Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor

Rakefet Kol

Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor

RESEARCH THESIS

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science

Rakefet Kol

Submitted to the Senate of the Technion - Israel Institute of Technology
Elul, 5757 Haifa September 1997

To my mother, with endless love.

The research was done under the supervision of Dr. Ran Ginosar from the Department of Electrical Engineering, and Prof. Michael Yoeli from the Department of Computer Science.

My deepest gratitude to Dr. Ran Ginosar and Prof. Michael Yoeli for their devoted guidance and invaluable help and comments. Their encouragement to explore new and exciting areas has taught me a lot and is greatly appreciated.

Table of Contents

Table of Contents (cont.)

Table of Contents (cont.)

List of Figures

List of Figures (cont.)

List of Tables

Abstract

Future semiconductor technology (such as 1 billion transistors per chip and over 1 GHz clocks planned for the year 2010) places severe new constraints on the design of high performance microprocessors. In particular, the chip is too large and the clock is too fast for single clock synchronous operation. Rather, new forms of distributed architectures and asynchronous interconnects are called for.

This research describes the architecture of the asynchronous microprocessor *Kin*, which supports out-of-order and deep speculative *Avid* execution. *Kin* contains unique architectural features, targeted to achieve high performance by utilizing generous hardware resources that will become available by future technology. The research concludes that technological constraints necessarily lead to asynchronous solutions. The development of *Kin* included addressing and solving problems at the architecture level, as well as developing architectural concepts and design methodologies for the required building-blocks. The highest level of *Kin*'s architecture is asynchronous, while the various units of *Kin* may be implemented internally as either asynchronous or synchronous.

A new branching lookahead strategy, *Avid* execution, is shown to offer reduced misprediction penalty and increased performance. *Avid* execution prefetches and executes the predicted path as well as some of the non-predicted paths. Unneeded paths are dynamically pruned. The depth of the alternative paths is dynamically adjusted according to the branch prediction accuracy and confidence. Pathmarks are added dynamically to instructions for identification and efficient pruning. Analytical study shows that *Avid* execution can significantly increase performance over a single path speculative execution with similar resources. Simulation of SpecInt95 benchmark confirms the analysis results. *Avid* execution is most suitable for asynchronous processors like *Kin*, since it incurs dynamically changing computation loads at the various processor modules.

Decoding a variable length instruction set is a bottleneck in a high performance microprocessor. The architecture and design of an optimized asynchronous instruction length decoder (AILD) is presented as an example of a fully asynchronous module.

A novel doubly-latched asynchronous pipeline (DLAP) architecture is introduced. DLAP offers improved performance over previous asynchronous pipelines in important special cases. DLAP is also suitable as the target for an automatic synchronous-to-asynchronous conversion, and the proper algorithm is described.

As a transition from fully synchronous to fully asynchronous implementations, *Kin* can be implemented as a *multi-synchronous* system, wherein a common clock is distributed over thin wires, avoiding the massive power investment in clock distribution trees and circuits for phase matching and skew minimization. *Adaptive synchronization* reduces the probability of synchronization failures. In contrast with methods like clock stretching, adaptive synchronization adjusts data delays. We show that it is more widely applicable to high performance microprocessors than other synchronization methods. Training sessions are devised to minimize adaptation overhead.

Finally, statechart CAD methodology is adapted for the formal specification of asynchronous systems. It is also useful for generating simulation models, validations, and direct synthesis. Statecharts have been employed intensively in this thesis for the design and simulation of *Kin*, *Avid* execution, AILD, and DLAP.

List of Symbols and Abbreviations

μOp	-	micro-Operation
AILD	-	Asynchronous Instruction Length Decoder
A/S	-	Adaptive Synchronization
BU	-	Branch Unit
CAD	-	Computer Aided Design
CISC	-	Complex Instruction Set Computer
CL	-	Combinational Logic
DIC	-	Decoded Instruction Cache
DIT	-	Dynamic Instance Tag
DLAP	-	Doubly-Latched Asynchronous Pipeline
EU	-	Execution Unit
FF	-	Flip Flop
FIFO	-	First In First Out
FSM	-	Finite State Machine
GALS	-	Globally Asynchronous Locally Synchronous
ILP	-	Instruction Level Parallelism
LSU	-	Load/Store Unit
MRR	-	Most Recent Results
MTBF	-	Mean Time Between Failures
PMU	-	Prune Management Unit
PU	-	Prefetch Unit
qDI	-	quasi Delay Insensitive
RISC	-	Reduced Instruction Set Computer
ROB	-	ReOrder Buffer
ROU	-	ReOrder Unit
RRU	-	Register Renaming Unit
RS	-	Reservation Station
STG	-	Signal Transition Graph

Chapter 1 : Introduction

Microprocessor performance has risen over the past 20 years from 500 KIPS (thousand instructions per second), to 300 MIPS (million instructions per second) today. The industry plans to achieve 100 BIPS (billion instructions per second) by the year 2010, through the integration of almost one billion transistors on a chip operating at over 1GHz [SIA94, Wei96]. This explosive growth in performance has been made possible thanks to the rapid development of semiconductor technology [Yu96], and improvements in architecture. Technological progress has contributed both to higher clocking frequencies and to growing levels of integration. As more transistors are integrated, more architectural features (pipeline, superscalar processing, out-of-order execution, caches, etc.) are introduced into microprocessors, contributing to their growing utility.

While this impressive growth is expected to continue in the future [SIA94], we are facing a turning point in computer architecture. The methodology that has brought us from the early microprocessor days to the present is about to change. All microprocessors, past and present, are designed as synchronous, single clock machines. In the future, this will no longer be feasible: The basic axioms of synchronous design are intended for limited equipotential domains (where signal propagation times over all wires are negligible). Synchronous methodology has been stretched a bit further by able designers and power-hungry skew minimization techniques. But future large chips will extend well beyond the synchronous domain simply because it will take any signal (clock or data) many clock cycles to propagate from one part of the chip to another. In simple electrical engineering terms, the processors of the future will transcend from lumped systems into distributed ones.

This change has already started. Modern processors have introduced some elements of distributed computing, such as decoupling modules with FIFO buffers and executing out-of-order. However, at the circuit level, present day processors still insist on a single clock with minimal skew. In this thesis we show that this aging paradigm is best exchanged for asynchronous architecture, which is much more suitable for distributed systems. And in contrast with all previous research on asynchronous architecture, we emphasize the high-level architecture, rather than the asynchronous design of each and every circuit. In fact, while the physical constraints dictate asynchrony at the high level, the individual modules may still be synchronous, and we investigate this in the thesis. Unlike all other research projects on asynchronous processors, we do not promote a revolutionary shift in all aspects of design and CAD tools. Rather, we consider how contemporary synchronous designs can smoothly evolve into asynchronous ones. With this in mind, we search for the path

that the industry will be willing to adopt, namely one that will not enforce any change until it has become absolutely necessary.

This thesis investigates a processor architecture for the asynchronous future, including a novel aggressive speculative execution method (necessary for high speed and suitable for asynchronous processors). The thesis also delves into a number of associated issues: Fully asynchronous design of one module, algorithmic conversion of synchronous pipelines into asynchronous ones, mixed timed globally asynchronous locally synchronous systems, and the design methodology suitable for high level asynchronous design.

1.1 Future Technological Constraints and Asynchronous Design

In this section we summarize the relevant constraints that may be posed by 2010 technology and which directly affect this thesis [KG97].

While the electromagnetic field travels in vacuum at the speed of light (c = 30 mm / 100 pSec, in VLSI terms), the electric signals inside chips progress about 10-100× slower, depending on the drive strength (how much power and area are invested) and on the capacitive load of the bus. Let's assume c/20 signal (clock and data) propagation speeds; given a chip size of 25-35mm in 2010 technology [SIA94], typical signals will require 2.5-3.5 nSec to cross the chip end-to-end. If the chip is clocked at 2GHz, about 5-7 clock cycles may be required for signal propagation alone. As a result, it will no longer be feasible to separate the logical and physical design of the pipelines, as is done today; rather, today's wire buses will be transformed into explicit pipeline stages, whose only task is to move data around, and the number of stages per bus will depend strongly on where the various modules are placed on the VLSI chip. To make the situation even worse, the signal may arrive at the various receivers on multi-drop buses at different cycles.

Other effects of technological progress on processor speed relate to clock distribution. Several cycles may be required for the propagation of a single clock transition over the entire chip, compared to less than a cycle today. A worse aspect of this is that many transitions will be present simultaneously on the clock distribution wires. While this wavefront superpipelining is not impossible, it is highly undesirable. Optical clock distribution (e.g., a strobe light flashing at the chip from above, and multiple detectors and amplifiers spread over the chip) might provide a solution. Clock skew is also expected to be a very difficult issue. If the clock is not distributed optically, jitter of clock drivers and distribution lines will result in a skew much wider than the clock cycle. This skew can be balanced only at the very high area and power cost of phase lock circuits and powerful drivers. The power dissipated by complex VLSI chips increases as clock frequency rises [Hor93, Int94, Str94]. An increasing portion (currently over 40% [Bow95]) of

the power budget in a chip is required by the clock distribution network in order to contain clock skew problems. This ratio is expected to grow even higher, when processors are predicted to dissipate almost 200W [SIA94].

As technology progresses and operating frequency rises, only a small number (3-4) of logical gate delays fit into the shorter clock cycle time per pipeline stage. This implies deeper pipelines which are relatively difficult to design, and substantial effort must be devoted to balancing them. In addition, severe performance penalty is incurred for stalling such pipelines.

Even if clock distribution problems (skew and power dissipation) are solved, signal propagation delays will make it impossible to design a synchronous processor that operates with a common single clock. This thesis applies asynchronous microprocessor architecture to answer future technological and architectural constraints, forecast for the year 2010 and beyond, when feature size is less than 0.1μ, over one billion transistors are integrated on a single chip, and the clock (if used) operates at over 1GHz.

Asynchronous design has been studied for the past 40 years, and has attracted new interest in recent years [Async, Hau95]. Asynchronous systems eliminate or restrict the use of clocks, thus avoiding some of the problems of clock distribution and accommodating excessive data propagation delays. Instead of a global clock controlling when data can be safely moved from one unit to another, asynchronous units employ local handshake over asynchronous channels [Hau95, Sei80]. Asynchronous logic trades time for discrete events. Actual delays are hidden (abstracted), and only sequences of events (as depicted by transitions) matter. Thus, the correctness of computation is made independent of delays. Another advantage is that events can be treated hierarchically and local details can be abstracted, similar to hierarchical logic design, whereas the design of continuous timing is global and 'flat'.

Clocks typically switch over the entire chip, feeding into every flip-flop, and thus dissipate substantial power. Asynchronous handshake, on the other hand, is local, and happens only when needed, thus minimizing power dissipation and spreading it more evenly over time. The local handshake lines spread over shorter distances than the global clock distribution network. Thus, less area and less power are required. Some of the handshakes can be completed concurrently with computations, and they automatically accommodate for process variations and jitters.

Asynchronous circuits can be designed to operate according to the average case delay instead of the worst case, thus achieving typically a factor of two in performance [GM90]. For instance, while no carry is generated when adding 1+0, maximal carry propagation is needed for 1-1. Synchronous adders must always allocate ample time for the worst case. Asynchronous adders are *self-timed*, namely they detect and announce completion as soon as the computation is over.

In the case of 1+0, the asynchronous operation completes much earlier, saving time and power. Moreover, if there is no addition to perform (i.e. the adder receives no valid inputs), the asynchronous adder simply idles, whereas typical synchronous circuits compute every cycle, unless special enabling control logic is employed. As another example, assume the modules in a synchronous pipeline take *t* time units each to complete, but in 1% of the cases one module needs *2t* time units to finish its task. If the clock cycle is set to *t*, that rare case will cause the circuit to fail. If the clock cycle is set to *2t*, the performance is reduced by 50%. In an asynchronous implementation, the same scenario will only cause a 1% performance degradation. Indeed, some synchronous pipelines go into great complexity to achieve the same flexibility.

Asynchronous circuits are expected to achieve low power operation, resulting from eliminating the clock, and taking advantage of varying computational load of the application. Some 80% power saving has been reported by [vBB+94], by power supply voltage scaling of an asynchronous DSP chip designed for a battery-operated consumer product, and slowing down the operation of the circuit when appropriate. Other expected benefits from asynchronous designs [Hau95] include a modular (easily scalable) design, correct by design, and testability [DGY95].

Applying asynchronous methodology to a synchronous microarchitecture is not simply removing the clock. Rather, two issues should be addressed: The high level architecture, and the circuit implementation (inside modules and at their interfaces). In the former, any implicit timing assumptions made by system architects should be carefully reconsidered, to accommodate arbitrary inter-module delays. We claim that this process leads to distributed architectures, with data-flow flavor, and in the thesis we show how all relative timing assumptions are relieved. This is demonstrated on *Kin*[1], an asynchronous high performance processor. We also attend to the circuit level, where both locally-synchronous and fully asynchronous implementations of *Kin* are examined.

1.2 Timing Disciplines

Existing definitions of timing disciplines, as prevalent in the asynchronous literature, are insufficient for the purpose of this thesis. Some more precise definitions are needed:

Def. 1.1: A *self-timed* system is one which generates a completion signal.

[1] *Kin* was the God of Time of the Maya.

Def. 1.2: A *uni-synchronous (unisync)* system is one operated by a single global clock, where all modules receive the same frequency and phase of the clock.

Def. 1.3: An *asynchronous* system is one which is not uni-synchronous.

Def. 1.4: A *multi-synchronous (multisync)* system is an asynchronous system operated by a single global clock, where all modules receive the same frequency, but the relative phases are considered arbitrary and unknown.

Def. 1.5: A *multi-clocked* system is an asynchronous system in which each module is clocked independently of the others, and its local clock is not synchronized with the other local clocks.

Def. 1.6: A *mixed-timed* system combines various timing disciplines in the same system.

Kin architecture (Chapter 2) comprises multiple fast self-timed units, interconnected over asynchronous channels, using handshake communication protocols. The asynchronous microarchitecture is a high level description and it allows flexible and robust implementation. Although *Kin* is designed as asynchronous at its top level, its modules can be implemented according to the various timing disciplines defined above. In this thesis we investigate the various timing alternatives.

Multi-clocked and multisync systems both present two fundamental synchronization challenges: At the low level, data lines coming into any module must be synchronized with the local clock. At the higher level, inter-module data delays are long and in particular they are layout-dependent; unlike contemporary practices, the architect of processors for future technology will not be free to assume that data emanating during a particular cycle will arrive at their destination during the same cycle (or any other specific cycle). Thus, it is safer at the high level to make the data self-identifying and to introduce handshaking mechanisms. As a result, large integrated systems may evolve as Globally Asynchronous, Locally Synchronous systems (GALS) [Cha84]. Such systems consist of multiple synchronous modules (e.g., 100 modules, each having 10 million transistors), each driven by its own clock driver. The modules intercommunicate asynchronously, as they are ignorant of each other's clock. The multiple clocks may be completely independent, or they may be driven by a single clock source, where each module receives an arbitrary phase but all operate at the same frequency (*multi-sync* system). In a fully synchronous processor, substantial area and power are invested to keep the multiple clocks in full synchrony, with minimal relative skew, striving to ascertain the exact same phase in all parts of the chip. In GALS, this increasingly difficult goal is abandoned. Instead, each module is left to deal with asynchronous inputs on its

own. If the clocks of the various locally synchronous modules need to run at exactly the same frequency, a minimal area, minimal power network will be used to distribute a single source sine wave clock, and each module will derive its local clock from that sine wave. While such derived clocks are subject to arbitrary relative phase delays and jitter, the main advantage is that substantially less power and area are needed because sine waves incur less harmonics and reflections than square wave clocks.

1.3 Previous Asynchronous Processors

Most of the previously published asynchronous microprocessor designs have rather simple and straightforward architectures. Neither of those processors supports out-of-order execution nor considers performance enhancement by branch prediction. The architecture is either based on micropipelines or built from small building blocks (bottom-up design).

The first asynchronous microprocessor was built at Caltech [MBL+89]. It is based on four common busses connecting the various units in the data path. The architecture is a pipeline with only two stages (Fetch and Execute).

The AMULET [FDG+93, Pav94] is an asynchronous implementation of the commercial ARM processor. Its architecture is based on micropipeline [Sut89]. It does not support out-of-order execution, except for out of order completion of load instructions relative to normal ALU instructions. All instructions are conditional, but the AMULET does not contain a prediction mechanism. It suffers from a long penalty when a branch is taken.

The NSR RISC processor [Bru93] is based on a four stage pipeline (Fetch, Decode, Execute, and Write Back). The processor units are each implemented as a micropipeline, and they communicate through FIFO buffers (similar to what is designed in *Kin*). The processor supports a variable delayed branch, but has no branch prediction or out-of-order execution. Its successor Fred [RB96] has multiple functional units, but instructions are issued (in order) only one at a time, thus limiting the level of parallelism.

The ST-RISC [DGY93] processor is built of FSM and combinational logic basic blocks, and basically consists of Fetch-Execute stages. Modules communicate through FIFO buffers. A branch processor and an ALU operate concurrently and communicate only when a conditional branch is encountered. Neither out of order execution, nor parallel issue, nor branch prediction are supported.

TITAC [NUK+94] was developed as a processing element in a parallel computer system. It was

optimized for delay insensitivity, rather than performance. Its simple architecture is based on a single accumulator.

The Counterflow Pipeline Processor (CFPP) developed by Sun [SSM94], has an interesting architecture based on two pipelines, one for instructions (flowing in one direction) and the other for results (flowing in the opposite direction). Neither branch prediction nor out-of-order are supported, and performance may suffer from the many comparisons and arbitration required at each stage.

STRiP [Dea92] is a pipelined RISC, implementing a predicted prefetch for both instructions and data. Its implementation is based on a dynamic clock, where local clocks are generated in the circuit and their speed changes according to the instruction being executed.

The A3000 [Wol92] is an asynchronous version of the MIPS-R3000 processor, and has five pipeline stages. It is based on an extension of the micropipeline approach, by applying several processing elements between the FIFO units, and constructed of parallel micropipelines.

SCALP [End95] design goal was low power operation. As reported, this goal was not achieved, and the processor was 3-4 times slower than comparable processors. No branch prediction was employed (all branches are treated as not taken), which severely limited the performance, due to high latency when branch is taken (i.e., a high misprediction penalty).

All the previously designed asynchronous microprocessor architectures are targeted at current technology, and are not extendable to take advantage of growing amount of resources to become available by future technology. *Kin* architecture is aimed at exploiting future technology, and taking advantage of the asynchronous architecture, as discussed in the rest of this thesis.

1.4 Thesis Outline

The thesis starts at the high conceptual level, descends into the more particular issues, and ends by contributions in the area of Computer Aided Design (CAD).

Kin, a high performance processor architecture suitable for future technology (one billion-transistors per chip) is presented in Chapter 2. *Kin* takes advantage of asynchrony to allow aggressive *Avid* speculation, to overcome the limitation of branch predictions and dependencies, and to support multi-execution. The novel *Avid* execution concept is defined and analyzed in Chapter 3.

As an example of fully asynchronous design of a critical component of *Kin*, an asynchronous instruction length decoder is presented in Chapter 4.

Automatic conversion of synchronous pipelines into asynchronous ones is discussed in Chapter 5. A new pipeline scheme (a doubly-latched asynchronous pipeline) is introduced, which addresses some of the limitations of existing proposals.

Chapter 6 presents a novel methodology for multi-synchronous implementation of *Kin*, in which the individual modules are all synchronous. That methodology eliminates most of the drawbacks of existing clocking methods.

The design methodology of *Kin* is introduced in Chapter 7. Statecharts are employed for specification, and special design rules are defined to adapt them for asynchronous design. A complete model of *Kin* has been developed with the statechart tool, and has been used for simulation and performance evaluation. Validation statecharts are also introduced.

Chapter 2 : *Kin* Architecture

Kin is an asynchronous microprocessor that supports out-of-order and deep speculative (*Avid*) execution. *Kin* architecture comprises multiple fast self-timed units, interconnected over asynchronous channels, using handshake communication protocols. The architecture of *Kin* supports the *Avid execution* model (introduced in Ch. 3), where multiple alternative paths, including the targets of some taken and not-taken branches, are prefetched and speculatively executed in order to reduce misprediction stalls. A *dynamic instance tag* is associated with every instruction, to enable management of the multiple paths. Instructions of different paths are executed together in the out-of-order zone. Paths descending from the branch direction that was not chosen are *pruned* without preempting useful execution.

Kin has been designed at the conceptual level. A complete specification has been modeled, and full simulation of standard performance benchmark (SpecInt95) on the microarchitecture of *Kin* has been carried out. *Kin* is designed with future technologies in mind (such as 1B transistor chips); thus, we suspect that full physical implementation will not be feasible for some time.

The novel architecture of *Kin* is presented in this chapter. The microarchitecture details and instruction pruning are described in Sect. 2.1. Race and deadlock problems and possible solutions are discussed in Sect. 2.2. *Kin* support of multi-execution is presented in Sect. 2.3, and Sect. 2.4 describes the model constructed for simulating *Kin*. Section 2.5 discusses possible implementation methodologies for *Kin*, which are further described in other chapters in this thesis.

2.1 Microarchitecture

2.1.1 General Description

Kin is a general purpose microprocessor that supports out-of-order and deep speculative (*Avid*) execution. The instruction set supported is not restricted and can be that of either a CISC or a RISC processor [HP96a, Joh91]. Each machine language instruction fetched from the memory is decoded in the processor and translated into (one or several) internal micro-operations (μOps); i.e., the processor uses a 'RISC' instruction format internally. Note that the only module of the processor that needs to be changed when a different instruction set is required is the instruction decoder; the rest of the processor remains unchanged. As an example, we simulated the support

of three generic instruction types (ALU, Load-Store, and Branch).

Kin's architecture is based on a distributed network of asynchronously interconnected modules, with no central control. The architecture is designed as asynchronous at the top level, but modules can be implemented according to various timing disciplines, as described in Sect. 2.5. Each module operates at its own speed, and communicates with other modules only when needed. The communication is done through asynchronous channels (which may contain FIFO buffers), by using handshake protocols [Hau95, Sei80]. The data is encoded (e.g., as dual-rail, or bundled data [Hau95]) to signal its validity, and is acknowledged. The use of FIFO buffers decouples the processor. Ideally, all modules are balanced (i.e., on average they have similar delays). However, if one of the modules is temporarily slow, it will not affect the other modules. Only when a FIFO is full will the sending module be stalled.

