Design and Modeling of Mixed Synchronous-Asynchronous and Hardware-Software Systems

Weiwen Zhu

McGill University, Montreal Canada

December, 2001

A Thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering

0-612-79108-4

To my family

Abstract

This thesis presents the design of a hardware/software co-simulator and a case study in the comparison of synchronous and asynchronous design styles of digital VLSI circuits. Adopting the design pattern approach of software design, our simulator software package, based on PtolemyII, extracts the temporal causality of software in embedded systems to perform fast timing estimation of functionality partitioning of hardware/software in embedded systems. Our package can simulate system features such as task prioritization, message passing, resource sharing and task blocking. We demonstrate the proposed approach by two event-driven software applications. In this thesis we also discuss synchronous and asynchronous design styles of VLSI circuits. We use a CDMA correlator to illustrate the different aspects of these design styles. The comparison is presented in terms of area and power. Meanwhile, we also include a switching activity study for the evaluation of architecture tradeoffs.

Résumé

Cette thèse présente la conception d'un co-simulateur matériel/logiciel et une étude d'un cas de comparaison de styles de conception synchrone et asynchrone pour circuits digitaux VLSI. En adoptant le style 'design pattern' de conception de logiciel, notre module de simulation extrait la causalité temporelle du logiciel d'un système encastré, afin d'y effectuer une estimation de timing rapide de la division fonctionelle entre matériel et logiciel . Notre module peut simuler des charactéristiques du système telles que la prioritization des tâches, le transfert de messages, le partage de ressources, et le bloquage de tâches. Nous démontrons la méthode proposée avec deux applications de logiciels à base d'évènements. Dans cette thèse, nous traîtons des styles de conception synchrones et asynchrones pour des circuits de VLSI. Nous utilisons un corrélateur CDMA pour illustrer les différents aspects de ces styles de conception. La comparaison est faite en termes de dimensions et de consommation de puissance. En parallèle, nous incluons également une étude des activités de commutation afin d'évaluer les compromis au niveau de l'architecture.

Acknowledgements

First, I would like to thank my co-supervisors, Dr. Radu Negulescu and Dr. Zeljko Zilic, for their direction, advice and attention. This work is impossible without their continuous support and encouragement. Particularly, thanks to Dr. Radu Negulescu for his persistent tolerance of reviewing the drafts I wrote and for his ceaselessly pushing me to pursue higher level of quality, to Dr. Zeljko Zilic for his insight and determination in the course of my research. I am very grateful to them for providing me with good study and research facility.

I would like to thank all my fellows at the MACS lab, in particular Xiahua Kong, Nazmy Abaskharoun, Henry Chan, Ian Brynjolfson, Antonio Chan and Clarence Kar-Lun Tam. I wish also to acknowledge Ehab Lotayef, Ben Mihailescu, our system administrators and people at CMC of their support for my CAD support work at the lab.

My study is supported by the CAD tool support job at MACS lab. Dr. Gordon Roberts, Thank you! The fabrication of the VLSI chip is through the grant from CMC (Canadian Microelectronics Corporation).

Last but not least, I like to thank my family, specially, my wife – Jiong Chen – for supporting me throughout this endeavor. Also I like to mention that Bardia Pishdad prepares the translation of the French version of the abstract.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

VLSI technology has shown spectacular capacity improvements during the past decade. Today, a powerful computing system can be built from a few VLSI chips, which are complex systems themselves. The design of such VLSI chips becomes increasingly complex. Currently, the minimal feature size of VLSI designs is already in the deep sub-micron range. Technology of 0.18 micron minimal feature size is used in today's mainstream designs. The designers face more and more challenges to design such complex chips. At system level of VLSI chips, we observe the trend of the shifting the realization of the functionality from hardware to software. Simulating such designs poses new challenges to the designers who use traditional event-driven simulators. At the circuit level, the current digital circuit design paradigm — global synchronous design — is facing difficulties in solving the problem of increasing power consumption and in achieving even higher operation frequency of the circuit. New approaches in circuit design methodologies and circuit design styles are needed to achieve significant further improvements.

1.1 Motivation

This thesis consists of two parts of work related to the issues mentioned above. Results are presented in areas of system level modeling, simulation and design methods of VLSI circuits.

Present day integrated circuit development is facing significant challenges. Complexity of the integrated circuits is growing exponentially, as observed by Moore's law. Meanwhile, time-to-market demands are becoming stricter, while the sizes of the design teams remain essentially the same. Short product life cycles and customization to niche markets force designers to reuse not only building blocks, but also entire architectures. With the ability to mix processors, complex peripherals, custom hardware and software on a single chip at decreasing cost, the designers increasingly choose the embedded system design approach. To deal with complex embedded system design, most of the design effort has to be concentrated at a high level of abstraction, while increasingly reusing existing designs and leveraging design expertise. Forms of reuse include design patterns, component-based designs for object-oriented software and hardware IP (intellectual property) blocks for integrated circuits.

Designers face the options of implementing virtually any functional component of a system in terms of hardware, software, or (more likely) a mixture of hardware and software. The final product cost is often paramount, so the main incentive is to find the right combination of processors, memories and glue logics for efficient production in a short design time. An important part of the design consists of mapping the functionality (from the specification) to the architectural blocks (from IP suppliers and/or software library suppliers) in such a way that cost, power consumption and timing of the system can be analyzed.

Achieving optimal design of complex embedded system is a difficult task. Besides being application specific, such system design also needs to respect real-time constrains of the environment in which the system is operating. Designing and simulating embedded systems is difficult because of their heterogeneous nature. In such systems, software and hardware components must be simulated at the same time. The basic co-simulation problem is two-fold. Firstly, it is desirable to execute the software as fast as possible, often on a host machine that may be faster than the final embedded processor and certainly is quite different from it. Secondly, it is required to keep the hardware and software simulations synchronized, so that they interact just as they will in the target systems. Accordingly, there are different approaches in co-simulation with varying degrees of accuracy and performance. The main approaches can be summarized in the following table according to [EL+97].

Author	Hardware Simulation	Software Simulation	Synchronization Mechanism
Gupta [GCM92]	logic custom	bus-cycle custom	single simulation
Rowson [Row94]	logic commercial	host-compiled	handshake
Wilson [Wil94]	logic commercial	host-compiled	handshake
Thomas [TAS93]	logic commercial	host-compiled	handshake
ten Hagen [HM93]	logic commercial	host-compiled	handshake
ten Hagen [HM93]	cycle-based	cycle-counting	tagged message
Kalavade [KL92]	logic custom	host-compiled	single simulation
Kalavade [KL92]	logic custom	ISA	single simulation
Lee [KL92]	logic custom	host-compiled	single simulation
Suterwala [SP94]	logic commercial	ISA on HW simulation	single simulation

Table 1 : A comparison of Co-Simulation methods

For hardware simulation, the designers use commercial or custom logic simulator. To simulate software, they use either bus-cycle model of the target CPU, or use software compiled on the host processor interacting the hardware simulator via a bus-cycle emulator in the hardware simulator. Alternatively, they can use an ISA processor model augmented with interfaces within a hardware simulator. The synchronization mechanism of these methods can use either single simulation which both hardware and software are simulated in the same simulator, or use handshaking when hardware and software are simulated separately. Another synchronization approach keeps track of time in software and hardware independently and uses tagged messages to synchronize them periodically.

The choice of the model of computation to represent an embedded system depends strongly on the type of the system being constructed. For example, for a purely computational system that manipulates a finite stream of data into another finite stream of data, the necessary semantics that are common in programming languages such as C, C++, Java and Matlab are adequate. For modeling electronics hardware, the semantics need to be able to handle concurrency and time continuum, in which case the continuous-time models of computation such as Simulink, Saber, Hewlett-Packard's ADS, and VHDL-AMS are more appropriate.

For embedded systems, the most useful models of computation need to handle concurrency and time. This is because embedded systems typically consist of components that operate simultaneously and have multiple simultaneous sources of stimuli; and they operate in a timed (real world) environment, where the timeliness of their responses to stimuli may be as important as the correctness of the responses.

For the system performance evaluation, simulation-based methods are more used than the analytical methods. Analytical approaches have the advantages that they usually produce general results, which are valid for all system inputs and parameters. Simulation-based methods are inputs-dependent. Good simulation results responding to some input vectors may not guarantee the correctness of the design. Nevertheless, the design of any practical embedded system in real life is so complex to model and to solve in mathematics, if not impossible. Also, the analytical methods suffer from the fact that they do not scale up well with the scale of the complexity of the design.

The history of the VLSI circuit design is a record of technology inventions that push the use of existing processing to its limit, interlacing with the advance of processing technology. It should be no surprise that this revolution has had a profound impact on how digital circuits are designed. To design today's complex VLSI circuits, designers have increasingly adhered to strict design methodologies and flows that are amenable to design automation. Most of current sequential circuits belong to synchronous systems, in which the latching of data into the memory elements is coordinated by a globally distributed clock signal.

Synchronous design methodology faces increasing difficulties for today's deep sub-micron designs. First, with the shrinking of the transistor feature size, we observe that gate delays scale down quicker than wire delays. As a result, across-block communication incurs higher cost than in-block processing on the VLSI chip. In particular, clock distribution is becoming difficult to realize because of the existence of large skews. Meanwhile, the speed of the circuit is greatly limited by the global synchronous clock network whose speed is governed by the slowest element of the circuit. Also, this global synchronous clock network is often accountable for fair large portion of the circuit's total power consumption because of the power involved in the clock generation, synchronization, and distribution. High power systems are expensive in term of packages, cooling devices and battery life for mobile devices. High noise emission and Electro-magnetic Interference (EMI) in harmonics of the clock frequency are also increasingly becoming concerns in mobile communication applications.

The asynchronous design as an alternative to the synchronous design is gaining more attention recently. Several asynchronous circuit architectures only dissipate power when and where active, that is, any sub-circuit returns to standby mode whenever it is not in use. Performance can be better as it is based on the average-case delay rather than the worst-case delay. Power consumption can also be lower since power is only consumed when needed. Large digital systems can easily be maintained due to high modularity and composability as each block can be designed without knowledge of the timing characteristics of any other blocks. One such architecture is single-rail handshake circuits, which have been shown to consume only 1/5 of the power of their synchronous counterparts, at the cost of a small area overhead [Pee96].

On the other hand, asynchronous circuit designs have several shortcomings. Lack of design tools limits their applications. Also the difficulty of testing asynchronous circuits prevents their wide acceptance by the circuit design community.

1.2 Contributions

In this thesis, we explore ways to address the issue of functionality partitioning of embedded system design. For embedded system design, early validation is a necessary approach to reduce the design effort by having guarantees of correctness early in the design flow, instead of leaving such guarantees to testing after the design is completed. Early validation of functionality and performance requirements can reduce the amount of redundancy and waste of design effort in the design cycle, by making sure the architecture and the high level design are stable before the low level details are developed. Most of the methods listed in Table 1 require system simulation only after the completion of the system design. We propose a method to perform fast performance estimation in the system planning stage. This method can simulate various dynamic software run-time behaviors concurrently with hardware behaviors. We extract the timing behavior of the software and ignore its functionality to quickly estimate the performance of the system under design. This permits early resolution of the tradeoffs of allocating functionality between hardware and software, before the detailed implementation is developed. Our simulation package is designed as an extension of PtolemyII, a heterogeneous design and simulation environment developed at UC Berkeley. In designing our package, we use the software design pattern approach. Our package can simulate system features such as task prioritization, message passing, resource sharing and task blocking, which influence the timing dependencies of the simulated system. This method has been reported in [ZNZ01].

In this thesis, we use the CDMA correlator as a case study to compare various aspects of synchronous and asynchronous design styles. The CDMA correlator circuit was chosen because it is intensively used in the DSP design and has portable applications. For the synchronous design, the circuit is described using *hardware description language* (HDL), then implemented in ASIC by using commercial design tools. For the asynchronous design, we implemented it in the single-rail handshake circuits design style, using gates from a standard cell library. Several handshake components from [Pee96] are implemented and simulated in this thesis. Also, we have custom designed several asynchronous blocks for this correlator.

The CDMA correlator architecture from [SB98] was chosen to minimize power consumption. We have built the circuit models in PtolemyII to study the switching activity in the correlator and to explore the architecture tradeoffs for the correlator design.

1.3 Organization of the Thesis

The presentation of this work is organized as follows: in next Chapter, we present the background material for the work in this thesis. In Chapter 3, we describe our software package design of the hardware/software co-simulator for fast performance estimation, along with two examples to illustrate the usefulness of our method. Chapter 4 presents the handshake circuits as an alternative design style for low-power design and describes the design of CDMA correlator of asynchronous design -- single-rail handshake circuits — and of synchronous design. The comparison of these two designs and a study of switching activity by simulation are also included in Chapter 4. Finally in Chapter 5, we conclude the presentation of our work and indicate directions for future work.

Chapter 2

Background

In this chapter, we review the background of the work presented in this thesis. In the Section 2.1, we present the background material for our hardware/software (HW/SW) co-simulator software design. This includes the review of previous work, an introduction to PtolemyII and the design pattern approach of software design which is used in PtolemyII and in our software package. In the Section 2.2, we describe basic concepts in synchronous design and asynchronous design. In that Section, various tradeoffs of these two design styles are discussed and one particular class of the asynchronous circuits -- handshake circuits -- is introduced in detail. In this thesis, we describe the VLSI circuit design of the CDMA correlator as a case study for the comparison of these two circuit design styles. The background knowledge of the CDMA system and its correlator can be found in Section 2.3 of this chapter.

2.1 Hardware/Software Co-Simulation

With the advancing of circuit design and VLSI processing technology for higher packaging density, faster circuit speed and lower power dissipation, today's VLSI technology can place 10^7 – 10^8 transistors on a single chip [TB+97]. As the time-to-market pressure increases, with the decreasing cost of microprocessors and the flexibility of the software implementation, we observe a trend for designers to use mixed hardware and software solutions for their applications. Embedded systems that combine hardware and software are widely used in communication and multimedia applications.

The differences in the behaviors of hardware and software pose new challenges to system modeling, design and simulation. Traditionally, hardware operates in timed fashion and is physically structured. Usually, hardware is synchronously reactive to its environment. Multiple hardware components can operate in parallel [PPT00]. Meanwhile, software usually modeled as non-timed, which means its executing time is affected by factors like OS scheduler, memory size etc. Software processes can run interleaved. Some software can reconfigure themselves during their run-time. Also, software are usually resource independent that they can be evoked to execute whenever there is resource available [PPT00]. To simulate the performance of mixed HW/SW systems, we need a mechanism to bridge the representations of these two domains.

In the traditional HW/SW co-design flow, the system is modeled, simulated and compared to the specification to verify the correctness of its functionality. In the next stage, the mapping of each function to hardware or software implementation is carried out. To explore the design space of different hardware and software combinations, a fast performance estimation method is important. In particular, it is necessary to rapidly evaluate tradeoffs of various hardware and software implementation options to guide the refinement of the system.

2.1.1 Previous Work

The HW/SW co-design problem is receiving growing attention both in academic research and in industry. There exist several tools that address the HW/SW co-design [BM+97].

Simulating embedded systems is challenging, mainly because they are heterogeneous. In particular, most of these systems contain both hardware and software components that must be simulated at the same time. Several approaches to co-simulation have been proposed, with varying tradeoffs between accuracy and performance. According to [EL+97], the major approaches to co-simulation are:

- *Gate-level models.* They are the most accurate, but also are very slow. Thus, they are only suitable for small systems, where either the processor is very simple or only very little code needs to run on it, or both.

- *ISA (Instruction Set Architecture) models.* In such models, filtered information is passed between a standard processor simulator (often written in C) augmented with hardware interfaces and a hardware simulator.

- *Annotation models.* In these models, the software is represented by a "software graph" which contains static timing information regarding the processor configuration [SS96].

- *Translation-based models.* This approach converts the code for the target processor to the host processor, then simulates the converted code on the host computer. Passing the timing information to a hardware simulator is the major challenge.

- *Equation models.* This approach uses a set of linear equations to implicitly describe the possible execution paths. They are mostly used for conservative worst-case execution time estimation [Bal99].

Each method has its advantages and disadvantages. The drawback of the more accurate methods is that they require detailed models of hardware and software, which may not be available until in the later stage of the design cycle. In such models, performance analysis can be done only when the detailed timing information is obtained after finishing the design.

The advantages of high level, abstract models are small simulation overhead, ease of integration and the flexibility of porting to different systems. However, high level models usually suffer from low accuracy. Besides, it may be difficult to preserve the characteristics of software (dynamic, resource sharing, etc.) and hardware while abstracting them into the high level models.

A typical HW/SW design flow is that of the *Polis* development environment [Pol]. First, the system specification is defined in some formal languages. Then it is translated into a network of interacting finite-state machines, called co-design finite state machines (CFSM). This representation does not distinguish between hardware and software implementations. Afterwards, each CFSM is mapped to hardware or software implementation. Finally, the system undergoes a HW/SW co-simulation to verify its timing requirements. The whole design process needs to iterate several times before finding the optimal solution.

A fast performance estimation method is needed in this process to guide the functionality partitioning of the system, which is usually performed interactively by the designer [BM+97]. If only the timing estimation or time budget of the system can be simulated instead of the whole system functionality, then the simulation can run much faster. Hence, more architectural design options can be explored in a short period of time.

This need for fast performance estimation motivated us to find a novel way to abstract the timing behavior from the functionality of the software. This allows us to simulate hardware and software in their own domains, and then to link them together for timing simulation purposes, without the costs incurred by reference to the detail functionality simulation which is already verified by other methods.

2.1.2 Heterogeneous Modeling Framework – PtolemyII

PtolemyII is designed to support heterogeneous modeling and design of concurrent systems. In PtolemyII, a system is modeled as a collection of hierarchical and concurrent components. The executable components of a system are called *actors*. These *actors* include the semantics of message passing and execution. This semantics is shared by a group of models of computation, including CSP, discrete-event (DE) model, and etc.

Messages are encapsulated in *tokens*. *Actors* have ports, which are represented as instances of the *IOPort* and which can be input, output or both, depending on whether they can receive *tokens*, send *tokens* or both. An execution includes one invocation of *initialize*(), followed by an arbitrary number of iteration of *prefire*(), *fire*() and *postfire*(), followed by one invocation of *wrapup*(). The execution of the components of a *composite actor* is governed by a *director* object. A *director* may implement different models of computation. A model of computation is implemented as a domain in PtolemyII. There exists a rich set of models of computation that deal with the concurrency and time in different ways. For example, the *actors* in the continuous time (CT) domain represent components that interact via continuous-time signals. They are most suitable to model analog circuits, mechanical components and microwave circuits [Lee01].

An essential difference between various models of computation is their modeling of time and order of occurrence of events. Choosing the right computation model is critical for the system modeling and specification.

