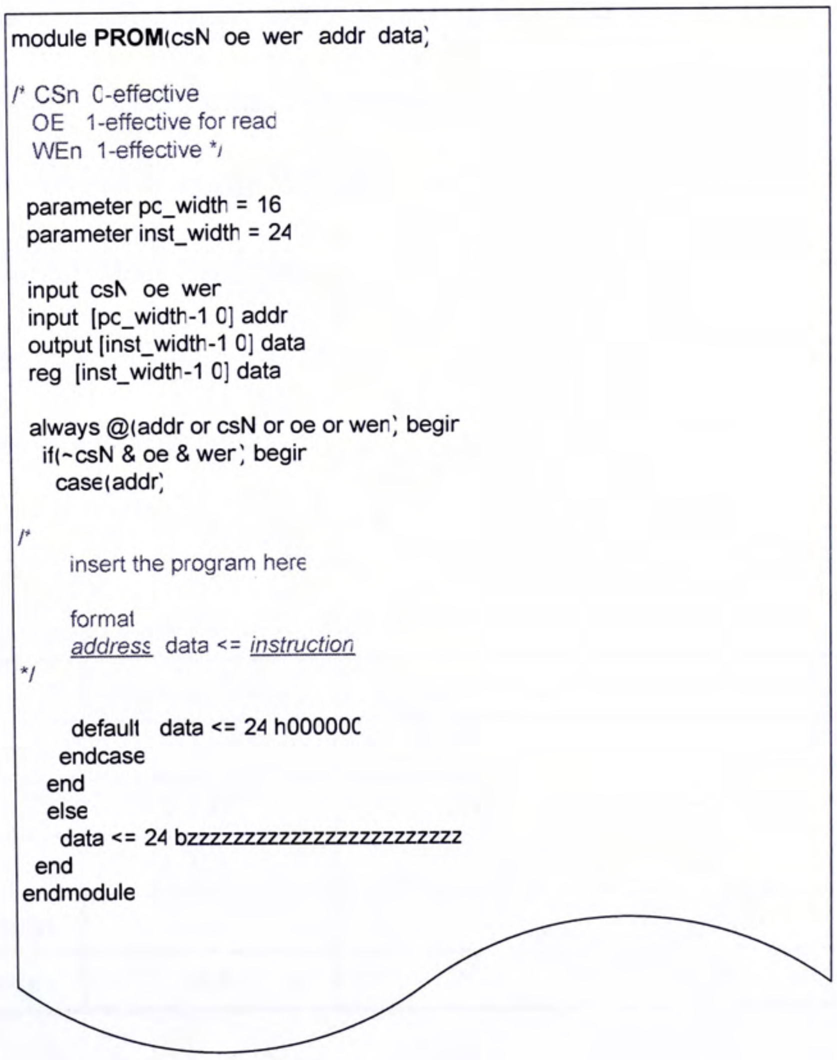


Figure 5.20: Co-Simulation Environment



```
module PROM(csN  oe  wer   addr  data)

/* CSn  0-effective
   OE   1-effective for read
   WEn  1-effective */

 parameter pc_width = 16
 parameter inst_width = 24

 input  csN  oe  wer
 input  [pc_width-1 0] addr
 output [inst_width-1 0] data
 reg  [inst_width-1 0] data

 always @(addr or csN or oe or wen) begir
  if(~csN & oe & wer) begir
   case(addr)

/*
     insert the program here

     format
     address  data <= instruction
*/

     default   data <= 24 h00000C
    endcase
   end
   else
    data <= 24 bzzzzzzzzzzzzzzzzzzzzzzzz
  end
endmodule
```

Figure 5.21: The memory model template

Results

According to the results from STA, the maximum operating frequency of the base platform is 86 MHz. The critical path is at the datapath, from the operand selection network, thru the 16×16 multiplier and 40-bit adder/substracter to the accumulator. On the other hand, the platform passed all the test programmes without any functional or timing error found.

A comparison to other processors of the similar class is made. The information of the other three widely used processors is from [48]. A summary of the comparison is shown in Table 5.7. The StrongARM has the worst performance due to the lack of a single-cycle multiplier. It has to take many instructions thus cycles to accomplish the kernel in sequence. The two TI DSPs show superior computational power in this benchmark. They can execute a FIR tap in one cycle. In contrast, our design needs two cycles for a FIR tap, which is apparently worse than the TI DSPs. An extra cycle is spent on loading data to the register file. The load store architecture of our design is beneficial when the memory becomes the bottleneck of the performance. This advantage can overweigh the cost when scaling to more advanced technologies.

Table 5.7: FIR Benchmark Results

Processor	StrongARM	TMS320C2x	TMS320LC54x	Base Platform
Technology (μm)	*0.35*	*0.72*	*0.6*	*0.35*
Frequency (MHz)	*169*	*20*	*40*	*86*
# of Multipliers	*0.5*	*1*	*1*	*1*
Throughput (cycle/tap)	*17*	*1*	*1*	*2*
Speed (M tap/second)	*9.9*	*20*	*40*	*43*

5.7. Summary

Base platform is the intermediate design. It is a good starting point for designing an optimized processor for particular application. The design goal is to maximize the degree of reusability by 1) minimizing the extent of modification needed for the base platform; 2) minimizing the impact on timing and the power consumption when changing the architectural parameters. Therefore, a flexible instruction encoding scheme, highly modulized datapath organization, scaleable instruction decoder and register file are designed to fulfil the requirement of complex and parallel instructions.

The organization of the pipeline is open to the programmers who are responsible for resolving any conflict in the pipeline. This approach can simplify the design by moving the conflict resolver from hardware to software, which can enable easier modification to the base platform. On the other hand, advanced programmers can make use of the delay slots to fully optimize the target application. A powerful compiler is of paramount importance in this case. Similar functionalities are available in the compiler of VLIW series of Texas Instruments, therefore, this approach is believed to be practical.

Finally, a benchmark shows that the performance of the base platform is comparable to other similar class DSPs. That means our design has demonstrated a good tradeoff point between performance and the architectural flexibility.

6. CASE STUDIES

6.1. Objective

For proving the feasibility of the proposed design methodology, two case studies were carried out. The objective of the case studies is to evaluate the impact of datapath optimization. The subjects are three well-known kernels which are widely used in sound and video processing domains, such as speech codec and digital video broadcasting.

6.2. Approach

A comparison of the base platform and the optimized one is given out. The measure focuses on the reduction of the average number of cycles needed to complete a task. The cycle used in the comparison is normalized to the cycle of the base platform. If the period of cycle of the optimized platform is different, it is presented as a fraction of a normalized cycle. In addition, the area overhead is also measured for estimating silicon cost. The measure of the area is normalized to equivalent area of NAND2 logic.

A comparison of the performance among the optimized processor and two advanced commercial DSPs is also provided. The comparison is based on the study of the instruction set architecture (ISA) of the two DSPs. To be fair and to avoid the influence from the compiler, the algorithm is programmed and optimized for the DSPs by hand. As the DSPs may have different strategies on interfacing memories resulting in different memory hierarchies, the effect of memory access is too complicated to be compared and it should be excluded when evaluating the datapath. Therefore, memory access is assumed to be accomplished in one cycle in the following study.

6.3. Based versus Optimized

6.3.1. Matrix Manipulation

This case is about the vector multiplication of two matrices. The general form can be written as:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & & a_{0n} \\ a_{10} & a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{20} & a_{21} & a_{22} & a_{23} & & a_{2n} \\ a_{30} & a_{31} & a_{32} & a_{33} & & a_{3n} \\ & \vdots & & & \ddots & \vdots \\ a_{m0} & a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & & b_{0m} \\ b_{10} & b_{11} & b_{12} & b_{13} & \cdots & b_{1m} \\ b_{20} & b_{21} & b_{22} & b_{23} & & b_{2m} \\ b_{30} & b_{31} & b_{32} & b_{33} & & b_{3m} \\ & \vdots & & & \ddots & \vdots \\ b_{n0} & b_{n1} & b_{n2} & b_{n3} & \cdots & b_{nm} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} + \cdots + a_{0n}b_{n0} & \cdots & a_{00}b_{0m} + a_{01}b_{1m} + \cdots + a_{0n}b_{nm} \\ \vdots & \ddots & \\ a_{m0}b_{00} + a_{m1}b_{10} + \cdots + a_{mn}b_{n0} & & a_{m0}b_{0m} + a_{m1}b_{1m} + \cdots + a_{mn}b_{nm} \end{bmatrix}$$

Each element in the resultant matrix is the scalar product of a row from the first matrix and a column from the second matrix. This operation involves an enormous amount of multiplications and additions. For the multiplication of an m×n and an n×m matrix, the calculation of one element needs n+1 multiplications and n additions. To enhance the computational performance, instruction level parallel is explored. Obviously, there is no dependence among the multiplications and also among the additions. It is possible to introduce an instruction for n+1 parallel multiplications and one for n parallel additions. For the consideration of practical implementation, the introduced instructions are partitioned into three instructions: 1) parallel execution of four multiply-and-accumulate; 2) parallel execution of four multiplications; and 3) parallel execution of four additions. The corresponding optimized datapath is shown in Figure 6.1.