In contemporary synchronous processors, the collective knowledge about an executed instruction is distributed among the pipeline stages, the controller, the channels into and out of the register file, etc. This location-dependent distribution of data and control information is unmanageable in large distributed systems like *Kin*. Rather, instructions flow through the system carrying their own identity tags. Each instruction is a self-sustained packet, containing all the information needed for its execution. It may leave some indications, e.g., an instruction entry in the reorder buffer, but these eventually reunite with the instruction, e.g., when it commits or is pruned. Each module receives instruction packets from its input queues, executes them at its own local rate, and sends them towards their next stop over one or more of its output channels. This model resembles the *data flow* architecture concept.

The *Kin* architecture is described in Fig. 2-1. It combines many known features, like multiple execution units, out-of-order execution, register renaming, etc. [HP96a, Joh91, Tom67], and some new ones (e.g., *Avid* Execution, Dynamic Instance Tagging, unified Multi-Execution, and Pruning). Multiple instructions are executed concurrently by employing multiple execution units, and instruction level parallelism is exploited by out-of-order execution, whereby independent instructions are executed before preceding ones which are waiting for data and/or resources. To preserve the serial nature of the code, instructions are committed (completed) in their original serial order, typically more than one at a time. Speculative execution is employed to avoid processor stalls; branches are predicted and code is prefetched from the more likely paths of the program.

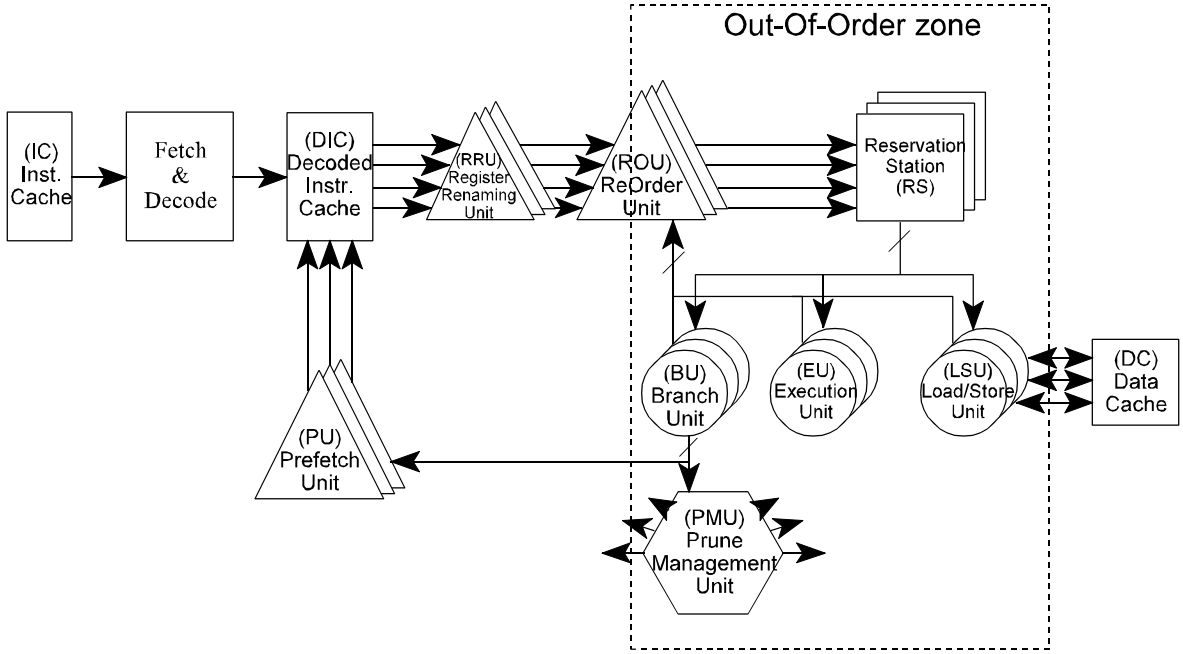


Figure 2-1: *Kin* asynchronous processor architecture.

2.1.2 Processor Modules and Instruction Pruning

The Prefetch Unit (Fig. 2-1) selects the address from where instructions should be brought and handled. The proper instructions are then either prefetched from the Decoded Instruction Cache, or (if not found there) are fetched from the Instruction Cache and are decoded before being further processed. After the decoded instruction registers are renamed (in a Register Renaming Unit), the instruction enters the ReOrder Unit (ROU), which manages the out-of-order execution. The instruction is sent to a Reservation Station to wait for its operands before it is executed and returns to the ROU to be committed (in-order). An instruction may be pruned before completing the whole path through the processor units. The operation of each unit is explained below.

Kin architecture supports the *Avid* execution described in Ch. 3. In *Avid* execution, the predicted path is prefetched and executed. In addition, a small portion of the non-predicted path is also prefetched and executed. Eventually, one of the two commits and the other is pruned. *Pathmarks* distinguish between the alternative paths. *Kin* generates dynamic instance tags and employs branch pruning (instruction purging) to discard unneeded instructions. The branching decision is made by the Branch Unit (BU, in Fig. 2-1). The BU notifies the Prune Management Unit (PMU), and the latter immediately issues a broadcast message *prune()* of the not-followed path. The message is sent over special channels spanning the entire processor. Each unit, as it receives the *prune()* message, scans its internal data structures and discards all redundant instructions identified by the *prune()* message, and also updates the pruning table it has at its input. Subsequently, any

instruction arriving at a unit is examined, and if it belongs to a pruned path, it is disposed of. This process needs not be exhaustive: An instruction which escapes pruning in one unit will be pruned eventually in another. At the latest, the instruction will be erased (without committing) when it comes back to the reorder unit. This process seems slow, but the cumulative rate of disposal meets the requirements. A beheading mechanism (explained in Ch. 3) is applied to control pathmark growth. The PMU generates and distributes (at proper times) a *behead()* message over the pruning channels. Every instruction arriving at a unit is checked to see if it should either be discarded or beheaded, before it is processed. Not all the modules in *Kin* receive *prune()* and *behead()* messages. Deciding which units should handle these messages and manage the proper prune and behead tables is a tradeoff between the module execution time, the overhead of handling prune/behead messages at that module, and the predicted performance gain by preventing redundant instructions from leaving the module. For example, computing an integer 'add' in an execution unit takes about the same time as doing the compare operation required to detect if the instruction should be discarded or executed. In *Kin*, we decided to handle prune/behead messages in the Prefetch Unit (PU), Decoded Instruction Cache (DIC), Register Renaming Unit (RRU), ReOrder Unit (ROU) and Reservation Station (RS). The PU must handle prune/behead messages since it generates the dynamic instance tags. The DIC should prevent unneeded instructions from entering the processor, to decrease the load. The RRU must keep track of the paths for maintaining coherent renaming tables. Every instruction handled by the processor passes through the ROU, and must be pruned there if necessary. Pruning is also important in the RS so that irrelevant instructions will not wait there forever for results of other pruned messages.

Prefetch Unit and Dynamic Instance Tagging

The pathmarks, which fully identify the path (see Ch. 3), are generated by the Prefetch Unit and are attached to each instruction at prefetch, as part of a unique *dynamic instance tag* (DIT), shown in Fig. 2-2. The Prefetch Unit determines which paths to follow according to the branch prediction [LS84, YP92] and *Avid* execution depth. It issues requests for fetching from the required instruction addresses, along with a proper DIT. The PU, having received a *behead()* message, henceforth modifies the generated DITs accordingly (as explained in Ch. 3).

Instruction		Dynamic Instance Tag (DIT)			
opcode	operands	root	path	context	pc

Figure 2-2: Dynamic Instance Tag structure.

The same basic block of code (or part thereof) may be fetched simultaneously multiple times. Consider a simple loop which ends with a conditional branch. Each time we reach that branch, we

should most likely prefetch the same loop again. Each time, the loop is prefetched (and tagged) as a new instance, and must be treated separately by the rest of the machine (e.g., proper register renaming), regardless of the fact that it is the same original code. Another example are instructions after an 'if' clause. They might have different dependencies and require different register renaming depending if the code in the 'if' clause was executed or not. This can only be decided at run time. Since we wish to prefetch multiple levels of the branching tree simultaneously, the instruction cache is multiported to provide all the required bandwidth, including multiple separate accesses from the same line. Of course, access optimization techniques are employed to replace brute force multiple reads of the same line by a single access and intelligent duplication, but this should be transparent to the PU. Since many instructions should be fetched and decoded more than once, and repeated decoding is inefficient, predecoded instructions are maintained in a Decoded Instructions Cache (DIC). The relevant μOps are fetched from the DIC, and are sent with their DIT to the register renaming unit. Requested instructions not found in the DIC are fetched from the Instruction Cache and are decoded by the decode unit.

Register Renaming Unit

A Register Renaming Unit (RRU) keeps and handles the renaming tables for the many possible execution paths avidly prefetched, to allow them to be speculatively executed out of order. The renaming process replaces architectural register names with virtual ones, thus filtering out false dependencies [HP96a]. The condition codes are treated as one of the registers and are being renamed accordingly. A new physical entry in the ReOrder Buffer (ROB) is allocated for each μOp destination (architectural) register. This entry number serves as the virtual name of the destination register. The μOp source registers are renamed according to the last name allocated to them on the same path or their ancestor path.

ReOrder Unit

The ReOrder Unit (ROU) manages the out-of-order execution in a processor, and enforces an in-order committing of instructions. A ReOrder Buffer (ROB) is used in the ReOrder Unit to keep track of the instructions from many possible execution paths, generated by avid execution. Instructions may be executed out of order, but they are committed (i.e., their results are written to the 'real' registers and to memory) in the same order they appear in the code. Since *Kin* supports *Avid* execution, the ROU needs to keep track of a binary tree of paths rather than just a linear sequence of instructions. The reorder unit also keeps a copy of what is usually referred to as the 'real' register file, containing the architectural registers. When a μOp arrives at the ROU, not all of its operands are necessarily valid, so the committed register values and speculative values from the ROB are searched in order to fill unresolved operands, before the μOp is forwarded to the reservation stations. After commit, ROB instruction entries and RRU allocations are released.

Reservation Station and Execution Unit

Several Reservation Stations (RS) are used by the instructions as waiting posts until their operands are available. These awaited operand values may arrive as a result of another instruction, or fetched from memory. Ready instructions (i.e., ones which have all their operands ready) are routed to one of several Execution Units (EU) in the processor. A scheduler (not shown explicitly in Fig. 2-1) is used to determine which instructions go to which execution unit. This scheduler can either be implemented as a simple router or as a sophisticated allocator. After execution, the result of the instruction is distributed to the ReOrder Buffer and to all the Reservation Stations, wherein other instructions might be waiting for it.

Load/Store Unit

The Load/Store Unit (LSU) handles memory access and bypass. It is used to take advantage of locality of references in data access. It is similar to another cache level, but is implemented as an independent smart associative table that tracks load and store operations. Ordering is enforced only when true dependencies are encountered, to guarantee correctness: For instance, Store(*X*) instructions can bypass Load instructions, but the LSU keeps a record of the previous value of *X* until Store(*X*) commits, in case it is needed by an earlier Load(*X*). Similarly, Load instructions can bypass Store instructions, except for Store to the same address, in which case the argument is forwarded from the Store instruction. Thus, the LSU can return values even before they are physically written to memory. Giving higher priority to Loads over Stores can increase the issue rate of instructions, because Loads results with values/operands for successive instructions, while Stores can wait without stalling any other instructions. Loads can be executed speculatively without affecting correct operation. However, Stores can only be done at commit, at which time it is known that the Store is on the actual true path of the program.

Branch Unit

The Branch Unit (BU) resolves a branch instruction and returns the result to the ReOrder Unit, the Prune Management Unit and the Prefetch Unit. Upon receiving branch results, the Prefetch Unit updates the prediction algorithm accordingly, and prefetches new instructions.

2.2 Race and Deadlock Problems

When designing a processor like *Kin*, one is likely to face the usual difficulties associated with distributed systems and algorithms, such as races and deadlocks. We have encountered several of them during the design and analysis of *Kin*. Knowledge and experience in the areas of distributed computing, operating systems, and communication networks already exist, and many solutions are applicable to distributed asynchronous processors.

Races

Races are a characteristic of an asynchronous microarchitecture. They might happen when two items (e.g., instruction and operand) must meet and merge, but one is in transition along a channel and the other chases the first. For instance, a μOp result might arrive at a unit before it is required by another μOp: Suppose that some number α, computed by an execution unit, is to be stored into architectural register *R*, and is also needed as operand for some instruction *I*. Assume that *I* has just passed through the ROU, and is travelling through some FIFO channel from ROU to one of the reservation stations. Just then, α is sent to all Reservation Stations and to ROU, where it is stored in *R*. The various reservation stations realize that none has any instruction in waiting for α, so they all discard it. When *I* finally enters one of the RSs, it is too late: α will not arrive again. This misfortune happens due to the fact that *I* could have met α in any one of multiple places (e.g., at the ROU before being sent to the reservation stations, or at the RS while waiting in a reservation table), an otherwise desirable feature. We resolve this non-determinism by maintaining multiple stored copies of α: each RS holds a *Most Recent Results (MRR)* table, wherein (even unclaimed) results are kept for a while. Thus, the RS can keep track of results arriving while the μOps needing them are on their way (e.g., waiting in the FIFO to be processed, or waiting to be written to the reservation table). If the architecture is to be delay insensitive, then the MRR might need to be as large as the ROB (because it might have to store the results of all executed μOps not yet updated in the ROB), and should be updated at commit and pruning time. However, it is enough if it is as big as the sum of the FIFO sizes along the path that might cause the race (e.g., the FIFOs on the channels from ROU to RS, from EU to ROU, etc.), with some safety margins depending on the delay of the modules in the path. The MRR can be cleared completely at certain safe points, e.g., when the affected FIFO channels are empty. Another solution to this kind of race could also be having the reorder unit send the results it receives to the reservation station. However, this will unjustifiably increase the communication bandwidth between the reorder unit and the reservation stations that already receive the same results through the bypass mechanism for performance reasons. Implementing the ROU and the RS as a unified module, results in a complex and large module, without eliminating the potential for similar race occurring when updating the shared table.

Mutual Exclusion

Mutual exclusion [Ray86] is required when executing some critical calculations such as updating FIFO pointers, or table entries. Arbiters are used to impose mutual exclusion. Although they may take a long time to resolve some conflicts, they never fail, and the asynchronous *Kin* is insensitive to arbitrary delays (they may affect performance, but not correctness).

A mutual exclusion mechanism for accessing shared tables is demonstrated by the passive FIFO shown in Fig. 2-3. While other types of FIFO channels are also implemented in *Kin* which do not require arbitration, the arbitrated FIFO is desirable when the FIFO is expected to contain a large

amount of data on average. Non arbitrated FIFOs consume substantial power when heavily loaded, due to the need to move all the data along shift registers. The arbitrated FIFO is implemented as a shared memory with pointers to its head and tail. The behavior of the FIFO control is presented by the statechart ([Har87] and Ch. 7) in Fig. 2-4. It defines three concurrently executing processes (separated by dotted lines): writing to and reading from the FIFO, and the arbiter that imposes mutual exclusion access. The operation of the fair arbiter is defined by the statechart in Fig. 2-5. The arbiter is implemented in hardware [Sei80, Mar85].

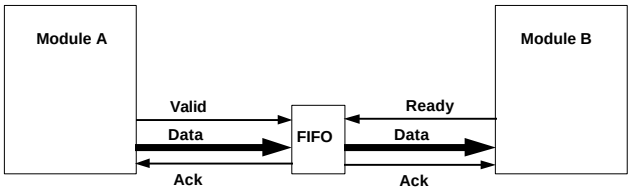


Figure 2-3: Inter-module communication.

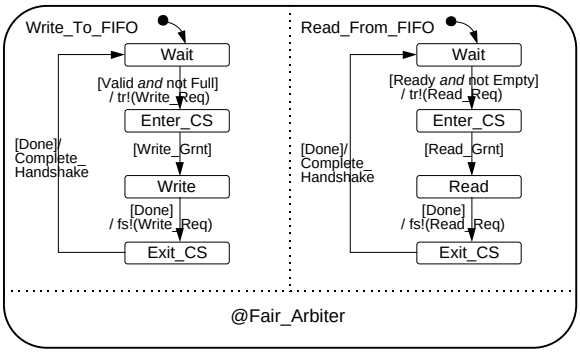


Figure 2-4: FIFO control statechart.

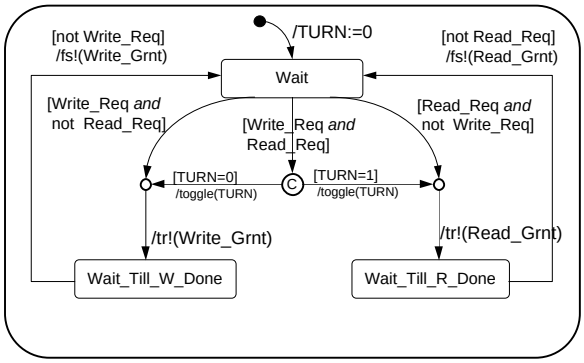


Figure 2-5: Fair arbiter statechart.

Deadlocks

Deadlocks in *Kin* might happen when cyclic buffers are clogged. This is the only kind of deadlock that is possible in the architecture of *Kin*, since the processor units are self contained, and do not depend on each other for resources or operation. The only correlation between the modules is by messages sent between them. A module receives a message, processes it and forwards it to its successor. The deadlock we have encountered in *Kin* was caused by a cycle of modules unable to complete sending a message because the FIFO at the input of their successor in the cycle was full. Each module is released only after is has successfully sent the message. When a module waits to complete communication, it cannot read a message from its input FIFO and free an entry in it. An abundance of distributed algorithms have been developed for prevention and avoidance of such deadlocks, as well as for detection thereof and recovery in case they do happen.

The simplest solution is to prevent, or avoid, deadlocks by making the FIFOs large enough. If unlimited FIFOs were available, there would be no chance for a deadlock to happen. The question is how to determine the practical size needed to prevent the deadlock. A heuristic algorithm can be applied to calculate the size of the FIFOs, as a function of the ROB size, RS size, modules execution time, instruction types and mixes, possible paths an instruction might take among the modules in the processor, etc. The ROB module serves as a 'sink' to the μOps in a sense that every μOp arriving to it will be consumed eventually, without a risk of deadlock (Since triggering the committing process is done by signaling, e.g., updating a mutually exclusive flag, rather than sending a message to it, there is no danger of having a closed cycle of FIFOs). The size of the ROB is an upper bound on the number of μOps being processed in the various modules. Thus, by making the FIFOs as large as that number, we can obviously prevent the communication deadlock from occurring. However, this bound can be tightened by noticing the way a μOp is being handled and processed by the modules in the processor: if a (μOp) message is waiting in one FIFO to be read and processed, it (or its result) cannot, at the same time, either be waiting in another FIFO, or be sent out. I.e., the sum of the sizes of the FIFOs along each possible cycle in the processor architecture should be equal to this bound. This method of choosing the size of the FIFOs is delay independent, and might be improved by considering the module delays (which change when using another implementation process), and statistical analysis of the instructions being executed by the asynchronous processor.

Another option to prevent a deadlock is to have a 'deadlock warning'. Since deadlocks always contain a cycle of full FIFOs, detecting a cycle which is close to this situation can be treated as a warning of a potential deadlock. This warning can trigger a mechanism to slow down (or even temporarily stop) new messages from entering the modules cycle (similar to a 'leaky-bucket' mechanism [BG92]). This can be done in the asynchronous architecture without affecting the correct operation of the system. The warning should be given early enough to enable some entries to free before letting new messages propagate into the suspected cycle, but not too early so that the processor will not have to slow down unnecessarily.

Algorithms to detect deadlocks and recover from them can either be centralized or distributively controlled. The messages needed to be transferred for detection and recovery, must use separate communication channels so they will not be stopped by the deadlock. An interesting idea might be adapted from the algorithm described in [JS89] for uni-cast communication networks. An 'emergency buffer' can be allocated in each module to be used only in case a deadlock is detected. Since all modules in the cycle are waiting for a free entry in their successor buffer, moving one message from the FIFO to the emergency buffer will temporarily release the deadlock, and might solve the problem if it enables some messages to be sent outside the cycle. However, for a reliable solution, this algorithm should be further developed and defined. Since a deadlock in the processor might be caused by a module outside the cycle (similar to multi-cast communication

networks), the suggested algorithm does not guarantee a recovery from the deadlock.

We should consider the effect on performance when selecting the proper solution for the deadlock problem. The processor modules are designed to be as balanced as possible to avoid creating bottlenecks which limit the performance. Variances of module delays and instructions flow rate can be used to simulate and analyze the architecture to select the proper FIFO sizes. Simulations of *Kin* showed that the FIFOs are most of the time either empty or have only a single message waiting in them. Thus, small FIFO sizes can be used and in some cases the FIFOs are redundant and a DLAP structure (introduced in Ch. 5) is quite adequate.

2.3 Multi-Execution on *Kin*

The high processing bandwidth of advanced processors is not fully exploited, because of true dependencies in the code and mispredicted branches. Three separate efforts, all aimed at increasing instruction processing rates by concurrent execution of multiple instructions are typically employed: Out-of-order superscalar execution (multiple instructions of the same serial code), multi-threading (multiple threads of the same program), and multi-processing (of different programs). All three (and more) can be unified under a single *multi-execution* model, and are handled almost identically by *Kin*. In addition, *Avid* execution (defined below in Ch. 3) is employed to execute speculatively multiple alternative paths of the same context.

In *Kin*'s unified multi-execution model, instructions from different contexts co-exist simultaneously within the processor. Each instruction is fully identified by its tag (Fig. 2-2): the specific path it belongs to and the program counter along that path, its specific thread, and its specific process. The prefetch, register renaming, and the reorder units take care of organizing all this. Others, such as reservation stations, schedulers, and the various functional units (all residing inside the Out-Of-Order zone), largely ignore the identification tag and simply process instructions as they come.

Instructions are fetched under control of the Prefetch Unit (PU). The PU maintains multiple contexts, one for each of the active processes and each of the active threads inside each process. To support *Avid Execution*, the PU initiates parallel prefetches of multiple basic blocks per thread and process, and also computes and assigns a unique *pathmark* per path. The PU prepares, for each instruction, a unique *dynamic instance tag* (DIT). The DIT identifies the context (process and thread), the pathmark (root and path), and the program counter (Fig. 2-2). The DIT and the instruction are henceforth packaged together. The non-preemptive pruning mechanism is managed by the PMU.