2.1.3 Design Patterns

Design patterns [GH+95] are a new design approach in object-oriented software design. A broadly applicable definition of term “pattern” is that a pattern is the abstraction from a class collaboration recurring in specific contexts. The notion of a pattern is geared toward solving problems in design. But a pattern is more than just a solution to a recurring problem. The problem occurs within a certain context in the presence of many competing concerns. A solution pattern involves a set of classes and their relationship (inheritance, aggregation and association) that balances these concerns in the manner most appropriate for the given context. Design patterns optimize maintainability of an object-oriented software design, rather than performance, memory, or other quantitative parameters. Through using patterns, it is easier to design and change software by reusing successful designs and architectures.

Several design patterns -- reactor pattern, non-blocked buffering pattern etc – are used in our software package design. For example, the reactor pattern addresses event-driven applications, such as protocol software in communication devices, which receive requests from multiple clients concurrently and iterate them without blocking indefinitely on any particular source. In the reactor pattern, the *reactor* defines an interface for registering, removing and dispatching concrete event handlers. A single-threaded application can use the *reactor* to wait synchronously for the arrival of events from multiple sources. By handling concurrent events at the *reactor*, there is no need for more complicated threading, synchronization or locking within the application.

2.2 Synchronous and Asynchronous Design

A digital circuit has the distinctive property that all signals in it are binary. By assuming that, Boolean logic can be used to describe and to manipulate logic constructs. For the time model, designers often assume that all events of interest for the functionality of a circuit occur at discrete times. The circuits designed with this assumption belong to the class of synchronous circuits, in which all signal transactions are synchronized to the rising and/or falling edge of the clock signal; one the other hand, circuits in which signal transactions can happen at any time along the continuous time axis are belong to the class of asynchronous circuits.

2.2.1 Synchronization in Digital System Design

Synchronous design uses the global clock signal to concert the operations of various components of a circuit. Instead of using a global synchronized clock network, asynchronous design uses communication channels of several types, which permit to exploit intrinsic timing relationships of the components and circuit topology to ensure the correct operation of the circuit. The synchronous design style is currently used in most of the designs because it is easy to understand and because of the availability of highly automated design tools. On the other hand, asynchronous design is getting more and more attention recently for their timing robustness and low power properties.

2.2.1.1 Basic Concepts

A basic approach in complex circuit design is to define abstractions that enable the designer to ignore the unnecessary details and to focus on the essential features of the design.

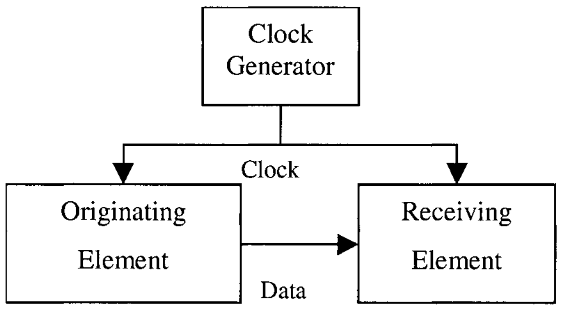


Figure 1: Synchronous interconnection

The role of synchronization is to coordinate the operation of various parts of a digital circuit. For synchronous design, each element (or module) is provided with a clock signal, as well as one or more signals that are generated with transitions synchronized to the clock. The global clock network controls the order of operations, ensuring correct data transfer throughout the circuit. This synchronous interconnection isolates the circuit behavior from timing details by setting the clock period ***T***. There exists a certainty period during which the output signals are guaranteed to be correct and stable, so they can be sampled. With synchronous interconnection, the irrelevant behaviors (multiple signal transitions and uncertain completion times) are hidden from the interface of a module. We can thus abstract the operation of a computational block, as viewed from the output registers, as an element that completes its computation precisely at the active transition of the second clock signal.

2.2.1.2 Pipelining

Pipelining is a common design technique to increase the throughput of the circuit. The pipelined structure has the ability to initiate a new computation at the inputs to a computational block prior to the completion of the last computation at the outputs of that block. Since this results in more than one computation in process within the block at any given time, pipelining is a form of concurrency. The number of the pipeline stages is defined as the number of concurrent computations that can be in process at any one time.

In practice, it is usually not possible to precisely divide a computational block into "equally-sized" stages. In that case, the throughput of a synchronous pipeline has to be adjusted to match the worst-case delay of a stage in the pipeline, resulting in a lowered throughput. There are a number of other factors, such as register setup time, which reduce even further the throughput of a pipeline and increase the overall processing time.

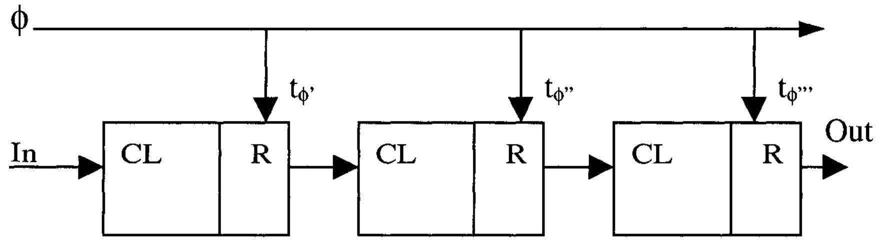


Figure 2: The pipeline structure

2.2.1.3 Clock Skew

Clearly, any increase in the uncertainty of clock phase (clock skew) will reduce the throughput of the circuit, since the sampling time of the output registers requires precise control of the clock phase within a vanishing certainty period. Conversely, any fixed delay in the interconnection will not necessarily affect the achievable throughput of the circuit, because it will increase the propagation and the settling time equally and thus it will not affect the length of the uncertainty period. In practice, however, for common digital circuit design, the effect of any uncertainty in clock phase is lumped with interconnection delays. The practical way to limit the problems caused by the skew is to ensure that the clock skew between communication registers is bounded. Careful routing of the clock signals is one possible solution to reduce the skew by equalizing the local clock delay.

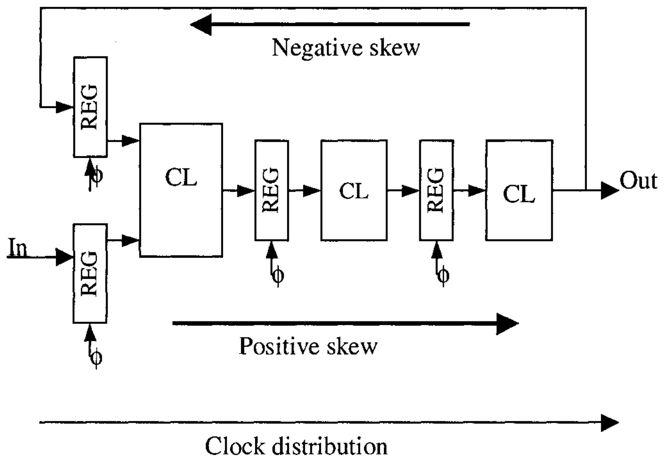


Figure 3: The cause of clock skew

Another approach to circumventing the skew is to introduce the Phase-Locked Loops (PLLs) or the Delay-Locked Loops (DLLs) in the clock distribution network. Such circuits synchronize the edges of the internal clock, generated from any specific location in the circuit relative to the edge of the reference clock with an adjustable offset [Bry01]. By controlling the offset value, the locally generated clock signal can eliminate the clock skew.

2.2.1.4 Clock Gating

Power optimization at high levels of abstraction has great impact on the reduction of overall power consumption at the final gate-level design. Clock gating is a widely used high level technique to reduce power.

A register bank is a group of flip-flops that share the same clock signal and synchronous control signals such as load enable, set and reset. Clock gating provides a power-efficient implementation of register banks that are disabled during some clock cycles. Without clock gating, register banks are usually implemented using a feedback loop and a multiplexer (see Figure 4). When such registers maintain the same logical values during multiple clock cycles, they use more power than necessary.

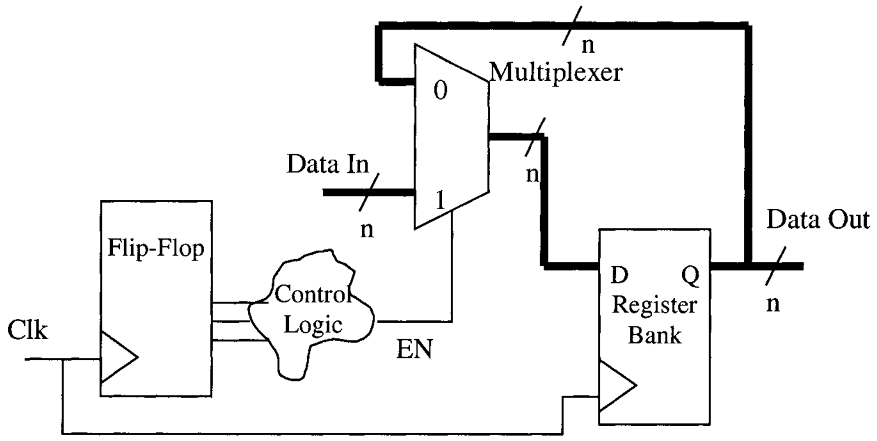


Figure 4: Register bank implementation without clock gating

Clock gating saves power by eliminating the energy dissipation associated with reloading register banks whose stored logical values do not change. Clock gating eliminates the feedback net and the multiplexer from Figure 4 by inserting a two-input gate in the clock net of the registers, as shown in Figure 5. Clock gating can also insert inverters or buffers to satisfy timing or clock waveform and cycle duty requirement.

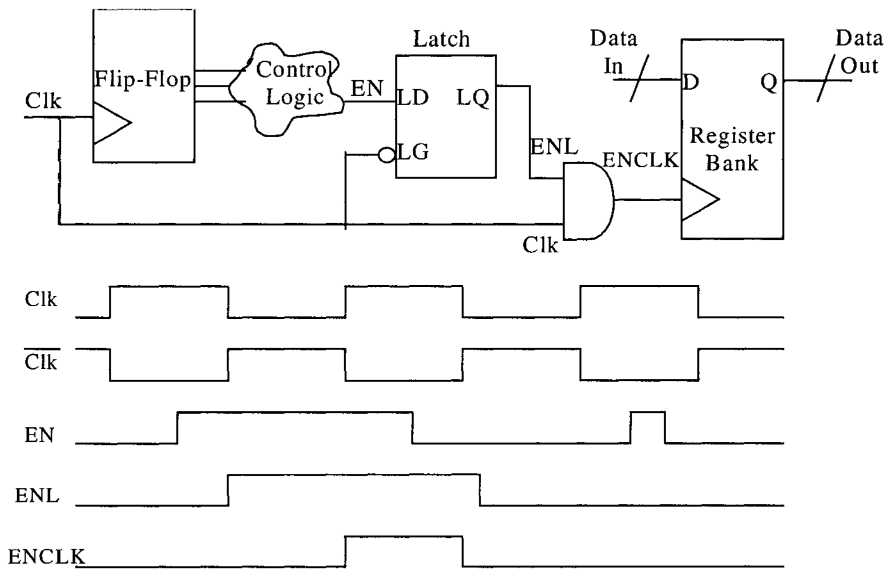


Figure 5: Register bank implementation with clock gating

2.2.2 Asynchronous Circuit Design Style

In this Section, we introduce some aspects of asynchronous design. Asynchronous design here refers to the design of digital circuits that operate correctly without relying on the global clock signal for synchronization. It is not possible to offer a complete overview here; instead a brief introduction to the basic concepts is provided with the emphasis on the single-rail handshake circuits, which will be used in the design example in this thesis. A full treatment of the subject of asynchronous design can be found elsewhere [Hau95].

2.2.2.1 Basic Concepts

By assuming that time is discrete, hazards and feedbacks can largely be ignored in synchronous circuits. However, as with many simplifying assumptions, circuits that can be designed and operate without these assumptions have the potential to generate better results.

There are several possible benefits by removing that assumption.

No clock skew – By definition, asynchronous circuits do not have globally distributed clock signal, so there is no need to worry about clock skew. In contrast, synchronous circuits often slow down their operating frequency to accommodate the skew. As transistor feature size shrinks and operation frequency increases, longer wire delays are taking a larger portion of the clock duty cycle. Efficiently distributing the clock network to avoid the clock skew will become a major design concern in high performance design.

Low power - Asynchronous circuits inherently cease their switching activity when no work needs to be done, and can go from idle (ideally zero power) to full activity (maximum throughput) instantaneously. This means that for some types of circuits, where there is a significant time period in which the circuits must be able to react quickly to their environment, also where there also is a significant time period in which the circuits are not doing anything useful, asynchronous design techniques are good candidates. For example, the most successful commercial asynchronous applications have been in portable electronics aiming for low power [Pee96].

Average-case instead of worse-case performance – Synchronous circuits must wait until all possible computation have completed before latching the results, yielding worst-case guarantees of performance. Many asynchronous circuits sense when operations have completed, allowing them to exhibit average-case performance.

Automatic adaptation to physical properties - The delay through a circuit can change with variations in fabrication, temperature and power supply voltage. Synchronous circuits must assume that the worst possible combination of these factors is present and clock the circuits accordingly. Many asynchronous circuits sense computation completion, and will run as fast as the current physical properties allow.

Although asynchronous circuits have all of these potential advantages, they have several drawbacks as well. Primarily asynchronous circuits are often more difficult to design in an ad hoc fashion than synchronous circuits. Designers of asynchronous circuits must pay a great deal of attention to the dynamic states of the circuits. Hazards must be removed from the circuits, or not be introduced in the first place. Asynchronous circuits generally require extra time due to their signaling policies, thus increasing their average-case delay. Whether this cost is greater or less than the benefits listed above is still unclear and more research in this area is necessary.

2.2.2.2 Basic Asynchronous Component – C-Element

Before we introduce asynchronous circuits and present the correlator designs in detail in the following sections, it is beneficial to briefly introduce the C-element here. Apart from standard Boolean gates, the C-element and its variations are the most widely used components in asynchronous design and will be used in various places in our circuit design.

$$a*b \rightarrow z\uparrow$$
$$a'*b' \rightarrow z\downarrow$$

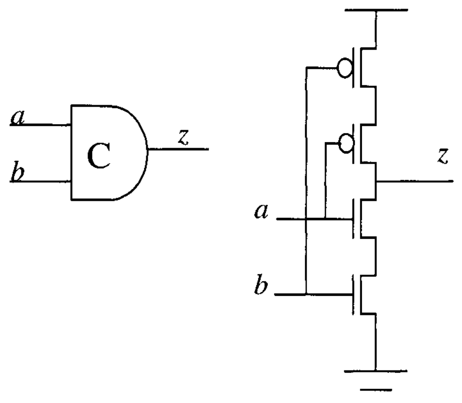


Figure 6: The C-element symbol and one of its implementation

A C-element is essentially an asynchronous memory element. A simple two-input generalized C-element can be represented by the production rules in Figure 6. The C-element operates as follows: when both inputs *a* and *b* are high, the output *z* is pulled high, and when both inputs *a* and *b* are low, the output *z* is pulled low; otherwise the output remains unchanged.

Besides the symmetric C-element, asymmetric C-elements is also very useful. The notation used for the asymmetric C-element indicates that an input controls both rising and falling edges of the output when that input is connected to the main body of the gate, an input controls only the rising edge of the output when that input is connected to the extension marked '+', and an input controls only the falling edge of the output when that input is connected to the extension marked '-'. This notation is illustrated in Figure 7 (a), and a possible transistor-level implementation of an asymmetric C-element is shown in Figure 7 (b).

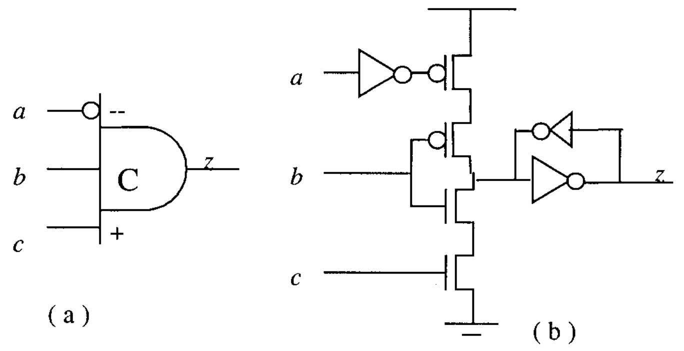


Figure 7: The asymmetric C-element

2.2.2.3 Data Encoding

The term "data encoding" refers to the data and the signaling scheme used for communication between circuit components. Different communication protocols and timing assumptions are used in different asynchronous design styles. In a typical handshake protocol, a sender will indicate the validity of data on one signal and the receiver will indicate the completion of processing data via another signal. This can be done by using either dual rail data coding, or bundled (single-rail) data coding.

Dual rail data coding involves encoding the data signal on two wires, and the wire that undergoes a transition indicates the logical bit's value. This method is, effectively, one-hot coding for the two states of each bit. The bundled data coding encodes the data on the conventional 1-bit per wire scheme, but uses an explicit *request* line to initiate the data validity signal. It is assumed that the delay on the explicit *request* line is greater than that on the data lines. In both cases an explicit *acknowledgement* line is needed. It can be seen from this that it is not possible in these types of circuits to encode a data bit with a single wire.

2.2.2.4 Two-Phase versus Four-Phase Protocols

The handshake initiation and completion signals use either a two-phase protocol or a four-phase protocol.

In the two-phase protocol, a handshake event occurs on a wire whenever there is a signal transition: no differences are made between a rising edge and a falling edge. The data must be valid before the sender initiates the *request* transition, and must remain valid until after the receiver indicates the completion of receipt by sending a transition on the *acknowledgement* line.

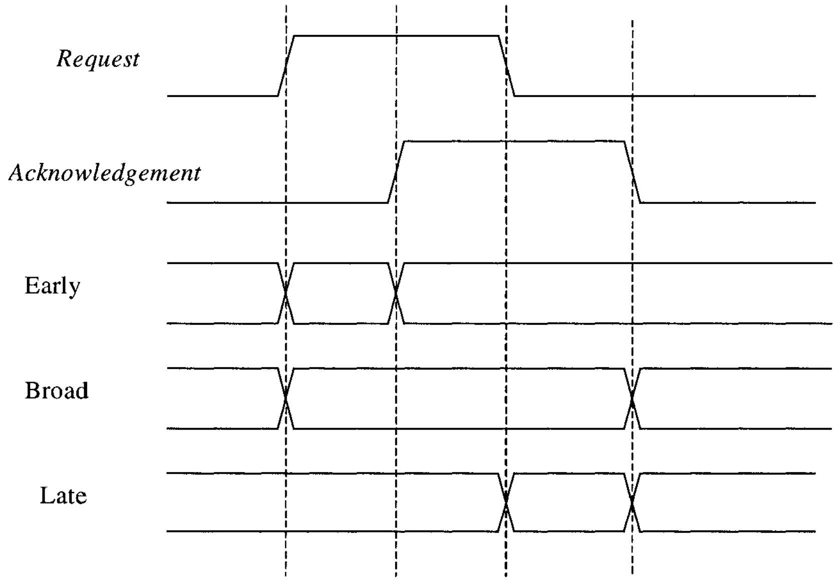


Figure 8: The "Early", "Broad" and "Late" data schemes of the four-phase protocol

The four-phase protocol has a return-to-zero phase for each signal, so the sequence, $0 \rightarrow 1 \rightarrow 0$, comprises only one transition on a four-phase protocol signaling. There are three data schemes of the four-phase protocol, which are “Early”, “Broad” and “Late” schemes. The “Broad” scheme has the advantage of simplifying datapath design, as data are valid as long as the *Request* signal is high, meaning that the *Request* signal can be used as a conventional enable signal.