To implement the optimized datapath, four MACs are used as seen in Figure 6.2. These MACs includes bypass connections in order to perform separated multiplication and addition operations. Having the three new instructions, the platform can perform vector-like operations to the data set for taking advantage of instruction level parallelism.

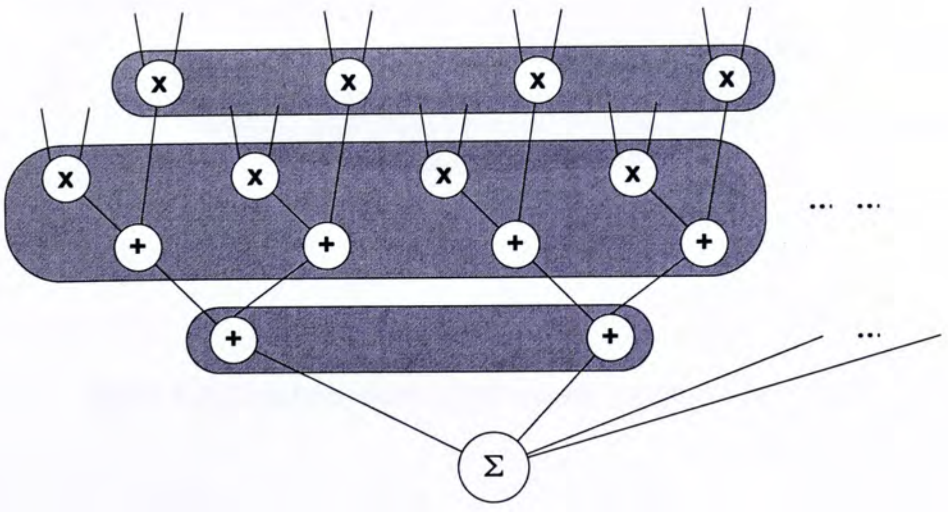


Figure 6.1: The partitioning of an optimized datapath

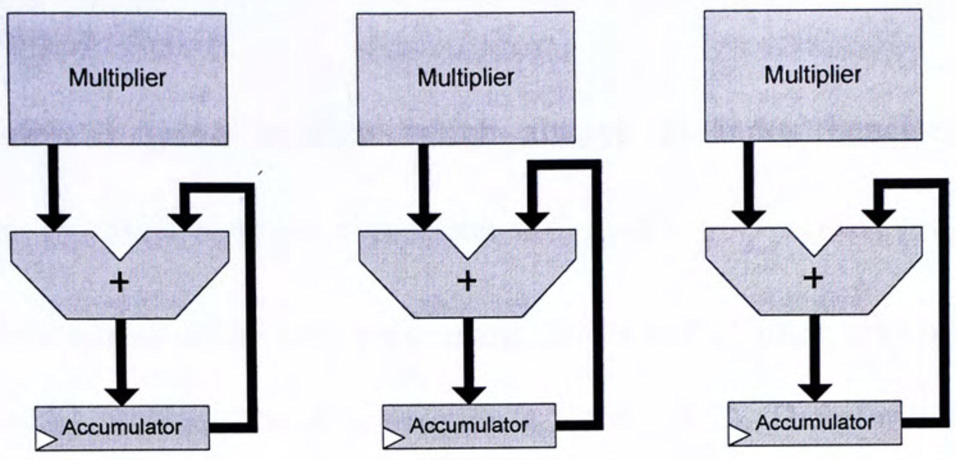


Figure 6.2: The implementation of the optimized datapath for vector multiplication

For evaluation purposes, vector multiplication of two 6×6 matrices is examined. The results are shown in Table 6.1. The loading distribution of the optimized task is shown in Figure 6.3.

Table 6.1: Results of the vector multiplication benchmark

Implementation	Base	**Optimized**
Area (NAND2 equ.)	*34k*	***52k***
Cycles per matrix	*364*	***222***

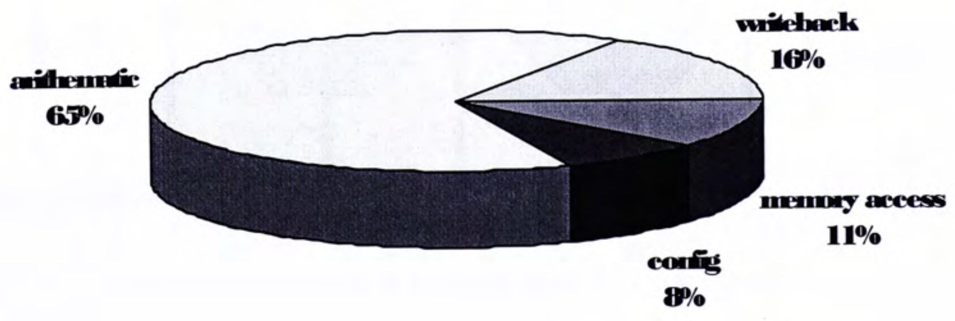


Figure 6.3: Task breakdown of optimized vector multiplication

6.3.2. Autocorrelation

Another case is autocorrelation which always an index benchmark for DSP evaluation. The mathematical expression is $C_{xx}(m) = \frac{1}{N} \sum_{i=0}^{N-1-m} x(i)x(i+m)$. The nature of this kernel and its data access pattern are similar to FIR filter, which are elaborated in detail in the previous chapters (section 4.2.1 & 5.5.2). Therefore, it was decided not to do a brief introduction here.

To optimize autocorrelation, software pipelining is preferable because it is less memory bandwidth hungry. A datapath for optimizing a 64-point autocorrelation is suggested as shown in Figure 6.4. One additional instruction is needed to command the software pipelined serial multiply-and-accumulate operations. The corresponding benchmark results and the breakdown of the optimized task are shown in Table 6.2and Figure 6.5 respectively.

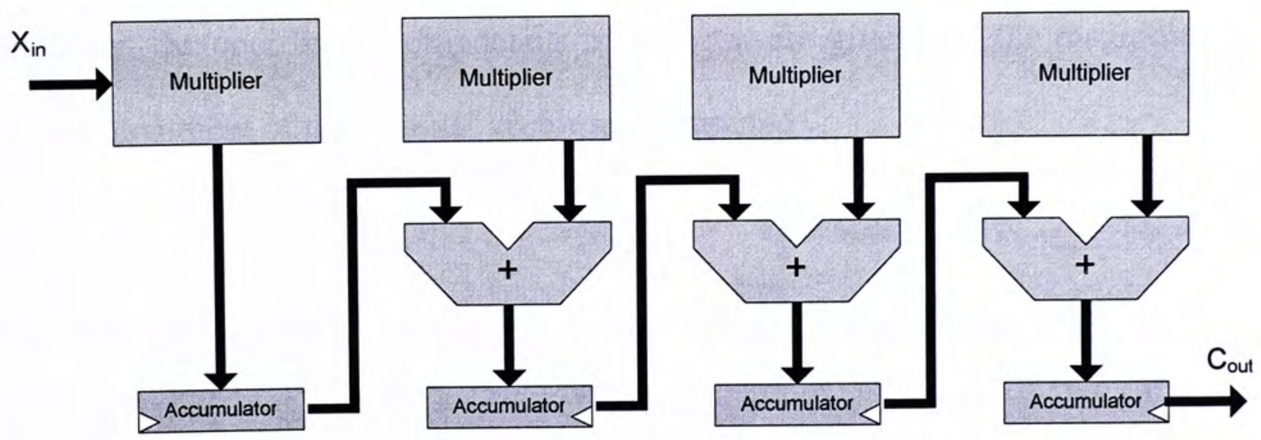


Figure 6.4: The implementation of the optimized datapath for autocorrelation

Table 6.2: Results of the autocorrelation benchmark

Implementation	Base	**Optimized**
Area (NAND2 equ.)	*34k*	***57k***
Ave. cycles per 64 points	*4226*	***1070***