Like the PU, the in-order instruction management units RRU and ROU are multiplied in *Kin* to support multiple contexts. Each instruction is channeled, based on the context portion of its DIT, to the appropriate RRU and ROU. Furthermore, the RRU and ROU internal data structures are implemented as associative tables, and their algorithms are enhanced to support the multiple simultaneous paths of the same context. All three units (drawn as triangles in Fig. 2-1) maintain binary trees of paths, rather than linear instruction sequences.

As described, *Kin* realizes minimal data dependencies (both register and memory), and exploits Instruction Level Parallelism (ILP) of a program dynamically during its execution. Since *Kin* is asynchronous, and operates as a distributed system, it is not limited by a single clock cycle period, so its resource allocation can be dynamically adjusted according to the ILP characteristics of the multi-execution paths it follows.

2.4 *Kin* Model

A model of *Kin* is specified based on statecharts [Har87] and C code. The architecture of *Kin* (i.e., the modules, the channels between them (including fork and merge of channels), FIFOs on the channels, etc.) is defined by using the Statemate MAGNUM tool [iLo96], which is based on statecharts. Asynchronous communication handshake protocols, as well as mutual exclusion mechanisms, are also implemented as statecharts. Internal algorithms completely specifying module behavior (e.g., branch prediction algorithm, internal tables handling, etc.) are described as program fragments in C code. The statechart model controls the operation of the model, and activates the C code parts by interfacing and triggering the required computations. This formal and operational specification of *Kin* enables us to execute event driven simulations, required for asynchronous design. Modules react to messages arriving at their inputs, process the data and generate proper outputs. Handshake protocols and mutual exclusions are controlled and executed by statecharts. The interface between the statechart model and the external C programs uses handshaking protocols for communications and regards the programs as self-timed modules. Each program may be assigned a (variable) delay as its operation duration (Ch. 7).

Kin's model has a (partly) synthesizable specification: The parts defined by statecharts can automatically be synthesized into VHDL or Verilog description, and be converted to asynchronous implementation afterwards (as explained in Ch. 5). The parts written in C can also be defined by statecharts and treated in the same way.

Kin's model was used for debugging and performance evaluations of the architecture and specifically for simulating the avid execution concept with various depths. Animation of the model helped us identify deadlocks, races and bottlenecks in earlier versions of the architecture. As

explained in Ch. 3, we used SpecInt95 traces for the simulations, and gathered information on average and worst case FIFO and table sizes, committing and pruning rates, and program execution times. The author developed and constructed the model of *Kin*, and defined the algorithms of all its modules. Some of the C programs were written by [Sha97], and used for simulating *Avid* execution in *Kin*, as described in Ch. 3.

2.5 Implementation Methodologies

As explained above in Ch. 1, a uni-synchronous implementation is not feasible for *Kin*. Technological constraints and sheer complexity dictate a distributed system, as opposed to single-clocked fully synchronized contemporary processors. *Kin* is designed as an asynchronous system at the architecture level and it is not based on a synchronous pipeline, i.e., there are no synchronized stages operating at the same rate, where one can say exactly what happens at each stage at specific points in time. Increasing performance by applying and managing avid execution in such an asynchronous architecture is described in Ch. 3. Each of the units in *Kin* can be internally implemented as an asynchronous circuit. As an example of a complete asynchronous design, from architectural level to circuit implementation, Chapter 4 presents the architecture of an asynchronous instruction length decoder. The processor architecture consisting of the modules and channels (with or without FIFOs) can be considered as having a complex, non-linear, pipeline structure. To increase performance, each module can be internally implemented as a pipeline. An asynchronous pipeline implementation is presented in Ch. 5. If the processor is implemented as a multi-synchronous circuit, the adaptive synchronization method, presented in Ch. 6, should be applied.

Chapter 3 : Avid Execution and Instruction Pruning

Performance of present processors is limited by a number of factors, including true and false dependencies, limits to inherent instruction level parallelism in serial code, and pipeline stalls due to misprediction of branches. Processors must concurrently execute and complete many more instructions than provided by the average serial code. Although many execution units may be available, data and control dependencies limit the instruction level parallelism. To exploit instruction-level parallelism in full, it is not enough to look just at the instructions in a single basic block (i.e., the instructions between consecutive branches). A wider window is required. On average, every fifth instruction is a branch. To enable the prefetch of sufficiently many instructions, processors have to look beyond the next branch even before it has executed. This can be done by using various branch lookahead strategies.

As explained above (Ch. 1), technology advances will provide many more transistors to be available for use in the design of complex processors. It will also dictate an asynchronous architecture. We propose to take advantage of the available hardware resources, and the asynchronous architecture, to gain higher performance, by applying a dynamically adjustable smart speculative *Avid Execution*, defined in this chapter. The asynchronous architecture of *Kin* is most suitable for handling the resulting variations of computational load, because it is not limited to a worst case operation.

In the following sections, we examine the limitations to the execution rate of a program, and analyze the effects of branch mispredictions and misprediction penalties. Previous work on speculative execution methods is reviewed, and the novel *Avid Execution* concept (supported by the asynchronous processor *Kin*, Ch. 2) is presented. Instruction pruning and beheading mechanisms, used as part of the implementation of Avid execution in *Kin*, are defined. Performance analysis, as well as simulation results, are given.

3.1 Introduction and Previous Work

3.1.1 Execution Model

Statistical analysis of program traces shows [HP96a] that 20% of all instructions are branches, i.e., on average, every fifth instruction is a branch. Processor architecture is generally based on

pipelines to increase parallelism and performance [HP96a]. Conditional branch instructions found in programs cause the processor pipeline to stall, since the next instruction to be fetched after the branch instruction is unknown until the branch is executed and resolved. To prevent this kind of stall in the processor, various branching lookahead strategies are applied. The branch result is predicted and instructions are fetched and handled from the predicted address. If the prediction is correct, the processor operation continues without any stall, as the right instructions are already in the pipeline. On a misprediction (when the prediction is incorrect), the pipeline is flushed to clear it of all the wrongly fetched instructions. Instructions are then fetched from the proper address and handled by the processor pipeline. ***Misprediction Penalty*** is defined as the time required for the pipeline to fill up after a flush, until instructions start to commit again. This time depends linearly on the pipeline depth, and the number of stages between the fetch and branch resolution stages.

Out-of-order execution takes advantage of the Instruction Level Parallelism (ILP) found in programs and reduces program execution time by executing independent instructions concurrently. Value prediction methods [LS96, LWS96, GM97a, GM97b] are suggested for speculatively reducing data dependencies in order to increase the available ILP. The higher the parallelism utilized by hardware resources, the higher the instruction execution rate. It also means that more branch instructions are executed at higher rate, and misprediction rate increases as well. This adverse effect is compounded by another setback: The deeper the processor and the higher the parallelism, the higher the penalty paid for misprediction. Most of the processor units are stalled while flushing the wrongly fetched instructions and waiting for the correct ones to arrive.

Instruction execution rate (defined as the number of instructions executed per time unit) is thus limited by several factors (mispredictions, available ILP, hardware resources parallelism, and misprediction penalty). Figure 3-1 defines the following execution model. If there were no mispredictions, the execution rate (presented by the dashed line) would be limited by the available instruction level parallelism and the hardware resources available. However, since mispredictions do happen, the processor is able to execute instructions only during $\boldsymbol{n}$ time units before a misprediction is encountered. During that time, E instructions (the shaded area in the graph) are committed. Following a misprediction, no instructions are committed for the next $\boldsymbol{m}$ time units (the misprediction penalty time). After that, instructions start to commit again. As described in the graph, misprediction might happen even before the maximum execution rate is achieved. This phenomenon repeats itself and the average execution rate is thus as presented by the dotted line in Fig. 3-1, and is given by

(3-1)
$$R = \frac{E}{n + m}$$

The number of instructions that are executed before a misprediction occurs is a function of the prediction accuracy of the branch prediction algorithm used, as can be seen from Fig. 3-2. Given that about every fifth instruction is a branch, and a prediction accuracy of p, misprediction occurs once every $E_{miss}=5/(1-p)$ instructions. E.g., for a prediction accuracy of p=0.9, one out of ten predictions is probably wrong and a misprediction is thus expected about once every 50 instructions. For a prediction accuracy of p=0.95, misprediction is expected once every 100 instructions, etc.

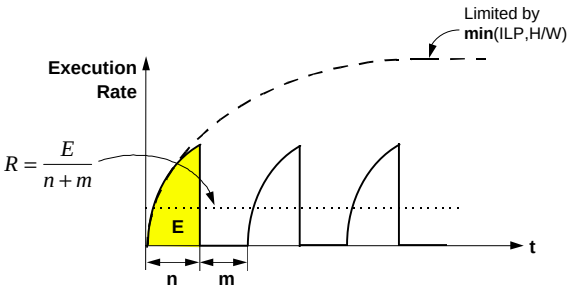


Figure 3-1: Instruction execution rate.

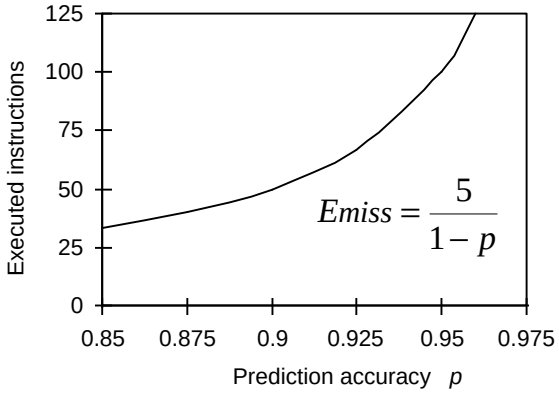


Figure 3-2: Number of executed instructions till misprediction.

From studies (cf. [HP96a]) of instruction level parallelism (ILP) found in programs (when there are no mispredictions), the ILP can be shown as a function of the instruction window size (Fig. 3-3). The '*window*' is the set of instructions examined as candidates for potential execution. The maximum window size is limited by the hardware resources available (e.g., the size of the reorder buffer). ILP is defined as the number of independent instructions in the window that can be executed concurrently. Figure 3-3 presents the ILP of four benchmark programs [HP96a], and their average ILP. When a program executes, the window size gradually increases, and the number of instructions candidates for execution depends on the number of new instructions entering the window every time unit. The rate of new instructions is a parameter of the processor width and parallelism (i.e., on the number of instructions that can be fetched, decoded, and renamed concurrently, in the processor pipeline stages at each time unit). The window size is also a function of the number of instructions leaving the window every time unit when issued to be executed. This number depends on the available ILP which, as explained above, depends on the instructions inside the window and the window size at that time.

In order to make the explanation and analysis clearer and easier to understand, we shall define the average time period required to complete handling instructions at a processor stage as a *'cycle'*.

Note that we use the term ‘cycle’ for convenience, and it does not necessarily imply that the processor design has either a synchronous or an asynchronous implementation. ‘Cycles’ should be thought of as time periods, which might be all equal, e.g., (one or several) clock cycles in case of a synchronous processor, or having an average length and variance in an asynchronous processor. The analyzed behavior, however, remains the same for synchronous and asynchronous cases, since both are designed to be as balanced as possible to prevent execution bottlenecks.

We define the possible parallelism available by the hardware as *w* (processor width), and say that at every cycle *w* more instructions enter the window. Also, every cycle some instructions (according to the ILP) leave the window and are executed. By analyzing the behavior of the average ILP as a function of the window size, we can conclude how many instructions can be issued for execution. Results of executed instructions enable other instructions from the window to be executed in the following cycles. The number of instructions executed every cycle is a function of the hardware parallelism and the ILP. As explained above, the ILP is in turn a function of the variable window size, which depends on the number of instructions entering and leaving the window every cycle. The result of this dynamic process is shown in Fig. 3-4, which plots the total executed instructions as a function of the number of cycles since the start of execution. The processor width (hardware parallelism) is a parameter.

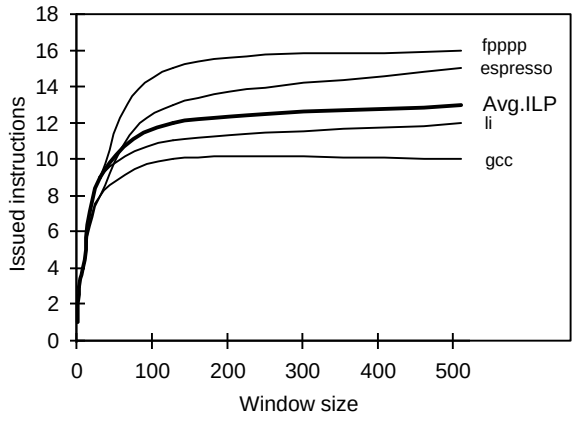


Figure 3-3: Studies of ILP as a function of window size.

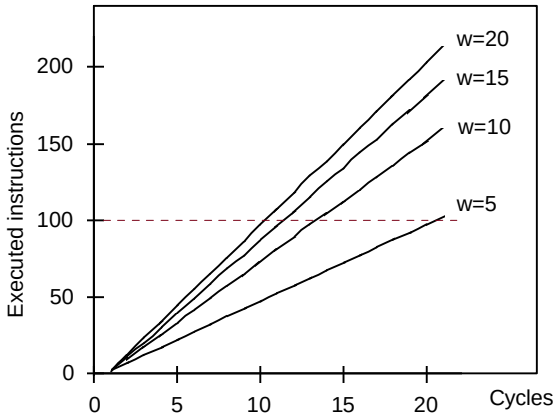


Figure 3-4: Cumulated executed instructions as a function of time cycles (assuming no misprediction), with hardware parallelism (*w*) as a parameter.

The dashed line in Fig. 3-4 marks E_{miss}=100 instructions executed (which, as explained above, is the number of instructions expected to be executed before a misprediction occurs at prediction accuracy of 95%). As can be observed from Fig. 3-4, the higher the hardware parallelism (the

more resources available), the less time (fewer cycles) required to complete E_{miss} instructions, but obviously there are diminishing returns. Further, it is clear that the higher the processor width (w), the more frequently a misprediction occurs. The lines in Fig. 3-4 are clearly linear, and regression analysis yields the following empirical trends for E (the number of executed instructions) as a function of the number of cycles n and the hardware parallelism w:

(3-2) $$E(n,w) = \alpha(w) \times n + \beta(w); \quad \alpha(w) = 4.1 \times \ln w - 1.5; \quad \beta(w) = -4.7 \times \ln w + 5$$

Assigning $E=E_{miss}$ and Eq. (3-2) into Eq. (3-1) yields:

(3-3) $$R = \frac{E_{miss}}{n+m} = \frac{E_{miss}}{\frac{E_{miss} - \beta(w)}{\alpha(w)} + m} = \frac{\alpha(w)}{1 + \frac{1-p}{5} \times [\alpha(w) \times m - \beta(w)]}$$

This expression is plotted in Fig. 3-5. Again, it can be seen in Fig. 3-5 that a high hardware parallelism w does not contribute to increase performance, since the limit is the ILP of the program. Increasing the maximum window size available in hardware does not help either, since the misprediction occurs long before the window fills up. As the misprediction penalty increases, the effect of a higher width w becomes negligible, since most of the time is spent while paying the misprediction penalty rather than doing computation. Thus, if the misprediction penalty is high, investing in higher parallelism of the processor does not improve the average execution rate and processor performance. The higher the hardware parallelism is, the higher the misprediction rate is, and the deeper the pipeline - the higher the penalty paid for misprediction.

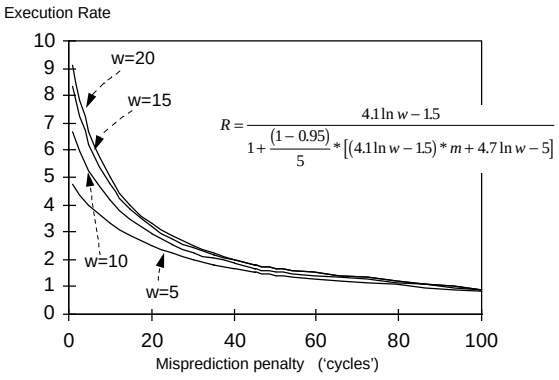


Figure 3-5: Hardware parallelism and misprediction penalty effect on execution rate.

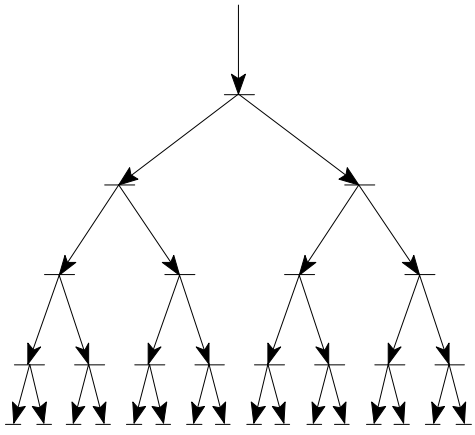


Figure 3-6: Tree of possible execution paths (vertices are branch instructions, edges are basic blocks).

The motivation for Avid execution is reducing the misprediction penalty, by utilizing 'spare' hardware resources not used due to limited ILP. Before defining and explaining Avid execution principles, we briefly survey and discuss previous works on applying speculative execution. These methods include *single path speculative execution*, *eager execution*, *multiple path exploration*, and *disjoint eager execution*.

3.1.2 Previous Work

The tree of possible execution paths is shown in Fig. 3-6. The vertices of the tree are the branch instructions. Each edge is a basic block, i.e., a sequence of nonbranch instructions terminated by a branch instruction.

Single Path Speculative Execution

Contemporary processors employ Single Path Speculative Execution, whereby each branch is predicted as either taken or not-taken, based on its past history [Cra92, HP96a, LS84, YP92]. The branch prediction can either be static (e.g., based on trace profiling, or always predicted as not taken) or dynamic (based on its behavior history at run time). For taken branches, the target address is also predicted. Only instructions from the predicted path are prefetched. On misprediction, the processor is flushed and the correct path is fetched. Current branch prediction algorithms are p=85%-95% accurate [HP96a]. For p=90%, E_{miss}=50 (Fig. 3-2). For example, if the misprediction penalty is 5 cycles, and on average 1.5 instructions are executed per cycle, then the slowdown due to misprediction is 15%:

(3-4) $$SlowDown_{misprediction} = \frac{Num.\ cycles\ with\ misprediction}{Num.\ cycles\ without\ misprediction} = \frac{50/1.5 + 5}{50/1.5} = 1.15$$

Figure 3-7 presents the misprediction effect on performance as a function of parallelism and misprediction penalty (with prediction accuracy as a parameter).

As explained above, single path speculative execution is highly sensitive to the quality of branch prediction and to pipeline depth. An execution tree of depth n contains n edges for single path speculative execution, so the cost is linear in the overall depth of prediction. However, the probability of correct prefetch over n levels falls off exponentially as p^n.

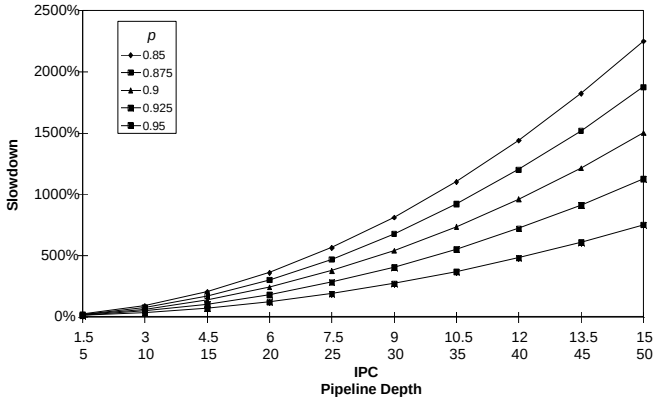


Figure 3-7: Misprediction effects on performance.

Eager Execution

In Eager Execution all paths are prefetched and (speculatively) executed. When a branch is encountered, execution proceeds down both paths of the branch. Multiple resources are required to support the parallel prefetch and execution of multiple paths. Once a branch is executed, its 'losing' sub-tree may be aborted and disposed of, and the corresponding resources can be released. As explained below, Eager execution is not practical and has not been implemented in any processor. Thus, no mechanism was developed for discarding irrelevant instructions. Since *Avid* execution (as explained below) requires efficient instruction discarding, a special non-preemptive *pruning* mechanism (Sect. 3.3) was developed for *Kin* (Ch. 2), and it is performed continuously and simultaneously with regular execution, without flushing the processor. The principal benefit of eager execution is that misprediction stalls are eliminated. However, eager execution is exponentially wasteful: Of the 2^n-1 edges of a *n*-level execution tree, only *n* edges are on the true path and eventually commit, while the remaining 2^n-1-n edges should be discarded. Since about 20% of all instructions are branches, the average basic block length is 5 instructions. If we consider an execution tree of depth n=3, then only 20 (out of 75) instructions are to be committed, while 55 should be purged. As the tree depth grows, the ratio between the required instructions and the irrelevant ones grows exponentially. Due to the enormous amount of resources required to implement eager execution, and the relative high accuracy of prediction algorithms available, eager execution is impractical.

Multiple Path Exploration

Multiple Path Exploration was suggested in [Mag80, MTM81]. The idea is to limit the eager execution to a tree depth of *m* levels. Thus, 2^m paths are explored simultaneously. After processing a set of 2^m paths, only one path remains valid, while all others are discarded. Another set of 2^m paths, generated from the valid path are explored next. A path code is used to identify each path, similar to the pathmarks (Sect. 3.3) used for Avid execution. However, these path codes have a constant length, and no pruning is applied. Only after all 2^m paths are completed,

the invalid ones are cleared. The system implementation presented in [Mag80, MTM81] contains 2^m processors (one per each possible path), and a central processor which generates the instructions of each branch path from the original program and issues them to the proper machines. While the instructions of *m* branch levels are being executed, the controller generates 2^{2m} paths of the next *m* branch levels. Only 2^m of these paths are used, and the rest are discarded, when the valid path from the previous *m* levels block is determined. The results from the data cache of the selected processor are copied into the shared memory, and the processors are then assigned to the next 2^m paths. Multiple Path Exploration requires approximately exponential hardware resources. The same instruction might be executed many times concurrently in different processors (since it belongs to many paths), even if eventually all of its copies are invalidated. This implementation requires a very high power dissipation.