The advantage of four-phase circuits is that, having the return-to-zero phase, circuit elements can be level-sensitive, rather than edge-sensitive, which can simplify their designs. On the other hand, two-phase circuits have, at least in theory, speed and power advantages over four-phase circuits since they have only half the number of transitions in their control paths. This advantage is not as great as might at first be thought, as the majority of delay is usually in the datapath logic, which must be included in whichever protocol is used. Also, two-phase control elements are generally slower than the four-phase ones.

2.2.2.5 Micropipelines

Micropipelines are a class of asynchronous circuits introduced by Sutherland in his Turing award lecture [Sut89]. The timing assumption used here is that for a small, isolated section of datapath, the delay is known, but that section can operate in an environment of unbounded delays.

A micropipeline, like any other pipeline, has stages separated by memory elements. The memory elements can be registers or latches, depending on whether the protocol used is two-phase or four-phase handshake. Due to the asynchronous nature of the circuit, these registers or latches need to indicate when they have latched the data values. As shown in Figure 9, the event registers latch the data in response to the *request* event on *Rin,* but only if the downstream registers have indicated that they have themselves latched the last data they were passed. The downstream registers did that by issuing an *acknowledgement* event on their *Ain* (connected to the *Aout* of the upstream registers). When processing is added to micropipelines, it can be put just in the datapath between event registers. In this case, the *Request* signal must be delayed by the time greater than the delay of the processing logic, which can be done by using either matched delay or completion detection.

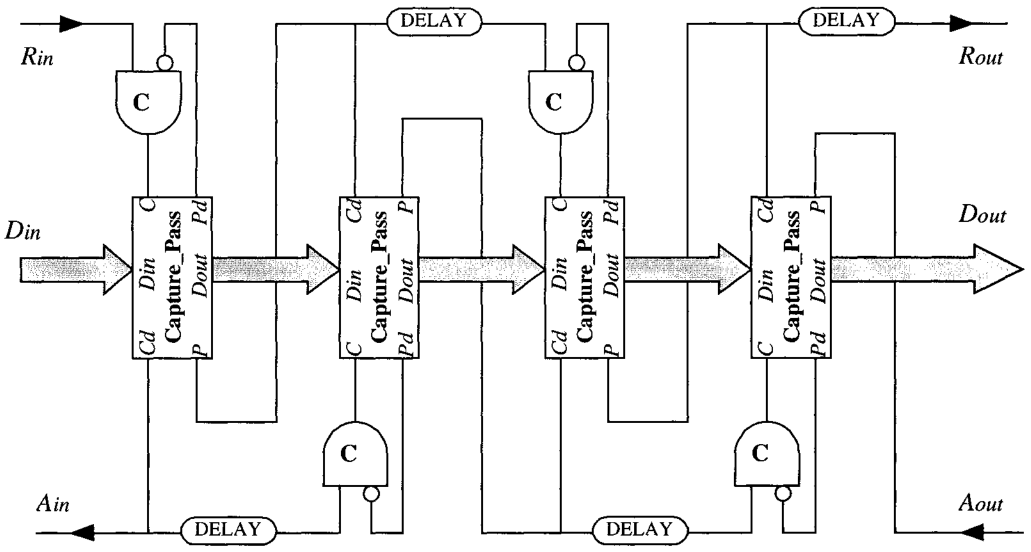
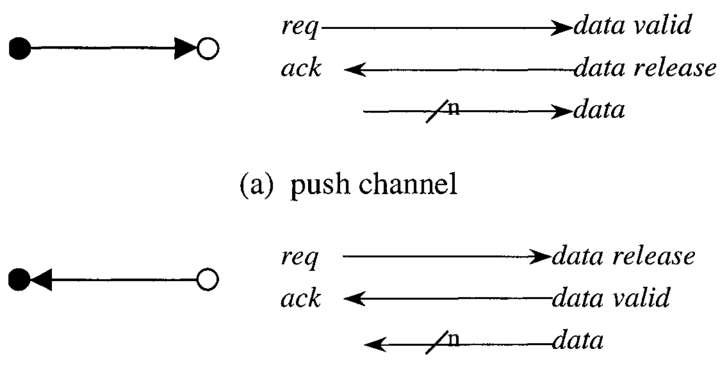


Figure 9: Micropipeline architecture

Micropipelines have an inherent advantage over other asynchronous circuits, which is that the datapath elements may be easily designed, using conventional combinational circuit design with conventional tools. Using matched delays, it is possible to simply take a block of conventional, synchronous style logic and to wrap it into a micropipeline shell, adding a matched delay, and it becomes an asynchronous component. This facility of module design means that it is easy to design small blocks and in turn to use these blocks to build very large devices [Sut89].

2.2.2.6 Handshake Circuits

Handshake circuits are another class of mature asynchronous circuits, mostly developed by the research group leaded by Kees van Berkel at Philips Research Labs [Ber93]. Handshake signaling as used in handshake circuits is a communication mechanism that establishes the point-to-point synchronization. A handshake involves two partners which play different roles, called *active* and *passive*. The partners exchange so-called *request* signal and *acknowledge* signals. The passive partner waits for a *request* signal to arrive and after receipt of a *request* signal responds with sending an *acknowledge* signal. The active partner starts with issuing a *request* signal and then waits for the corresponding *acknowledge* signal to arrive. Such an exchange of a *request* signal and an *acknowledge* signal is called a handshake.



(a) push channel

(b) pull channel

Figure 10: Handshake channel notions

A handshake essentially synchronizes the active and the passive partners. In addition to pure synchronization, handshake can also establish data communication between the partners by encoding data in the *request* signal, in the *acknowledge* signal, or in both.

Handshake channels with no data encoded are called *nonput* channels. They connect two so called *nonput* handshake partners, one active, another passive. A handshake on a *nonput* channel establishes synchronization only, no data is communicated.

The second type of handshake channels is that with data encoded in the *request* signal. These channels connect an active sender and a passive receiver. So, the sender takes the initiative for a communication. One might say that the sender pushes the data through the channel, therefore these channels are referred to as *push* channels. From a data-flow point of view, *push* channels are data driven.

On a *pull* handshake channel, data is encoded in the *acknowledge* signal. Such a channel connects a passive sender and an active receiver. The sender issues the data after receiving a *request* from the receiver, so one could say that the receiver pulls the data through the channel. From a data-flow point of view, *pull* channels are demand driven.

2.2.2.7 Design Tools

Currently, there are almost no CAD tools for asynchronous circuit design. Of the few available tools, none of them is integrated into the commercial EDA (Electronics Design Automation) tool suites. However, conventional synchronous tools may be used for many of the asynchronous design work.

In most cases, for the lower levels of the design, Place-and-Route and layout, there are in practice no differences between synchronous and asynchronous circuits. It is in the higher levels, architectural design and synthesis, that the lack of tools is most obvious. Nevertheless, the simulation tool LARD proved useful for architectural experimentation in Amulet project at University of Manchester [Amu], while the Petrify tool is used for verification and synthesis [Pet].

Silicon compilation, that is, the automatic generation of VLSI circuits from descriptions written in a high level programming language, demands a powerful programming language and a good compiler. The programming language should be abstract from VLSI circuits and technology details, thus allowing designers to concentrate on application programming issues. The *Tangram* VLSI language [Ber93] is one of the most successful languages in this category. Circuits described in *Tangram* programs first are translated into equivalent handshake circuits, then the silicon compiler compiles the handshake circuits into a gate-level netlist. This compilation is based on component-by-component substitution of handshake components by pieces of circuitry. The compilation also contains many gate-level optimizations, so called peephole optimizations [Pee96]. These optimizations replace combinations of circuit elements by simpler ones that are smaller, faster and more power efficient.

2.3 CDMA System and CDMA Correlator

In this Section, we briefly present the concept of the CDMA system and the role of correlators in a CDMA system. The discussion of asynchronous and synchronous implementations of the CDMA correlator is included in Chapter 4.

2.3.1 The Concept of CDMA System

The requirements of the capacity of data transmission through mobile radio interface is rapidly increasing. Currently, the second generation of mobile communication system – the digital data and voice system -- has almost replaced the analog system — the first generation mobile system. Most third generation mobile communication proposals suggest using Wide-Band Code Division Multiple Access (W-CDMA) technologies in order to support future multimedia transmission, which requires high data rate and wide bandwidth.

CDMA is a class of multiple access technology based on the spread spectrum technology. In a communication system, the task of multiple access technology is to transmit message to and from many users simultaneously without mutual interference. Traditionally, signals are considered from two aspects, time and frequency. CDMA system uses a different approach. This approach spreads the message from each user to a longer code. Codes from different users are mutually statistically uncorrelated, i.e. orthogonal. Therefore, a correlator with the property of the codes of specific user can extract the information from him/her even through the signals of all user are mixed.

Both Walsh code and "Pseudo-Noise (PN) code are employed in CDMA technology. The Walsh functions are a binary antipodal {+1, -1} sequence that are designed, by definition, to be orthogonal to each other [PM96]. However, Walsh codes, while having perfect cross-correlation properties, are extremely poor auto-correlated: they do not appear to be pure "white" noise. The desirable property of multipath rejection, which depends heavily on the autocorrelation performance, is completely lost. Timing recovery in the receiver is also exacerbated, since it becomes difficult to determine each user's baseband bit-timing without employing large over-sampling factors. Although this problem can be overcome in the situation that a base station is capable of providing synchronization for all of its users the Walsh sequences in the downlink by maintaining a pilot tone.

The PN sequences have the key property that they appear to be "white" in nature: their power spectrum is flat, and their autocorrelation is nearly an impulse function. Furthermore, the resulting spectrum of the concatenated Walsh/PN code is also white, to good approximation. Basically, it is a similar situation as the case of sinusoidal modulation of white noise: the periodicity of the Walsh codes, spectrally concentrated, does not significantly affect the randomness of the spectrally flat PN code. Multiplication in the time domain by periodic function results in a shift in frequency domain, and shifting a white spectrum results the same.

The IS-95 standard is a typical CDMA standard, which is widely used in today's public CDMA mobile phone network. In the IS-95 standard, the CDMA operations are slightly different in uplink and downlink. In downlink, the base station uses Walsh codes to generate codes, these codes are deterministic, strictly orthogonal. In uplink, each mobile uses a long PN code to spread the message. The PN sequence generator is seeded with the data received in the message send from the base station. The same seed is used in both directions. In IS-95 standard, the basic user channel rate is 9.6kbit/sec. This is spread to a channel chip of 1.2288Mchip/sec (a total spreading factor of 128, a chip is single bit within the PN code in CDMA terminology) by using a combination of techniques. More details can be found in [Vit95].

The basic codes used in this design – PN code and Walsh code – are the same codes employed by the IS-95 standard. However, the detail coding scheme – a hybrid PN-Walsh code – and channel rate do not conformed to the IS-95 standard. This aggregate coding scheme can simultaneously achieve 100% capacity, while still maintaining near white spectral performance [Vit95], each user bears a signaling rate of 1M bit/s, with a minimum spread rate of 64 Mchip/s (64 chip are transmitted per user bit) to accommodate 64 users simultaneously [SB98].

The correlator performs the correlation function at the receiver end in this CDMA system. It recovers the sequence of data coded with the same Walsh function and PN code as it is selected to decode from the aggregated transmitting signal, so the data coded with different pairs of Walsh function and PN code are rejected as noise. It samples the input signal at 64 Mchip/s and outputs 1 Mbit/s decoded data stream.

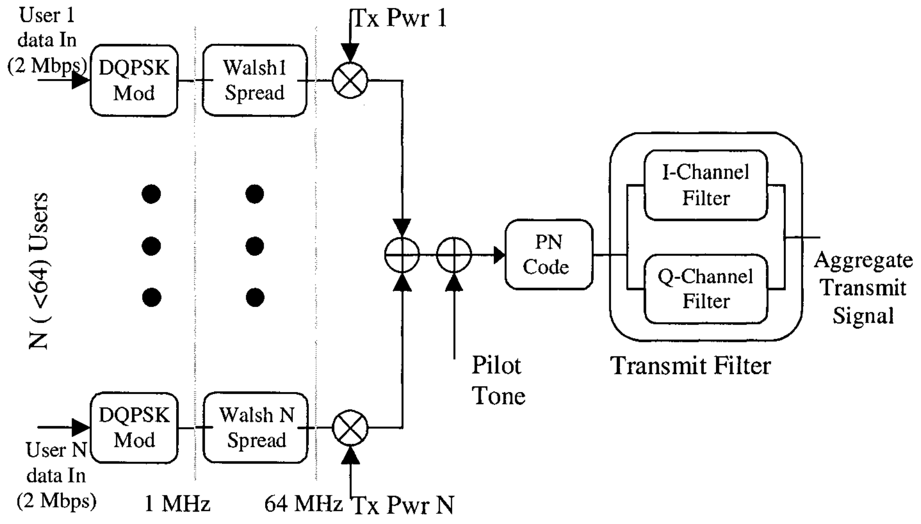


Figure 11: CDMA transmitter schematic

In this thesis, we present the design of the correlator in two different digital circuit design styles, namely, synchronous and asynchronous design style as introduced in section 2.2. A word of caution should be mentioned here to clarify the terms of "asynchronous" and "synchronous" used in this thesis. The synchronous and asynchronous terms also are used in the communication domain. Most noticeably, there are two different communication modes, the synchronous communication mode and the asynchronous communication mode, which indicate the synchronization method at the system and/or architecture level. They are two totally different concepts, used in different contexts. All the terms "asynchronous" and "synchronous" appeared in this thesis are in the context of digital VLSI circuit design.

Chapter 3

Modeling and Simulation of Heterogenous Systems

In this chapter, we describe the design of a software package that performs system timing estimation of embedded HW/SW systems. Two aspects of interest are the functionality and the timing constraints of the simulated system. Detailed HW/SW co-simulation is a time-consuming task in which both software and hardware are usually simulated in their own time domains, then simulated jointly for both functionality and timing. In contrast, we extract the timing behavior of the software and ignore its functionality to quickly estimate the system performance. This permits early resolution of the tradeoffs of allocating functionality between hardware and software, before the detailed implementation is developed. In designing our package, we adopted the reactor design pattern [SS+00], which permits to represent distributed objects in such a way that facilitates maintenance of the models.

Our package is built as an extension of PtolemyII [Pto]. We model each hardware element as a component in the discrete-event (DE) domain of PtolemyII. We view the software part of the system as another DE component that represents just its timing behavior. The simulated software is assumed to consist of several concurrently executing threads, and each thread is represented by a fixed number of actions.

Our package can simulate system features such as task prioritization, message passing, resource sharing and task blocking, which influence the timing dependencies of the simulated system. In combination with the other existing domains in PtolemyII, our package provides a convenient way to model and estimate the tradeoffs of the HW/SW functionality partitioning of embedded systems.

3.1 Modeling of Hardware/Software Systems

"Software" mainly implies the sequential execution of a single instruction stream. That is, the same hardware resources are multiplexed in time to perform different functions. "Hardware" in contrast, denotes primarily parallel execution [Lee00]. A dedicated DSP with its own instruction memory, which is processing the incoming data stream, may be treated as hardware component rather than a software component. The software in embedded systems is dynamic in nature. Its responsibilities include run-time components instantiation and termination, run-time allocation and de-allocation of resources (CPU, memory, communication channels), management of access to these system resources, as well as the possibility to guarantee bounds on the execution time. Most of these services are offered by a real-time operating system (RTOS).

Typically, the architecture of software in embedded systems is layered [CC+99]. Each layer provides the functions and services to the next layer. Software tasks are mostly implemented as threads running under the control of the RTOS kernel, which offers basic services, such as task scheduling and inter-task message passing. Also, a RTOS contains the communication protocol stack, which provides the communication between the application layer and the hardware layer, event and alarm trapping and management to handle hardware interrupts, and external IO stream management and other functions. High-level system functions are made available to user applications via Application Program Interfaces (API).

Software is widely used to implement control loops, user interfaces and protocol stacks in the embedded system. Software either interactively responds to the inputs from the user or executes upon occurrence of events from the environment. Typically, the specifications of this type of software consist of the interacting state machines rather than of sequences of arithmetic operations to be scheduled within a bounded time frame. On one hand, such specifications exhibit FSM-style behavior, as in the RTL (register transition level) description for hardware; on the other hand, they have the behavior pattern of software (dynamic, resource sharing etc). Object-oriented software generated by the modern compilers also has certain reactive-type features, in which objects can be dynamically created and destroyed to implement certain functions in reaction to the message they receive.

In [GM93], Gupta suggested to model software as a set of fixed-latency concurrent threads. We model the timing behavior of the software threads as a combination of several atomic constructs corresponding to their timing behavior relative to the hardware components and other software threads. We catalogue these atomic software actions into following types:

- *Execution* – The software thread executes for a fixed amount of time, which has no timing dependency relationship with the operation of hardware or the execution of other concurrent software threads.

- *Creation* –This action creates the message that will invoke the message handler.

- *Wait* – This action expresses the temporal relationship with the hardware or the other threads. It is blocked by result of the output of the hardware or the message generated by the other threads.

- *Send* – This action sends the output signal to the output port.

Our model of the software is a set of concurrently executing threads, which is made of a combination of these actions. These threads are assembled into a temporally and causally ordered task graph. To the event-driven software, the top-level loop is the message dispatch mechanism that is implemented by using the reactor pattern, as will be described in more detail in Subsection 3.3.2. Each task can be assigned a priority. The execution of the tasks is scheduled by the RTOS according to a scheduling policy. Different scheduling policies can be enforced, depending on how to build the task graph, how to add a new task to the task graph and how to select the next task from the task graph to execute. The timing information of each action, used for linking back to DE domain, is determined by the parameters of each action. These parameters can be set to the worst-case data, average-case data or even a distribution pattern extracted from the time-stamped trace of the previous implementation.