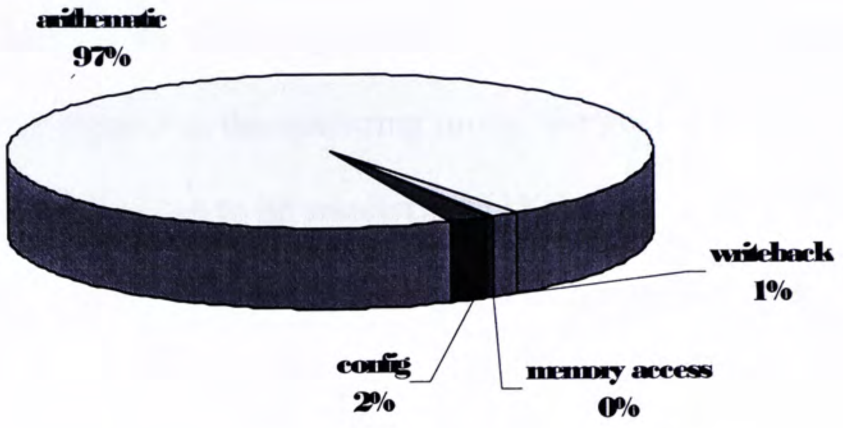


Figure 6.5: Task breakdown of optimized autocorrelation

6.3.3. CORDIC

The COordinate Rotation DIgital Computer (CORDIC) algorithm for trigonometric computing was first introduced by Volder [49]. There are two computing modes, rotation and vectoring. In the rotation mode, the coordinate components of a vector and an angle of rotation are given and the coordinate components of the original vector, after rotation through the given angle, are computed. In the second mode, vectoring, the coordinate components of a vector are given and the magnitude and angular argument of the original vector are computed.

The CORDIC algorithm computes iteratively on the following equations:

$$X_{i+1} = X_i \mp 2^{-i} Y_i$$
$$Y_{i+1} = Y_i \pm 2^{-i} X_i$$
$$\theta_{i+1} = \theta_i \mp \tan^{-1}(2^{-i})$$

Where X_i and Y_i are the two coordinate components in the plane coordinate system, and θ_i is the angle of rotation.

In the rotation mode, the goal of the each operation is to let $\theta_N \rightarrow 0$ in N iteration, thus $|\theta_{i+1}| < |\theta_i|$. Therefore, the upper signs of the three equations are chosen if $\theta_i > 0$, otherwise the lower signs. For the vectoring mode, the goal is to obtain $Y_N \rightarrow 0$ in N iterations. Similarly, $|Y_{i+1}|$ has to be smaller than $|Y_i|$ in each iteration. Therefore, if the signs of X_i and Y_i are different, the upper signs are chosen, otherwise the lower signs.

After N iterations, the magnitude of the resultant vector has been increased compared to the start vector by a factor of K_N where $K_N = \prod_{i=0}^{i=N-1} \sqrt{1+2^{-2i}}$. Therefore, the resultant vector in the rotation mode has to be scaled by K_N^{-1} for correct magnitude.

Obviously, CORDIC algorithm is computationally intensive and allows simultaneous calculation of X_{i+1}, Y_{i+1} and θ_{i+1}. However, the domination of control operations prohibits the use of parallel instructions. In this scenario, hardware accelerator is the best candidate for optimization. A suggested accelerator is shown in Figure 6.6.

Practically, setting the number of iterations up to the bit length of the operands can obtain adequate accuracy. Having a fixed number of iterations allows pre-calculation of the scaling factor K, which can reduce the number of multipliers. In contrast to software loop control, a flow controller is built inside the accelerator for handling iterations. This arrangement can take full advantage of GALS design style: 1) the accelerator can operate at full throttle; 2) early completion is possible once the desired accuracy is met. In our design, early completion detection is not implemented. The number of iterations is fixed to 16.

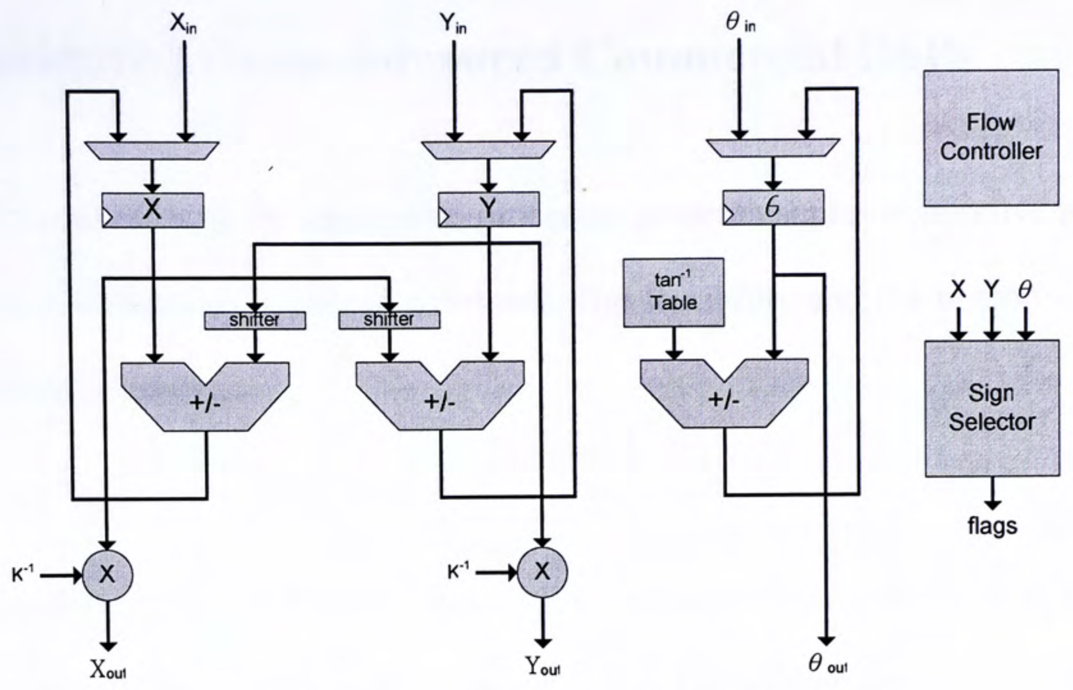


Figure 6.6: The structure of the CORDIC accelerator

Besides the cordic instruction, three more additional instructions are introduced to put the X, Y and θ to the datapath. But they can be encoded using the configuration argument of the complex instruction encoding. A benchmark of Rotation CORDIC algorithm is performed. The results are shown in Table 6.3. The loading distribution of the optimized task is shown in Figure 6.7.

Table 6.3: Results of the Rotation CORDIC benchmark

Implementation	Base	**Optimized**
Area (NAND2 equ.)	*34k*	***41k***
Ave. cycles per CORDIC	*253*	***19.8***

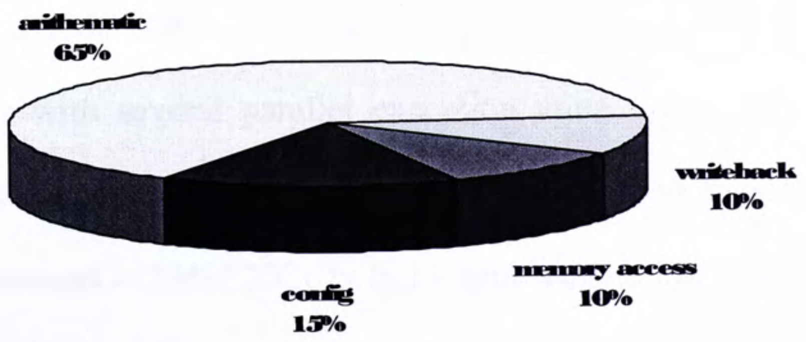


Figure 6.7: Task breakdown of optimized CORDIC

6.4. Optimized versus Advanced Commercial DSPs

In the previous section, the optimized processor presents supreme improvement on performance without great silicon overhead. The flexibility and the potential of the base platform are proven.

In this section, a study among advanced commercial DSPs and the optimized processor is performed. This study is aimed to provide another angle to evaluate the impact of the datapath optimization. TMS320C62x [50] of Texas Instruments Inc and SC140 [51] of Freescale Semiconductor Inc are chosen as the subjects in the study. The reasons are: 1) they are highly optimized to exploit parallelism in DSP applications; 2) they are designed for conveying high performance over general-purpose DSP applications; 3) their architecture details and instruction sets are open to the public. The approach of this study is to compare the performance of the DSPs on the three kernels aforementioned according to their instruction set architecture.

6.4.1. Introduction to TMS320C62x and SC140

Before the comparison, an introduction to TMS320C62x and SC140 is given. The architecture features of TMS320C62x and SC140 is summarized in Table 6.4. Both are VLIW processors with several parallel execution units. They have dedicated hardware for memory transfer and for address generation. For the register file, they have different organizations -- TMS320C62x has a split register file with 32 bits data width, but SC140 has a unified register file with 40 bits data width.