Disjoint Eager Execution

Disjoint Eager Execution [US95] is based on calculating cumulative prediction probabilities, and assigning resources to the most likely code to be executed over all unexecuted code. The basic blocks which have the highest cumulative probabilities are fetched and executed. On a misprediction the processor is flushed. When the prediction accuracy $p \rightarrow 1$, disjoint eager execution practically converges into single path speculative execution, since the first alternative path will be selected only after taking *n* levels deep down in the execution tree, when $(1-p) > p^n$. For values of *p* close to 1, the number of levels in the tree that will be followed before considering any alternative path is very high. E.g., for p=0.9, the condition holds for n=22. But since p=0.9, misprediction is expected about once every 10 branch predictions. For p=0.95, the condition holds for n=59, while misprediction is most likely to happen about once every 20 branches. Thus, this model is inconsistent for typical levels of *p*.

Implementing disjoint eager execution requires a dynamic computation of cumulative prediction accuracies, every time the root of the execution tree changes (i.e., every time a branch instruction is committed). Because of difficulties with dynamic computation of those probabilities, static profile-based probabilities are proposed instead. The architecture presented in [US95] is based on a static instruction window, which is replicated along with bookkeeping hardware matrices per each execution unit. Unit latencies are assumed for the model, and low misprediction penalty is expected if the misprediction does not change the contents of the static instruction queue. It is stated that the effect on performance of using non-unit latencies in the model is not clear. Dependency lists are computed and maintained, so that if a branch mispredicts, its dependent instructions are squashed. A result is written to memory only when all of an instruction's depending branches have been resolved.

Avid Execution, defined in the following section, is designed to avoid the pitfalls of all these methods.

3.2 Avid Execution

3.2.1 Avid Execution Concept

Avid Execution results from combining the single path speculative and eager execution methods, such that the probability of misprediction is kept very low, while the exponential cost of eager execution is replaced by an approximately linear cost. The Avid execution method is basically an eager execution whose eagerness is limited, based on prediction. As in single path speculative execution, the predicted path is prefetched and executed. In addition, for each branch encountered and predicted, parts of some *k* levels subtree which is predicted as not-taken are also fetched into the processor, and are speculatively executed.

The number ***k*** of prefetched levels in the non-predicted subtree is adjustable. Figure 3-8 shows two examples of Avid execution depth, for k=2 and k=5. The main predicted path is marked by solid thick lines, and the extra (avidly handled) paths are drawn as dashed lines. Note that if k=0, Avid execution is reduced to single path speculative execution. For k=1, about 50% of all instructions fetched will be pruned, since for every predicted basic block another basic block from the not-predicted path is also fetched. The price of exponential demand for resources is avoided in case of Avid execution, and is replaced by an approximately linear one: For Avid execution with depth k, the number of edges in the execution tree is $(k+1)\times n-1$. Avid execution can produce instructions at a sufficient rate to reduce or even eliminate the stall on misprediction, as analyzed below in Sect. 3.2.2. The unneeded instructions are pruned asynchronously, without preempting continuous operation of the processor, as described in Sect. 3.3.

Selecting the *Avid* depth can be done either statically (e.g., all conditional branches have the same alternative path depth), or dynamically. Dynamic adjusting of Avid depth can be done per each branch instruction, and can be based on statistics collected at run time. If confidence is applied to prediction [JRS96, Smi81], the Avid depth can be adapted accordingly. For example, when a saturating up-down counter is used for making prediction [LS84, YP92], predictions made at the counter extremes are more accurate (about 85%-95% of them are correct) than predictions made away from the counter extremes (when only about 60%-70% are correct) [Smi81]. When the prediction confidence level is low, a deeper Avid depth should be used, and for high confidence prediction a small Avid depth (or non at all) might be better. Obviously, k=0 for unconditional branches.

Observe that the first edge of each alternative path described in Fig. 3-8, originating from each branch instruction (a tree vertex), is the branch direction predicted as not being followed. The following edges of the alternative paths are selected by branch prediction. Another option for selecting the alternative paths in Avid execution (instead of following ‘single path’ alternatives),

is to span a (limited depth) sub-tree from the path predicted as not taken. Avid execution can be recursively applied to the alternative paths as well. Obviously, if the alternative sub-tree is as deep as the predicted one, and follows all possible paths, then Avid execution becomes eager execution.

The more alternative paths followed by Avid execution, the more resources required. Our simulations verify that following single path alternatives is quite adequate when prediction accuracies are high. Spanning more alternative paths results in diminishing returns.

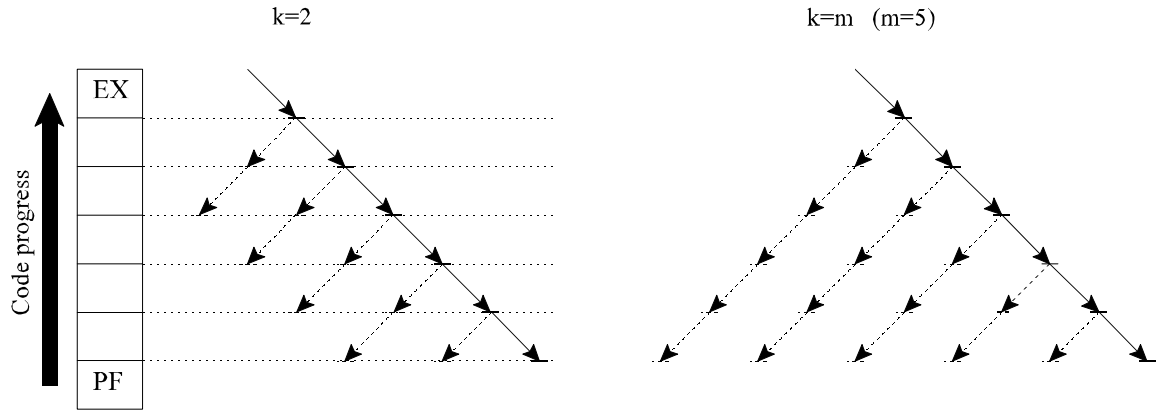


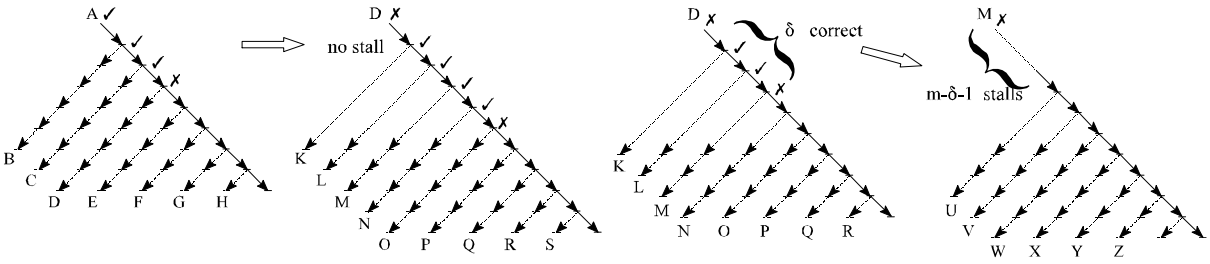
Figure 3-8: Examples of *Avid* Execution depth (*k*). *m* is the number of processor pipeline stages between prefetch (PF) and branch resolution (EX) stages.

Some related methods have been previously proposed. IBM RS/6000 [Cra92, HP96a] predicts every branch as not taken, but fetches instructions from the other path in case the prediction is wrong. Instructions from the alternative path are not issued for speculative execution. Similarly, IBM 360/91 [Cra92, HP96a] could prefetch instructions from both branch directions, but speculative execution of instructions from both paths was not possible. The IBM 370 [Flo74] had a small auxiliary instruction buffer for speculative prefetched instructions. Branches were predicted as taken or not taken based on their type (i.e., opcode). Once a branch was resolved as mispredicted, the content of the auxiliary instruction buffer was copied into the main instruction buffer.

3.2.2 Performance Analysis of Avid Execution

Assume there are *m* stages in the processor between the prefetch stage (marked as ‘PF’ in Fig. 3-8), in which instructions are fetched into the processor, and the execution stage (marked as ‘EX’), where the branches are resolved. These stages include operations such as decoding, renaming, operand fetching, scheduling, etc., of the instructions, as described in Ch. 2. If the Avid depth is *k*=2 (Fig. 3-8 on the left), then each processor stage must be able to handle instructions

from three basic blocks. For $k=m$, each processor stage handles instructions from $m+1$ basic blocks. After the instructions of a basic block commit in-order, the entire subtree is shifted and progresses in the processor stages, while new paths are predicted and prefetched. For every branch on the predicted path, alternative paths (emerging from it) of length k are also predicted. When a branch is resolved at the execution stage (possibly out-of-order), the redundant path is pruned. Execution continues along the resolved path uninterrupted, regardless of whether it was predicted or not. The misprediction penalty of stalling all processor stages while waiting for the correct instructions to be fetched and handled is avoided, or reduced, as follows. Two cases of misprediction penalty for Avid execution are presented in Fig. 3-9.



(a) Case I: No penalty. (b) Case II: Reduced penalty.

Figure 3-9: Misprediction penalties in Avid Execution.

In this example, the Avid depth $k=m$. In Case I (Fig. 3-9(a)), execution is following the path marked *A*. Assuming there is no misprediction of several previous branches, the Avid subtree (including the alternative paths marked *B* to *H*) has been fetched and is handled in the various stages of the processor. The first two predictions are proved correct (checkmarked '✓'), execution continues along path *A*, and paths *B* and *C* are pruned. The next prediction is incorrect (marked '✗'), and the rest of path *A* is pruned along with all the alternative paths emerging from it (*E* through *H*). After the misprediction, execution starts following path *D*. Since path *D* has already been predicted, fetched, and partially handled for depth *m*, none of the processor stages is stalled.

Note that none of the alternative paths of the new main speculative path *D* has been prefetched, so they must now be predicted and fetched. These 'holes' in the Avid tree are presented by dotted lines. If there is no subsequent misprediction for at least *m* branches, then the Avid tree is completed again. If a misprediction happens at a later time, e.g., at the '✗' mark leading to path *O*, then again no misprediction penalty is incurred, since path *O* already has *m* basic blocks in the processor.

Figure 3-9(b) shows Case II. After switching to path *D*, a second misprediction occurs as early as after only δ correctly predicted branches. Execution should now follow path *M*, which has not

yet reached the execution stage. Some of the processor stages are stalled while waiting for the *M* path to proceed through them. While path *M* moves forward towards the execution stage, its alternative Avid paths are predicted and prefetched to restore the proper Avid tree. Since part of the alternative path *M* does exist in the processor at the time of the misprediction, only a reduced penalty is incurred. This is in contrast to what happens after misprediction on a single path speculative execution processor, wherein the whole processor is flushed and stalled.

Average execution rate (as defined in Sect. 3.1.1) is used to measure the performance improvement that can be achieved by Avid execution (this measure is similar to the 'instructions per cycle' parameter used for synchronous processors performance). We identify the possible cases of mispredictions, and weigh the different misprediction penalties according to the probability of their occurrence. We concentrate on cases of having two mispredictions, separated by some correctly predicted branches. For prediction accuracies as high as can be achieved today (i.e., $p>0.9$), the probability of three successive mispredictions is $(1-p)^3$, which is negligible.

If i is the number of consecutive basic blocks executed without a misprediction between two successive mispredictions, then i-1 consecutive branches were correctly predicted. The misprediction penalty depends on the number of stages between the prefetch and execution stages (m), on the Avid depth (k), and on the number of basic blocks executed without a misprediction (i). The last two parameters affect the size of the 'bubble' in the processor. Thus, the misprediction penalty (in 'cycles') is given by

(3-5) $$M_i = m - \min(i, k)$$

and the average rate R_i is defined as

(3-6) $$R_i = \frac{E_i}{n + M_i}$$

where n is the time (number of 'cycles') required to execute E_i instructions between the two mispredictions.

Mispredictions which occur once every m branches or more (i.e., $i \geq m$) have the same reduced penalty, which depends on k (if $k=m$, then there is no penalty, as explained above). The total average execution rate is defined as the weighted sum of execution rates and probabilities of all possible cases, and is calculated as:

(3-7) $$R_{avg} = \sum_{i=1}^{m-1} (1-p)^2 p^{i-1} \times R_i + \left(1 - \sum_{i=1}^{m-1} (1-p)^2 p^{i-1}\right) \times R_m$$

where p is the prediction accuracy.

As an example of performance improvement achievable by Avid execution, consider a case of m=5, p=0.95, and enough hardware resources available (for the alternative paths as well) so execution is limited only by ILP and mispredictions. As can be seen from Tab. 3-1, up to 50% increase in performance can be achieved by Avid execution, depending on the avid depth applied.

k	Average Rate [#instr/time]	% Performance
0	6.62	100
1	7.09	107
2	7.64	115
3	8.28	125
4	9.03	136
5	9.93	150

Table 3-1: Average execution rates achieved by various Avid depths (k), for m=5, p=0.95, and high bandwidth (execution limited by ILP).

Obviously, Avid execution can contribute more to performance gain when misprediction penalty (m) is higher, since reducing that penalty has a larger effect on overall performance. The lower the prediction accuracy (p), the higher the increase in performance that is achievable by Avid execution, because mispredictions happen more often and Avid reduces the penalty paid.

On the other hand, even though Avid execution can optimally use any 'spare' processing bandwidth which is not utilized due to limited parallelism in code, it might also hamper performance if there are insufficient resources. The deeper the avid depth, the more resources are required. Most of the instructions are pruned at early stages of the processor, but if the bandwidth (w) is not wide enough the extra instructions might slow the execution down. Still, if the reduction of misprediction penalty increases performance more than the decrease in execution, the overall performance is increased. Some examples are presented and further explained by simulation results in Sect. 3.5.

3.3 Pathmarks, Pruning Management, and Beheading Mechanism

Avid execution prefetches and executes both directions of each branch. Eventually, one of the two commits and the other is pruned. Many of the instructions flowing through the processor should

be discarded. As explained in Ch. 2, *Kin* performs clean-up tasks (‘***pruning***’) on the fly, without preempting execution, and without stalling the processor for a centralized flush. Pruning removes the unneeded instructions, while the others (the relevant ones and the ones that are still speculative) remain untouched. Pruning does not stall the processor: Rather, it is executed concurrently while the processor continues to fetch, issue, execute, and commit instructions. Since *Kin* is an asynchronous processor, it cannot rely on a central clock or control to be used for simultaneously flushing unneeded instructions. Non-preemptive pruning employs ***pathmarks*** and a distributed algorithm to discard useless instructions.

Pathmarks distinguish alternative paths. Each edge of the execution tree is assigned a unique pathmark, based on prefix notation of binary trees. If an edge (a basic block, terminated by a branch instruction) is marked by *m*, then the sequentially following edge and the branch target edge are marked *m0* and *m1*, respectively (Fig. 3-10). The root is marked by the empty string. The pathmark is described by accumulating these bits as a road map to follow from the root until the edge the instruction is on. Note that the marks of all edges in the (dashed) subtree of node *n* are prefixed by *n*. The pathmarks, which fully identify the path, are generated dynamically during program execution and are affixed to each instruction at prefetch, as part of the *Dynamic Instance Tag* (DIT), as explained in Ch. 2.

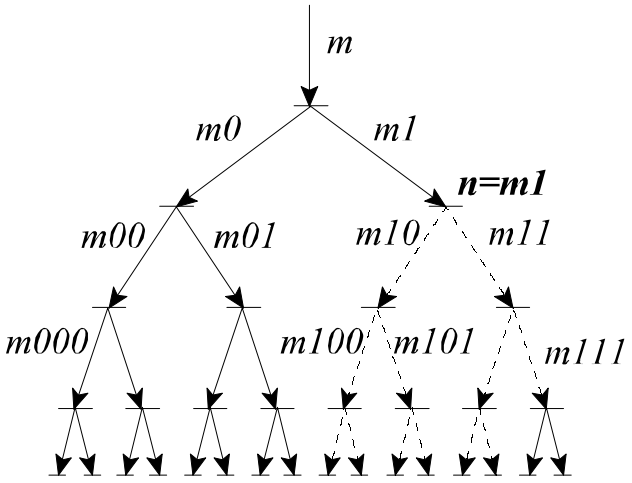


Figure 3-10: Pathmarks based on prefix notation.

When a branch is resolved (i.e., executed), one of the directions (according to the branch result) is made obsolete, and all the sub tree emerging from it should be pruned. All instructions from the obsolete subtree that have already been fetched and issued into the processor (none of them has been committed of course) can be ‘marked for deletion’ by broadcasting a single *prune()* message to all the units. This message contains the prefix that defines the sub tree to be expunged. Pruning is performed by comparing pathmark prefixes. All instructions having a matching pathmark prefix are discarded. Once the branch marked *m* is executed, the subtree that must be pruned is known:

m1 if the branch was not taken, *m0* otherwise. Respectively, a message *prune(m1)* or *prune(m0)* is distributed to the entire processor over the pruning channels, as described in Ch. 2. Out-of-order is supported: Assume that a branch *mk* (which is the *k*-descendant of *m*, where *k* is some binary word) is encountered before *m*. An appropriate *prune(mki)* message ($i \in \{0,1\}$) is sent. When, at a later time, the *prune(m)* message is sent, it overrides the former one, which is contained in the latter.

The pathmark length grows very fast, as one more bit is attached on every branch. On the other hand, much of the information of the pathmark becomes irrelevant when processing progresses down the tree. Consider a mark *m*, L-bits long. Once a branch instruction marked *m* commits, the pathmarks of all useful subsequent instructions in the processor will be prefixed by *m*. Since the *m* prefix is now redundant, it should be beheaded. A distributed ***beheading*** algorithm is used to contain pathmarks growth.

To behead prefixes, a *root mark R* is added to the DIT (Fig. 2-2). Occasionally (e.g., every *L* committed branches), a *behead(R,m)* message (*R*=path root, *m*=path prefix) is generated (by the PMU) and distributed over the pruning channels. Following the receipt of such a message, each unit modifies each instruction it handles as follows: If the instruction's DIT contains root mark *R* and pathmark prefix *m* (of *L* bits), then the root mark is updated to *R+1* and the pathmark is left-shifted by *L* bits. In effect, linear pathmark growth is thus replaced by logarithmic growth of the root mark.

A similar prefix notation of binary trees is used in the multiple path exploration scheme [Mag80, MTM81], to distinguish between the 2^m paths that are executed. However, in that application all paths have the same length, and there is no need to use beheading since the next new 2^m paths are introduced into the processor only after one of the previous paths has committed, and the entire processor is cleared of old instructions.

FIFOs can be used for the implementation of the pruning and beheading tables. Thus, a limited number of messages are kept at all times. New messages arriving will 'push' older ones out. This way the old and redundant messages are automatically discarded.

Since *Kin* processor is an asynchronous and distributed processor, races such as a new prune message being tested against an instruction which has not been beheaded yet should be resolved. This can be handled by either having the beheading process update the pruning tables, or rechecking for pruning after beheading an instruction. Cumulative beheading is also handled.

The number of the execution tree levels which exist simultaneously in the processor, although changing dynamically, is limited by the available hardware capacity. The number of bits in the

pathmarks reflects this limit. It need not necessarily accommodate the worst case; upon saturation, execution may be delayed until the beheading mechanism frees some bits in the pathmarks. The number of different root values that might exist in the processor is also limited, and the root mark may be allowed to 'wrap around'. A beheading message can be sent whenever a branch commits. But, to reduce the overhead of behead messages, the beheading information is accumulated by the PMU (see Ch. 2), and broadcasted only after L branches commit. The value of L is a function of the commit rate in the processor and the number of bits in a pathmark.

Kin's Prefetch Unit generated the DITs, but it has no knowledge that an instruction is a branch, until it is decoded and recognized as a branch in the decode unit. Since no prediction is available for branches when they are encountered for the first time, some extra bits are used (as part of the DIT) to indicate a basic block id. The basic block id is changed whenever the decoding unit sees a branch instruction for the first time (it means, of course, that this branch has been treated as predicted not-taken, but the other edge of the branch has not been prefetched). This way the pruning (if needed) can be done to a part of an edge, starting after the mispredicted branch.

3.4 Asynchronous Architecture for Avid Execution

Avid execution is more suitable to asynchronous architectures than to synchronous ones. Since a lot of hardware resources are utilized and the architecture is complex, a large chip is required (or even a multi chip) to implement the processor design. As explained in Ch. 1, due to signal propagation delays and clock distribution problems, the processor will have an architecture of a distributed system, most suitable for asynchronous design.

The workload that the processor needs to handle is largely varied. Since an adaptive Avid depth is applied, the amount of instructions handled varies a lot over time. Handling the register renaming for instructions from variable paths on the execution tree requires flexible comparisons and updating. Even the 'in-order' commit process is not the same as currently done in contemporary synchronous processors, since the instructions to be committed are not necessarily saved continuously in the ReOrder Buffer; rather, the ROB contains 'holes' and should be searched associatively. A variable number of instructions can be ready to be committed at different times. To increase performance, it is better to be able to operate at speeds close to the average case and slow down occasionally, than to always work as slow as the worst case of the variable load. This is easily achieved by a self timed asynchronous architecture.

Broadcasting *prune* and *behead* messages (required as part of Avid execution) to various processor units may slow down a synchronous processor because of long signal transmission time, affecting the clock cycle time. A dynamic and distributed algorithm is applied for instructions

pruning, and it is most suitable for an asynchronous architecture. There is no central control, and each unit handles the pruning at its own speed. There is no stall of the whole processor while pruning is done, since there is no need to wait until all the units acknowledge that they have completed the pruning. The rate in which *prune* and *behead* messages are generated and handled is not constant, and it can vary significantly as a function of some properties of the executed program (e.g., the length of the basic blocks), and even during the execution of a code (e.g., as prediction accuracies change). An asynchronous architecture is tolerant to such variances, while a synchronous design will either have to work much slower all the time, or will fail when occasionally a long computation occurs.

Avid Execution requires a wide memory bandwidth to prefetch and decode many instructions concurrently. Increasing code density, thus reducing the number of bytes fetched from memory, is achieved by using a variable length instruction set. Chapter 4 describes the design of an asynchronous instruction length decoder that speeds up the decoding of such instructions.

Avid execution was developed for asynchronous processors like *Kin*. Although it can be adapted for use in a synchronous processor (if such a complex and large processor is ever feasible as a single synchronous processor), its performance potential will not be fully exploited by a synchronous processor, for the reasons explained above.