An important point of this model is the separation of the functionality of the software from its run-time behavior. We represent the software code by the *Process* class, which is just a sequential collection of actions (see Figure 13 for the class diagram). The run-time behavior is represented by the *Task* class, which uses the *process* object as prototype, together with the run-time information, which is updated at the completion of each action. A task graph is constructed by an *RTOS* class, whose responsibility is to control the run-time behavior of the software execution.

3.2 System Modeling

A typical model of the reactive embedded system built in PtolemyII is as follows. The functionality provided by the software implementation is run under the control of the system RTOS. The hardware component is only aware of the existence of the glue logic between the hardware itself and the interfaces to which it is attached. The glue logic circuitry may include the control registers that the CPU can address to, the buffer to hold the temporary data and the interface protocol logic. The hardware component interfaces with a virtual machine in the sense that the response of this virtual machine is the value of its outputs with a certain delay.

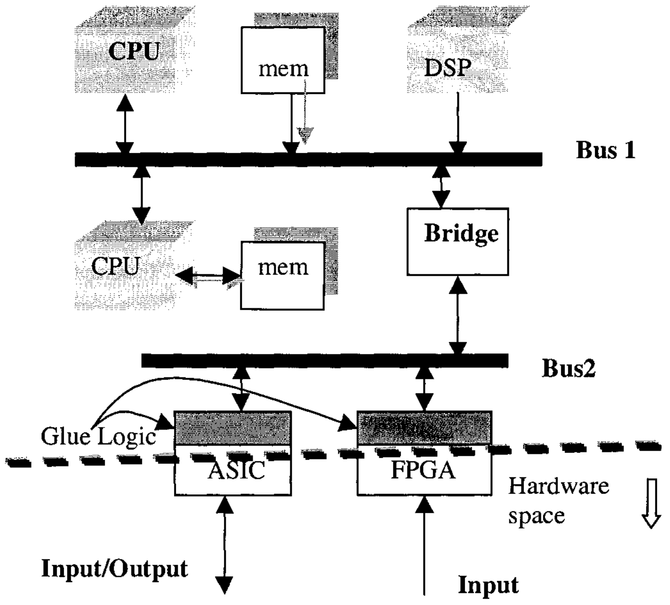


Figure 12: The graph of a system model

A HW/SW co-simulation model must capture the typical behavior described above. In PtolemyII, the model of certain behavior is implemented as a component in certain domain. For our timing behavior modeling of embedded systems, we use a proxy component to represent all the software threads and the hardware elements on which these threads are running, and the hardware components which can run concurrently (no resource sharing) along with the software threads in DE domain. The timing information of the thread execution will be called back to the proxy component to link it to the virtual time in the DE domain.

3.3 Package Implementation

Our package is designed as an extension of the DE domain in PtolemyII. In the DE domain, the actors interact with each others through sequences of event placed in time. An *event* has a value and a time stamp. Actors fire when new events are present. This model of computation is appropriate for specifying concurrent hardware.

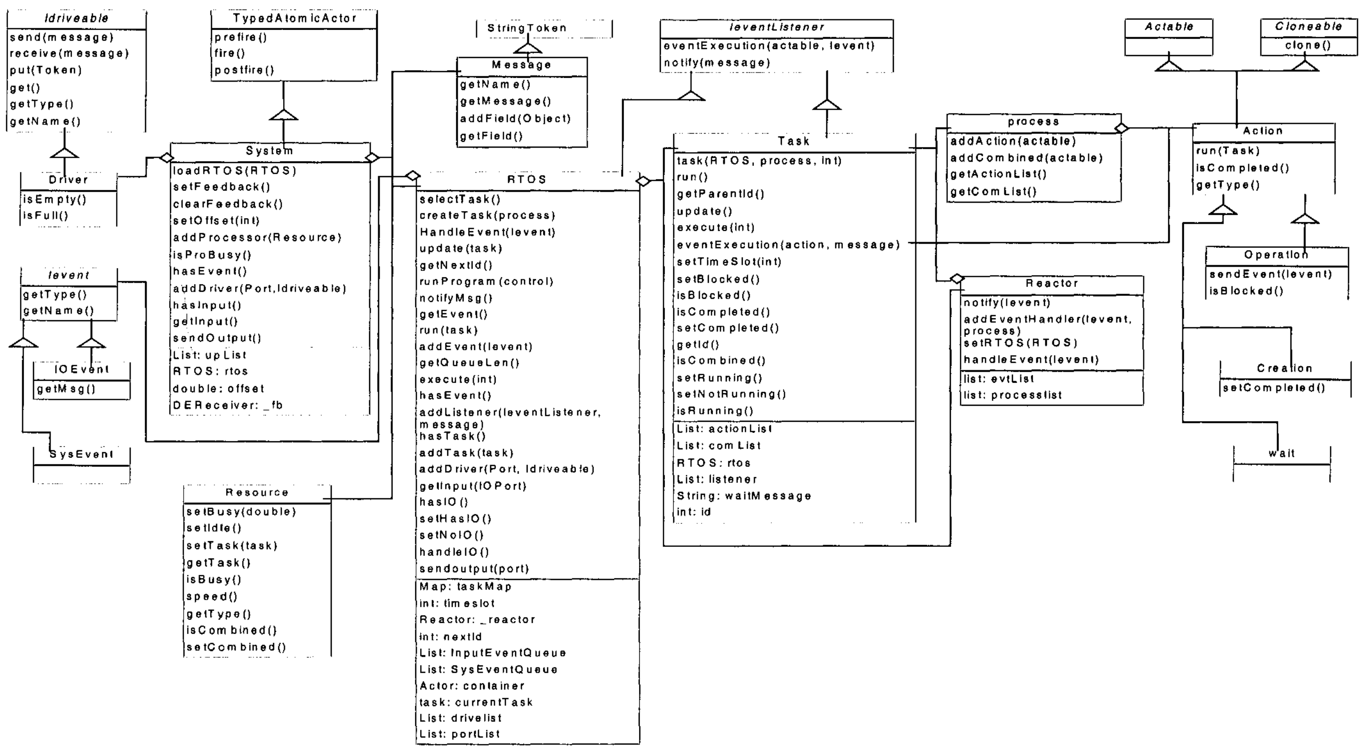


Figure 13: UML class diagram

3.3.1 Class Design

The different software actions mentioned in Subsection 3.1. are represented as a Java class. (For a full description of the Java language, see for instance [AGH00].) Unified Modeling Language (UML) is a general-purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of software systems [Boo94]. A UML class diagram visually shows the universal relationship of a set of classes to each other. A UML sequence diagram formalizes the behavior of the software systems and visualizes the communication among objects. Figure13 shows the UML class diagram of the class structure in this package. Below, we briefly list the functionality of the major classes.

System class - The System class is a subclass of the *Actor* class of PtolemyII. It represents the proxy class of the software virtual machine. This is the place where timing information of the execution of software action is passed back to the virtual time in the DE domain. It has input and output ports to receive and send *Token* to other components in the DE domain. It contains a reference to the RTOS and other system resources (CPU, memory etc).

RTOS class – This is where the functionality of the RTOS is modeled. This class can provide different scheduling policies, such as priority-based or round robin. The preemption feature is not supported at present, but it can be emulated by using different task priorities. The RTOS class includes an event queue that supports the message passing. A task graph is constructed under the control of this class to represent the run-time status of the software threads.

Process class – It includes a list of actions, which can be treated as the static software code.

Task class – This is the representation of run-time behavior and states of the thread. Task objects can be dynamically created by the RTOS object by prototyping the process object using the clone method of the *Process* class.

Reactor class – It implements the message dispatch mechanism in the event-driven software. Messages and their handlers (processes) are registered to it by the user.

Driver class – It simulates the functionality of the driver in the software virtual machine. It includes a buffer to hold the temporary input/output data and to perform the flow-control.

Resource class – It represents the hardware resource used by the software tasks such as memory, and processors which keeps the information (task id, next action completion time) of the tasks which are running on them to indicate and update their status.

All action classes implement an interface called *Actable*. An *Event* interface is defined for the classes used for internal communication between different classes (System, RTOS etc). Through using interfaces, our design is more extensible, and easily accommodates design changes and adding new features.

3.3.2 Design Patterns

Following is a brief discussion of the main software design patterns that are used in this package.

Reactor pattern – It is an event demultiplexing mechanism which channels all external events in an event-driven application through a single demultiplexing point. It allows handling of multiple events concurrently. Using this pattern, a task graph can be created and different tasks prototyped from the same process can co-exist in a single application. It is well suited for simulating the dynamic behavior of the event-driven software. In the initialization phase, messages and their handlers are registered to the

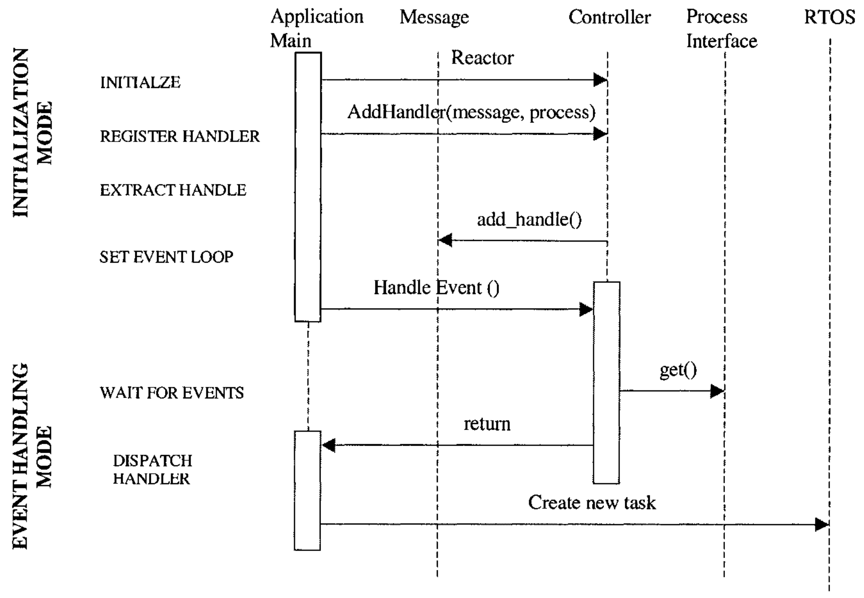


Figure 14: The sequence diagram of the reactor pattern

control object; during event handling phase; handlers are extracted to create tasks by the RTOS object.

Non-blocked buffering pattern – This pattern decouples input and output mechanisms, so it avoids blocking the application processing. This pattern is used in the system class and the driver class. In the driver operation phase, the data is put into the buffer of the driver and the driver registers its next output action time to the system object. When time arrives, the system wakes up the drivers to execute the output actions to the IO ports.

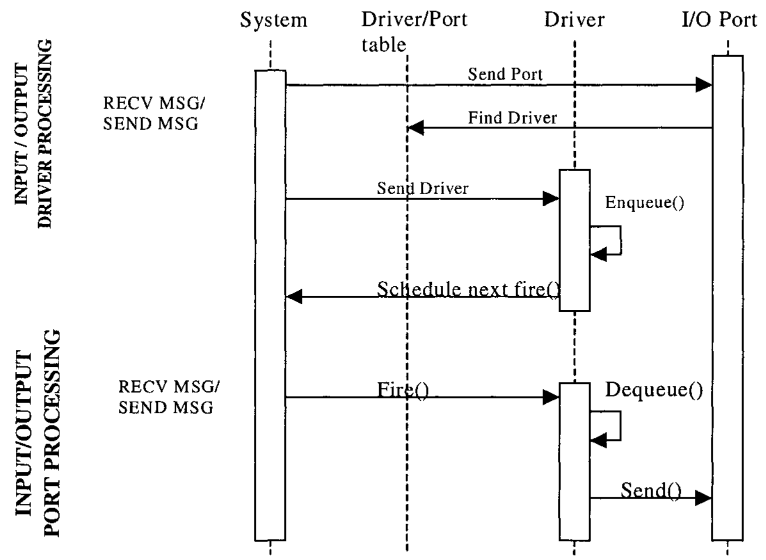


Figure 15: The sequence diagram of the non-blocking buffer pattern

3.4 Examples

We use two examples from the literature to illustrate how to use our method in realistic applications.

Our first example consists of the dashboard controller of a car, taken from Balarin et al [BC+97]. The controller consists of the following modules. The speed meter displays the speed of the car in a certain range. The odometer incrementally records the distance traveled starting from 0 km. The driver can reset the trip odometer. The only input is the wheel pulse that is generated from every rotation of the wheel.

After receiving the input from the wheel pulse, the system schedules the execution of these tasks to meet their timing constrains. Both the computation and the filtering can be implemented in software while filtering may also be implemented in hardware. A fast time budgeting can be carried out to try the different functionality partitioning of the system before starting the detail design of the system.

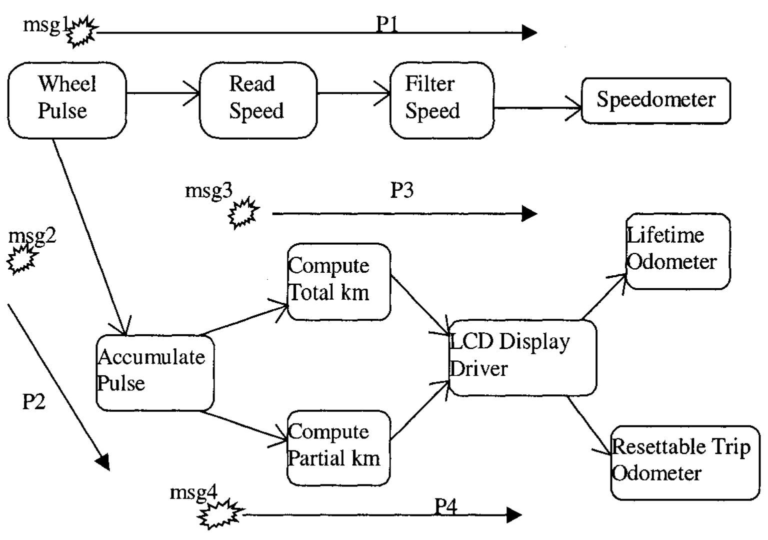


Figure 16: The dashboard example

We first decompose the system into different messages and processes. Then each process is refined to a list of actions. A control object is constructed to register all the message process pairs. Figure 16 shows the detailed message and process graph after the system decomposition.

A *Source* component in the DE domain is built to simulate the wheel pulse that generates the input message. The wheel pulse changes its frequency during the simulation. To each process, we assign a priority. From the simulation, we find that the task execution time remains the same within certain frequency range of the wheel pulse input, and starts to

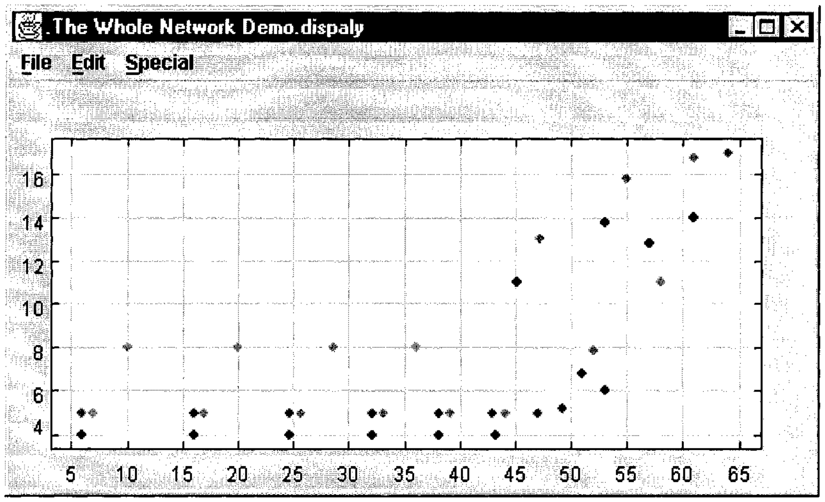


Figure 17: The software execution time plot of example 1

change when the frequency higher than a cut-off frequency. When we change the priority of each process, we observe the change in the pattern of the execution time. The higher priority processes finish faster than before and the lower priority processes get accumulated in the event queue in the RTOS as shown in Figure 17. By shifting functions between different software processes and/or between hardware and software, we can change the input frequency that causes the tasks to miss their deadlines. Through this example, we demonstrate the options of relocation of the functionality between software processes and between software and hardware to get the optimal design. Also, it is easy to observe the effect of the event queue in the RTOS which may take millions cycles to simulate few seconds behavior by using the cycle accurate simulation method.

The second example is a simplified model of a communication device, shown in Figure 18. There are several components in this example. The RX module is the hardware receiver that receives, amplifies and/or samples the receiving signal. The RXCU module is the RX control unit that controls the work of the RX module and generates some data to the local and remote system (like the stats report, control signaling etc). Some of the functions of the RXCU can be implemented in software, and some can be in hardware. The system includes a microcontroller (controller) to execute the reactive control software. Functions of the control software can either be implemented in hardware or in software. The SP module is the signaling processing module that implements certain signaling conversion functions in hardware to relieve some workload of the controller so that it can meet the real-time requirements. The TX module is the transmitter module that transmits the RF signal converting from the digital baseband signal. This is a complex system that may have hundreds of feature and finite-state control function.

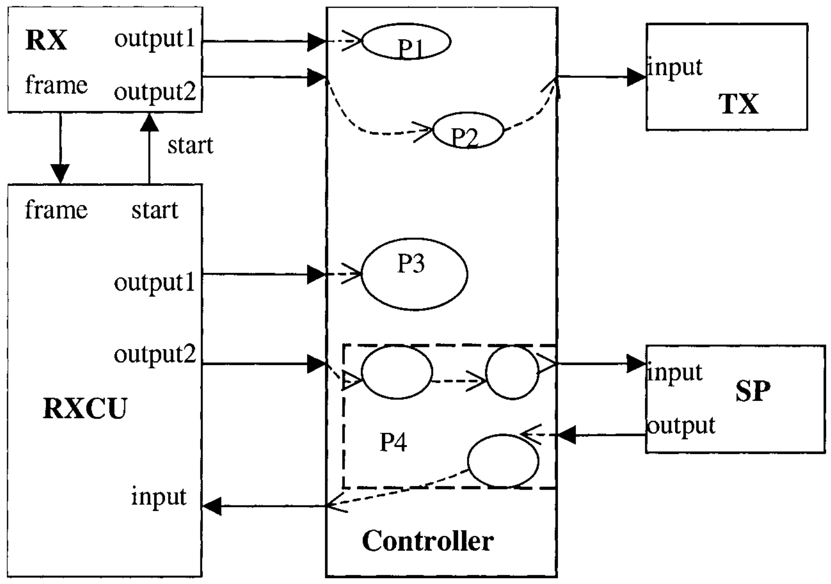


Figure 18: A communication device

Following is the explanation of how our example works. When the system starts, the RXCU module sends message via “start” port to RX, then sends some signaling messages to the controller with different fixed delay through output1 and output2 ports. The RX module sends messages to the controller via 2 ports periodically which acts sending data. The data is organized in frames. Each frame contains a fixed number of bits to represent data or signaling. The frames compose the hyper-frame that has the data frames and the signaling frames arranged in fixed patterns. After certain amount of data frame sent, it sends a frame message to RXCU to inform the start of the next hyper-frame. There are several processes running on the controller that respond to the inputs of these modules. After receiving the message from the first port of RX, it starts executing process one that only includes execution actions. Process two is invoked after receiving the message from second port of RX and the completion of the process one. Process two sends the message to the TX module after some delay that can represent the using of controller to execute some functions like data compression, adding error check bits, which have the option to be shifted to the TX module. Processes three and four are called after receiving the inputs from two ports to RXCU, respectively. Process three is a simple process like process one. Process four is a more complex one: it starts with some operations, then sends the message to the SP and waits for the response from the SP. After receiving the input from the SP, process four sends a message back to the output port to RXCU. The RXCU will iterate another cycle of sending messages to the controller only when it receives the response from the controller before receiving the frame signal from the RX. If the RXCU does not receive such a response, the RXCU has to wait for the next hyper-frame to send its signaling frames.