Table 6.4: Processor features of TMS320C62x and SC140

	TMS320C62x [50]	SC140 [51]
Architecture	*VLIW*	*VLIW*
Instruction width	*256 bits*	*128 bits*
Register file size	*2×16×32 bits*	*16×40 bits*
Floating point support	*No*	*No*
# of parallel exec units	*8*	*6*
Parallel memory transfer	*2*	*2*
SIMD support	*No*	*No*
# of multipliers	*2*	*4*
# of ALUs	*4*	*4*
# of Address generation units	*2*	*2*

The datapath of TMS320C62x and SC140 are depicted in Figure 6.8 and Figure 6.9 respectively. These figures illustrate the degree of parallelism of their datapath, in addition the interfaces to data memory are also shown. Obviously, these two DSPs are aimed to provide as much parallelism as possible for getting outstanding performance.

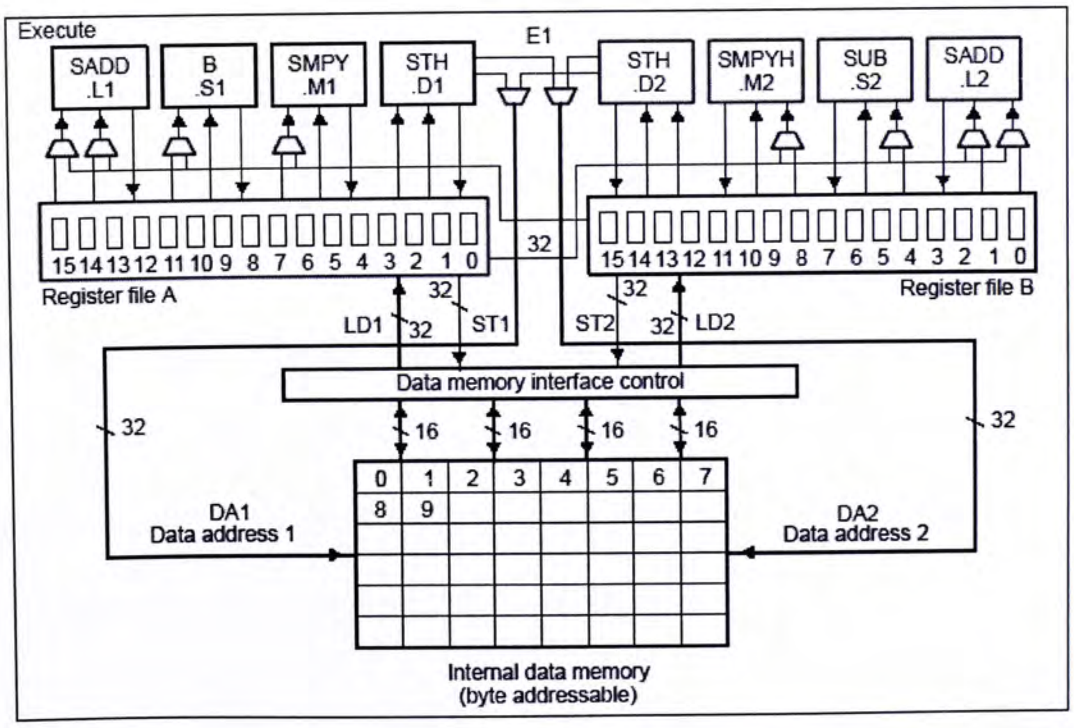


Figure 6.8: Datapath of TMS320C62x (source: [50])

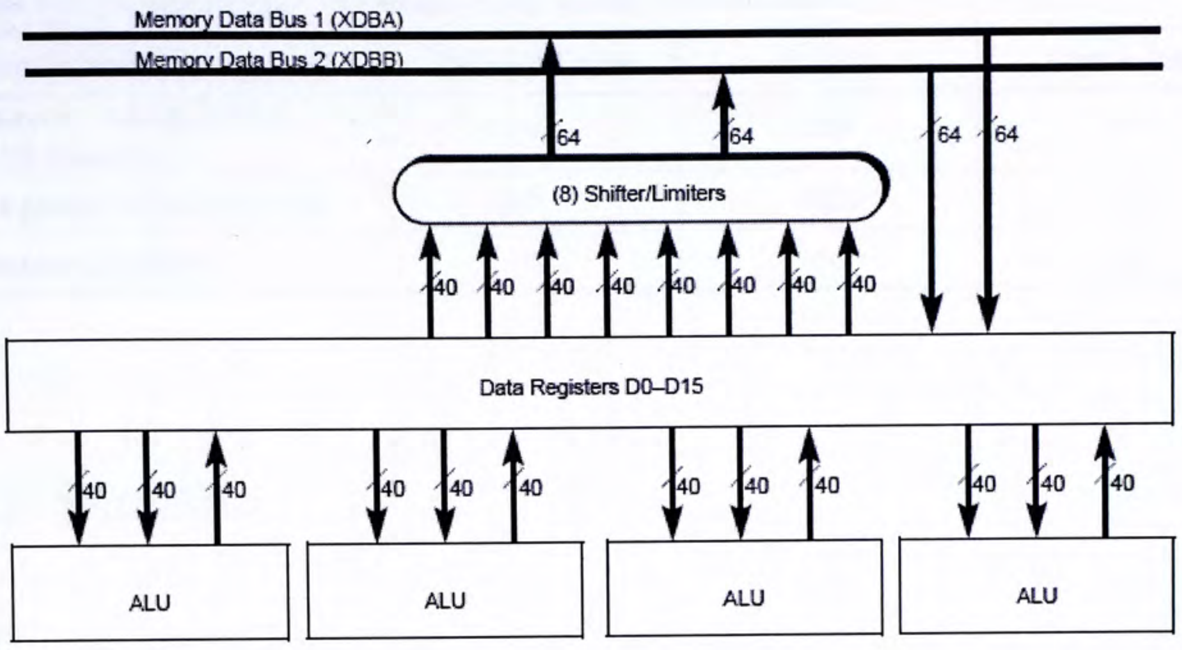


Figure 6.9: Datapath of SC140 (source: [51])

6.4.2. Results

The benchmark tests in the last study are also applied to the subject processors. The tests are: 1) vector multiplication of 6×6 matrices; 2) 64-point autocorrelation; 3) CORDIC algorithm. The results of the benchmark are shown in Table 6.5. The last column is our design.

According to the results, our design has comparable performance to the other two advanced processors in the first two benchmarks. Although our design is scalar in nature, the additional parallel datapath boosts the performance to the level of parallel architectures. For the CORDIC algorithm, our design performs much better than the others. It shows that the CORDIC algorithm hampers the parallelism exploration for parallel architecture. Our design uses dedicated hardware accelerator to handle this task that is always a blind spot in parallel architecture.

Table 6.5: The summary of benchmark results among different DSPs

Benchmark (ave. cycles)	TMS320C62x	SC140	**Optimized**
Vector multiplication of 6×6 matrices	*182*	*169*	***222***
64 points autocorrelation	*2052*	*1028*	***1070***
Rotation CORDIC	*82*	*66*	***19.8***

6.5. Summary

To evaluate our design methodology, case studies for 1) optimized verses base platform and 2) optimized platform versus two commercial DSPs were conducted. The first study demonstrates the ways to optimize the base platform to kernels of different natures. The results are summarized in Table 6.6. In the first case, the kernel is relatively small. The configuration operations occupy up to 8 % of the computation power, which leads to low gain in performance. In the second case software pipeline technique is used. The requirement of memory bandwidth is relaxed, and therefore simultaneous execution of memory access and arithmetic operations is possible. On the other hand, as the loop involved is sufficiently large, the configuration overhead becomes neglectable. Hence the performance gain is up to 400 %. In the last case, dedicated hardware accelerator is used. Using GALS interface, the hardware accelerator runs at its full speed, which is faster than the base platform. Extra performance gain due to asynchronous technique is resulted.

To sum up, the performance gain is of several folds and the silicon overhead is at most 68%. This is due to the contribution of effective design of the flexible platform and the efficient design methodology.

Table 6.6: The summary of the results of benchmark for base and optimized platform

	Vector multiplication of 6×6 matrices	64-point autocorrelation	Rotation CORDIC
# of new instructions	*3*	*1*	*1*
Performance Gain	*164 %*	*395 %*	*1278 %*
Silicon Overhead	*53 %*	*68 %*	*20 %*

The base platform is essentially a scalar architecture.However, after optimization its performance is comparable to advanced commercial VLIW DSPs. The second study compares the performance of TMS320C62x, SC140 and our optimized design. It shows that our optimized design can have performance at the level of these highly parallel architectures without the need of expanding the instruction width. In addition, our design performs several times better in control-dominated kernels, which are always the weakness of VLIW architectures.