3.5 Simulation Results

A model of *Kin* with Avid execution was developed by the author, as described in Ch. 2. The author defined the Avid execution and pruning handling algorithms that were implemented, and specified the tests to be made, as well as analyzed their results. Simulations of synthetic traces, and SpecInt95 benchmark programs, were carried out on the *Kin* model by [Sha97], with various parameters, as explained below. This section summarizes and analyzes the simulation results.

A branch prediction algorithm, based on [YP92] was implemented. Various prediction accuracies were obtained by changing the size of the branch target buffer (implemented as a 1-way set associative). The pathmarks were limited to 32 bits, and 16 bits (of which at most 5 were used during simulation run) were allocated for the root mark. Beheading was issued every two branch commits.

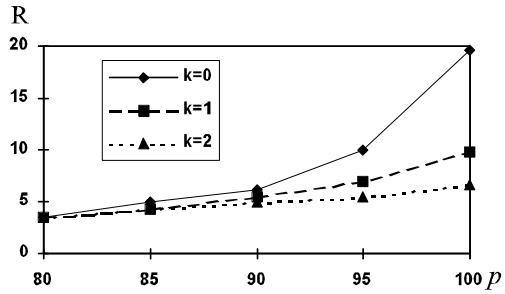
Avid execution was simulated for three possible (fixed) Avid depths: k=0, 1, or 2. The processor hardware width (i.e., the number of instructions that can be handled concurrently in each processor unit) was varied as either 20, 40 or 80 instructions. Avid execution spanning an ‘eager’ subtree for k=2 was also simulated, and demonstrated a diminishing return, as expected. The

results were at best the same as those obtained by Avid execution spanning a 'single path' for k=2, and at times worse, due to high prediction accuracies and lack of resources.

Two synthetic traces were generated and simulated on a processor model of width w=20, assuming several prediction accuracies (p=85%-100%). Each of the traces contained basic blocks of length 5. The first trace had no dependencies between instructions, thus the performance was limited only by branch mispredictions. The second trace was built with full dependency (i.e., every instruction depends on its predecessor), hence execution was limited by data dependencies and branch mispredictions.

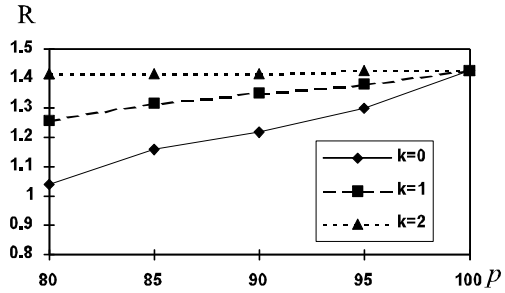
The simulation results are summarized in Fig. 3-11. The values in the tables are the average execution rate (i.e., the number of instructions committed per time period), and the performance percentage of Avid execution (for k=1,2) relative to the case of a single path speculative execution (k=0). When there are no dependencies (Fig. 3-11(a)) and no mispredictions (i.e., p=100%), Avid execution reduces the execution rate (from 19.69, which is almost equal to the processor width w=20), to 50% and 33%, for k=1 and 2, respectively, as expected, since it uses resources that can be used for the 'true' path. Since there are no dependencies, the execution along the 'true' path can proceed with no stall, except when a branch is mispredicted. When the prediction accuracy is reduced to p=80%, Avid execution does not decrease the performance any more. With even lower prediction accuracies, the overall performance can be increased by applying Avid execution
.

p	80%		85%		90%		95%		100%	
	R	%	R	%	R	%	R	%	R	%
k=0	3.56	100	4.97	100	6.22	100	9.95	100	19.69	100
k=1	3.56	100	4.35	88	5.52	89	6.99	70	9.90	50
k=2	3.56	100	4.35	88	4.97	80	5.54	56	6.61	33



(a) Trace with No Dependencies.

p	80%		85%		90%		95%		100%	
	R	%	R	%	R	%	R	%	R	%
k=0	1.04	100	1.16	100	1.22	100	1.30	100	1.43	100
k=1	1.26	121	1.32	114	1.35	111	1.38	106	1.43	100
k=2	1.42	137	1.42	122	1.42	116	1.43	110	1.43	100



(b) Trace with Full Dependencies.

Figure 3-11 : Synthetic traces simulation results (for w=20).

When there are many dependencies between the instructions (Fig. 3-11(b)), the available resources in the processor are not fully utilized, and execution is mainly stalled due to the data dependencies. When the branch prediction is not perfect (i.e., p<100%), Avid execution can use the 'spare' idle resources to work concurrently on some of the instructions from alternative paths. For p=95%, Avid of depth k=1 is 6% better, and Avid of depth k=2 is 10% better than the single path speculative case. The performance improvements grow to 21% and 37% (for k=1,2, respectively), when p=80%. Similar behavior was observed for other tested synthetic traces, which had smaller and larger basic block sizes.

Obviously, non-synthetic programs have a mixture of dependencies between the instructions they contain. All SpecInt95 programs were simulated with several Avid execution depths (k=0, 1, 2) and various processor widths (w=20, 40, 80) [Sha97]. The simulation results for w=40 are presented in Fig. 3-12, and analyzed below. Not all simulations yielded a prediction accuracies higher than 90%, thus they do not reflect the diminishing return point, expected at higher prediction accuracies, where Avid execution utilizes resources that might be better used in the 'main' predicted path. The highest prediction accuracy achieved was 97%, for the Ijpeg program.

The simulation of Compress95 (Fig. 3-12(a)) shows that for k=2, performance is best up to p=86% (except for the case of p=84%, where k=1 performs better). Above p=88%, k=0 is best. For Gcc (Fig. 3-12(b)), k=2 is better than k=1, up to p=83%, where they become equal. At that point, they both give 11% higher performance than k=0. For Go program (Fig. 3-12(c)), k=2 shows better performance than either k=1, or k=0, up to p=85%. Simulation of Ijpeg (Fig. 3-12(d)) resulted in highest performance for k=2 up to p=77%, then k=1 gives better performance up to p=97%. They are always better than k=0. A similar behavior was seen for the Li program (Fig. 3-12(e)), where k=2 performs better than k=1, up to p=92%. At p=93% they switch, but are still both better than k=0. M88ksim program (Fig. 3-12(f)) also behaves the same, but the switch in performance gain between k=2 and k=1 occurs at p=76%. For Perl (Fig. 3-12(g)), k=2 is best up to p=86%, and then k=1 is better but k=0 becomes the best. Vortex (Fig. 3-12(h)) always resulted in best performance for k=2 (up to p=95%), while even k=1 was better than k=0.

As explained in Sect. 3.1, the average execution rate is affected by several parameters, namely the instruction level parallelism, the prediction accuracy, the processor width and the misprediction penalty (pipeline depth). The effects of prediction accuracies and different instruction level parallelism are shown in Fig. 3-12. To demonstrate the effects of the other parameters on the execution rate and the performance gain by applying Avid execution, the same program (M88ksim) was simulated on the *Kin* model, with different parameter values. The results are summarized in Tab. 3-2.

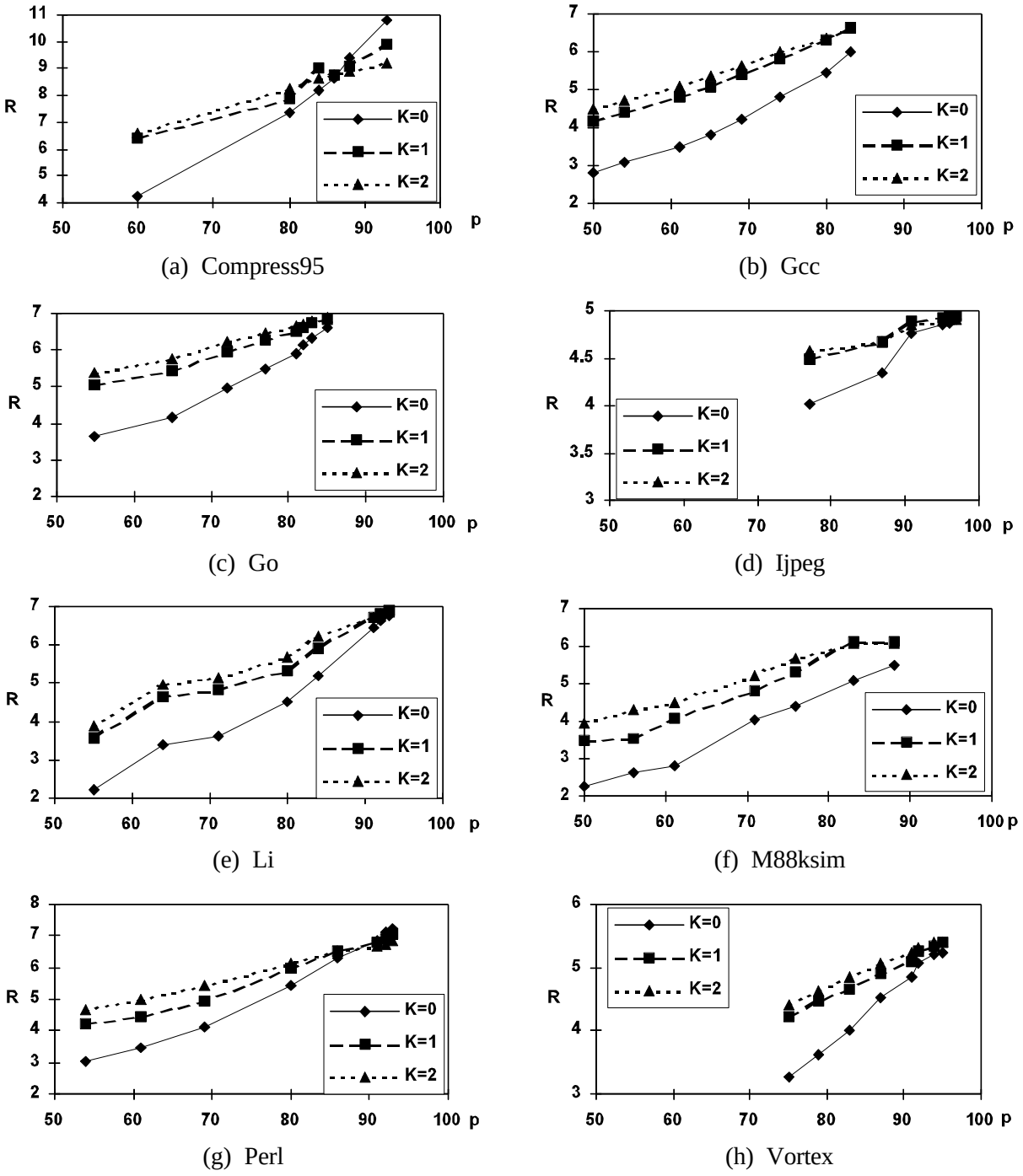


Figure 3-12: SpecInt95 simulation results (for *w*=40). The graphs describe the average execution rate (*R*) as a function of prediction accuracy (*p*), with Avid execution depth (*k*) as the parameter.

Table 3-2 contains both the absolute values of the execution rates measured for every case, as well as the performance percentage relative to what was achieved by *k*=0 in each comparable case. As can be seen, when the processor width is doubled from *w*=20 to *w*=40, the execution rate (for *k*=0) increases by less than 8% in the best case. This is due to execution rate limited by instruction

level parallelism (data dependencies) and misprediction penalty. However, Avid execution has more free resources to utilize and for *p*=88% increases the performance by 6% in case of *w*=20, and by 12% in case of *w*=40. For *p*=71%, performance is better by 20-30% when Avid execution is applied. Further increasing the processor width to *w*=80, results in a diminishing return, due to the limited parallelism found in the program. It is worth noting that the simulation model suffered from a limited number of pathmark bits (which causes the processing of *Kin* to stall when pathmark bits are exhausted, until some are beheaded). This can be avoided if more bits are allocated for the pathmarks, but it is not expected to change the results very much.

The wider the processor, the more instructions are handled concurrently. Thus, more branch instructions are executed, more mispredictions happen per time period, and more often the misprediction penalty has to be paid. As can be seen from Tab. 3-2, *k*=2 is better than *k*=1 only up to *p*=76% for *w*=20, and *w*=40, but it remains better for *p*=88% for *w*=80. Similar behavior was observed for the other traces simulated, where the 'switch' between the performance gains of *k*=0,1,2 occurs at different prediction accuracies, depending on the processor width.

Kin does not have a pure pipeline structure, but the units in it can be viewed abstractly as pipeline stages. By changing relative timing of the units we could change the pipeline effective 'depth', and affect the misprediction penalty. Setting the processor width to *w*=20, and changing the pipeline depth (the misprediction penalty) is also described in Tab. 3-2. Although the execution rates increase because of the lesser stall on each misprediction, the performance increase gained by Avid execution remains relatively the same (comparing by percentage of improvements).

Due to the limited number of pathmark bits implemented in the simulated model, we tested the effect of deeper Avid depth by using a very short pipeline (equal to 2-3 stages deep, compared to maximum Avid depth of *k*=2). As can be noticed from the results in Table 3-2, the execution rates increased due to the even smaller misprediction penalty, and Avid execution resulted in an average performance improvement of 25% for the case *p*=88%, and much more (up to 80% better) for lower prediction accuracies.

Another interesting effect of Avid execution found in the simulations was regarding the total number of instructions (and bytes) fetched from the memory. In several cases k=1 actually resulted in less instructions being fetched, since *k*=0 had to flush many of the instructions brought from memory. Hence, Avid execution not always resulted in an increased memory bandwidth.

We only implemented and simulated a fixed Avid depth scheme. It indicates however that better performance can be achieved when an adaptive Avid depth is used, based on prediction accuracy (or prediction confidence) of each branch, as defined in Sect. 3.2.

p	50%		56%		61%		71%		76%		83%		88%	
	R	**%**	**R**	**%**	**R**	**%**	**R**	**%**	**R**	**%**	**R**	**%**	**R**	**%**
***w*=20**														
k=0	2.29	100	2.65	100	2.79	100	3.87	100	4.10	100	4.78	100	5.32	100
k=1	3.35	146	3.43	129	3.88	139	4.54	117	4.91	120	5.55	116	5.65	106
k=2	3.73	163	3.92	148	4.24	152	5.00	129	5.14	125	5.51	115	5.62	106
***w*=40**														
k=0	2.27	100	2.63	100	2.83	100	4.03	100	4.42	100	5.07	100	5.48	100
k=1	3.51	155	3.55	135	4.08	144	4.84	120	5.34	121	6.14	121	6.14	112
k=2	3.94	174	4.31	164	4.50	159	5.22	130	5.69	129	6.09	120	6.09	111
***w*=80**														
k=0	2.39	100	2.70	100	2.96	100	4.01	100	4.39	100	4.70	100	5.48	100
k=1	3.63	152	3.66	136	4.23	143	4.96	124	5.22	119	5.68	121	6.00	109
k=2	4.13	173	4.47	166	4.64	157	5.27	131	5.47	125	5.94	126	6.05	110
***w*=20, Shorter pipeline**														
k=0	3.46	100	3.91	100	4.12	100	5.44	100	5.65	100	6.74	100	7.51	100
k=1	5.04	146	5.12	131	5.65	137	6.64	122	6.93	123	7.81	116	7.83	104
k=2	5.49	159	5.74	147	6.14	149	7.12	131	7.38	131	7.86	117	7.95	106
***w*=20, Very short pipeline**														
k=0	3.51	100	4.04	100	4.21	100	5.38	100	6.06	100	6.93	100	8.38	100
k=1	5.51	157	5.53	137	6.26	149	7.19	134	7.83	129	9.13	132	10.34	123
k=2	6.35	181	6.77	168	6.98	166	7.99	149	8.62	142	9.57	138	10.63	127

Table 3-2: M88ksim trace simulations performance results.

3.6 Concluding Remarks

Greater investments in hardware resources might result in diminishing returns, when no meaningful performance increase is achieved and performance might even suffer because of larger chips and deeper pipelines. Avid execution is aimed at better utilization of the available resources.

We have developed, analyzed, and simulated the Avid execution concept, along with proper pruning and behead mechanisms. The Avid parameters (e.g., depth) can be adjusted dynamically

according to prediction confidence. We introduced the Dynamic Instance Tag (DIT) to uniquely define a path, and defined a set of operations on the DIT to insure that useful computation is executed and useless computation is discarded. Avid execution applies pathmarks and pruning to execute instructions from many paths as soon as their operands are ready, but stop executing the remaining instructions on a path as soon as it is known that it will not be taken.

We have simulated a fixed Avid scheme, but other alternatives (e.g., dynamically adjusted, and various spanning trees) were defined, and should be further simulated and analyzed regarding their effect on performance. The structure of the spanned tree (the *Avid* depth and width) can be dynamically adjusted, depending, for example, on the prediction accuracy, the prediction confidence, and the distance (in tree levels) from the main predicted path (without computing cumulative probabilities). We expect higher performance improvements when a dynamic Avid is applied.

Asynchronous architectures (such as *Kin*) are best suitable for Avid execution, because of the design complexity (large chips) and great variance of computation load. As explained in Sect. 3.5, Avid execution does not necessarily result in excessive memory bandwidth requirements.

As any speculative algorithm, Avid execution is limited by branches whose target address varies and is not predictable because it is calculated at run-time (e.g., a jump through a register). These branches can either be handled by the compiler (e.g., by replacing a multiple target branch with a sequence of compare and jump instructions), or by better predictors implemented in hardware. When Avid execution detects a mispredicted branch target, all instructions in the processor are pruned, and execution continues from the correct address with a new root number, so no centralized flushing is required.

Avid execution can use ‘hints,’ which may be provided to it by the compiler, regarding what to do with the branch the first or each time it is encountered. The help provided by a compiler can include initial information about the target address, the branch prediction, the branching probability, and the recommended depth of Avid execution, per each branch instruction.

Chapter 4 : An Asynchronous Instruction Length Decoder

Kin architecture is asynchronous at its highest level, while the implementation of each module can follow almost any timing discipline. To complete this investigation at all levels of the design, a fully asynchronous implementation of a non-trivial module in *Kin* has also been attempted. This author has been fortunate to take part in such a study performed at Intel Corporation. While the Intel team focused on 'hacking' an asynchronous instruction length decoder (AILD) for best performance, the author investigated a strictly formal specification of the same module, employing the statechart tool (Ch. 7). Thus, the architecture of the AILD presented in this chapter is not the same as the circuit designed at Intel. A secondary purpose of this effort was to provide a verification frame for the Intel project. The third goal was to investigate the applicability of the statechart tool and of our methodology to the complete design cycle. At the same time, the author also contributed to the Intel design, as described below.

Avid Execution (Ch. 3) requires a wide memory bandwidth to prefetch and decode many instructions concurrently. While a variable length instruction set reduces the number of bytes that need to be fetched from memory, its decoding is difficult and may pose a bottleneck in a high performance microprocessor. The AILD architecture employs extensive parallelism to accelerate this operation as much as possible.

4.1 Introduction

The Intel x86 processor family implements a variable length instruction architecture wherein instructions can vary in length from one byte to eleven bytes or more [Int94]. However, memory systems, and in particular the cache memory used to store instructions prior to execution, typically store data in fixed size blocks, such as 16 bytes. Instructions are fetched in 16 byte lines aligned on 16 byte boundaries. Accordingly, in a variable length instruction architecture, each fixed sized line fetched from memory contains instructions of various lengths that may start anywhere within the line and may even cross line boundaries.

The decoding of a variable length instruction is, by nature, a serial operation, since the beginning of a particular instruction can be determined with certainty only after the beginning and length of a previous instruction have been determined. To decode several instructions concurrently, a fast length decoding mechanism is needed to mark the beginning of each instruction, before the

relevant bytes can be sent to be decoded. Decoding variable length instructions is rather simple if the length is explicitly stated at the beginning of each instruction, or a 'control field' is associated with each group of instructions, indicating the layout of the instructions within the group, as in [End95]. Neither of these cases applies to the x86 instruction set, and cannot be used for existing programs code. In the AMD-K5 [Chr96], which is an x86-compatible microprocessor, the decoder partially decodes instructions in a serial manner when they enter the instruction cache. Five bits of information are stored along with each byte to indicate instruction boundaries, distinguish between key bytes such as prefix versus opcode, etc. This adds a large amount of overhead in the cache size, and does not avoid the serial decoding of instruction length. In this chapter we describe the architecture of a highly parallel asynchronous instruction length decoder. Lengths are speculatively calculated in parallel, and a fast marking mechanism is used to detect the first byte of each instruction.

Current implementations of the variable length instruction decoding circuits are synchronous (clocked). Each unit in a synchronous system must complete its calculation within the given clock cycle, and the clock cycle is determined according to the worst case delay of all units. Hence, the design should be optimized for the worst case, even though only a small subset of all the instructions in the instruction set are encountered most of the time. Unlike synchronous circuits, the asynchronous instruction length decoder can be optimized for the most frequently used instructions, by handling the common cases very fast, while rare cases are handled more slowly.

The self-timed design of the instruction length decoder is optimized for the common (most frequent) instructions, and its speed is not limited by the slowest path. Thus, average case (instead of worst case) performance can be achieved. There is no need to optimize the rarely used circuits and computations, and data dependency is exploited for faster operation. The design takes advantage of the fact that a small subset of the instructions are executed most frequently, and the length decoding is optimized for this subset. Statistical analysis [BGK+97] of instruction traces of SpecInt92 benchmark programs shows that only 30% of all possible opcodes are used for 90% of the time. The statistical analysis also indicates [BGK+97, HP96a] that 99.8% of the instructions are seven bytes long or less, and 92% are no longer than 5 bytes. Only 1% of the instructions include a prefix.

The format of an instruction from the x86 instruction set [Int94] consists of an opcode, ModR/M and SIB bytes, and displacement and immediate fields. All the parts except the opcode are optional. The opcode is either one- or two- bytes long. The existence of a ModR/M byte depends on the opcode, with no easy-to-decode rule. When present, the ModR/M byte indicates the existence of the SIB byte and the existence and length of the displacement and immediate fields (up to four bytes each). Some instructions have a displacement field even though there is no ModR/M byte. An instruction may be preceded by prefix bytes which affect the instruction. Only

two prefixes affect the instruction length. The maximum valid instruction length is 11 bytes (excluding prefixes). The length of an instruction is defined by up to four bytes (1-2 bytes opcode, and more optional bytes), however, in many cases a single byte suffices.