When the system is simulating, multiple software task instances exist, possibly in different stages. We start with the initial frequency of the data frames of the RX, increase its frequency gradually, because process one and two have strict time constrains, so we set the priority of these processes higher than that of processes three and four. With the reduction of the cycle time of the output of RX, it will increase the delay of the execution time of the lower priority process. At a certain point, the input of the RXCU will miss the next hyper-frame signal. We let the simulation to stop after two continuous hyper-frame misses of the RXCU module. From the simulation, we can observe that a long delay in process four causes the simulation to stop. We have the options to avoid it by shifting the function from software to hardware, or by using faster CPUs or multiple CPUs.

A more complex and refined model can be built by using this method. Instead of using the fixed execution time, one can use the execution time of a distributed pattern or use the data derived from the reference implementation.

To decompose the software processes into a sequence of actions and estimate their execution time, one can reference the design from the previous implementations if the new processes are just improvements of previous versions. Or, one can decompose the software into sub-modules that mainly use the function library: by annotating the function library with some distribution pattern, one can get the estimated time.

One the other hand, we can use this method to do the time budgeting of different software architectures before starting the detailed code design of the software.

3.5 Summary

We presented a software extension to PtolemyII that performs the fast timing estimation of the HW/SW embedded systems. Two examples are shown to illustrate the usage of our methods. The experiments show that:

- This package can well capture the dynamic characteristics of the event-driven software.

- The pure timing behavior simulation is useful for system performance estimation and time budgeting in system development.

Chapter 4

Synchronous and Asynchronous VLSI Design

For digital circuit design, how to synchronize the operation of different parts is an important issue. Choosing the right synchronization scheme will have the critical impact on the final performance of the circuit. Until now, using the clock signal to concert the operation of the whole circuit has been the most widely used design practice. With the transistor feature size shrunk into deep sub-micron range, this design style faces new challenges of timing closure for ever higher operation frequencies and of constraints on the power consumption of a single chip. Asynchronous design styles with their robustness of timing characteristics and low power properties are promising alternatives in low power applications.

In this chapter, we first discuss the architecture design of a CDMA correlator. Then, two different implementations -- asynchronous design using single-rail handshake circuits, and synchronous design using edge-triggered registers -- are presented. Finally a comparison is carried out to illustrate the differences between these two design styles, along with a study of the circuit switching activity.

4.1 The Correlator Design

The viability of a 64 Mbps all digital CDMA receiver hinges upon the development of a low power matched filter correlator: a receiver DSP may employ many such correlators in timing recovery, adjacent cell scan and data recovery units. The operation of the system requires all or some of these correlators working simultaneously.

$$Y = \sum_{i}^{N} W[i].X[i]$$

The correlator essentially functions as an accumulate-and-dump of N weighted inputs, where $X[i]$ is an input sample and $W[i]$ is the weight. For the receiver DSP, the input sample from the analog front-end is a 4-bits wide word clocked at 64 MHz. A sign-magnitude representation of data was chosen because the received data samples are spectrally white due to the spread spectrum nature of the system; from [Cha94], it has been shown that a sign-magnitude correlator results in 30% lower power consumption than an equivalent two's complement one. This is because of the nature of the weighting function $W[i]$, which is a 1 bit stream of +/- 1's corresponding to a Walsh code overlaid on a PN sequence. A simple toggling of the sign bit can easily realize the multiplication by +/- 1, it is extremely efficient from the power standpoint.

4.1.1 Specification

This chip has 7 input pads and 11 output pads along with the power supply pads. The input signals to the circuit are of a 4-bit wide data stream in sign-magnitude representation, a 1-bit wide Walsh code stream and a 1-bit wide PN stream. The clock pad connects to the clock signal of 64MHz. The output from this circuit is a 10-bit wide data stream in 2's complement representation synchronized at 1 MHz with the outside environment. It should operate at 3.3V power supply. Several extra pads are needed to act as test points to access to signals inside the chip. The power consumption and silicon area are not specified for this circuit The main purpose of this experiment is to compare the aspects of different implementation using synchronous and asynchronous design style, rather than push the edge of the CDMA systems.

4.1.2 Architecture Design

In [SB98], it presents the architecture exploration of a full-custom correlator design for low power. The main ideas are restated here for clarity since we adopt the similar approach. As the weighting function of the correlation is simply a sign-toggle, the operation effectively reduces to an accumulation. The main element of functionality is the addition/subtraction of 3 magnitude bits over 64 samples at 64 MHz, which requires 9 bits of magnitude plus 1 bit of sign for dynamic range, with a resulting output dump frequency at 1 MHz. As a simple-minded first attempt, one possible implementation is to use a straightforward two's-complement ripple adder and literally accumulate the data after multiplication by the weighting sequence. Since the incoming data is sign-magnitude, it would need to be converted to two's complement in this approach. This sign-extension causes significant additional power consumption, but it is not the worst aspect of this design. The carry chain

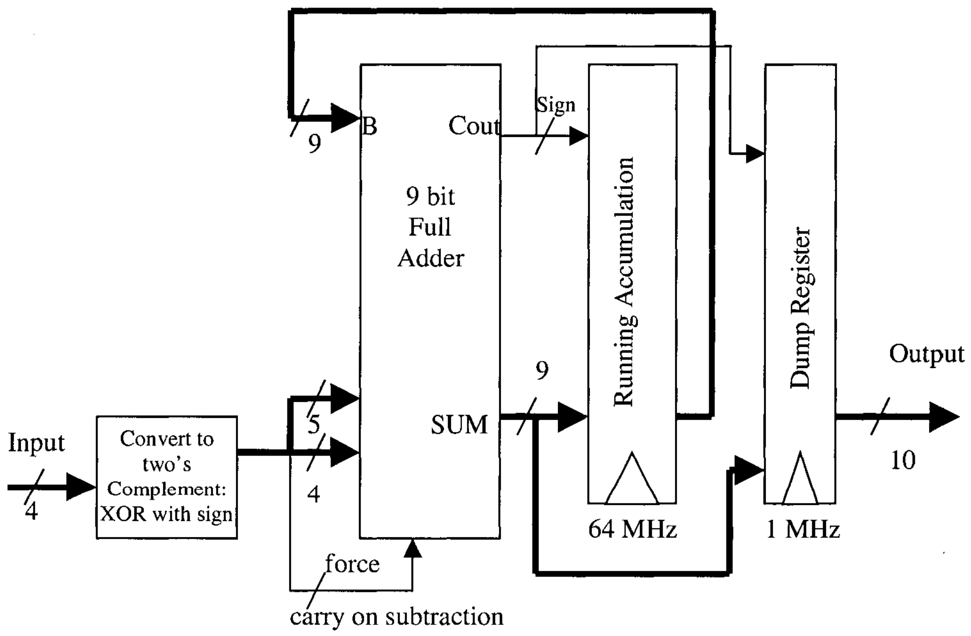


Figure 19: Two' complement implementation of the CDMA correlator

of a ripple adder must complete in 15.6 ns minus the register setup and delay times and any clock skew; this is tight time budget in the worst case that the carry signal must propagate from the lowest bit to the highest bit.

To preserve the advantage of the sign-magnitude nature of the incoming data, an alternative architectural approach is to immediately break up the accumulation into two parts: one accumulation of all incoming positive data and another accumulation of all incoming negative data. The sign bit can be used to multiplex the 3 bits of magnitude to either the positive or the negative accumulator, and the difference between positive and negative values can be computed after dumping at the 1 MHz rate by simply including a subtractor after the dump register. This approach has the advantage that the final subtractor will take negligible power at 1 MHz and has plenty of time to compute, but increases the area

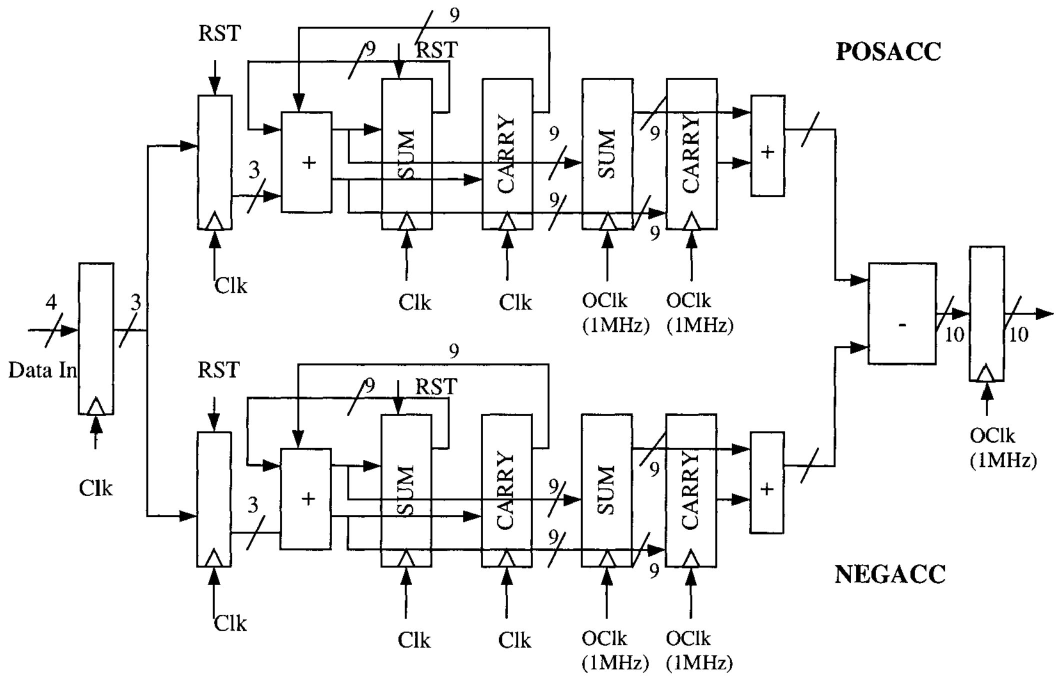


Figure 20: The alternative architecture with POSACC and NEGACC two datapaths

slightly. In the following discussion, the positive-data accumulator will be designated POSACC, and the negative-data accumulator will be designated NEGACC as shown in Figure 20.

Another architectural optimization is to cut down the critical path by pipelining the entire carry chain. Thus, the bit-level carry pipelining of the carry-save reduces the critical path to that of a single bitslice of a full adder cell. A carry-save architecture needs the use of two register banks, one to hold the temporary sum vector and another to hold the temporary carry vector. The cost of this replication is extra area for registers and a separate adder to recombine the sum and carry vectors after the 1 MHz dump operation. Not surprisingly, the adder is one of the most studied digital arithmetic blocks, and an overview of the more common designs can be found in the reference [Rab96].

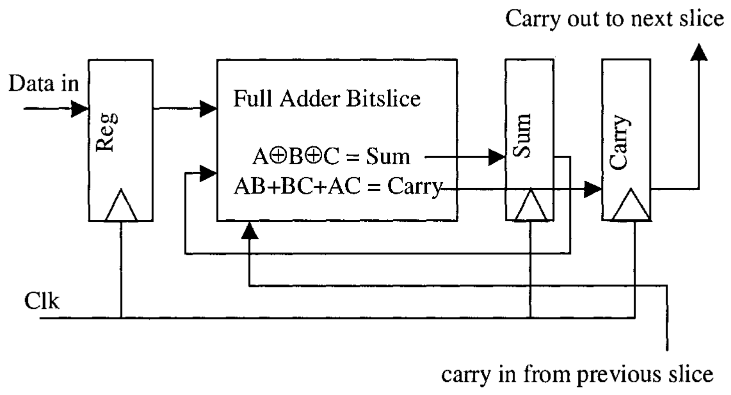


Figure 21: The carry-save adder

4.2 Asynchronous Design

Both the asynchronous and the synchronous designs of the CDMA correlator are presented in this thesis. The asynchronous implementation uses the single-rail handshake circuits which will be discussed in this section; the synchronous implementation is presented in the section afterwards. In this section, we present the design of several handshake components, the top level schematic and the design of several sub-modules. We also discuss in this section some important design issues and the asynchronous circuit design flow.

4.2.1 Handshake Components

The handshake components are computational modules that communicate on their input and output channels via handshake protocols. These handshake circuits must coordinate the *request* and *acknowledge* signaling exchanges with their environment and implement the required computational functionality as well. In the following, we list the handshake components used in our design. Some component designs refer to the work of [Pee96].

4.2.1.1 Case Components

Case components are used in the implementation of case construct in VLSI programming to decode the expression to select the appropriate statement to be executed. For the implementation of the case component, the data-valid scheme on data channels is critical. If the data-valid scheme on *a* is broad, then valid data may be assumed during the complete handshake on *a*. This implies that the handshakes on *b* and *c* apparently do not directly influence this data. With the broad data-valid scheme on *a*, the case component can be implemented as shown below. The AND-gates decode the incoming data, and depending on the a_0 either initiate a handshake along *b* or along *c*.

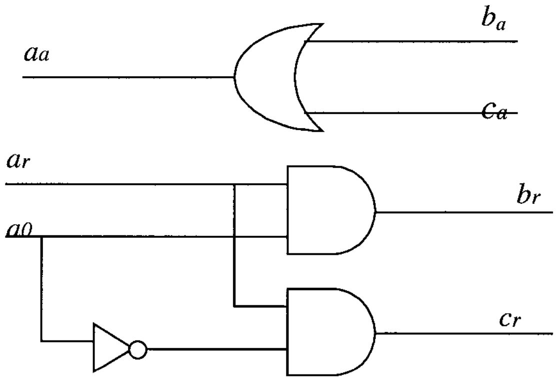


Figure 22: The case component

4.2.1.2 Passivator

The passivator is used to synchronize two active partners during a data transfer. It synchronizes data exchange between an active sender and an active receiver.

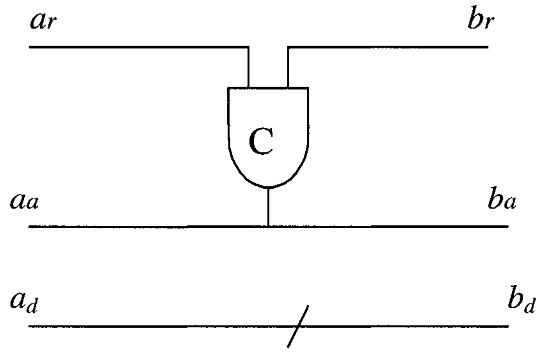


Figure 23: The passivator component

A four-phase implementation of the passivator is shown above. The data-release signal of channel a is directly connected to the data-valid signal of channel b. This circuit can be used to combine the broad data-valid scheme along a with the early scheme along b.

4.2.1.3 Mixer

The isochronic fork is introduced by Burns and Martin [BM88] and is considered to be an essential and the "weakest possible" compromise to true delay insensitivity. An isochronic fork is a fork for which we assume that the difference in delays between the two branches is less than the delay through the gates to which the fork is input. Circuits in which the only timing assumption is that of the isochronic fork are generally called *quasi delay insensitive* (QDI). The term speed independent basically refers to the same class of circuits. An example of the application of isochronic forks is the four-phase implementation of the nonput mixer. The nonput mixer requires the handshake on a and b to be mutually exclusive. For the four-phase protocol, this implies that the environment guarantees that a_r and b_r will never be high simultaneously.

The efficiency of the mixer implementation (in term of area, delay and power) can be improved by replacing the symmetric C-elements by asymmetric C-elements. The mixer implementation illustrates that circuit realizations can be made more efficient by making additional timing assumptions. Especially in the application of isochronic fork the replacing symmetric C-elements by asymmetric variants is an interesting one that is often applied in the implementation of handshake components.

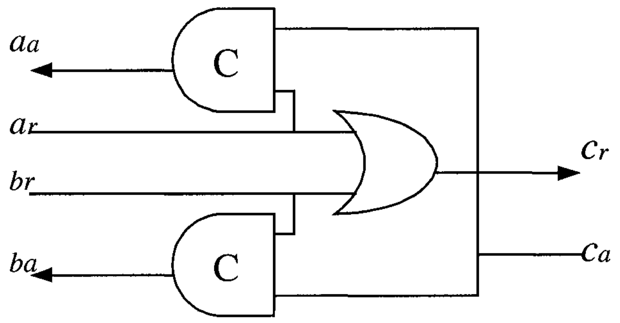


Figure 24: The mixer implementation

4.2.1.4 Four-Phase Micropipeline Controller

As introduced before, mircopipelines are a very commonly used pipeline architecture in the practical asynchronous circuits design. Micorpipelines are viewed as being composed of a control circuit employing the two-phase or four-phase handshake protocol and a datapath

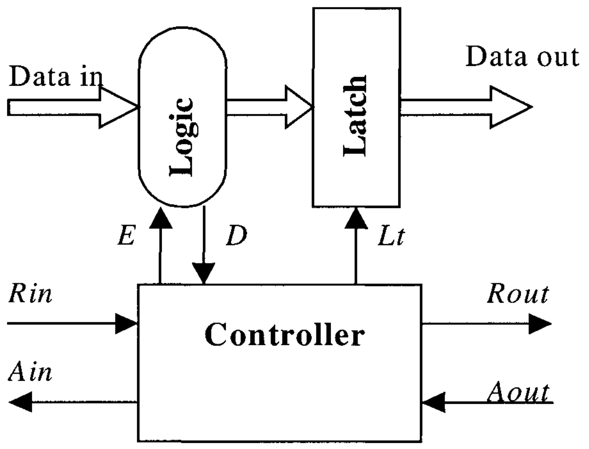


Figure 25: Single micropipeline stage

using the bounded delay model.