These two case studies prove the effectiveness and the efficiency of our design methodology and our base platform. They also show that the optimized processor approaches or even surpasses the performance of today's advanced DSPs.

7. Conclusion

7.1. When ASIPs encounter asynchronous ……

Application-specific instruction-set processors (ASIP) are today's enabling technology for tackling increasing complexities of embedded systems together with tightening time-to-market constraints. It combines the high design productivity of software approach and the high performance of hardware approach, which brings to us programmable devices with dedicated hardware features for real-time constrained applications. A major obstacle of ASIP design is the larger design space compared to pure hardware or pure software implementations. This makes it hard for the designers to search for large amounts of architecture alternatives in order to identify an optimal implementation in competitive design time.

When ASIP design meets asynchronous design methodology, the mindset seems to be changed. We find that searching for alternative architectures has become much easier than before. Thanks to the synchronization mechanism of asynchronous techniques, global timing requirements are broken into timing requirements of local modules. Different architectures can be built by putting different modules together rapidly without worrying about global timing. Designers only have to pay attention to verifying individual modules and their interfaces.

With this in mind, a platform-based design methodology for asynchronous ASIPs is developed. Platform-based design is the design methodology that starts in the middle of the whole process. It is based on the foundation provided by the platform. Using asynchronous techniques, modules can be added on the platform easily to scale up the functionality of the datapath. It is a straightforward way to design a complex system. However, asynchronous technique does not mean all to the platform. Some other design strategies are deployed to ensure the platform has maximum room for optimization and induces less impact on timing and the power consumption during datapath scaling.

The proposed design methodology is proven to be effective in the case studies. It shows that the base platform can be scaled up easily to speed up different kinds of kernels dramatically, which can reach the performance of some advanced parallel DSPs. The benchmark of rotation CORDIC algorithm even illustrates further performance gain by using asynchronous design methodology for seamless cooperation between two different clock domains.

7.2. Contributions

In this thesis, we develop a new design methodology for asynchronous ASIPs. The globally-asynchronous locally-synchronous design style is chosen for this design methodology. A fully synthesized asynchronous wrapper is designed to facilitate rapid development of ASIP. The asynchronous wrapper only introduces 3.1 ns overhead in the communication between two modules.

A highly extensible and flexible base platform is designed based on the study of the nature of DSP applications. The platform is aimed at providing a high degree of parallelism and powerful complex instructions for different DSP applications. A mechanism of conveying parallel instructions and parallel data using 'narrow' width instruction set is devised. To support the parallel datapath, the instruction decoder and the register file are specially designed. The instruction decoder is modulized in order to isolate the changes due to addition of application specific instructions. The register file is highly extensible in terms of size and number of ports. Novel register file organization is developed so that each register bank can keep an acceptable number of ports while expanding the number of global ports. Much effort has been put into strengthening the flexibility and the extensibility of the base platform, but the performance of the base platform is still kept up. The platform operates at 86 MHz and is able to handle 43 filter taps per second, which is comparable to other DSPs of the same class.

Finally, we presented two case studies using three different types of kernels, showing significant performance improvement after datapath optimization, which proves the effectiveness of our design methodology.

7.3. Future Directions

The work presented in this thesis is limited to the hardware part of the ASIP design. The scenarios being considered in development and testing are relatively small pieces of codes or a fraction of application that can be handled by hand. To have a full picture of ASIP design and to further evaluate our design methodology, software generation and EDA tool development can be directions for the future.

As the silicon technology becomes more and more advanced, having multiple processor cores in a chip is increasingly possible. ASIP should not be limited to single core design. As asynchronous design methodology is an excellent technique to interface different components, it is interesting to develop a methodology for designing asynchronous multi-core ASIPs.

A SYNTHESIS OF EXTENDED BURST-MODE ASYNCHRONOUS FINITE STATE MACHINE

The 3D machines of the input- and output-port controller are represented by Karnaugh maps as in Figure A.1 and Figure A.2 respectively. The red arrows in the map indicate the sequential flow of the machines, and the red numbers are the state number. The blank cells can be treated as don't care. Synthesizing these two Karnaugh maps can result in the formula shown in Figure 3.15and Figure 3.16.

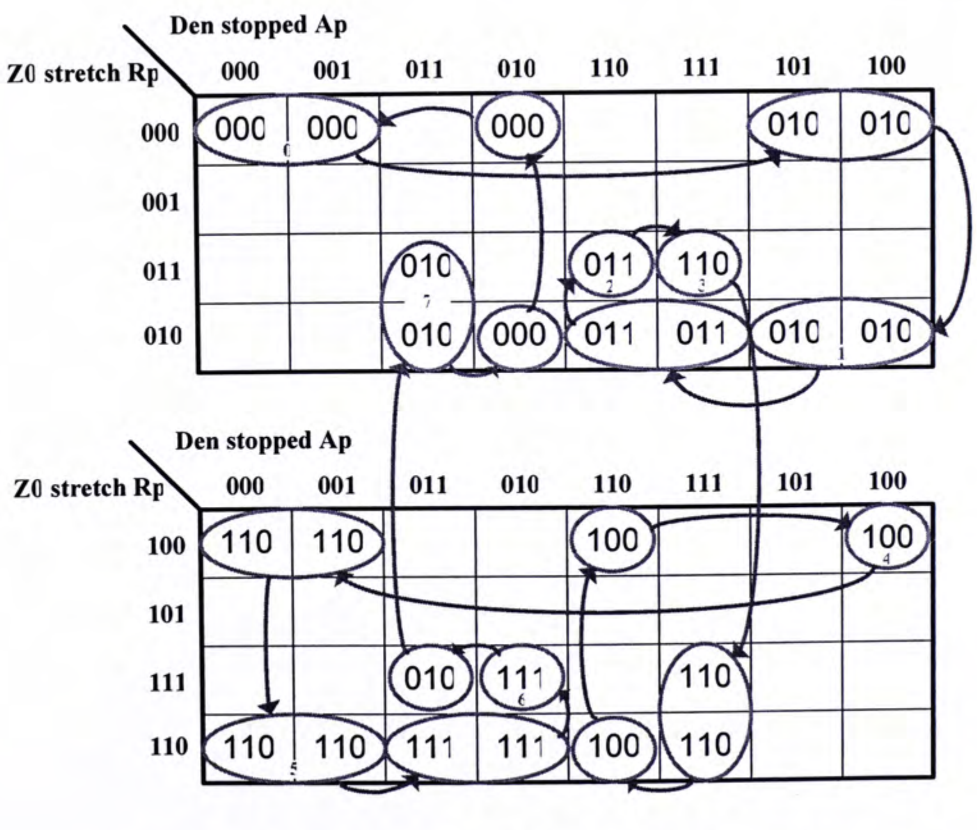


Figure A.1: The 3D machine of the input port controller

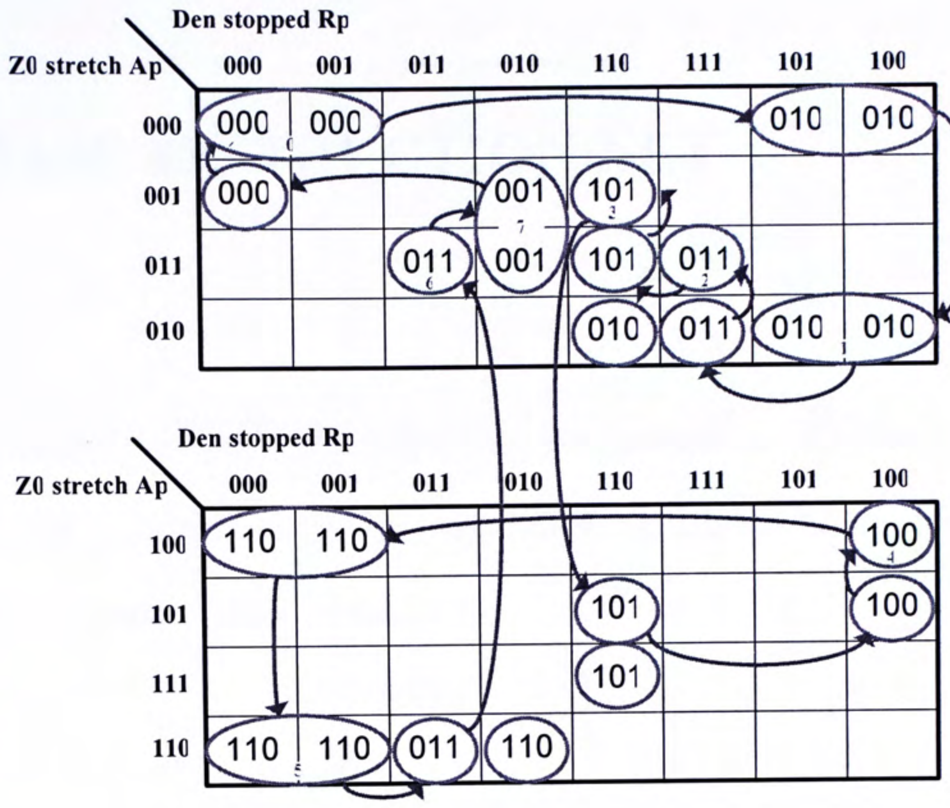


Figure A.2: The 3D machine of the output port controller

B Base Instruction Set

The base instructions can be divided into six categories. All the six kinds of instructions are summarized in the six following tables. Some acronyms and definitions are presented after the tables.