The AILD design is optimized for the common cases (common instructions and lengths), while rare cases are handled more slowly. The goal was to decode 3-4 instructions in one nanosecond, which is five times faster than a 200MHz PentiumPro processor.

4.1.1 Author's Contribution

The design and implementation of an asynchronous x86 instruction length decoder were undertaken at Intel Corporation by a group of researchers [BGK+97]. This section seeks to clarify the contribution of the author to the work described in this chapter.

The author participated in the design of the AILD, and contributed to the architecture definition, design methodology, and logic design. Considering alternative designs, making design decisions, and choosing an implementation for the AILD were done by the whole group but with substantial input from the author.

The entire AILD architecture and interfaces between the modules were specified and codified by the author using statechart models (some examples are shown in Sects. 4.2.1 and 4.2.5). It contributed to the completeness and correctness of the architecture and better understanding of the control protocols needed to be implemented. It defined the module behaviors at various levels of details.

Section 4.2.1 contains the description of the AILD architecture. The author contributed to the design by defining a regular structure of the columns, including the routing of the many marking lines that flow between columns and rows. The author also defined the logic for decoding the marking and priority encoded switches for the steering circuit, based on the one-hot encoding of the lengths.

The author contributed to the definition and characterization of handling special cases, such as prefixes, long instructions, and branches, which are described in Sects. 4.2.2-4.2.4.

The author did not directly contribute to the implementation synthesis, described in Sect. 4.3; it is brought for completeness of the description. However, the statechart model built by the author was used to help understanding the architecture and changes made during the implementation phase.

4.2 AILD Architecture

4.2.1 General Description

A block diagram of the asynchronous instruction length decoder architecture is shown in Fig. 4-1. It consists of multiple identical units arranged in ‘columns’, corresponding to each byte in the input instruction line. Thus, for a 16-bytes instruction line, there are 16 columns. Each column includes a byte latch (which stores the byte from the instruction line), length decoding logic, marking units, and cross-bar switches.

Once a new instruction line is fetched from the fixed line-size memory, each byte of the instruction line is separately input to a corresponding byte latch, and processed in the length decoder of that column. Once all bytes are latched, a new instruction line can be made available. Each length decoder processes the respective byte, together with any additional bytes as may be required by the length decoding algorithm. The length decoders compute the length of the instruction, assuming that the byte being processed is the first byte of an instruction. Since all length decoders perform the calculations in parallel, all speculative lengths are available, and once a column is marked as being the first byte of an instruction, the proper bytes can be steered to the instruction decoder unit.

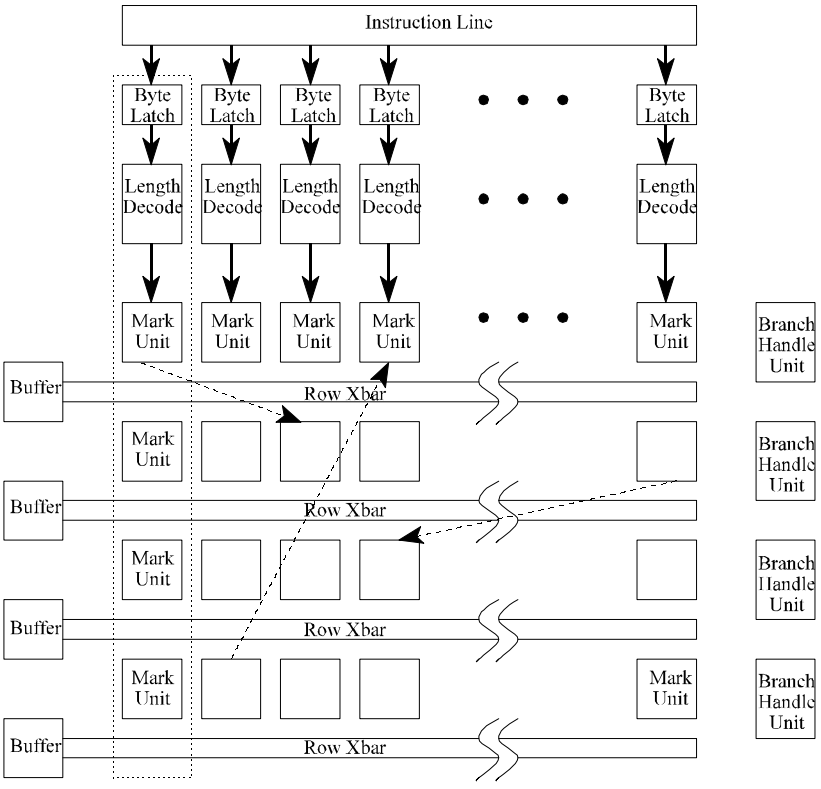


Figure 4-1: Block diagram of the Asynchronous Instruction Length Decoder architecture.

The AILD design is optimized for common cases, and the handling of rare cases involves communication between length decoders, as explained below. The handshake signals and interconnections between length decoders, and between length decoders and marking units are described in Fig. 4-2. Thin lines indicate single lines, and thick lines denote groups of several lines. The use of each signal is described in the following sections.

The length decoder outputs the speculative length to the marking units in its column. The length is used to mark the first byte of the next instruction, if the current column is found to be the first byte of an instruction. The marking lines are directly coupled to the proper marking units of subsequent instruction bytes. The marking outputs of the marking units in columns close to the end of the line, are wrapped around to the marking units at the beginning of the line, and therefore mark the first byte of the first instruction in the next fetched line. The generation and transmission of the marking information flow through the marking units in a self-timed manner. Since the marking scheme is very fast, a possible bottleneck might occur at the steering circuit which transfers the instruction bytes to a buffer at the instruction decoder unit. The marking propagates and wraps around before the bytes are steered forward and replaced by new bytes from next line. To avoid such a stall, several 'rows' of marking units and steering circuits are used. The marking signals from row *k* are hardwired to the marking unit of row *k+1*, modulo the number of rows (see Figs. 4-1, 4-3). The number of rows is defined by the timing of the marking process. Dashed lined in Fig. 4-1 demonstrate examples of marking between columns and rows.

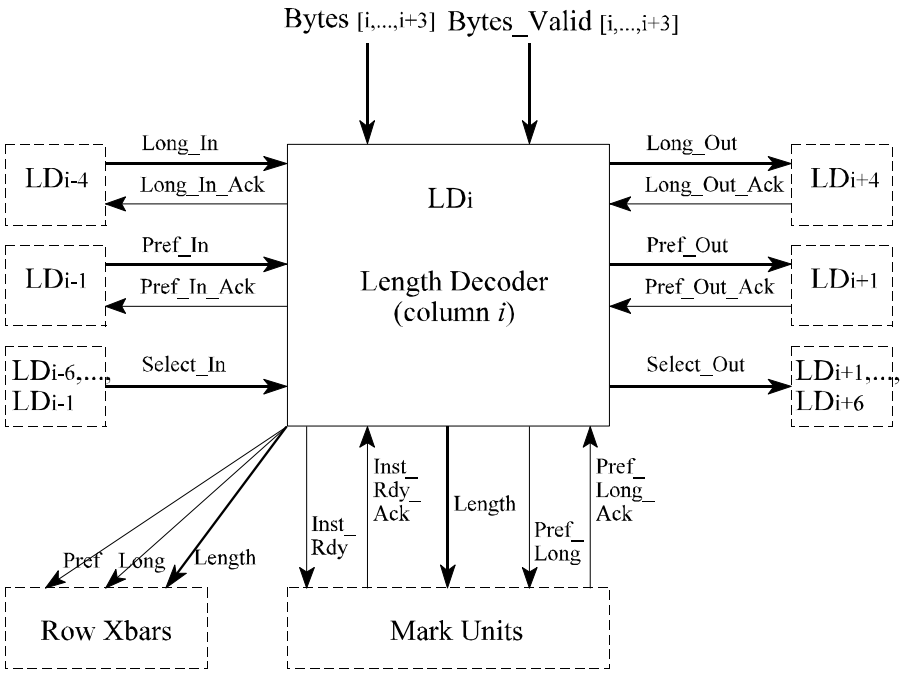


Figure 4-2: Length decoder interconnections and handshake signals.

A marking unit waits for an indication that its column contains the first byte of an instruction, as provided by the marking signal received over the marking lines from previous marking units (Fig. 4-3). A marking unit also waits for an indication that the bytes that comprise the instruction have been loaded into their respective byte latches and are ready for transmission. This indication is given by the *Inst_Rdy* signal, provided by the length decoder over the handshake lines (see Figs. 4-2, 4-3). The *Buf_Avail* signal, produced by the instruction steering circuit, indicates to a marking unit that the instruction steering circuit is available to receive an instruction for decoding and execution. These signals can arrive in any order. The instruction bytes are transferred to the output buffer of the same row as the marking unit that has processed those instruction bytes, and thus are incrementally spread across each output buffer, and can be fetched in order by the instruction decoder unit. The length of the instruction, as well as some other indications (such as *Long* and *Pref*, as explained below), are also transferred to the buffer along with the instruction bytes.

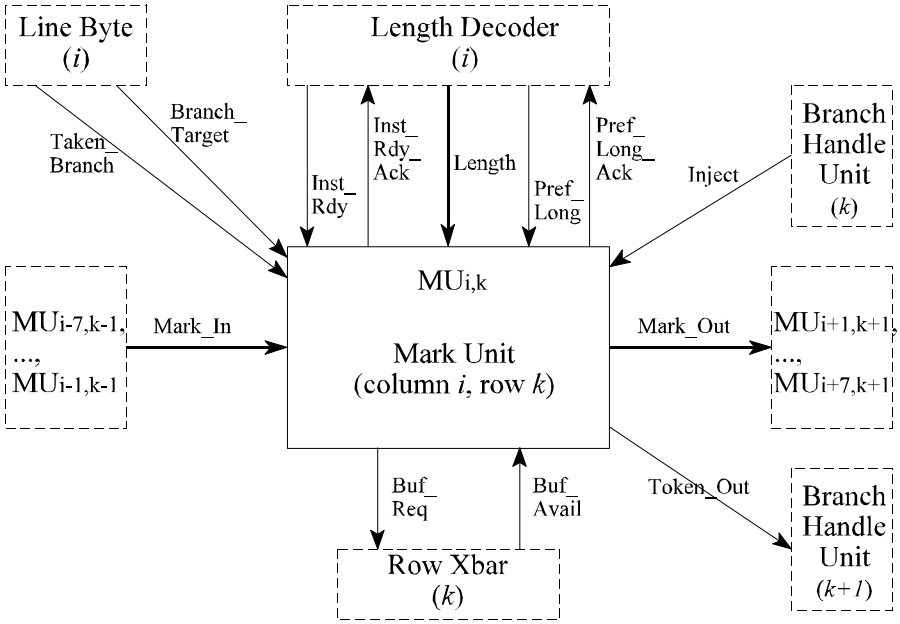


Figure 4-3: Marking unit interconnections and handshake signals.

4.2.2 Handling Branch Instructions

Branch instructions can change the control flow in a program. As explained in Ch. 3, branch prediction is applied to prevent stall in the processor operation, by predicting whether a branch instruction will be taken, and what its target address will be. The proper information is added to the bytes fetched from the cache line. A branch instruction can appear at any byte location in a

cache line, and the bytes following a taken branch are not used. Similarly, bytes preceding a branch target in a line are irrelevant. Each byte in the line can have at most one of three mutually exclusive indication bits set: *taken_branch* (the byte is the first byte of a branch instruction predicted as taken), *branch_target* (the byte is the first byte of an instruction which is a target of a branch), *unused* (the byte is not part of any instruction and should be ignored, e.g., bytes between the branch instruction and the target instruction).

When an instruction indicated as a taken branch is marked, the next marked instruction should be the instruction at the branch target, rather than the next instruction in sequential order. Branch handling units (Fig. 4-1) are used to control the marking of branch instructions in the rows of the instruction marking circuit. When a *taken_branch* indication is set, normal marking of the next instruction is aborted and the branch handling logic is activated. Instead of sending a marking signal to a marking unit on the next row, a *token_out* signal (Fig. 4-3) is sent to the branch handling unit of the next row, indicating that a target instruction should be marked as the next instruction. The branch handling circuit for the next row signals all marking units of that row (by the *inject* line, Fig. 4-3) that the first byte of the target instruction is identified by a set *branch_target* bit. The column containing the byte having its *branch_target* bit set will thus be marked by the branch marking process. The normal self-timed marking process then continues with subsequent columns, as explained in Sect. 4.2.1. A branch target FIFO buffer is used to avoid conflicts caused when multiple instruction bytes have their *branch_target* bits set in a single instruction line [BGK+97].

4.2.3 Handling Long Instructions

The marking and steering mechanisms described above can handle instructions of any pre-determined instruction length. However, the x86 instructions are typically short (i.e., length 1-7), while the maximum instruction length is 11. If the marking and steering are implemented for length $n+1$ instead of length n, then $n+1$ additional wires are needed and cross each column, each row. Having dedicated marking lines for lengths 8 to 11 will require about 40 more lines per marking unit row, which implies more area, wires, power, and latency, and will complicate the design of the marking circuit. Thus, the AILD is optimized for the most frequent instruction lengths, and long instructions are handled by a special mechanism (without using dedicated marking lines) as described in this section.

Since the most frequent instruction lengths are short, the AILD accordingly implements the self-timed marking and steering for up to length 7. Longer instructions are handled by being separated into two parts, head and tail, each at most 7 bytes long. There are two possible solutions to handle the two parts of a long instruction. The first one handles long instructions by

having an instruction head of a fixed length (equal to 4), with a variable length of the instruction tail (4-7 bytes long). The second solution handles long instructions by having an instruction tail of a fixed length (equal to 7), with a variable length of the instruction head (1-4). As described in Fig. 4-2, the first solution was preferred for the AILD architecture, since it requires the length decoder to communicate with only one other length decoder (rather than four), and thus less handshake lines are needed.

The algorithm of handling a long instruction according to the first solution (fixed head, variable tail) is as follows. The length decoder at column *i* (LD_i) decodes the actual length (say 10). When column *i* is marked as containing the first byte of an instruction, LD_i sends to the length decoder of column *i+4* (modulo the number of bytes in the memory line) a long instruction indication, with the tail length (say 10-4=6). This is done via the *Long_Out* lines, which are the *Long_In* lines for the receiving column (Fig. 4-2). The four possible tail lengths are encoded as one-hot lines, for the reasons explained below. LD_{i+4} acknowledges the message, via the *Long_Out_Ack* line (Fig. 4-2). LD_i sets its length output to 4, while LD_{i+4} sets its length output to the tail length, as indicated by its *Long_In* lines (say 6). The marking and steering of the head and tail operate as described above in Section 4.2.1. The head is placed in output buffer *k* (where *k* is the row number of the marking unit that was indicated as the first byte of the instruction). The tail is placed in output buffer *k+1* (modulo the number of rows). A *Long* indication bit is also sent to the output buffer *k*, so that the instruction decoder unit would know the bytes in that buffer are only the head of the instruction.

4.2.4 Handling Prefixes

The opcode of an instruction may be prefixed by (up to four) prefix bytes [Int94]. The prefixes are one byte long each, and are uniquely defined by that one byte. Prefixes have neither value nor operands, and typically the instruction decoder sets a single bit flag when a prefix is encountered. There are only two prefix values, operand size (66H) and address size (67H), which affect the length of an instruction. Each of these prefixes affects the instruction length in a different way.

Since each length decoder speculatively calculates the instruction length assuming that an instruction starts at the byte of its column, prefixes cause a problem with this paradigm. However, prefixes are hardly used at all (statistical analysis shows only 1% of the instructions to have a prefix). The solution (for the AILD) is to consider a prefix as an instruction of length one. A prefix is decoded as a separate instruction, and information about it is forwarded to the affected instruction.

The length decoder in column *i* detects whether the byte in its column is a prefix. Each prefix

(length affecting or not) is decoded as an instruction of length one. The information about length affecting prefixes is accumulated and passed on until the first byte of the instruction is reached, and these indications are then used to recalculate the instruction length. The indication passed on by length decoder *i* (LD_i) to the next length decoder (LD_{i+1}) is based on the indication received from the previous length decoder (LD_{i-1}) about prefixes detected, and the current byte (in column *i*). Handshake lines, as described below, are used for communication in case of a length affecting prefix.

The algorithm of prefix handling is as follows. The length decoder at column *i* (LD_i) detects byte *i* (B_i) as a prefix. LD_i uses any previous prefix indication and current byte information to determine which information (if any) to send to LD_{i+1}. When column *i* is marked, LD_i sends the prefix information to the length decoder in the next column (LD_{i+1}) on the prefix lines (encoded as one-hot lines for the three possible cases: 66, 67, or both). The prefix indication is done via the *Pref_Out* lines. *Pref_Out* lines of column *i* are the *Pref_In* lines of column *i+1* (Fig. 4-2). LD_{i+1} acknowledges the prefix lines (via the *Pref_Out_Ack* line, see Fig. 4-2) and latches that information. LD_i sets its length output to one. If the byte in column *i* is not a prefix, namely it is the first byte of an instruction, then LD_{i+1} redoes its length calculation according to the prefix indications it has received. The marking and steering in both columns *i* and *i+1* proceed as described in Sect. 4.2.1.

4.2.5 The Length Decoder Operation

The entire AILD architecture was specified by using statecharts, to guarantee completeness and correctness of the architecture and interconnections between the modules. The statechart of the length decoder (Fig. 4-4) is presented in this section as an (abstract) example. This is only a partially detailed statechart (as some details are eliminated to make it more clear), but it summarizes the various options the length decoder handles. For the syntax and use of statecharts, refer to Ch. 7. The following operational description depends on the detailed discussions of the previous sections.

When a new byte is latched in the byte latch, the length decoder starts to decode it. If it receives an indication over the *Select_In* lines (case marked by '①' in Fig. 4-4), it knows the byte is one of the bytes of an instruction which started at some previous column. When the current byte is steered to one of the output buffers (i.e., when *valid_ack* becomes false, marked by '①①' in Fig. 4-4), the length decoder operation is restarted to prepare to handle the next byte.

A *Long_In* indication ('②' in Fig. 4-4) means that the byte is the first byte of the tail of an instruction, and the length is forced to the value received. An input at the *Pref_In* lines ('③' in

Fig. 4-4) indicates that a length affecting prefix has been detected, and the length is recalculated accordingly.

If none of the above three cases occur, further handling depends on the column being marked. If the byte is recognized as a prefix (refer to '❶' in Fig. 4-4), the length is forced to 1 (as explained above), and for a length affecting prefix, the information is sent to the length decoder at the next column by a proper handshake protocol. Otherwise, the byte is handled as the first byte of an instruction (provided, of course, it is not an *unused* byte). If the length calculated is longer than 7 ('❷' in Fig. 4-4), the handshake over the *Long_Out* line is initiated, and the length is forced to 4 (the fixed head length), as explained above. When either the local length is short, or forced to a short value, ('❸' in Fig. 4-4) the marking process continues by the marking units, as detailed above.

In summary, the events detailed above (and described in Fig. 4-4) are mutually exclusive: Either a byte is selected as part of an instruction (if one of the *Select_In* lines is activated), or it is marked. If it is marked, it is either signaled to respond to a previous detected prefix (if one of the *Pref_In* lines is set), or signaled (by *Long_In* lines) to be the tail of a long instruction, or (when *Inst_Rdy_Ack* arrives) to use the result of its local computation regarding prefix, long and short instructions.

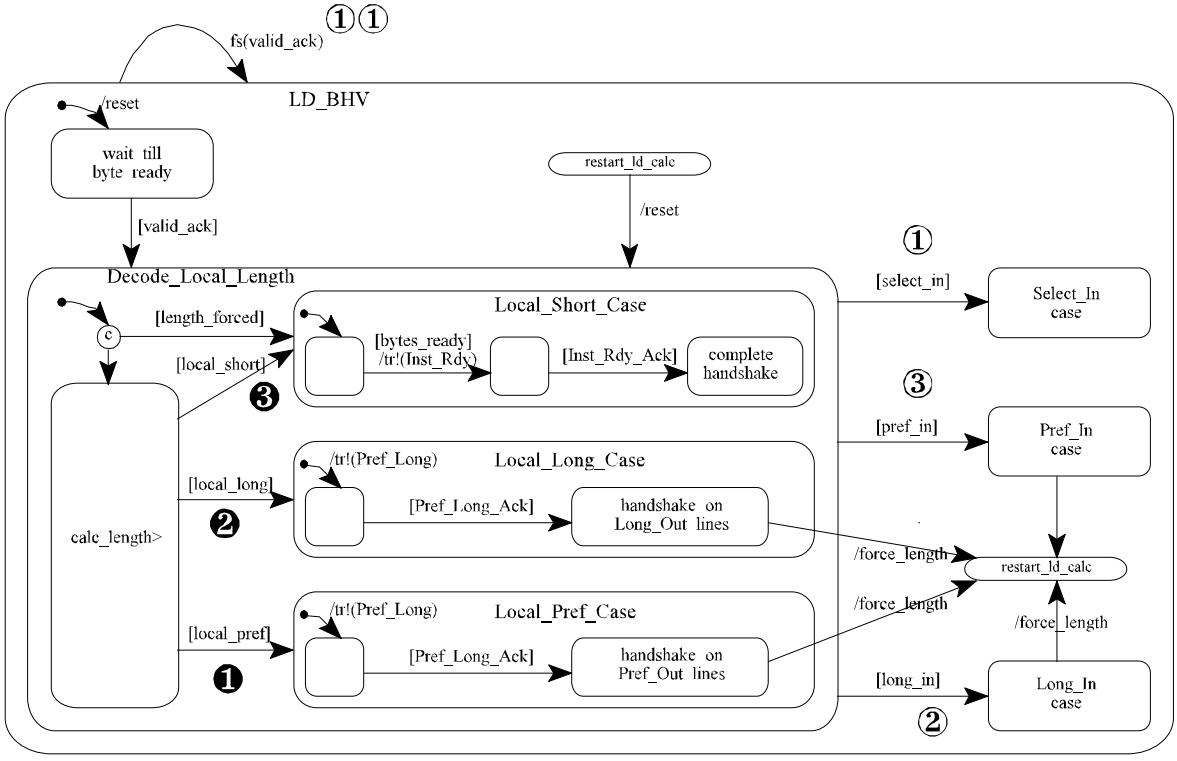


Figure 4-4: A simplified statechart of length decoder behavior.