Figure 25 shows a general micropipeline stage structure. The latch control circuit communicates with neighboring pipeline stages on both its upstream link (*Rin*, *Ain*) and its downstream link (*Rout*, *Aout*). The control links (*E*, *D*) connect with associated combinational logic block. In addition to these three handshake links, a latch control wire (*Lt*) is needed to open and close the latch when it is low or high, respectively. The rising edge of *Rin* that indicates "data available" is usually used to activate combinational logic.

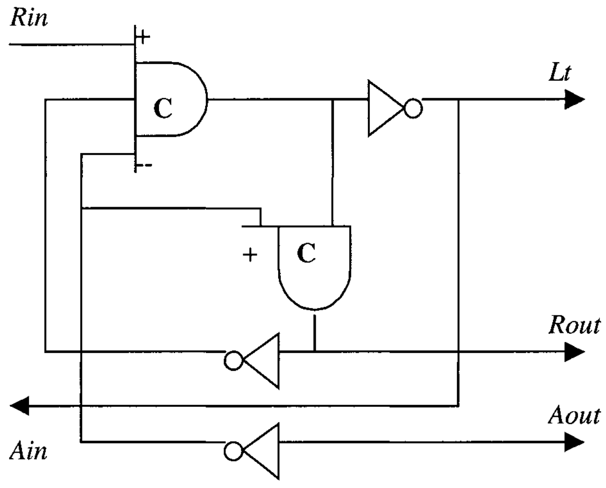


Figure 26: The early scheme and semi-decoupling micropipeline controller

To implement the micropipeline, another important factor that needs to be considered is the degree of decoupling. A micropipeline stage is said to be decoupled if it satisfies the following two conditions: (1) a new communication coming along its upstream link cannot be "initiated" until the current communication going along its downstream link has been "completed", (2) and it is "suspended" if the new communication along its downstream link has not been "initiated". A micropipeline stage becomes semi-decoupled by removing the first condition, and it becomes fully-decoupled by also removing the second condition [Liu98]. A new communication along the input link of a fully-decoupled latch control circuit may be "completed" before the new communication along its downstream link has been "initiated".

In our correlator design, several variants of micropipeline controller are used. The early data scheme and semi-decoupling version is shown in Figure 26. Within this circuit a new communication on the upstream link can be "initiated" before the current communication on the downstream link has been "completed", but it is suspended until the new communication on the downstream link has been "initiated".

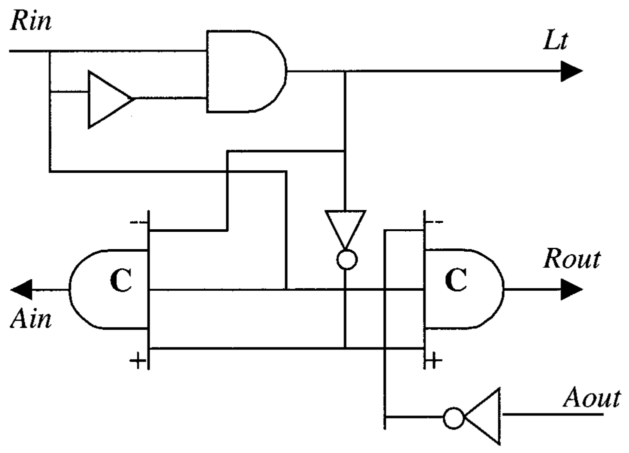


Figure 27: Another implementation of micropipeline controller

Another kind of micropipeline controller used in our design is shown in Figure 27. In this implementation, the input *Rin* signal will create a pulse at the latch control output (*Lt*). Then it triggers signal on *Rout* to go through the 4-phase handshake signaling exchange.

4.2.2 Top Level Architecture

The top-level schematic of our asynchronous correlator design consists of a control path circuit and two data path circuits. One data path is called accumulator data path along which input data samples are summed up; another data path is called counter data path along which a counter counts how many cycles have been processed in one frame and generates the corresponding control signals.

The handshake components steer the data flow along the data paths. The micropipeline controllers control the memory elements that represent the variables along the data path.

They separate the previous stage and the next stage of the micropipeline, so that they can operate in various degree of parallelism depending on the type of micropipeline controller used. Without pipelining operations of the circuit, the handshake signals issued by the handshake components just form a single handshake channel, in which the *acknowledge* signal to the first handshake component in the chain will appear only after the completion of all the handshake signaling of the rest handshake components in the chain. This reduction of parallelism would limit the speed and the throughput of the handshake circuit. By using the micropipeline structure in asynchronous circuits, the throughput of the circuit will increase similarly to the way it does in a pipelined synchronous circuit. Micropipeline controllers are also used in the control path for the correction operation of latches in some cases.

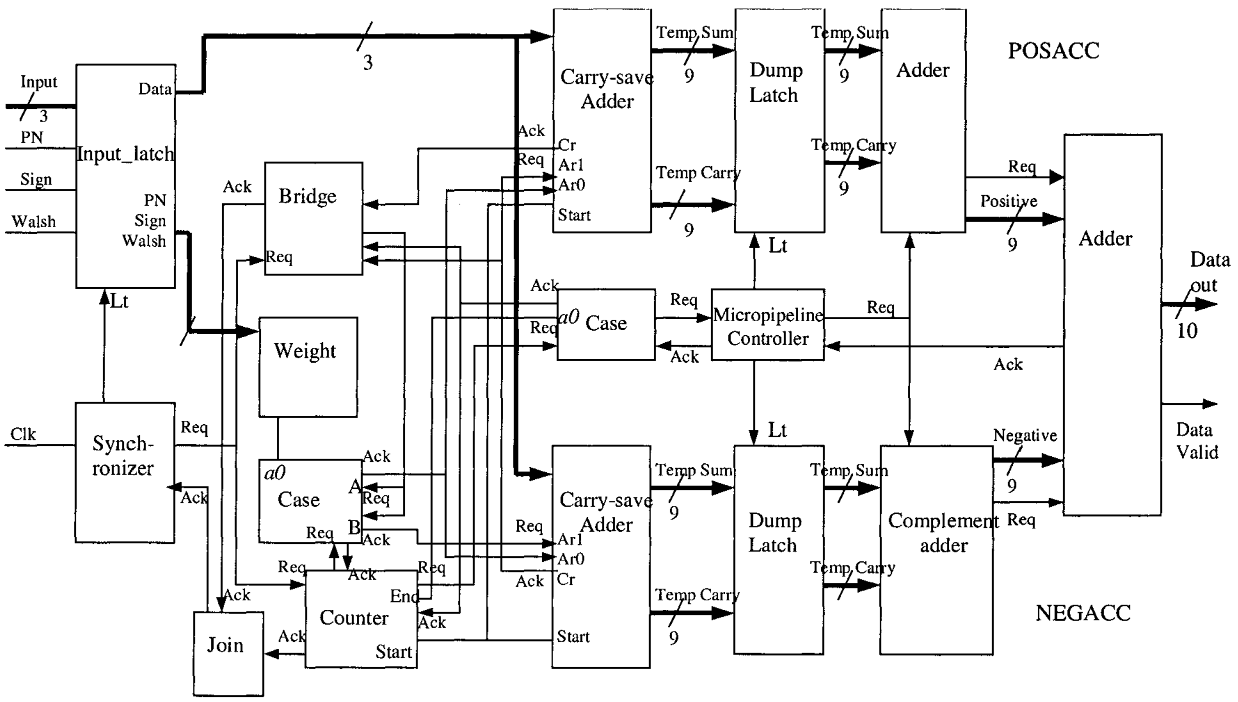


Figure 28: The top level schematic

According to this concept, the accumulator data path needs one micropipeline controller which controls dump latches. When it is the last cycle of one data frame, the micropipeline controller controls the dump latches to latch the sum and carry vectors from the positive and the negative adder networks. So the addition operation of accumulation adder networks can run parallel with the addition operation of the dump adders which have the option to choose simple and low speed adder structure. For the counter data path, which generates the control signals according to the value of the counter, there are two micropipeline controllers. One micropipeline controller pipelines the increment operation of the counter which is implemented as a carry-save adder network; another controller is used to control updating the control signal latches. Our asynchronous correlator is designed to accept inputs from a clocked environment. When the rising edge of the *Clk* signal comes, the synchronizer latches the data and generates the *request* signal to initiate the actions along the accumulator data path and the counter data path. Along the counter data path, the counter updates the *End* signal and the *Start* signal which indicate the last and the first cycle of the data frame, when its value is high, respectively.

The *End* signal of the counter is used by the downstream case component to initiate the operation of adding the positive and negative values to generate the circuit output. When the *End* signal is high, the case component sands a *request* signal to the micropipeline controller in the accumulator data path. The micropipeline controller starts the operation of dump addition where the dump latches latch the sum and carry vectors from the positive and the negative accumulation adder networks, then one adder adds positive sum and carry vectors, another adder performs a 2's complement of the addition of negative sum and negative carry vectors, afterwards the third adder adds them together as the output of the circuit. The availability of results at the output is indicated by the "Data out" signal. Synchronization of the circuit output to a clocked environment can be realized by sizing the delays of the circuit to match a specific number of periods of the clock. When is *End* signal is low, nothing happens, the *request* signal from case component is used directly as *acknowledge* signal to the case component itself.

After the counter has updated the *End* and *Start* signals, it sends the *request* signal to another case component. Meanwhile the weight component generates the selection signal to this case component to steer the input data to either the positive or the negative part of the accumulator data path depending on the sign of the input sample, the Walsh code and the PN. The bridge component synchronizes the *acknowledge* signals from the two accumulator data paths which indicate the completion of adding the new input sample. The join component merges the *acknowledge* signals from both the bridge component and counter data path to indicate that the circuitry is ready for accepting the next input sample.

4.2.3 Major Sub-Circuits

For our asynchronous correlator design, we designed several new handshake components, in addition to using common handshake components from [Pee96]. These new components will be presented in the following subsections.

4.2.3.1 Synchronizer

To interface the asynchronous circuit to the synchronous inputs, a synchronizer has been designed. It accepts the clock signal as input from one side and initiates the handshake

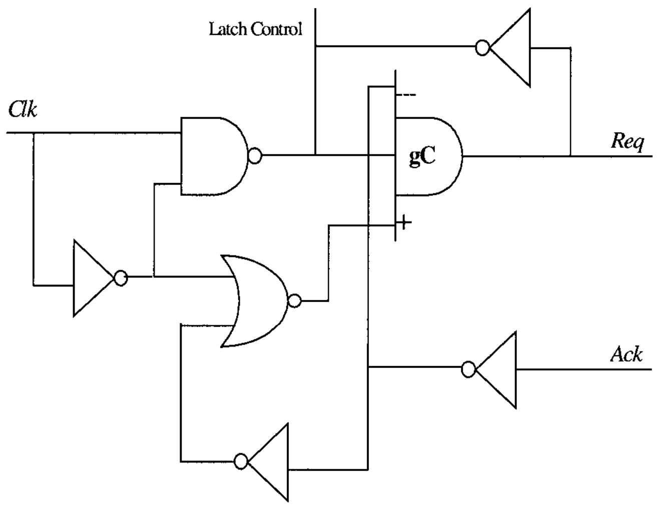


Figure 29: The schematic of the synchronizer

signals to another side of the circuit. If the *acknowledge* signal for one cycle comes too late, after the rising edge of the next cycle clock signal, the synchronizer will not generate the *request* signal for the next clock cycle, which means that the speed of circuit is too slow to finish the first stage of the action within one clock period.

4.2.3.2 Counter

The counter unit in this design needs to generate the control signals in two continuous cycles of the data inputs to indicate, the last cycle of the frame, and the start of the new frame, which are called *End* signal and *Start* signal, respectively.

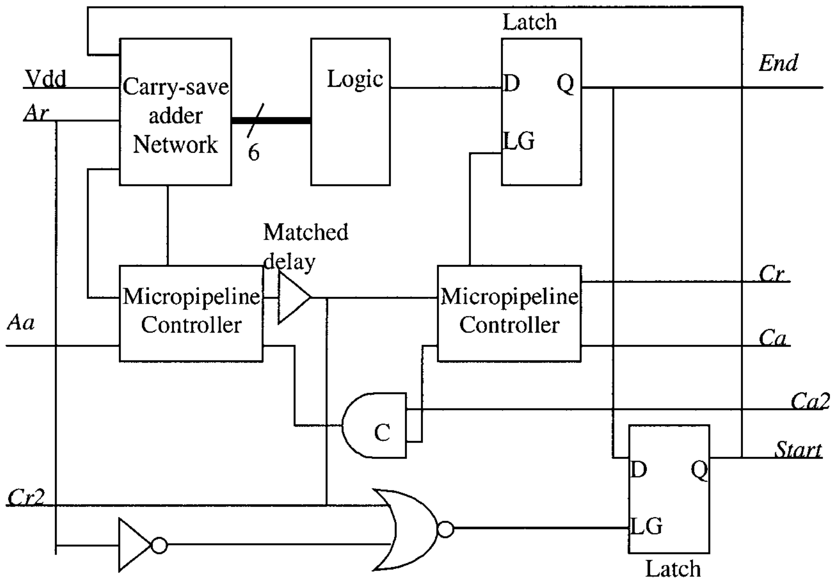


Figure 30: The schematic of the counter component

The counter consists a carry-save adder network that increments its value whenever a *request* signal *(Ar)* comes. There are two latches in the counter to represent the *End* and *Start* signals. One micropipeline handshake controller controls the pipelined carry-save adder and another micropipeline controller controls the *End* and *Start* latches. The *Start* latch captures the signal of *End* latch before it changes to the new value. The adder network in the accumulator data path has to wait for the *Start* signal from the counter which indicates whether the adder needs to reset to zero before starts to add the new data item. The *Start* signal also resets the adder in the counter itself to zero. The counter uses the handshake signals *Cr2* and *Ca2* to communicate with the case component in the upstream of the accumulation adder networks and utilizes the *Cr* and *Ca* signals to control the case component in the downstream of the dump adder network. These two handshake signals merge at the first micropipeline controller using a passivator in the counter unit as shown in the Figure 30.

4.2.3.3 Accumulation Adder

The accumulation adder in the accumulator data path consists a N-bit wide carry-save adder and the control logic. A N-bit wide carry-save adder is composed of N one-bit full adder slices. Each carry-save adder unit has one sum and one carry latch. The carry of the low bit adder connects to the carry input of the higher bit adder. The control logic circuit is the modified asynchronous latch controller. The latch is opaque between two handshake signalings. When the *request* signal comes, it generates a short pulse signal to make the latch transparent to latch the input signal. Both POSACC and NEGACC adder network receives the handshake signal to itself and the handshake signal to its counterpart.

Either POSACC or NEGACC adds the new input data to its temporary sum and generates the new temporary sum and carry vectors when the *request* signal to it comes and the *Start* signal is low. When the *Start* signal is high, the adder will reset itself to zero when *request* signal to other part of the adder networks comes, otherwise the adder network clears the previous temporary sum and carry vectors and adds the new input data sample.

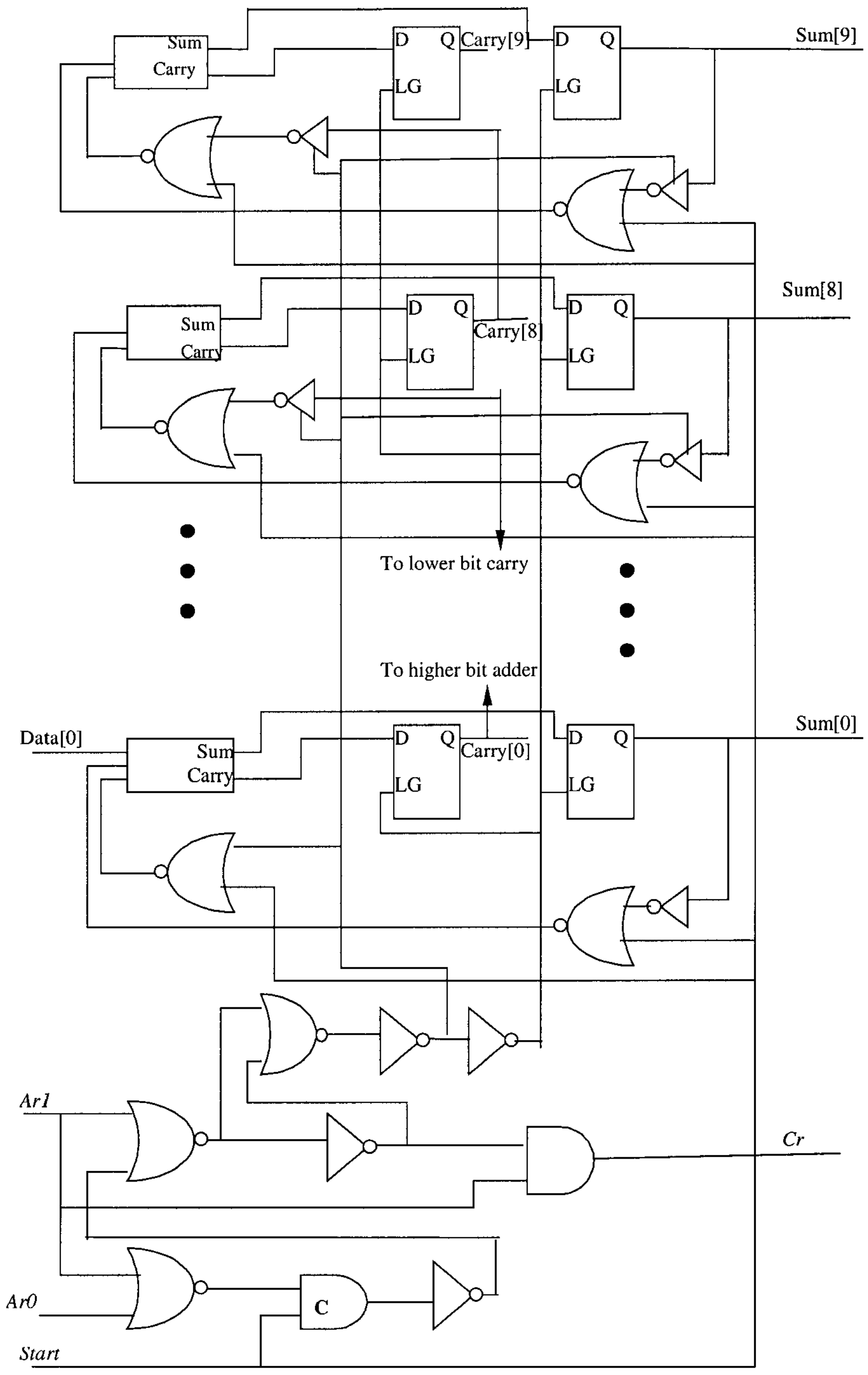


Figure 31: The schematic of the carry-save adder

4.2.4 Asynchronous Circuit Design Flow

The full chip design is composed of logic design stage and physical design stage. In the logic design stage, the circuit is designed and simulated to generate the gate-level netlist description of the circuit. In the physical design stage, the circuit components in the netlist are placed and routed to generate the physical layout information.