Table B.1: The data processing instructions

Mnemonic	Input → Output	Description
MAC	**(Reg, Reg, Acc) → Acc**	Multiply two values of registers and accumulate
MPY	**(Reg, Reg) → Acc**	Multiply two values of registers
ADD	**(Reg, Reg) → Acc**	Add two values of registers together
SUB	**(Reg, Reg) → Acc**	Subtract one value of registers from another one
ADDC	**(Reg, Reg, Flag) → Acc**	Add two values of registers together with carry
SUBB	**(Reg, Reg, Flag) → Acc**	Subtract one value of registers from another one with borrow
ADDA	**(Reg, Offset, Acc) → Acc**	Add an offset-able value of register to accumulator.
SUBA	**(Reg, Offset, Acc) → Acc**	Subtract an offset-able value of register from accumulator.
NEG	**Acc → Acc**	Invert the sign of the accumulator.
ABS	**Acc → Acc**	Take the absolute value of the accumulator.
EXP	**Acc → SReg**	Determine the exponent of the accumulator.
NORM	**(Acc, SReg) → Acc**	Normalize the accumulator to the exponent stored in the special register.
SH	**(Acc, Reg) → Acc**	Shift the accumulator by the value of the register.
SHK	**(Acc, Value) → Acc**	Shift the accumulator by the immediate value.

Table B.2: The bit manipulation instructions

Mnemonic	Input → Output	Description
NOT	**Acc → Acc**	Bitwise NOT of the accumulator
OR	**(Reg, Offset, Acc) → Acc**	Bitwise OR of the accumulator with an offset-able value from register.
AND	**(Reg, Offset, Acc) → Acc**	Bitwise AND of the accumulator with an offset-able value from register.
XOR	**(Reg, Offset, Acc) → Acc**	Bitwise XOR of the accumulator with an offset-able value from register.

Table B.3: The Boolean operation instructions

Mnemonic	Input → Output	Description
CMP	**(Reg, Reg) → Flag**	Compare two values from registers and assert the condition flag.
CMPACC	**(Reg, Offset, Acc) → Flag**	Compare the accumulator with a value from register and assert the condition flag.

Table B.4: The flow control instructions

Mnemonic	Input → Output	Description
BEQ	**Flag → PC**	Branch if "equal to" flag is asserted.
BNE	**Flag → PC**	Branch if "not equal to" flag is asserted.
BLT	**Flag → PC**	Branch if "less than" flag is asserted.
BGT	**Flag → PC**	Branch if "greater than" flag is asserted.
BLE	**Flag → PC**	Branch if "less or equal to" flag is asserted.
BGE	**Flag → PC**	Branch if "greater or equal to" flag is asserted.
SR	**Value → (PC, stack)**	Subroutine call
JP	**Value → PC**	Unconditional jump
LOOP	**(size, cycle) → (PC, stack)**	Zero-overhead looping
RET	**stack → PC**	Return from subroutine call or break a zero-overhead loop.
NOP	**NA**	No operation

Table B.5: The configuration instructions

Mnemonic	Input → Output	Description
CONF4	**(Value, Pos) → SReg**	Write a nibble to a special register without altering other bits.
CONF8	**(Value, Pos) → SReg**	Write a byte to a special register without altering other bits.
CONF16	**Value → SReg**	Write a word to a special register.

Table B.6: The memory manipulation instructions

Mnemonic	Input → Output	Description
MOV	**Reg → Reg**	Move a register content to another register
LOAD	**Mem → Reg**	Load a value from data memory to register.
STORE	**Reg → Mem**	Store a register content to data memory.
LDACC	**(Value, Pos) → Acc**	Load a word to the higher or lower word of the accumulator.
STACC	**Acc → Reg**	Store the accumulator to register.

Reg – register content

Acc – accumulator content

Flag – status and conditional flags

Offset – shift the value to the left by 16 bits

Offset-able – a value can be set to be offset

SReg – special register content

PC – programme counter

Stack – programme stack

Value – immediate value

Size – the number of instructions within a zero-overhead loop

Cycle – the number of iterations of a zero-overhead loop

NA – not available

Pos – a position of a nibble or a byte in a 16 bits value.

Mem – data memory

C SPECIAL REGISTERS

The organization of special purpose registers is shown in Table C.1. The leftmost column is the addresses of the registers. The acronyms used in the table are presented after the table.

Table C.1: The organization of special purpose registers

	15							8	7		0
0	INT (RESERVED)										
1	ov 40	ov 32	EQ	NE	LT	GT	LE	GE	0	0	EXP
2	LSU XDATA address										
3	LSU XDATA size										
4	LSU XDATA step										
5	LSU YDATA address										
6	LSU YDATA size										
7	LSU YDATA step										
8	0	0	LSU XREG LD address						0	0	LSU XREG LD size
9	0	0	LSU YREG LD address						0	0	LSU YREG LD size
10	0	0	LSU XREG LD step						0	0	LSU YREG LD step
11	0	0	LSU XREG ST address						0	0	LSU XREG ST size
12	0	0	LSU YREG ST address						0	0	LSU YREG ST size
13	0	0	LSU XREG ST step						0	0	LSU YREG ST step
14	LSU Configuration										
15	0	XY	SFU LOP address						0	0	SFU LOP size
16	0	XY	SFU ROP address						0	0	SFU ROP size
17	0	0	SFU LOP step						0	0	SFU ROP step
18	0	XY	SFU WB address						0	0	SFU WB size

19	0	0	0	0	SFU Conf	0	0	SFU WB step
20	PFU0 ~ PFU3 Configuration							
21	0	X Y	PFU0 LOP address			0	0	PFU0 LOP size
22	0	X Y	PFU0 ROP address			0	0	PFU0 ROP size
23	0	0	PFU0 LOP step			0	0	PFU0 ROP step
24	0	X Y	PFU0 WB address			0	0	PFU0 WB size
25	0	0	0	0	PFU0 interval	0	0	PFU0 WB step
26	0	X Y	PFU1 LOP address			0	0	PFU1 LOP size
27	0	X Y	PFU1 ROP address			0	0	PFU1 ROP size
28	0	0	PFU1 LOP step			0	0	PFU1 ROP step
29	0	X Y	PFU1 WB address			0	0	PFU1 WB size
30	0	0	0	0	PFU1 interval	0	0	PFU1 WB size
31	0	X Y	PFU2 LOP address			0	0	PFU2 LOP size
32	0	X Y	PFU2 ROP address			0	0	PFU2 ROP size
33	0	0	PFU2 LOP step			0	0	PFU2 ROP step
34	0	X Y	PFU2 WB address			0	0	PFU2 WB size
35	0	0	0	0	PFU2n interval	0	0	PFU2 WB size
36	0	X Y	PFU3 LOP address			0	0	PFU3 LOP size
37	0	X Y	PFU3 ROP address			0	0	PFU3 ROP size
38	0	0	PFU3 LOP step			0	0	PFU3 ROP step
39	0	X Y	PFU3 WB address			0	0	PFU3 WB size
40	0	0	0	0	PFU3 interval	0	0	PFU3 WB step
⋮								
	PFUn-3 ~ PFUn Configuration							
⋮								
	0	X Y	PFUn LOP address			0	0	PFUn LOP size
	0	X Y	PFUn LOP address			0	0	PFUn ROP size
	0	0	PFUn ROP step			0	0	PFUn ROP step
	0	X Y	PFUn WB address			0	0	PFUn WB size
	0	0	0	0	PFUn interval	0	0	PFUn WB step

INT – interrupt flags. Reserved for future implementation.

OV – overflow flag. The following number indicates the bit location of the happened overflow.

EQ – condition flag. Equal to.

NE – condition flag. Not equal to.

GT – condition flag. Greater than.

LT – condition flag. Less than.

GE – condition flag. Great than or equal to.

LE – condition flag. Less than or equal to.

EXP – exponent value. Obtained after EXP instruction.

LSU – load store unit.

XDATA – X bank data memory.

YDATA – Y bank data memory.

XREG – X bank register file.