4.3 AILD Implementation

Four phase handshake protocols are used for the communication between length decoders, and between length decoders and marking units, as can be observed from the different signals and acknowledge lines in Figs. 4-2, and 4-3. All units in the AILD are designed as self-timed circuits. Completion detection and data transferred could be implemented using the bundled-data approach [Hau95]. However, it would require design effort to guarantee fulfillment of the bundling condition between the units, and prevent taking advantage of variable data dependent delays. When the possible data values to be communicated are few, it is better to encode them using a delay insensitive code [Ver88]. We have chosen to use one-hot encoded lines for the lengths sent from the length decoders to the marking units, and for the long and prefix indications sent between length decoders. When one of the input lines is set, the receiving unit is signaled to handle the input, according to the value uniquely defined by the set line. Using the one-hot encoding for the instruction length, for example, makes the marking and selecting (for steering) logic implementation simpler [BGK+97].

For higher performance, the delay-insensitive operation of some of the signals was compromised, and they were implemented as self-resetting signals. These signals are pulsed, based on timing assumptions, to accelerate operation by eliminating the use of handshake protocols (e.g., there is no acknowledge of a mark being received) [BGK+97].

The length decoding logic was designed as a self-timed circuit, optimized for the common instructions, thus achieving better average case performance [BGK+97]. Statistical information was used to implement the length decoding logic as an unbalanced tree of gates (skewed monotonic logic), where frequently occurring instructions are decoded using shorter paths of logic and rare instructions are decoded using longer paths. Dual-rail and self-timing approach are used for a monotonic and hazard free domino logic implementation. Completion detection is done by having the output length encoded as one-hot lines. The contribution to the length calculation from following bytes (other than the byte from the length decoder column) is calculated separately, and combined only at the last stage, so that if they are not needed and not ready they do not stall the local calculation.

The controllers of the various AILD units were specified as either burst-mode or timed circuits, and were synthesized by the appropriate tools [Yun96, Mye95]. Several levels in the design hierarchy were formally verified [Ste94].

4.4 Concluding Remarks

Decoding variable length instruction sets is a major bottleneck in high performance processors. The design of an asynchronous instruction length decoder was described as an example of applying self-timed techniques and using asynchronous design for high performance circuits. The design is optimized for the common cases, thus achieving a better average case performance. The complex design of the AILD was achieved by integrating various concepts of asynchronous design methodologies. The architecture design of the AILD is independent of the implementation style of its modules.

The AILD architecture is based on speculative parallel length computation with a fast marking system. It is optimized to handle the common cases (instruction lengths and types) very fast, and provides proper mechanisms to handle special cases (i.e., rare instructions, prefixes, long instructions, and branches). Asynchronous control, based on one-hot encoding and handshake communication are applied, and the circuit operation is event driven, reacting to computation completion.

Currently, the AILD design is being implemented in Silicon and sent for fabrication. It is expected to be tested and analyzed within a few months.

Chapter 5 : A Doubly-Latched Asynchronous Pipeline

Synchronous and asynchronous systems use pipelines as their basic architecture. The faster the pipeline, the better the performance. The pipeline structure is widely used as the basis of computer architecture, and other processing modules, in order to increase performance [HP96a, WH90]. *Kin* architecture (described in Ch. 2) can be considered a complex, non-linear, pipeline. Each of the modules in the processor can be implemented internally as a pipeline, for higher parallelism and performance.

In this chapter we present the Doubly-Latched Asynchronous Pipeline (DLAP), developed as part of our research. The DLAP is an asynchronous pipeline with master-slave (dual) registers, which offers improved performance. DLAP is capable of truly decoupled operation: All pipeline stages can shift data simultaneously, and execution is faster than previous asynchronous pipeline designs when variable delays are encountered. Implementations based on both edge triggered registers and transparent latches are shown, fully analyzed, and compared to previous designs.

Converting synchronously designed circuits into asynchronous ones has the advantage of using existing synchronous synthesis tools, and achieving data dependent and low power operation, without redesigning the circuit. DLAP was found suitable for automatic synchronous-to-asynchronous conversion.

5.1 Introduction

Asynchronous micropipelines were first introduced in [Sut89]. They were based on a 2-phase communication protocol. Four-phase handshake protocol pipelines are presented in [MBM89, MBM91], where edge triggered registers are employed. Various similar control structures were proposed in order to enhance the performance of the asynchronous pipeline [DW95, FD96, FL96, YBA96], based on either edge triggered registers or transparent (level sensitive) latches. However, all those asynchronous pipeline designs suffer from one of the following drawbacks: They either achieve only 50% utilization of the pipeline stages (only every other stage is active at any one time, while idle stages contain bubbles), or (in some cases) incur a long backwards propagation of the acknowledge signals. The backwards latching scheme cannot be avoided when only a single register is used in each stage, since a storage element cannot release its value until the following stage has signaled that it is ready for another value. This might result in a major

performance problem for deeply pipelined circuits, e.g., rings or linear pipes with data dependent stage delays. In [SS93] it was shown that pipeline performance depends on the number of bubbles, namely registers ready to accept new values. When only a single bubble exists in the above mentioned designs, their performance is limited by the lack of bubbles. Synchronous pipelines, on the other hand, are not limited: If each register is master-slave, then a bubble is always available, and all values can propagate simultaneously.

Asynchronous circuits are expected to achieve lower power consumption and/or higher performance, by eliminating the driving clock. In addition, computational delays can be data dependent. While synchronous designs are timed according to the worst case delay over all stages, asynchronous circuits can be designed to determine and signal its own completion time, typically saving time and power. On average, self-timed operation with completion detection results in about 2× speedup of the individual units; power saving depends on the particular application, but can reach as high as 80% [vBB+94]. We have developed an algorithm for converting synchronous circuits into asynchronous ones, thus exploiting some advantages of asynchronous circuits while retaining investments in synchronous designs and tools.

A general description of the DLAP structure and operation is presented in Sect. 5.2. The design of edge-triggered registers and transparent latches based DLAP and implementation details are described in Sects. 5.3 and 5.4, respectively. Simulation results and comparative analysis are reported in Sect. 5.5. Section 5.6 extends the DLAP concept to non-linear pipelines, and Sect. 5.7 defines the synchronous-to-asynchronous conversion algorithm.

5.2 The Doubly-Latched Asynchronous Pipeline

DLAP (Doubly-Latched Asynchronous Pipeline) is shown in Fig. 5-1. It is designed for a single rail, 4-phase communication protocol between the stages. DLAP imitates the operation of a synchronous master-slave pipeline, by decoupling the pipeline stages. If the pipeline is balanced, DLAP operates the same as a synchronous pipeline: since all pipeline stages finish their computation at the same time, they can all latch the values concurrently into the master part of the registers, while the slave parts retain the previous values. Subsequently, all values latched into the masters are simultaneously transferred to the slaves. DLAP takes advantage of variable delays, as other asynchronous pipelines do. However, unlike other implementations, DLAP is truly decoupled: Thanks to double latching, a stage that has completed early can start processing the next data even if the following stage is still occupied. This is demonstrated in Sect. 5.5 below.

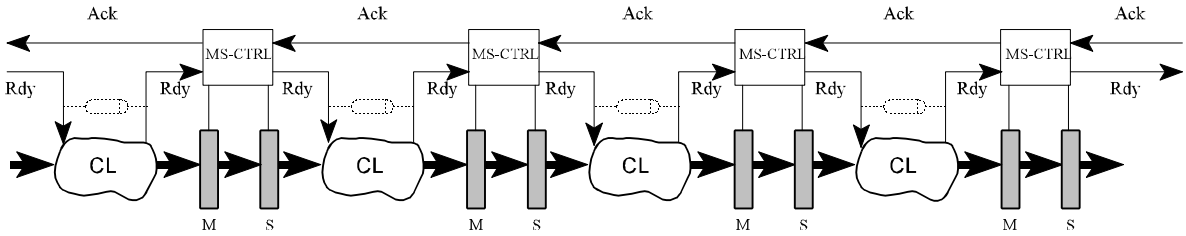


Figure 5-1: A Doubly-Latched Asynchronous Pipeline (DLAP).

A single stage controller for DLAP is shown in Fig. 5-2. The controller communicates with neighbor stages by *Ready* (Ri, Ro) and *Acknowledge* (Ai, Ao) lines. The latching of data into the master and slave registers is controlled by appropriate signals (Lm, Ls). The Done lines (Dm, Ds) signal when latching has occurred. If the Done signals cannot be generated by the registers, they can be created by routing back the Latch signals after passing through all the registers in the column, ensuring that all have been triggered [Pav94].

An example DLAP test circuit is shown in Fig. 5-3. The *ReadyOut* signal emerging from stage *i* is delayed before entering stage *i+1*. That delay matches the computation delay of the combinational logic between the two stages. We employ an asymmetrical delay [Sei80] in order to make the reset phase as short as possible. If the logic generates a completion signal (e.g., DCVSL [MBM89] or dynamic logic [FL96]), there is no need to add a special matched delay in the control circuit: *ReadyOut* feeds the combinational logic and the completion signal serves as *ReadyIn* for the following controller.

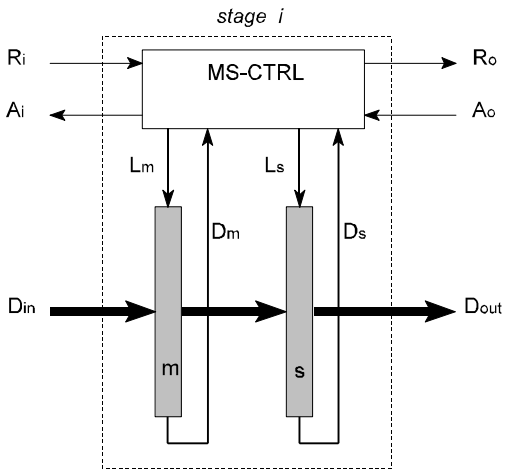


Figure 5-2: A DLAP stage structure.

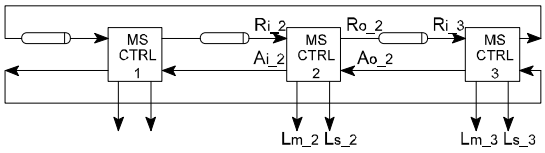


Figure 5-3: The DLAP test circuit.

DLAP can be implemented with either edge-triggered registers or transparent latches, as discussed in the following two sections. In the former case the control is simpler, while in the latter case simpler registers may be employed.

5.3 Edge-Triggered DLAP

The behavior of a controller for an edge triggered register based DLAP stage is defined by the Signal Transition Graph (STG) [Chu87] of Fig. 5-4. STG nodes represent signal transitions (underlined signals are inputs), directed edges are precedence relations, and the black dots are tokens, shown at the initial marking. The graph is 'executed' by moving tokens around. A transition is enabled by the presence of tokens on all edges leading to it. The transition removes those tokens and places new ones on all edges emanating from it (thus, the number of tokens may change). As described in Fig. 5-4, the master register is activated by the rising edge of *Lm* when a new value is ready (*Ri* is set), and the previous value has been moved to the slave (as marked by the internal signal *B*, for 'bubble'). Similarly, the slave register is activated by the rising edge of *Ls*, when a new value is ready at the master, and the previous value has been consumed by the following pipeline stage. *Petrify* [CKK+96] has been employed to ensure that the STG is safe, persistent, and has a complete state coding so it can be implemented as a speed independent circuit with no hazards. The control circuit implementation synthesized by *Petrify* is depicted in Fig. 5-5.

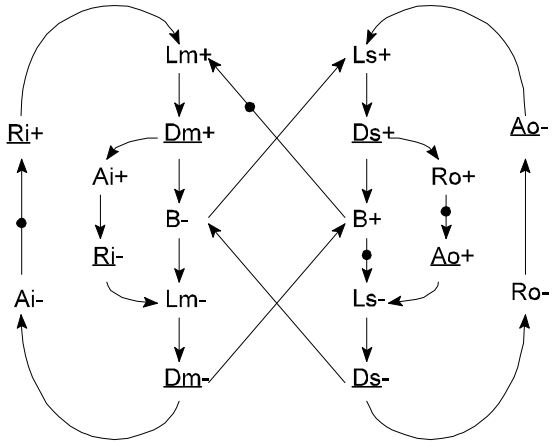


Figure 5-4: STG for a Master-Slave Edge Triggered stage controller.

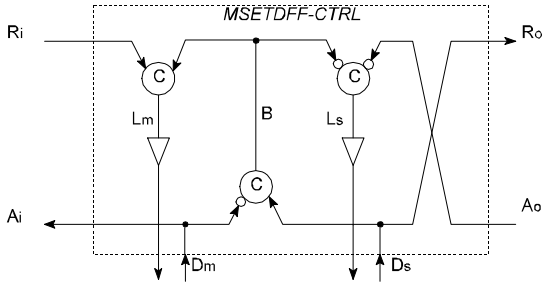


Figure 5-5: A Master-Slave Edge Triggered stage controller circuit implementation.

Some waveforms obtained from the simulation of a DLAP test circuit (Fig. 5-3) based on edge triggered registers are presented in Fig. 5-8. We have designed the two types of DLAP (edge

triggered and latched), and have simulated them with SPICE for a 0.8μ, 5V, typical CMOS process. Transistor sizing are optimized for speed and symmetric transitions. We have loaded the latch control signals (*Lm, Ls*) to simulate the drive of 32 bit registers. The simulated register driving delay time is 0.9nS. Note that this delay is included in the cycle time (to ensure the correct operation of the control circuit). The relative timings are summarized in Sect. 5.5. Observe that since the pipeline is balanced, all *Ri* lines are set simultaneously. Consequently, all masters are triggered simultaneously (*Lm* lines). After completing the handshake on the *Ai/Ao* lines, all the slaves are triggered (*Ls* lines), and *Ro* signals are set. Following the computational delays, the *Ri*s are set again. The cycle time (from Ri+ to the following Ri+) is 10.9nS, which includes a combinational logic delay of 5.03nS, a logic reset delay of 1nS, and four times 0.9nS for driving the two registers. In other words, the control overhead is only 1.27nS. The response time (passing the data through the double registers, i.e., Ri+ to Ro+) is 2.9nS (which also includes twice 0.9nS register driving delay).

5.4 Latched DLAP

Transparent latches are simpler than edge triggered registers. To save power, the latches are used according to the 'blocking latch' scheme [YBA96], i.e., they are kept closed at all times except when data must be latched. Power is saved because hazards are blocked. Note that since the latches are transparent, master and slave cannot be both open at the same time. Consequently, the controller is a bit more complex than for edge triggered DLAP. An extra internal signal (*G*, for 'gate') is needed to mark which of the two latches has been opened last, and to ensure that the STG has a complete state coding. The proper STG is presented in Fig. 5-6, and the implementation (synthesized by *Petrify* [CKK+96]) is presented in Fig. 5-7.

The waveforms obtained from the simulation of a DLAP test circuit (Fig. 5-3) based on transparent latches are presented in Fig. 5-9, and the relative timings are summarized in Sect. 5.5 below. Comparing this to the waveforms of the edge triggered DLAP, one can see that the pipeline is balanced and all the masters are enabled at the same time (*Lm* signals), followed by a simultaneous transfer of the data through the slaves (by activating the *Ls* signals). Note also that the *Lm* and *Ls* signals are mutually exclusive. The cycle time (from Ri+ to the following Ri+) is 12.06nS (including the combinational logic delay of 5.03nS, the logic reset delay of 1nS, and 3.6nS for driving the two registers; thus, the control overhead is only 2.43nS). The response time (Ri+ to Ro+) is 6.9nS, including four times the 0.9nS for driving the registers.

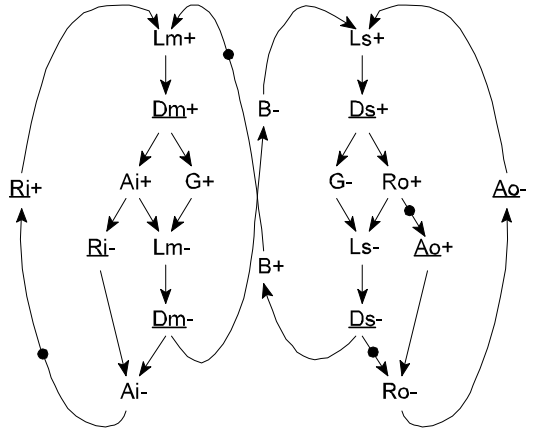


Figure 5-6: STG for a Master-Slave Latch stage controller.

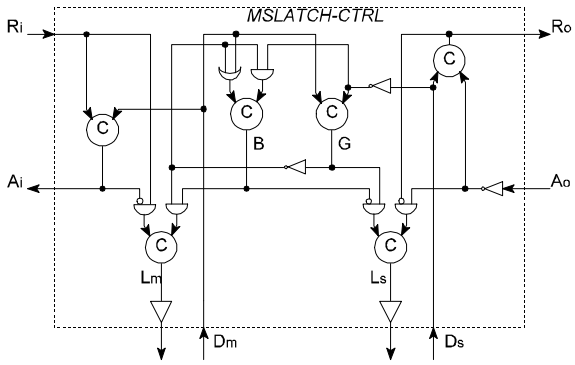


Figure 5-7: A Master-Slave Latch stage controller circuit implementation.

5.5 Comparative Analysis

As explained above, we have designed the two types of DLAP (edge triggered and latched), and have simulated them with SPICE. The basic test circuit with several stages and the resulting waveforms are presented in Figs. 5-3, 5-8, and 5-9, respectively. The measured times are summarized in Tab. 5-1. Note that the register driving delays are included in the cycle time.

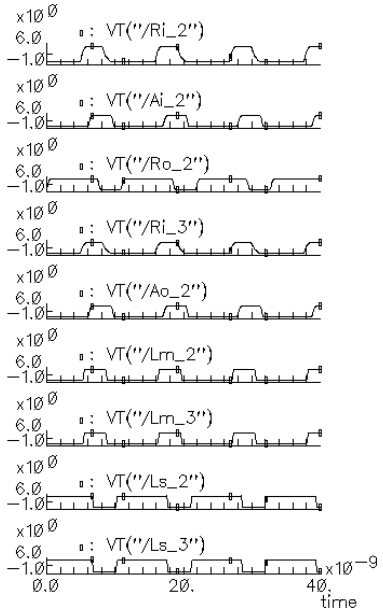


Figure 5-8: Waveforms of Master-Slave Edge Triggered DLAP test circuit.

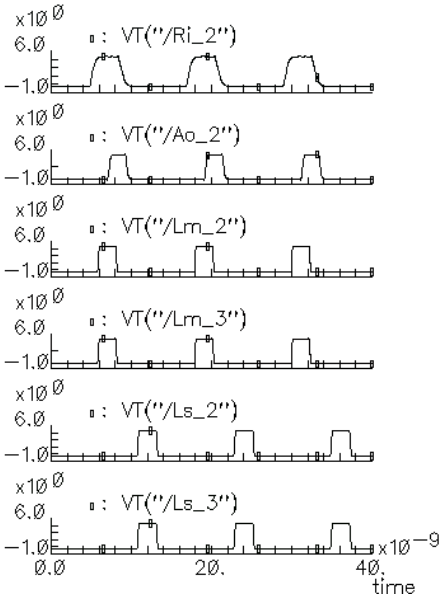


Figure 5-9: Waveforms of Master-Slave Latch DLAP test circuit.

		Edge Triggered DLAP [nS]	Latch DLAP [nS]	Comments
1	Ri+ → Ai+	1.10	2.06	
2	Ai+ → Ri-	2.24	1.62	Including logic reset delay of 1nS
3	Ri- → Ai-	1.13	0.60	
4	Ai- → Ri+	6.43	7.78	Including computational delay of 5.03nS
5	Cycle time (Ri+ → Ri+)	10.90	12.06	Sum of lines 1-4 (include the delay set and reset times)
6	Response Time (Ri+ → Ro+)	2.90	6.90	Measured on a single empty pipe stage (i.e., the time to pass data through the master and slave)

Table 5-1: DLAP SPICE simulation results.

The latched DLAP (relative to the edge triggered DLAP) incurs slightly longer cycle time (about 1nS longer), due to the need to precisely sequence more transitions. The total overhead is still negligible compared to typical computational logic delays.

Relative to synchronous pipelines, DLAP requires about twice as many registers and a small control circuit per stage. However, when replacing each edge-triggered FF with double latches, the area overhead is kept to a minimum. The timing overhead required (for edge-triggered DLAP) is one more register loading delay. The return to zero phase of the handshake protocol is kept to a minimum. These times are typically negligible compared to the logic computational time.

Four phase handshake protocol pipelines with edge triggered registers are also used in [MBM89, MBM91], where two types of control circuits are presented: 'Half handshake' utilizes only 50% of the pipe, as only every other stage operates at a time. 'Full handshake' is more efficient, and acknowledge signals propagating backwards sequentially, can sometimes overlap stage operation. Four phase pipelines with transparent latches are presented in [DW95, FD96, FL96] (the latter employs dynamic logic). The latches are left open most of the time, resulting in possibly higher power dissipation due to data hazards. Their 'semi-decoupled' and 'fully-decoupled' schemes are similar to the 'half-handshake' and 'full-handshake' of [MBM89], respectively. A 2-phase protocol micropipeline using double edge triggered registers is presented in [YBA96], as well as a 4-phase protocol micropipeline using latches and 'blocking latch' scheme. The design is reportedly faster than [DW95], but it is still a semi-decoupled circuit, limited to 50% pipeline utilization.

The cycle time of a semi-decoupled pipeline includes approximately twice the processing delay of the combinational logic because of its 50% duty-cycle operation (i.e., a stage must wait for the following stage to clear before initiating its own next calculation). In a fully decoupled pipeline, even if all stages finish evaluation at almost the same time, a stage cannot latch the result in its output register until the following stage has. When the pipe is full, operation is limited by a single 'bubble' flowing backwards, and the latency overhead is relative to the length of the pipe. The master/slave action of the storage in a DLAP serves the purpose of interleaved bubbles in the pipeline, and relaxes the coupling between the stages. Thus, DLAP is more tolerable to changes in the output rate from the pipe than the other asynchronous pipelines.