For the correlator design, the function of the correlator is first described in some sort of the programming language. Then the structure of the source code is analyzed and optimized to maximally explore the parallel operations. Then the optimized code is manually refined to a handshake network composed of handshake components communicating through handshake channels. Afterwards the design of each handshake component and module is captured in the schematic editor. These designs use the gates chosen from the standard cell library. The circuit level simulator – SPICE -- is used to tune the delay match of these circuitry. The whole design is also simulated in SPICE to verify its functionality and timing.

Afterwards, the design is exported from the schematic editor in EDIF (electronics design interchange format) format. This file is used by the synthesis tool to link each element to its Verilog model file. The output of the synthesis tool is the Verilog netlist description of the circuit. Then it is gone through the Auto Place&Route using the commercial tools (Design Planner and Silicon Ensemble from Cadence Design System) as done in the synchronous design flow to generate the physical layout data.

4.3 Synchronous Design

In this section, we briefly introduce the synchronous design of the correlator for the purpose of comparison against the asynchronous design. First we present the digital design flow to put this design in perspective. Then we discuss our design. Comparing to its asynchronous counterpart, the synchronous version is quite straightforward and simple.

4.3.1 Digital Design Flow

Currently, the most effective and successful design methodology of digital circuit design is to use the high level description language (HDL) to design and simulate the circuit, then synthesize it into different implementations with the aid of electronics design automation (EDA) tools. A digital circuit is first described in the source code of some type of HDL, either VHDL or Verilog. Then the functional and timing correctness of the code are verified against the specification by using the HDL simulator. Afterwards the source codes are fed into the synthesis tool to be synthesized to generate the gate-level netlist according to the chosen implementation. What synthesis does is translation plus optimization. In most situations, the problem of finding a optimal solution to the design is so hard that run-time of algorithms for the overall problem is prohibitive long or it is impossible. It is better to use the two stage procedures of translation plus optimization to approximate the optimal solution in a short run-time.

The steps mentioned above are usually called front-end steps since they do not involve any of the physical information of the implementation. The steps that follow the front-end steps are usually called back-end steps in which gate-level netlists generated at the front-end stage are mapped to the physical layout information.

The main challenge in the current design flow is the timing closure that is to meet the timing constrains of the design in the back-end stage. Since they are two separate stages of the current design flow the high level synthesis and the physical level placing-and-routing, if the constrains of design are set too aggressive, it may take very long time to close the timing or even unable to close the timing, if the constrains of the design are set too loose, it may sacrifice the performance. To achieve the best performance, it may need several iterations and design expertise.

4.3.2 Synchronous Design

The design is described in the Verilog source code. It uses mixed structural style coding and RTL style coding. The adder network is described as a group of carry-save adder units connected directly. The counter is written in RTL code. The top level RTL code is written to reflect the two paths architecture as described in Figure 20.

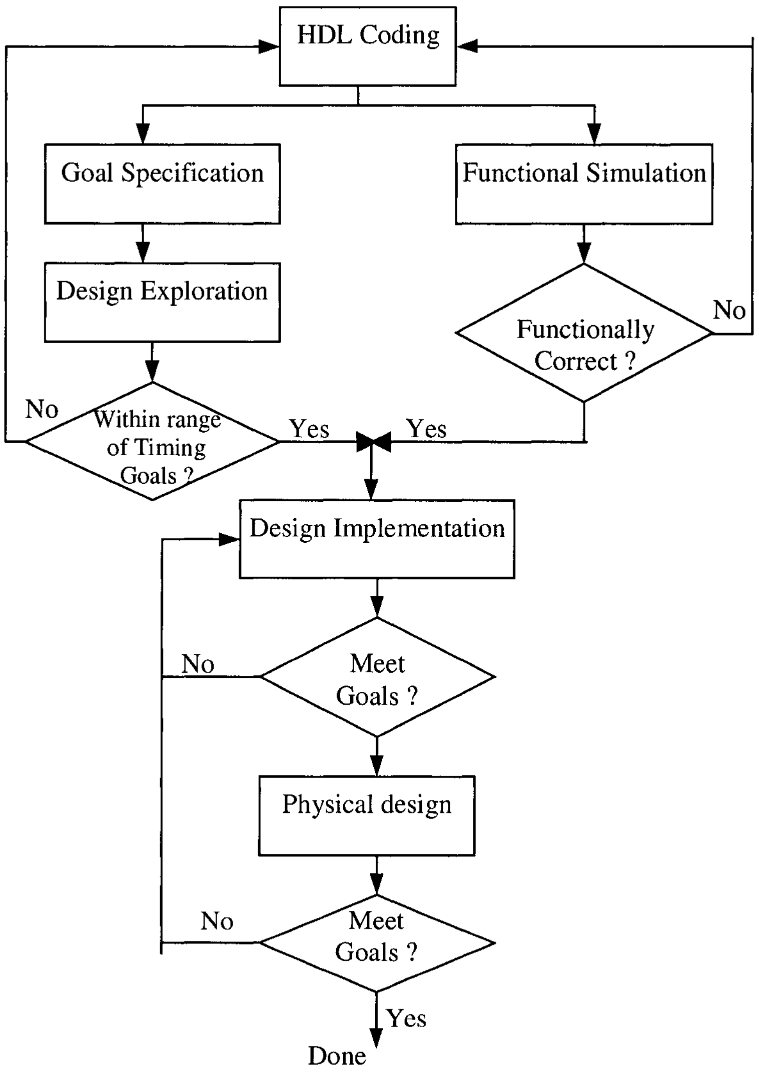


Figure 32: The synchronous digital design flow

Also a testbench is developed for simulation proposal. It generates the source signals which are fed to the correlator model, then the correlator outputs are compared against the result of the direct correlator function to verify the design. It is simulated in a commercial event driven simulator, the Verilog-XL from Cadence, to simulate its functionality. Then the source code is undergone the synthesis step in the synthesis tool, in this case, the Design Compiler from Synopsys. Afterwards the design is mapped to specific technology, the ASIC implementation using the standard cell library from Canadian Microelectronics Corporation (CMC).

4.4 Power Analysis for Architecture Exploration

The power consumption is an important design concern in the modern VLSI design, especially in the deep sub-micron design. High power consumption increases the cost of the chip in term of package and cooling technology. In addition to cost, there is the reliability issue. High power systems often run hot, and high temperature tends to exacerbate several silicon failure mechanisms [Ped96].

The power dissipation of digital CMOS circuits is mostly caused by following four sources:

- *Leakage current*, which is mainly decided by the fabrication technology. It consists of two parts, the first is the reverse bias current in the parasitic diodes formed between source and drain diffusions and the bulk region of a MOS transistor, and the second is the subthreshold current that arises from the inversion charge that exists at the gate voltage below the threshold voltage.

- *Standby current*, which is the DC current drawn continuously from power source to ground.

- *Short-circuit current,* which is the current gone through the DC path during the output transitions; and

- *Capacitance current,* which flows to charge and discharge the capacitive loads during the logic changes.

The dominant factor of the power dissipation of the CMOS circuits is the charging and discharging of the capacitive nodes. It can be represented by the following formula:

$$P = C_L \times V_{dd}^2 \times Sw$$

where C_L is the total physical capacitance at the output of the node, V_{dd} is the power supply voltage, Sw is the average number of signal transitions on that node. Reducing the supply voltage is the most effective way of reducing power consumption, but at cost of reduced performance. When the supply voltage is set by the external requirement, or when the performance degradation caused by lowering the supply voltage is not acceptable, the only way to reduce the power consumption is by lowering the physical capacitance or switching activity, or both. While the physical capacitance is mainly decided by the circuit family and processing technology, the architecture and logic designs have the most significant impact on the switching activity.

Calculation of the switching activity of a logic circuit is difficult since it depends on several circuit parameters and technology dependent factors that are not available or precisely characterized. Some of these factors are input pattern, gate delay model, logic function, logic style and circuit structure etc. Still the simulation-based method is the most commonly used method for switching activity analysis today.

As mentioned before, architectural level design decision will have the most important factor affecting the total amount of switching activity of the final design, thus affecting the total power consumption of the chip. Here we use the PtolemyII to simulate and analyze the switch activity of two different architectures, one using 2's complement to represent the data with one data path, and another using the sign bit plus the magnitude to represent the data with two data paths. The simulation environment is set up as shown in Figure 33.

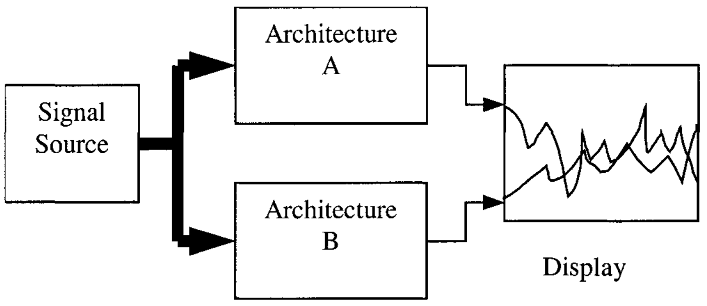


Figure 33: The switching activity analysis simulation setup

The source, architecture A and architecture B are components in the DE domain of PtolemyII. Architecture A is the one using 2's complement adder, and architecture B is the one using two data paths. The source component generates the PN, Walsh code and the clock signal with the random data samples. These signals are fed to both architecture A and architecture B. According to their different architecture, each node is simulated to record its switching activity. One snapshot of the simulation is shown in Figure 34. The dot is number of switching activity of the architecture A and the cross is that of the architecture B. From the figure, we can see on average architecture A has about 10~20% more switching activity than that of architecture B. The reasons for that are, first the sign extension in the input data converting of architecture A may cause more switching activities, second, the adder network of architecture A may cause more switching activities when the accumulated data changes sign. From the figure, we also observe that sometime architecture B has more switching activity than architecture A. This is because architecture B needs to latch both the temporary sum and carry data vector, whereas architecture A only needs to latch the temporary sum. At the end of the frame, architecture B needs to perform the subtraction operation which is absent in architecture A.

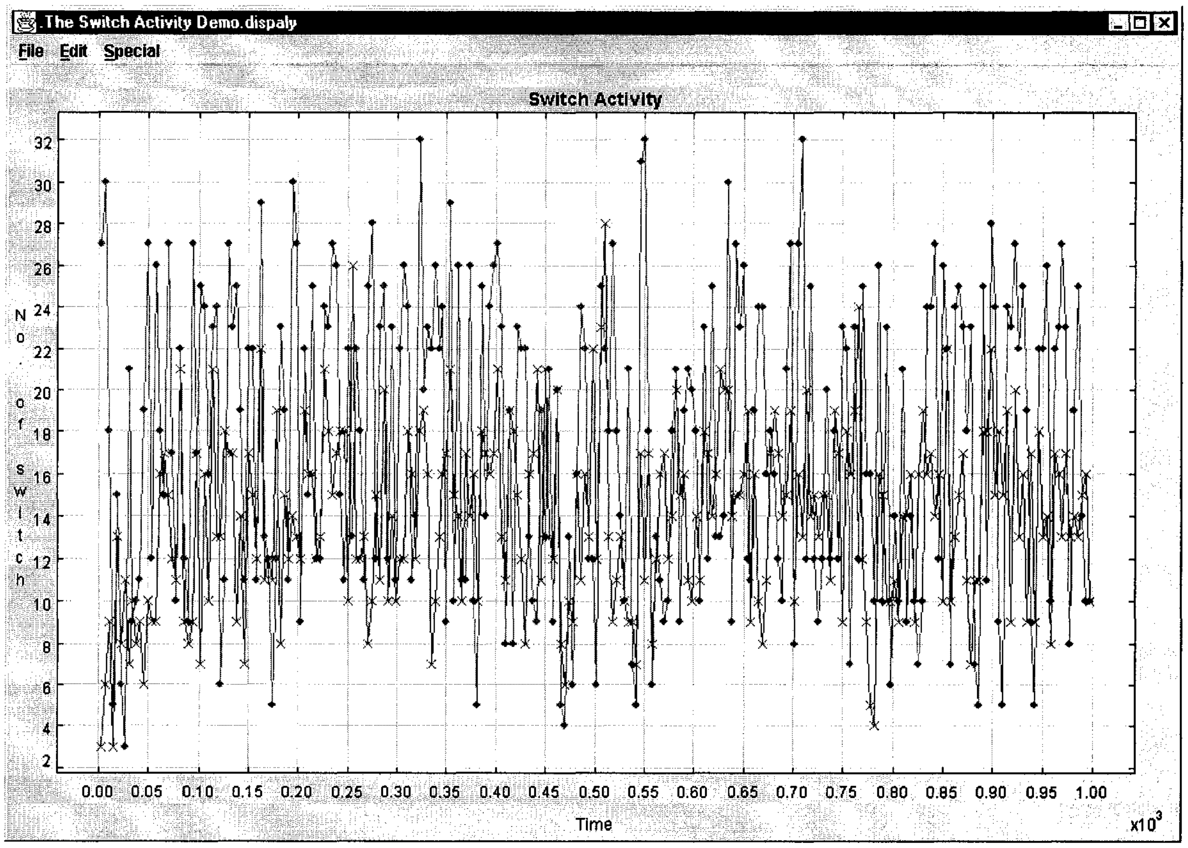


Figure 34: One example of switching activity plot

4.5 Performance Comparison

The performance comparison is mainly based on the results of simulation and synthesis of the design. For digital design, mostly we use the gates from the standard cell library, as we did for this design. The library provides various characteristic data of the cells in the library, which includes the operation condition, max, min and typical delay and other useful information needed by the simulation and synthesis tools. Comparing to the real result from the silicon, these results from simulation and synthesis are accurate enough to indicate the difference of the designs.

There are a lot of points worth noticing in these two design experiments. First, asynchronous design uses more area compared to the synchronous design. It uses 813 cell instances from a standard cell library, whereas the synchronous design uses only 654 cell instances from the same standard cell library. For the power consumption, the asynchronous design uses average 17mw at 3.3 supply voltage for the 64 MHz data stream, and 0.9 mw at 1.5v for the 25M Hz data stream from the SPICE simulation. The die size of this chip is 3mm x 2 mm as shown in the layout view below. We can see from the layout that this chip is pads bounded, leaving a real core area is of 130855 square microns.

	Area (μm^2)	Power(64M/s, 3.3v)	Power(25M/s, 1.5v)
Asynchronous	130855	17 mW*	0.9mW
Synchronous	114935	5.4076 mW	not available.

Table 2: The asynchronous design and synchronous design comparison

A word of notice should be mentioned for reading the power value of this table. The value of synchronous design is obtained from the synthesis tool, and the value of asynchronous design is obtained by using circuit-level simulation of SPICE. Because of the prohibitive long simulation time of whole frame of 64 cycles, this value is obtained by setting per frame to four cycles. We can expect that the power value will be much smaller if we use per frame of 64 cycles. Also the power simulation is input pattern dependent as mentioned before. To verify the design, the input pattern that we used for simulation tries to sensitize more paths in the design, it may also cause more switching activity for this input pattern and increase the power consumption.

Another interesting point in this comparison is the area. Because this application is computation intensive, the area overhead of handshake circuits is not that critical. Another fact reflected in this value is that the memory elements are implemented as latches in the handshake circuit, but as flipflops in the synchronous design. Usually the flipflops use more area than that of the latches. Therefore, for wider data paths, the area ratio would change in favor of asynchronous circuits.

For the asynchronous design the delay of the pipeline stage of that of the data path is about one iteration of bit slice adder. In this case, the delay of the micropipeline controller is comparable to or even larger than the delay of the data path that it controls. So reducing the delay of the micropipeline controller can speed up the circuit. As seen from the schematic, various C-elements are used in micropipeline controller design. These C-elements are implemented by using gates from the standard cell library. By using full custom design C-element, we can predicate faster speed and less power consumption of the resulted circuit.

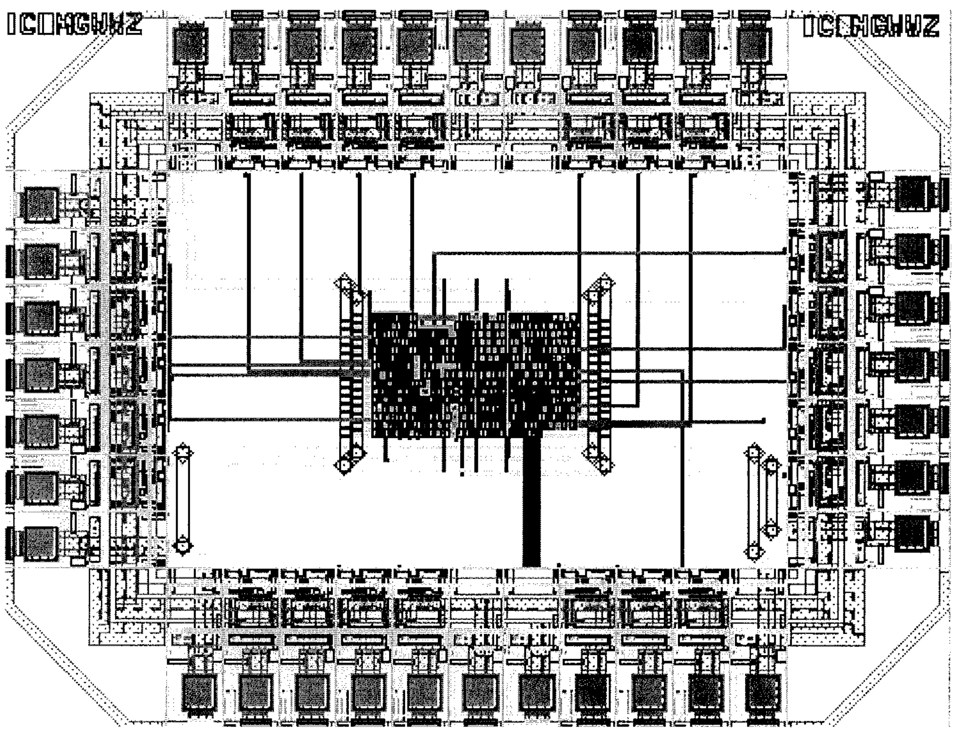


Figure 35: The layout of the asynchronous design

Figure 35 shows the layout of the asynchronous design. The layout of physical design of the synchronous circuit is quite straightforward according to the steps of CMC digital design flow. This stage is omitted in this thesis. This circuit itself is a fairly small design for the modern mature digital design flow. The simulation result and the estimation values from the synthesis tools are good approximation of the final design that can be achieved by the experienced designer.

The detailed breakdown lists of the cell usage of both designs are attached as Appendix 1 at the end of this thesis.