YREG – Y bank register file.

LD – load operation.

ST – store operation.

SFU – scalar functional unit. For base and complex instructions.

PFU – parallel functional unit. The following number indicates the ID of the functional unit.

XY – bank selector. One means X bank. Zero means Y bank.

LOP – left hand side operand.

ROP – right hand side operand.

WB – writeback.

Conf – configuration.

D Synthesizable Model of GALS Wrapper

```
module GALS_wrap1_1I1O(rstN, pclk, Den_out, Den_in, Rp_out, Ap_out, Rp_in,
                      Ap_in);

 input  rstN, Den_out, Den_in, Ap_out, Rp_in;
 output pclk, Rp_out, Ap_in;

 wire stretch, stopped;
 wire stretch_out, stopped_out;
 wire stretch_in, stopped_in;

 wire stretchN;
 wire stopped_outN;
 wire stopped_inN;

 pausibleClk1
  U_pclk1(.clk(pclk), .rstN(rstN), .stretch(stretch), .stopped(stopped));

 outport
  U_output(.rstN(rstN), .Den(Den_out), .Rp(Rp_out), .Ap(Ap_out), .stretch(st
  retch_out), .stopped(stopped_out));
 inport
  U_inport(.rstN(rstN), .Den(Den_in), .Rp(Rp_in), .Ap(Ap_in), .stretch(stret
  ch_in), .stopped(stopped_in));

 NOR21 U_NOR21_stretchN(.A(stretch_out), .B(stretch_in), .Q(stretchN));
 INV1  U_INV1_stretch(.A(stretchN), .Q(stretch));

 NAND21 U_NAND21_stopped_outN(.A(stretch_out), .B(stopped), .Q(stopped_outN));
 NAND21 U_NAND21_stopped_inN(.A(stretch_in), .B(stopped), .Q(stopped_inN));
 INV1  U_INV1_stopped_out(.A(stopped_outN), .Q(stopped_out));
 INV1  U_INV1_stopped_in(.A(stopped_inN), .Q(stopped_in));
endmodule


module inport(rstN, Den, Rp, Ap, stretch, stopped);

  input  rstN, Den, Rp, stopped;
  output Ap, stretch;

  wire Z0, Ri, Ai;
  wire nAp, nRp, nAi, nDen, nZ0;

  assign stretch = Ri;
  assign Ai = stopped;

  INV1 U_INV2(.A(Ap), .Q(nAp));
  INV1 U_INV3(.A(Rp), .Q(nRp));
  INV1 U_INV4(.A(Ai), .Q(nAi));
  INV1 U_INV5(.A(Den), .Q(nDen));
  INV1 U_INV6(.A(Z0), .Q(nZ0));

  wire Ri_1, Ri_2, Ri_3;
  NAND33 U_NAND_Ri1(.A(Ri_1), .B(Ri_2), .C(Ri_3), .Q(Ri));
  NAND31 U_NAND_Ri2(.A(Rp), .B(Ri), .C(rstN), .Q(Ri_1));
```

```
NAND31 U_NAND_Ri3(.A(nDen), .B(Z0), .C(rstN), .Q(Ri_2));
NAND41 U_NAND_Ri4(.A(Den), .B(nAp), .D(nZ0), .C(rstN), .Q(Ri_3));

wire Ap_1, Ap_2;
NAND23 U_NAND_Ap1(.A(Ap_1), .B(Ap_2), .Q(Ap));
NAND31 U_NAND_Ap2(.A(Ai), .B(Rp), .C(rstN), .Q(Ap_1));
NAND31 U_NAND_Ap3(.A(Ai), .B(Ap), .C(rstN), .Q(Ap_2));

wire Z0_1, Z0_2, Z0_3;
NAND33 U_NAND_Z01(.A(Z0_1), .B(Z0_2), .C(Z0_3), .Q(Z0));
NAND31 U_NAND_Z02(.A(nRp), .B(Z0), .C(rstN), .Q(Z0_1));
NAND31 U_NAND_Z03(.A(nAi), .B(Z0), .C(rstN), .Q(Z0_2));
NAND41 U_NAND_Z04(.A(Den), .B(Ap), .D(nRp), .C(rstN), .Q(Z0_3));
endmodule

module outport(rstN, Den, Rp, Ap, stretch, stopped);

 input  rstN, Den, Ap, stopped;
 output Rp, stretch;

 wire Z0, Ai, Ri;
 wire nRi, nAp, nRp, nAi, nDen, nZ0, nrst;

 assign stretch = Ri;
 assign Ai = stopped;

 INV1 U_INV1(.A(Ri), .Q(nRi));
 INV1 U_INV2(.A(Ap), .Q(nAp));
 INV1 U_INV3(.A(Rp), .Q(nRp));
 INV1 U_INV4(.A(Ai), .Q(nAi));
 INV1 U_INV5(.A(Den), .Q(nDen));
 INV1 U_INV6(.A(Z0), .Q(nZ0));
 INV1 U_INV7(.A(rstN), .Q(nrst));

 wire Ri_1, Ri_2, Ri_3;
 NAND33 U_NAND_Ri1(.A(Ri_1), .B(Ri_2), .C(Ri_3), .Q(Ri));
 NAND31 U_NAND_Ri2(.A(nZ0), .B(Den), .C(rstN), .Q(Ri_1));
 NAND31 U_NAND_Ri3(.A(Ri), .B(Ap), .C(rstN), .Q(Ri_2));
 NAND31 U_NAND_Ri4(.A(Z0), .B(nDen), .C(rstN), .Q(Ri_3));

 wire Rp_1, Rp_2, Rp_3, Rp_21, Rp_22, Rp_31, Rp_32;
 NAND33 U_NAND_Rp1(.A(Rp_1), .B(Rp_2), .C(Rp_3), .Q(Rp));
 NAND41 U_NAND_Rp2(.A(Rp), .B(Ai), .C(nAp), .D(rstN), .Q(Rp_1));
 NAND21 U_NAND_Rp3(.A(Rp_21), .B(Rp_22), .Q(Rp_2));
 NAND21 U_NAND_Rp4(.A(Rp_31), .B(Rp_32), .Q(Rp_3));
 NOR41 U_NOR_Rp5(.A(Z0), .B(nRi), .C(Rp), .D(nrst), .Q(Rp_21));
 NOR31 U_NOR_Rp6(.A(nDen), .B(nAi), .C(nrst), .Q(Rp_22));
 NOR31 U_NOR_Rp7(.A(nZ0), .B(nRi), .C(nrst), .Q(Rp_31));
 NOR41 U_NOR_Rp8(.A(Rp), .B(Den), .C(nAi), .D(nrst), .Q(Rp_32));

 wire Z0_1, Z0_2, Z0_3, Z0_4;
 NAND43 U_NAND_Z01(.A(Z0_1), .B(Z0_2), .C(Z0_3), .D(Z0_4), .Q(Z0));
 NAND31 U_NAND_Z02(.A(Z0), .B(Den), .C(rstN), .Q(Z0_1));
 NAND31 U_NAND_Z03(.A(Z0), .B(nAp), .C(rstN), .Q(Z0_2));
 NAND31 U_NAND_Z04(.A(Z0), .B(nRp), .C(rstN), .Q(Z0_3));
 NAND41 U_NAND_Z05(.A(Rp), .B(Den), .C(Ap), .D(rstN), .Q(Z0_4));

endmodule

module pausibleClk1(clk, rstN, stretch, stopped);

 input  rstN, stretch;
 output clk, stopped;

 wire clk_delayed, clk_in;

 delay1 U_delay1(.i(clk), .o(clk_delayed));
```

```
me
 U_me(.request1(clk_in), .request2(stretch), .grant1(clk), .grant2(stopped));

 assign clk_in = ~(clk_delayed|~rstN);

endmodule

module me(request1, request2, grant1, grant2);

 input  request1, request2;
 output grant1, grant2;

 wire n1, n2;

 wire tied_low, tied_high;
 assign tied_low = 1'b0;
 assign tied_high = 1'b1;


 NAND21 U_NAND1(.A(request1), .B(n2), .Q(n1));
 NAND21 U_NAND2(.A(request2), .B(n1), .Q(n2));

 MUX22 U_MUX2_1(.A(n2), .B(tied_low), .S(n1), .Q(grant1));
 MUX22 U_MUX2_2(.A(n1), .B(tied_low), .S(n2), .Q(grant2));

endmodule

module delay1(i, o);

 input  i;
 output o;

 DLY12 U_DLY1(.A(i), .Q(o));

endmodule
```

REFERENCE

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, vol. 38, no. 8, 1965

[2] C. Ttistram, "It's Time For Clockless Chips," *MIT's Technology Review magazine*, October 2001

[3] B. Cole, "Asynchronous design gets a second look," *EE Times*, 6 June 2003.

[4] P. A. Beerel, "Asynchronous circuits: an increasingly practical design solution," *Proceedings of International Symposium on Quality Electronic Design 2002*, pp. 367 - 372, 18-21 March 2002.

[5] Tensilica Inc., Xtensa V – A Proven Configurable Processor, http://www.tensilica.com/html/xtensa_v.html

[6] R. E. Gonzalez; "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20 , issue. 2, pp. 60 – 70, March-April 2000

[7] P. Kievits, E. Lambers, C. Moerman and R. Woudsma, "R.E.A.L. DSP Technology for Telecom Baseband Processing," *Proceedings of ICSPAT 1998*

[8] ARC Cores Ltd. ARCtangent Processor, http://www.arccores.com

[9] Improv Systems Inc., Jazz PSA/Jazz DSP, http://www.improvsys.com

[10] Target Compiler Technologies, Chess/Checkers is a retargetable tool-suite, http://www.retarget.com/products-more.html

[11] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink,H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, issue. 11, pp. 1338 – 1354, November 2001

[12] Institute of Integrated Signal Processing Systems, Aachen University of Technology, Germany, LSIA Processor Design Platform, http://www.iss.rwth-aachen.de/lisa/lpdp.html

[13] J. H. Yang, B. W. Kim, S. J. Nam, Y. S. Kwon, D. H. Lee, J. Y. Lee, C. S. Hwang, Y. H. Lee, S. H. Hwang, I. C. Park and C. M. Kyung, "MetaCore: an application-specific programmable DSP development system," *IEEE Transactions on VLSI Systems*, vol. 8 , issue. 2 , pp. 173 – 183, April 2000

[14] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima and M. Imai, "PEAS-III: an ASIP design environment," *Proceedings of International Conference on Computer Design 2000*, pp. 430 – 436, 17-20 Sept. 2000

[15] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple, "AMULET1: an asynchronous ARM microprocessor,' *IEEE Transactions on Computers*, vol. 46 , issue. 4 , pp. 385 – 398, April 1997

[16] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day and N. C. Paver, "AMULET2e: an asynchronous embedded controller," *Proceedings of Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 290 – 299, 7-10 April 1997

[17] S. B. Furber, D. A. Edwards and J. D. Garside, "AMULET3: a 100 MIPS asynchronous embedded processor," *Proceedings of International Conference on Computer Design 2000*, pp. 329 – 334, 17-20 September 2000

[18] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80C51 microcontroller," *Proceedings of Forth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 96 - 107 , April 1998

[19] H. Terada, S. Miyata, and M. Iwata, "Ddmps: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, vol. 87, issue. 2, February 1999

[20] A.J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings, "The design of an asynchronous MIPS R3000 microprocessor," *Proceedings of Seventeenth Conference on Advanced Research in VLSI*, pp. 164 - 181 , September 1997.

[21] P. Clarke, "University spinouts revive clockless processors," *EE Times*, 25 October 2001

[22] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno and T. Nanya, "TITAC-2: an asynchronous 32-bit microprocessor based on scalable-delay-insensitive model," *Proceedings. of IEEE ICCD '97*, pp. 288 – 294, 12-15 October 1997

[23] A. Takamura, M. Imai, M. Ozawa, I. Fukasaku, T. Fujii, Kuwako, M. Y. Ueno and T. Nanya, "TITAC-2: an asynchronous 32-bit microprocessor," Proceedings of the ASP-DAC '98, pp. 319 – 320, 10-13 February 1998

[24] J. Sparso and S. Furber, "Chapter 2: Fundamentals," *Principles of Asynchronous Circuit Design: A Systems Perspective*, Kluwer Academic Publishers, 2001

[25] C. H. Van Berkel; M. B. Josephs and S. M. Nowick, "Applications of asynchronous circuits,"

Proceedings of the IEEE, vol. 87, issue 2, pp. 223 – 233, February 1999

[26] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, issue 6, June 1989.

[27] K. Y. Yun, "Recent advances in asynchronous design methodologies," *Proceedings of the ASP-DAC '99*, vol.1, pp. 253 - 259, 18-21 January 1999.

[28] C. S. Choy; J. Butas, J. Povazanec, and C. F. Chan, "A fine-grain asynchronous pipeline reaching the synchronous speed," *Proceedings of the 4th International Conference on ASIC 2001*, pp. 547 – 550, 23-25 October. 2001

[29] C. S. Choy; J. Butas, J. Povazanec, and C. F. Chan, "A new control circuit for asynchronous micropipelines," *IEEE Transactions on Computers*, vol. 50, issue. 9, pp. 992 – 997 , September 2001

[30] K. M. Chu and D. L. Pulfrey, "A comparison of CMOS techniques: differential cascade voltage switch logic versus conventional logic," IEEE Journal of Solid-State Circuits, vol. 22 , issue 4, pp. 528 - 532 , August 1987

[31] A.J. McAuley, "Dynamic Asynchronous Logic for High-Speed CMOS Systems, " *IEEE Journal of Solid-State Circuits*, vol. 27, no. 3, pp. 382 – 388, March 1992

[32] M. Renaudin, B.E. Hassan and A. Guyot, "A New Asynchronous Pipeline Scheme: Application to the Design of a Self-Timed Ring Divider," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 7, pp. 1001 – 1013, July 1996

[33] J. L. Yang , C. S. Choy and C. F. Chan, "A self-timed divider using a new fast and robust pipeline scheme," *IEEE Journal of Solid-State Circuits*, vol. 36 , issue 6, pp. 917 – 923, June 2001

[34] J. Butas, C. S. Choy; J. Povazanec, and C. F. Chan, "Asynchronous cross-pipelined multiplier," *IEEE Journal of Solid-State Circuits*, vol. 36, issue 8, pp. 1272 – 1275, August 2001

[35] C. W. Lee, C. S. Choy, J. Butas and C. F. Chan, "A pipelined dataflow small micro-coded asynchronous processor and its application to DCT," *Proceedings of the* ISCAS 2001, vol. 4, pp. 910 – 913, 6-9 May 2001

[36] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," *Proceedings of 36th Design Automation Conference 1999*, pp. 873 – 878, 21-25 June 1999

[37] J. Muttersbach; T. Villiger and W. Fichtner, "Practical design of globally-asynchronous

locally-synchronous systems," *Proceedings of. Sixth International Symposium on ASYNC 2000* , pp. 52 – 59, 2-6 April 2000

[38] K. Y. Yun, D. L. Dill, "Automatic synthesis of extended burst-mode circuits. I. (Specification and hazard-free implementations)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, issue. 2, pp. 101 – 117, February 1999

[39] "Dialogic ADPCM Algorithm", Dialogic Corporation, 1988,

[40] M. K. Jain, M. Balakrishnan, A. Kumar, "ASIP design methodologies: survey and issues," *Fourteenth International Conference on VLSI Design 2001*, pp. 76 - 81, 3-7 January 2001

[41] W. K. Chan, C. S. Choy, C. F. Chan and K. P. Pun, "An asynchronous SOVA decoder for wireless communication application," *ISCAS 2004,* publishing

[42] P. K. Leung, C. S. Choy, C. F. Chan and K. P. Pun, "A low power asynchronous GF(2^{173}) ALU for elliptic curve crypto-processor," *Proceedings of the ISCAS '03*, vol. 5 , pp. V-337 - V-340, 25-28 May 2003

[43] P. L. Siu, C. S. Choy, J. Butas and C. F. Chan, "A low power asynchronous DES," *Proceedings of the ISCAS 2001*, vol. 4, pp. 538 –541, 6-9 May 2001

[44] J. H. Tseng, K. Asanovic, "Banked multiported register files for high-frequency superscalar microprocessors," *Proceedings of 30th ISCA, 2003*, pp. 62 – 71, 9-11 June 2003

[45] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," *Proceedings of MICRO-35*, November 2002.

[46] V. Zyuban and P. Kogge, "The energy complexity of register files," *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, pp. 305 - 310, August 1998.

[47] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi and J. D. Owens, "Register organization for media processing," *Proceedings of International Symposium on HPCA-6*, pp. 375 - 386 , 8-12 January 2000

[48] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, J. Rabaey, "Evaluation of a Low-Power Reconfigurable DSP Architecture" *Proc. of the Reconfigurable Architecture Workshop,* Orlando, Floride, USA, March 1998

[49] J. E. Volder, "The CORDIC Trigonometric Computing Technique", *IRE Transaction on Electronic Computers,* EC-8, p.330-334, 1959.

[50] *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Texas Instruments Inc.

[51] *SC140 DSP Core Reference Manual*, Freescale Semiconductor, Inc.