The final physical design of the asynchronous chip is done in Cadence design environment by using the standard cell and pad library developed by CMC. The circuit passed the LVS (Layout via Schematic) check without including the extracted pad models and DRC (Design Rule Check) with the pad layout. We sent this design for fabrication by TSMC (Taiwan Semiconductor Manufacture Corporation) in March 2001 and got the chip back in July 2001. During test, we found there are short paths existing inside the power supply network. After diagnosis, we found that there exist short path between VSS ring pads and the VDD code pads in this fabrication run.

4.6 Summary

In the Chapter, we have discussed the advantages and disadvantages of synchronous and asynchronous design styles.

We use the CDMA correlator as a case study to illustrate many aspects of the practical design issues. For asynchronous design, we have described our single-rail handshake circuit design in detail. For synchronous design, we have presented the digital design flow in today's VLSI design.

Also a switching activity study is carried out to show the importance of the architecture selection on the final system performance. Although we have not been able to reproduce the power economy reported in [Pee96], overall we can see that both design styles have its advantages and disadvantages, depending on factors such as application area, width of data path, idle time, the availability of design tools and etc.

Chapter 5

Conclusions and Future Work

This thesis has presented the research work of a simulator software design for modeling, simulation of hardware/software embedded systems and research work of VLSI circuit design of synchronous and asynchronous circuits. First, the simulator can be used for the fast performance estimation in functionality partition of embedded hardware/software systems. In this thesis, we illustrate the software design and its use of design patterns, we also include two examples to demonstrate the usefulness of our method. Second, after the general discussion of synchronous and asynchronous circuit design style, a case study – the CDMA correlator – is designed to compare various aspects of this two different design styles. Though the nature of work is mainly engineering, there are some ideas and contributions gained in the course of the work.

In the simulator design, we propose to abstract the temporal relation of the complex software design. Such abstraction will still keep the inherent temporal relationship between the hardware, software components of the system under design, meanwhile it will greatly simplify the task of the system simulation. Its integration into the PtolemyII – a heterogeneous simulation framework – makes it easy to combine with other models of computation. Through two examples, we show how embedded system with dynamic run-time behavior of event-driven software can be effectively simulated.

Using software design patterns is another important gain in this software package design. It makes software easy to maintain and to extend. Through using reactor and non-blocked buffering patterns, complicated algorithms (like task tree construction, priority ordering and concurrently execution etc.) can be wrapped into a simple interface. New features can be easily added.

For the comparison of synchronous circuits design versus asynchronous circuits design, several new asynchronous handshake circuits have been designed for the CDMA correlator. They are quite general handshake components. They can be used in other handshake circuits as well. A comparison of the ASIC implementations in term of the area and power is also presented. Each design style has its advantages and disadvantages. Most current circuits use synchronous design due to its smaller area, high performance, wide support of EDA tools and matured design flow. Meanwhile synchronous design faces more and more challenge for the power dissipation and clock skew in the deep sub-micron design. Conversely, very few commercial designs use asynchronous design. Lack of tool limits its adoption by the industry. However its excellent power properties and timing robustness make it a potential alternative for digital circuit design in the deep sub-micron range. For asynchronous design to become a mainstream design style, more research in circuit, signaling techniques and design methodology are need. Other alternative design styles such as global-asynchronous-and-local-synchronous [Cha84] might emerge as well.

There is still lots of work that can be done to extend the current work. For the simulator software design, various algorithms can be added, features can be enriched. For example, for the dynamic task tree construction, cycle detection can be added to indicate the deadlock in possible program execution. It will be useful that given a system model and a set of constrains, the simulator can interactively help designer find the optimal solution in hardware/software functionality partition.

Asynchronous circuit design is an active research area. One of the interesting new circuit family is so called "Gasp" circuits developed at SUN Microsystem Labs. In the "Gasp" circuits, they use only one wire to represent both the *request* and *acknowledge* signals. It can be used in the micropipeline controller circuit. The full custom design “Gasp” circuit has only three inverter delay of the one iteration of handshaking.

By replacing the four phase micropipeline controller circuits with those of “Gasp” circuits, it will increase the speed of the design. To do that, new interface circuitry between the four phase handshake circuits and the “Gasp” micropipeline controllers has to be designed.

Bibliography

[AGH00] K. Arnold, J. Gosling and D. Holmes, The Java Programming Language. 3rd Edition. Addison Wesley 2000.

[Amu] http://www.cs.man.ac.uk/amulet.

[Bal99] F. Balarin, Worst-case analysis of discrete systems. Digest of Technical Papers of the 1999 IEEE international Conference on CAD, Nov. 1999.

[BC+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, K. Suzuki and B. Tabbara, Hardware-Software Co-design of Embedded System: The POLIS approach. Kluwer Academic Boston, MA. 1997.

[BC+99] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich and K. Suzuki, Synthesis of Software Programs for Embedded Control Applications. IEEE Transactions on Computer-Aided design of Integrated Circuits and Systems. pp834-849 Vol. 18 No. 6, June 1999.

[Ber93] K. van Berkel, Handshake Circuits: an Asynchronous Architecture for VLSI Programming. Volume 5 of International Series on Parallel Computation. Cambridge University Press, 1993.

[BM88] S. M. Burns and A. J. Martin, Syntax-directed translation of concurrent programs into self-timed circuits. Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, pages 35-50. MIT Press, 1988.

[BM+97] I. Bolsens, H. de Man, B. Lin, K. van Rompaey, S. Vercauteren and D. Verkest, Hardware/Software Co-Design of Digital Telecommunication Systems. In Proceedings of IEEE, VOL. 85, NO. 3, Mar. 1997.

[Boo94] G. Booch, Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.

[Bry01] I. Brynjolfson, Dynamic Clock Management Circuits for Low Power Applications. M.Eng. Thesis, McGill University, April 2001

[CC+99] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly and L. Todd, Surviving the SOC Revolution – A Guide to Platform-Dased Design. Kluwer Academic, 1999.

[CG+94] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli and L. Lavagno, Hardware-Software Codesign of Embedded Systems. IEEE micro pp. 26-36, 1994.

[Cha84] D. M. Chapiro, Globally-Asynchronous Locally-Synchronous. PhD thesis, Stanford University, Oct. 1984.

[Cha94] A. Chandrakasan, Low Power Digital CMOS Design. Ph.D. Thesis, U. C. Berkeley, Berkeley, CA, 1994.

[DRG98] A. Dasdan, D. Ramanathan and R. Gupta, A Timing-Driven Design and Validation Methodology for Embedded Real-Time Systems. ACM Trans. Design Automation of Electronic Systems., pages 533-553 Oct.1998.

[EL+97] S. Edwards, L. Lavagno, E. Lee and A. Sangiovanni-Vincentelli, Design of Embedded System: Formal Models, Validation, and Synthesis. Proc. of IEEE pages 366-390 Vol. 85, No. 3. March 1997.

[GCD92] R. K. Gupta, C. N. Coelho Jr. and G. De Micheli, Synthesis and simulation of digital systems containing interacting hardware and software components, in Proc. Of the Design Automation Conf., June 1992.

[GH+95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley 1995.

[GM93] R. Gupta and G. De Michell, Hradware-Software Cosynthesis for Digital Systems. IEEE Design & Test of Computers. pp. 29-41 Sept. 1993.

[Hau95] S. Hauck, Asynchronous design methodologies: an overview. Proceedings of the IEEE, Vol. 83, pp: 69-93, 1995.

[HB+93] J. Haans, K. van Berkel, A. Peeters and F. Schalij, Asynchronous multipliers as combinational handshake circuits. Proc. IFIP Working Conf. Asynchronous Design Methods, Manchester, U.K., Mar. 31–Apr. 2, 1993.

[HM93] K. ten hagen and H. Meyr, Timed and untimed hardware/software cosimulation: application and efficient implementation, in Proc. of the int. Workshop on Hardware-Software Codesign, Oct. 1993.

[KL92] A. Kalavade and E. A. Lee, Hardware/software co-design using Ptolemy – a case study, in Proc. of the Int. Workshop on hardware-Software Codesign, Sept. 1992.

[Lee00] E. A. Lee, What's Ahead for Embedded Software?. IEEE Computer, September 2000, pp. 18-26.

[Lee01] E. A. Lee, Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.

[Liu98] J. Liu, Arithmetic and Control Components for an Asynchronous System. PhD Thesis, Dept. of Computer Science, University of Manchester, 1998.

[LS] IS-95 North American Standard – A CDMA Based Digital Celluar Systems. http://citeseer.nj.nec.com/32841.html

[Mes90] D. Messerschmitt, Synchronization in Digital System Design. IEEE Journal on Selected Areas in Communications. Vol 8. No. 8. Oct. 1990.

[MJ+97] C. E. Molnar, I. W. Jones, B. Coates and J. Lexau, A FIFO ring oscillator performance experiment. Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1997.

[Pet] http://www.lsi.upc.es/~jordic/petrify/petrify.html.

[Ped96] M. Pedram, Power Minimization in IC Design: Principles and Applications. ACM trans. on Design automation of Electronic Systems, Vol. 1, pp 3-56, January 1996.

[Pee96] Ad M. G. Peeters, Single-Rail handshake circuits. Ph. D. thesis, Edinhoven University of Technology, June 1996.

[PM96] J. G. Proakis and D. G. Manolakis, Digital Signal Processing: Principles, Algorithms, and Application. Prentice-Hall. Inc. 1996.

[Pol] http://www-cad.eecs.berkeley.edu/~polis/

[PPT00] J. M. Paul, S .N. Peffers and D .E. Thomas, A codesign virtual machine for hierarchical, balanced hardware/software system modeling. Design Automation Conference, 2000. Proceedings 2000 , 2000 , Page(s): 390 –395

[Pto] http://ptolemy.eecs.berkeley.edu/ptolemyII/

[Rab96] J. M. Rabaey, Digital Integrated Circuits. Prentice Hall, 1996.

[Row94] J. Rowson, Hardware/Software co-simulation, in Proc. Of the Design Automation Conf. 1994, pp.439-440.

[SB98] S. Sheng and R. Broderson, Low-Power CMOS Wireless Communications – A Wideband CDMA System Design. Kluwer Academic Publisher, 1998.

[SGR99] K. Stevens, R. Ginosar and S. Rotem, Relative timing. Proceedings of. International Symposium on Advanced Research in Asynchronous Circuits and Systems ASYNC'99, pp. 208-218, 1999.

[SP94] S. Sutarwala and P. Paulin, Flexible modeling environment for embedded systems design, in Proc. of the int. workshop on Hardware-Software Codesign, 1994.

[SS96] K. Suzuki and A. Sangiovanni-Vincentelli, Efficient software performance estimation methods for hardware/software codesign. In Proc. Design Automation Conf., pages 605-610, Jun. 1996.

[SS+00] D. C. Schmidt, M. Stal, H. Robnert and F. Bushmann, Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2. New York, NY: Wiley & Sons. 2000

[Sut89] I. E. Sutherland, Micropipelines. Communications of the ACM, 32(6):720-738, June 1989.

[TAS93] D. E. Thomas, J. K. Adams and H. Schmitt, A model and methodology for hardware-software codesign, IEEE Design and Test of Computers, vol. 10 no. 3, pp. 6-15, Sept. 1993.

[TB+97] Y. Taur, D. A. Buchanan, W Chen, D. J. Frank, K. E. Ismail, S. H. Lo, G. A. Sai-Halasz, R. G. Viswanathan, H. C. Wann, S. J. Wind and H. Wong, CMOS Scaling into the Nanometer Regime. Proc. of IEEE, pp. 486-504 Vol. 85,No.4, April 1997.

[Vit95] A. J.Viterbi, CDMA: principles of spread spectrum communication. Reading, Mass.: Addison-Wesley Pub. Co., 1995. 245 p.

[Wil94] J. Wilson, Hardware/software selected cycle slution, in Proc. Of the Int. Workshop on Hardware-Software Codesign, 1994

[ZNZ01] W. Zhu, R. Nagulescu and Z. Zilic, Using Design Patterns for Fast Hardware/Software Performance Estimation. IEEE ICT2001 June, 2001.

Appendix 1

A. The cells breakdown list of the asynchronous design of the CDMA correlator

```
CELL INSTANCE DATA
  #Master      #Instances
  wand2_1       192
  wand2_2        36
  wand2_4         5
  wbuf_1          1
  wbuf_2          3
  wbuf_4         15
  wdp_2          11
  winv_1         19
  winv_2         16
  winv_4         39
  winvzp_2       47
  wlp_2           8
  wlrp_2         86
  wnand2_1       32
  wnand2_2        6
  wnand2_4        1
  wnand3_1        2
  wnand3_2        1
  wnor2_1         1
  wnor2_2        54
  wnor2_4         4
  wnor3_1         2
  wnor4_1         1
  wnor4_2        29
  wor2_1         42
  wor2_4          2
  wor3_1         51
  wor4_1          3
  wxor2_2       104
  Total: #Masters=37 #Instances=861
```

B. The cell list breakdown of the synchronous design.

```
****************************************
Report : cell
Design : core
Version: 2000.05
Date   : Wed Jun 20 17:45:31 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
core_DW01_add_10_0              6105.000000       1   6105.000000  h
core_DW01_add_10_1              6105.000000       1   6105.000000  h
core_DW01_sub_10_0              7212.000000       1   7212.000000  h
counter                         7545.000000       1   7545.000000  h, n
p_adder_0                      39391.000000       1  39391.000000  h, n
p_adder_1                      39387.000000       1  39387.000000  h, n
wdp_2              wcells        360.000000      22   7920.000000  n
winv_1             wcells         67.000000       4    268.000000
wnor2_1            wcells         88.000000       7    616.000000
wxor2_2            wcells        193.000000       2    386.000000
-----------------------------------------------------------------------------
Total 10 references                                 114935.000000

****************************************
Report : reference
Design : counter
Version: 2000.05
Date   : Wed Jun 20 17:50:01 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
counter_DW01_inc_7_0            1845.000000       1   1845.000000  h
wand2_1            wcells        109.000000       6    654.000000
wdp_2              wcells        360.000000      10   3600.000000  n
winv_1             wcells         67.000000       2    134.000000
wnand2_1           wcells         88.000000       4    352.000000
wnand3_1           wcells        109.000000       1    109.000000
wnand4_1           wcells        130.000000       2    260.000000
wnor2_1            wcells         88.000000       4    352.000000
wnor3_1            wcells        109.000000       1    109.000000
wnor4_1            wcells        130.000000       1    130.000000
-----------------------------------------------------------------------------
Total 10 references                                   7545.000000

****************************************
Report : cell
Design : counter_DW01_inc_7_0
Version: 2000.05
Date   : Wed Jun 20 17:56:28 2001
****************************************

Cell                      Reference       Library             Area  Attributes
------------------------------------------------------------------------------
U5                        winv_1          wcells             67.00
U6                        wnor2_1         wcells             88.00
U7                        wnand2_1        wcells             88.00
U8                        winv_1          wcells             67.00
U9                        wnor2_1         wcells             88.00
U10                       winv_1          wcells             67.00
```

```
U11                       wnand3_1        wcells            109.00
U12                       wxor2_2         wcells            193.00
U13                       wxor2_2         wcells            193.00
U14                       wxor2_2         wcells            193.00
U15                       wxor2_2         wcells            193.00
U16                       wmux2_2         wcells            172.00
U17                       wmux2_2         wcells            172.00
U18                       wnand2_1        wcells             88.00
U19                       winv_1          wcells             67.00
-------------------------------------------------------------------------------
Total 15 cells                                             1845.00

****************************************
Report : reference
Design : p_adder_0
Version: 2000.05
Date   : Wed Jun 20 18:00:19 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
adder_u_0                       302.000000       1    302.000000  h
adder_u_1                       302.000000       1    302.000000  h
adder_u_2                       302.000000       1    302.000000  h
adder_u_3                       302.000000       1    302.000000  h
adder_u_4                       302.000000       1    302.000000  h
adder_u_5                       302.000000       1    302.000000  h
adder_u_6                       650.000000       1    650.000000  h
adder_u_7                       650.000000       1    650.000000  h
adder_u_8                       302.000000       1    302.000000  h
wand2_1            wcells       109.000000       2    218.000000
wdp_2              wcells       360.000000      72  25920.000000  n
winv_1             wcells        67.000000       5    335.000000
wnand2_1           wcells        88.000000      54   4752.000000
wnor2_1            wcells        88.000000      54   4752.000000
-----------------------------------------------------------------------------
Total 14 references                                 39391.000000

****************************************
Report : reference
Design : core_DW01_add_10_0
Version: 2000.05
Date   : Wed Jun 20 18:03:29 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
wand2_1            wcells       109.000000       1    109.000000
winv_1             wcells        67.000000       6    402.000000
wmux2_2            wcells       172.000000       3    516.000000
wnand2_1           wcells        88.000000      23   2024.000000
wnand3_1           wcells       109.000000       1    109.000000
wor2_1             wcells       109.000000       4    436.000000
wxor2_2            wcells       193.000000      13   2509.000000
-----------------------------------------------------------------------------
Total 7 references                                   6105.000000

****************************************
Report : reference
Design : adder_u_7
Version: 2000.05
Date   : Wed Jun 20 18:09:27 2001
****************************************
```

```
Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
wnand2_1           wcells        88.000000       3    264.000000
wxor2_2            wcells       193.000000       2    386.000000
-----------------------------------------------------------------------------
Total 2 references                                    650.000000

****************************************
Report : reference
Design : adder_u_1
Version: 2000.05
Date   : Wed Jun 20 18:11:23 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
wand2_1            wcells       109.000000       1    109.000000
wxor2_2            wcells       193.000000       1    193.000000
-----------------------------------------------------------------------------
Total 2 references                                    302.000000

****************************************
Report : reference
Design : core_DW01_add_10_1
Version: 2000.05
Date   : Wed Jun 20 18:13:40 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
wand2_1            wcells       109.000000       1    109.000000
winv_1             wcells        67.000000       6    402.000000
wmux2_2            wcells       172.000000       3    516.000000
wnand2_1           wcells        88.000000      23   2024.000000
wnand3_1           wcells       109.000000       1    109.000000
wor2_1             wcells       109.000000       4    436.000000
wxor2_2            wcells       193.000000      13   2509.000000
-----------------------------------------------------------------------------
Total 7 references                                   6105.000000

****************************************
Report : reference
Design : core_DW01_sub_10_0
Version: 2000.05
Date   : Wed Jun 20 18:15:38 2001
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
wand2_1            wcells       109.000000       3    327.000000
winv_1             wcells        67.000000       7    469.000000
wmux2_2            wcells       172.000000       3    516.000000
wnand2_1           wcells        88.000000      23   2024.000000
wor2_1             wcells       109.000000       9    981.000000
wxor2_2            wcells       193.000000      15   2895.000000
-----------------------------------------------------------------------------
Total 6 references                                   7212.000000
```