We employ a scheduling notation (Fig. 5-10) to compare the latency incurred by four kinds of two stage pipelines, namely a synchronous pipeline, a 'semi-decoupled' ('half handshake') asynchronous pipeline, a 'fully-decoupled' ('full handshake') asynchronous pipeline, and DLAP. Three tasks (*i, j, k*) are to be processed, and the computational delays of each task per each pipeline stage are listed in Tab. 5-2. A synchronous design requires the clock cycle time to accommodate the worst case of all calculations over all stages, namely two time units, thus requiring eight time units to complete the computation (Fig. 5-10(a)). The semi-decoupled (or half handshake) pipeline [DW95, FD96, MBM89] achieves only 50% utilization. Since the pipeline contains only two stages, and they must operate alternatively, the computation takes eight time units (Fig. 5-10(b)). In a fully-decoupled ('full handshake') asynchronous pipeline [FD96, MBM89] task *k* cannot start execution at stage *A*, since task *j* is stalled there until task *i* frees stage *B*. Thus, the computation requires six time units to complete (Fig. 5-10(c)). The DLAP completes the computation in only five time units (Fig. 5-10(d)), since stages *A* and *B* are decoupled by the double latches between them, and task *k* is not stalled.

We designed a three stage fully-decoupled pipeline [FD96] and a full-handshake pipeline [MBM89, MBM91] in MOSIS 0.8μ CMOS process, and ran detailed SPICE simulations to compare the performance with DLAP. The results are summarized in Tab. 5-3. Values in the overhead columns are calculated as the difference between the measured cycle time and the slowest (stage or output) delay. The pipelines were implemented as balanced, i.e., all stages of the pipeline had the same delay. The processing delay of the combinational logic (the data path in each pipeline stage) was measured as 11.2nS, and its resetting time as 1.8nS. Thus, its total contribution to the cycle time was 13nS. The cycle time (i.e., Ri+ → Ri+) was measured for both an empty pipeline case, and a full one.

	Task i	*Task j*	*Task k*
Stage A	1	1	2
Stage B	2	1	1

Table 5-2: Processing times (in relative time units).

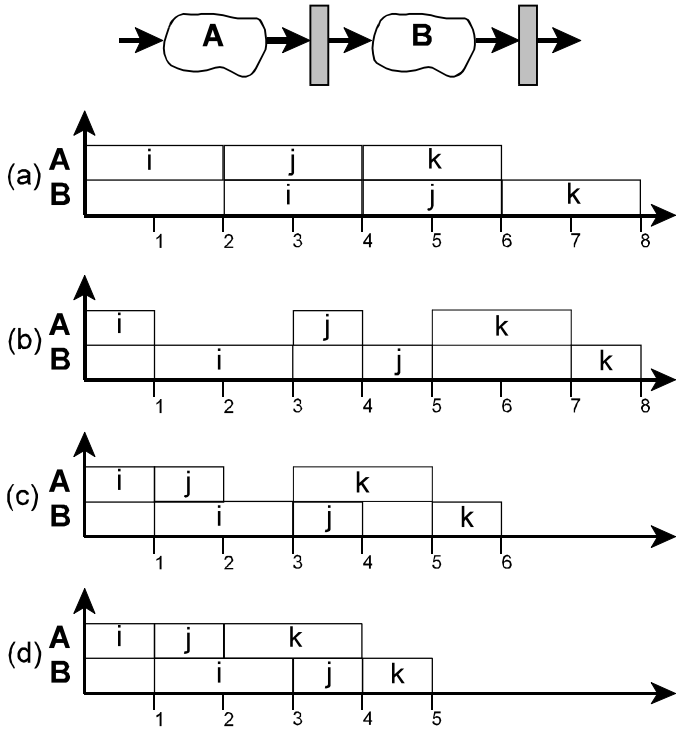


Figure 5-10: Scheduling comparison of alternative pipelines: (a) synchronous; (b) semi-decoupled asynchronous; (c) fully-decoupled asynchronous; and (d) DLAP.

The tested empty pipeline had an output rate lower than the processing rate, so the pipe never filled up, and operation was not limited by lack of bubbles. The cycle time was determined by the stage logic delay and the pipeline control circuit (including the latching time). The results show that DLAP cycle time is slightly slower than other asynchronous pipelines. However, it is only 1.5-2nS slower, mainly because of the extra register load required. Since DLAP is best for data-dependent delays, this extra overhead is tolerable.

When the pipe starts from an empty state, the acknowledge propagates backwards concurrently to data forwarding, and thus it does not depend on the number of stages in the pipe. Since the pipeline is balanced (i.e., all stages in the pipe take the same time to complete), stage *i* starts processing before stage *i-1* does, and it also finishes its work earlier. Therefore the acknowledge from stage *i* is ready when stage *i-1* needs it. When a DLAP starts working from an empty pipe state it has a similar behavior, and although it has many bubbles - they do not help, since the bubbles are not a limiting factor. Since DLAP has one more latch to load in each stage over fully-decoupled, and its latches are normally kept closed and not normally open, then obviously the overhead of the cycle time includes the time of opening and closing of that latch, even if the control circuit takes the same time. Driving the latches usually takes longer than the control. Thus, DLAP is not faster than other asynchronous pipelines when the pipeline is empty most of the time.

The tested full pipeline had an output rate higher than a stage processing rate, which caused the pipe to fill up. The measured delays of the slow output processing and resetting times were *t(Ro +→ Ao+) = 16.8nS*, and *t(Ro -→ Ao-) = 2.7nS*, respectively. When the delay of the pipeline sink stage is longer than each pipe stage, the pipeline becomes full, and its throughput and cycle time are limited by the output rate. All the pipe stages and pipeline source are stalled until the bubble propagates from the last stage to the first one. The longer the pipeline, the longer the backwards acknowledge delay. Since the DLAP starts with *n* bubbles, it takes a longer time until the number of bubbles is gradually reducing and all the *2n* registers are filled, before the pipe is stalled. Because is has more bubbles, DLAP is more tolerable to temporarily slow outputs than the other asynchronous pipelines. However, when the pipeline sink is slow for long enough, the DLAP will eventually be stalled just the same.

DLAP is best for cases of variable (data-dependent) delay stages, as showed by the scheduling analysis. It is also suitable for automatic conversion from (balanced) synchronous pipelines, to asynchronous ones, without redesign, as described in Sect. 5.7.

	Edge Triggered Registers				**Transparent Latches**			
	Full-Handshake		DLAP		Fully-Decoupled		DLAP	
	Cycle	Overhead	Cycle	Overhead	Cycle	Overhead	Cycle	Overhead
Empty Pipe	17.0	4.0	18.7	5.7	17.0	4.0	19.2	6.2
Full Pipe	20.2	0.7	22.0	2.5	20.8	1.3	22.3	2.8

Table 5-3: Cycle time and overhead SPICE simulation results (measured in nS).

5.6 Non-Linear DLAPs

Evidently, a linear DLAP is not enough to implement complex data-path structures, which are needed in a processor design. Non-linear DLAP data paths can be created by using *Fork* and *Join* interconnection circuits. A *Fork* is basically a two output pipeline stage. Figure 5-11 shows the implementation of an edge triggered *Fork* DLAP. Note that both following stages share the same *Ro* line, while their *Ao* lines are combined by the C-Element. Similarly, the *Join* is a two input pipeline stage, as presented in Fig. 5-12. The *Ri* signals are combined by the C-element.

Many synthesized circuits have complex structures that contain loops (aka rings). A DLAP ring can be constructed by employing *Join* and *Fork* circuits as shown in Fig. 5-13.

A ring structure based on fully-decoupled pipeline scheme must contain an extra register (i.e., an empty stage) to prevent a deadlock [MBM91]. The single bubble going backwards around the loop might limit and slow down the ring operation. DLAP ring is faster than a fully-decoupled one when stage delay is shorter than the acknowledge round trip delay. Using semi-decoupled pipeline to construct a feedback loop yields a ring with only half stages full, since only alternate blocks can store valid values [MBM91]. DLAP rings have enough many bubbles and can operate the same way as synchronous rings.

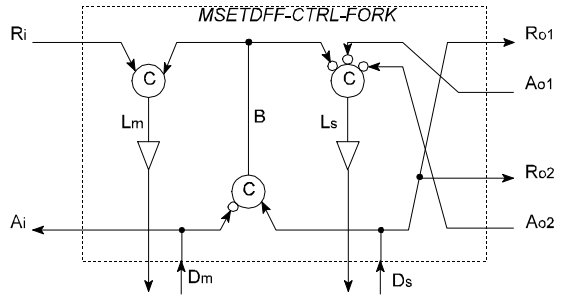


Figure 5-11: A Fork stage implementation, two output pipeline interconnection circuit.

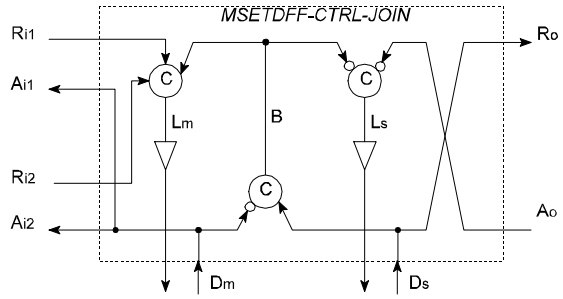


Figure 5-12: A Join stage implementation, two input pipeline interconnection circuit.

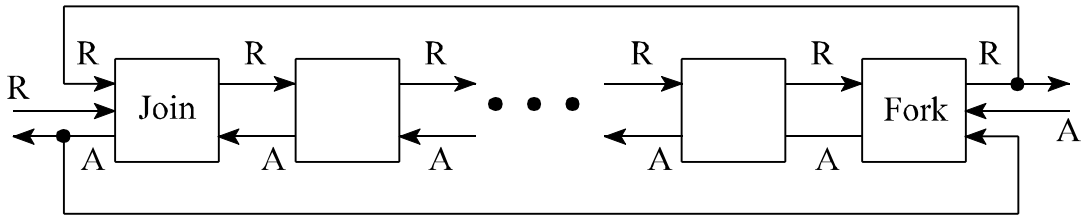


Figure 5-13: A ring DLAP.

5.7 Synchronous to Asynchronous Conversion

5.7.1 Motivation

Several advantages can be achieved by converting synchronous circuits to asynchronous ones. The clock signal in synchronous circuits always switches (i.e., it has 100% activity) and it must arrive to all parts of the chip with the same phase. The power dissipated by complex VLSI chips increases as clock frequency rises [Hor93, Int94, Str94]. Growing portion (currently over 40% [Bow95]) of the power budget in a chip is required by the clock distribution network in order to

reduce clock skew problems. Asynchronous logic does not use a clock. Logic elements can be *self-timed*: they can detect and announce when the computation is complete and the outputs are ready [Hau95, Sei80]. They also wait for inputs to be announced before starting the computation. Registers load their inputs under local control, rather than on a global clock edge. Thus, eliminating the clock and replacing it with local handshakes can save power.

Since power is a square function of the operating voltage, much power can be saved, even with the same circuit design, by applying a scalable power supply [NNS+94]. If the circuit is self timed, reducing the power will only cause it to work slower, but will not affect its correct operation. This can be exploited in portable devices, e.g., at standby mode or while processing data requiring variable computation loads [vBB+94].

Asynchronous pipelines can potentially have higher performance than synchronous pipelines because they are not restricted to operate according to the worst case delay. Using self-timed logic enables the circuit to operate according to actual data-dependent delays, i.e., the average case delay. The worst case delay might be very rare and the average case delay is usually about half of the worst case delay [GM90].

When self-timed logic is not available, a delay (matching the worst case delay of the pipeline stage) can be used to signal the end of the combinational logic evaluation time. The worst case delay is matched per each pipeline stage and does not affect the delay of other stages. Even in this case complex asynchronous pipelines can be faster than synchronous pipelines. In a complex pipeline (e.g., a microprocessor) data can go through short or long alternative paths, depending on the instruction: A pipeline stage containing an ALU with an adder and a multiplier can have a variable matched delay, since only one of the units is active at a time. Thus, each possible path operates at a rate affected only by the slowest unit in its path, and not affected by other paths.

Synchronous synthesizers are widely used as CAD tools for designing VLSI circuits. They are proven reliable and able to handle large and complex designs. Currently available asynchronous synthesis tools [Async] typically generate only rather small asynchronous controllers, and do not handle data path elements. Using a post-synthesis conversion, enables us to take advantage of the design structure and combinational logic of the data path synthesized by the synchronous tool, while achieving the benefits of asynchronous circuits.

Synchronous to asynchronous circuit conversions can also be used in *mixed-timed* designs. Converting the synchronous modules to asynchronous ones eliminates the synchronizers needed at the synchronous module inputs, and prevents synchronization failures (cf. Ch. 6). Changing one or some of the modules requires no changes in other parts of the design.

We consider synchronous logic synthesized into netlists according to the common architecture of ‘register-and-cloud’ pipelines [Per94], where ‘clouds’ of combinational logic are separated by clocked registers. We would like to take advantage of the general pipeline structure and of the combinational logic clouds, but we need to get rid of the clocked registers, thus converting a synchronous circuit into an asynchronous one. To that end, we must identify the best target asynchronous pipeline. The DLAP architecture was found to be most suitable to such a synchronous-to-asynchronous conversion, since it can imitate the synchronous operation, and can also benefit from variable computation load. The master-slave architecture of the DLAP operates the same as synchronous pipeline does. DLAP is also ‘synchronous compatible’ when values are loaded in parallel before the pipeline operation starts.

This section describes the methodology to convert a synchronous design to an asynchronous one, at the gate level, based on a DLAP architecture. Our previous work on post synthesis conversion, targeted at different implementation styles is described in [KGS96, KGS97].

5.7.2 Post-Synthesis Conversion Algorithm

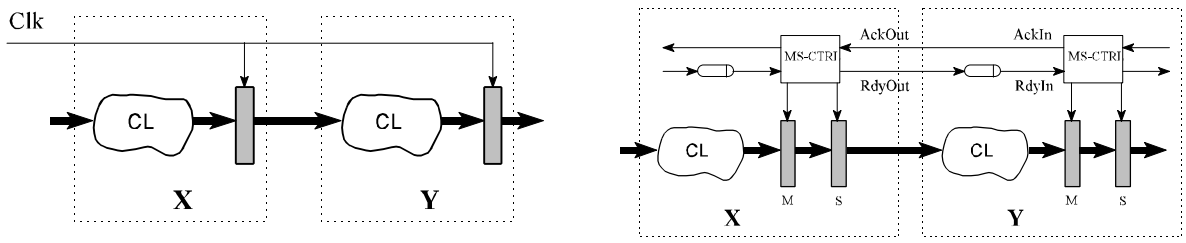
The conversion algorithm applies to synchronous netlists which are typically synthesized by a tool like Synopsys. The algorithm retains the (possibly complex) pipeline structure as generated by the synthesizer. The combinational logic (the ‘cloud’) between the registers is not altered. However, if the combinational logic of the design is not self timed (i.e., it does not generate a completion signal), matching delays are generated by the conversion algorithm and are used to generate proper completion signals, as explained in Sect. 5.2.

The same algorithm is suitable for either edge-triggered registers or transparent latches based DLAP as the conversion target architecture. The only difference is replacing each original edge-triggered flip-flop with either double edge-triggered flip-flops, or double transparent latches, and adding the proper DLAP stage controller. As explained above, the double latches architecture has less area overhead, and is only slightly slower than the edge-triggered version.

The input to the conversion algorithm is a synchronous netlist, containing edge-triggered D flip-flops, and combinational logic blocks. The netlist includes the definitions of the design primary inputs, primary outputs, and list of registers. The algorithm also requires as input the delays of the combinational logic blocks between pipeline stages. These delays are obtained from a timing analysis tool that analyzes the worst case delay of each data path between all pairs of points connected by logic. Such a data path begins at either a primary input or an output of a flip-flop, and ends at either a primary output or an input to a flip-flop.

Every flip-flop (FF) bit of every original register is replaced with two latches (or FFs), the first one feeding the second. The clock signal is replaced by either of the two latch signals (*Lm, Ls*, as explained in Sect. 5.2) connecting to the clock input of the first (master) FF and second (slave) FF, respectively. The required *Done* signals are generated by returning the latch signals (driven by the DLAP controller), which are delayed by the register driving delay (Fig. 5-2).

A DLAP stage controller circuit is added for every pair of latches (or a group of pairs, treated as a bus, as explained below). The DLAP stage controllers are interconnected as follows: If stage *X* produces data for stage *Y* (cf. Fig. 5-14(a)), then the *ReadyOut* signal of the stage controller *X* is connected to the *ReadyIn* input of stage *Y* (cf. Fig. 5-14(b)) via a proper matched delay unit. The *AcknowledgeIn* output of stage *Y* is connected to *AcknowledgeOut* input of stage *X* (refer also to Figs. 5-1 and 5-2).



(a) Original circuit. (b) Post conversion circuit.

Figure 5-14: Synchronous-to-asynchronous conversion.

Hence, each flip-flop in the original netlist is replaced by a DLAP stage, containing either two edge-triggered D-FF, or two transparent latches, with a proper DLAP stage controller. Each stage controller generates the local 'clock' signals, to latch data into the stage. The control lines are connected (through properly matched delays) to reflect the data flow. The architecture of the netlist, and interconnections between the DLAP controllers determines the dependencies among them. Splitting computation paths (from a single source to several destinations) is done by *Fork* controllers, and merging multiple computation paths requires *Join* controllers, as described in Sect. 5.6.

Designs usually contain many busses, i.e., several bits emerging from a set of flip-flops, and passing through a combinational logic block before entering a set of flip-flops (e.g., a 32-bit number). All signals that are buses are aggregated and treated as one, using the worst delay for the bus delay. This peep-hole local optimization significantly reduces the number of paths (and therefore controllers and delay units) that have to be handled. For example, two 32-bit busses feeding an adder require only two DLAP controllers rather than 64. In case some bits are

extracted from the bus towards different destinations, the relevant delays are calculated and separate delay units are used.

The post synthesis conversion algorithm has been implemented and tested on a collection of small circuits, e.g., a simple finite input response (FIR) filter design. The converted design was tested by simulation, after modifying the testbench program (dropping the dependence on a single clock signal and adopting a single rail, 4-phase communication protocol). The design was verified by using test vectors, and comparing the results to those generated by the original synchronous design. The modified testbench controls the response of the environment to events and responses of the design under test, and can be used to test various environment behaviors. Work is underway to complete the programming and perform large scale tests.

5.8 Concluding Remarks

The DLAP scheme is not limited to the implementation of an asynchronous microprocessor. It applies to any asynchronous circuit structured as a simple or complex pipeline. In our quest for high performance asynchronous implementations for *Kin*, as well as for efficient conversion of synchronous circuits into asynchronous ones, we have examined various asynchronous pipeline schemes [MBM89, MBM91, DW95, FD96, FL96, YBA96, SS93], and have found that none operates as efficiently as a balanced synchronous pipeline. Consequently, we have developed the doubly-latched asynchronous pipeline (DLAP) which employs master-slave registers. DLAP is capable of truly decoupled operation: All pipeline stages can shift data simultaneously, and execution is faster even if variable delays are encountered. We have shown implementations based on either edge-triggered registers or transparent latches. Both designs have been defined with STGs, verified, and fully simulated and compared with previous architectures.

DLAP is best for variable (and data-dependent) delays, both of the internal stages and the output process, and it is best for rings which suffer from lack of bubbles. It is suitable for balanced pipelines. However, if the computational load per stage is small (relative to the overhead), it is slightly slower than a fully-decoupled pipeline due to the extra registers.

DLAP is suitable for automatic synchronous to asynchronous conversion from (balanced) synchronous pipelines, in order to eliminate clocks (for power saving, or for easier interface to other asynchronous units), without redesign.

The DLAP controllers described in this chapter operate according to the four-phase communication protocol. However, DLAP controllers can also be implemented for two-phase protocol operation, which might be faster (due to the fact that only half the number of handshake

transitions are required [AML97]). The DLAP pipe control logic is fully contained, hence the handshake overhead can be reduced without affecting anything else.

A variance of DLAP can be constructed by employing a semi-decoupled pipeline with logic units only at every other stage. However, it appears that this would require more hardware, and consume more power, than the DLAP scheme presented in this paper, because of the redundant internal handshake transitions.

Newer versions of *Petrify* and other synthesis tools may be applied to the STGs presented in this paper, to synthesize simpler DLAP control circuits. The circuits generated can be implemented with Set-Reset FFs instead of C-elements for standard cell implementation. Faster DLAP controllers might be implemented based on generalized C-elements, or designed to operate according to non-blocking schemes, as in [FD96], or semi-blocking, where the first latch is normally open, while the second one is normally close. The control circuits we used were designed to be delay insensitive. However, simple engineering optimization techniques can be applied for lower latency overhead, e.g., overlap control circuit timing with latch operation.

Chapter 6 : Adaptive Synchronization for Multi-Synchronous Systems

While the highest architectural level of *Kin* is asynchronous, as described above in Ch. 2, the various units in it may be implemented according to different timing disciplines. Chapters 4 and 5 presented asynchronous design methodologies, where the circuits are either originally designed as asynchronous, or converted from synchronous designs. In this chapter we present another possible implementation for *Kin*, as a *multi-synchronous* system. This proposed implementation can be used as an alternative migration path from a complete synchronous design to a complete asynchronous design.

Multi-synchronous clocking discipline is based on a common clock distributed over thin wires, avoiding the massive power investment in clock distribution trees and circuits for phase matching and skew minimization. Hence, all the processor units operate at the same clock frequency, but have an arbitrary clock phase. *Adaptive synchronization* is used to substantially reduce the probability of synchronization failure (when data are sent between the units), and reduce performance degradation caused by synchronizers. In contrast with clock-manipulating techniques, such as clock stretching, adaptive synchronization adjusts data delays. Thus, adaptive synchronization can be used to handle synchronization problems that are harder to solve by clock phase adjusting methods, e.g., when data are received from more than one source.

6.1 Introduction

With the advance of technology, the integration levels of VLSI chips grow from millions of transistors per chip towards the hundreds of millions. VLSI chips, e.g., microprocessors, grow larger and run faster. Over half a billion transistors on a die and clock rates in excess of 1 GHz are predicted for year 2010 [KG97, SIA94]. Distributing a single 1 GHz clock to all parts of a 0.5B transistors chip becomes very expensive: To assure minimal skew and short rise and fall times, a growing portion of total power is dissipated by the clock distribution network, including the phase lock loops, buffers, and tuning circuits [Fri95]. For instance, currently over 40% of the power budget in an Alpha chip are consumed by a clock distribution network for reducing clock skew problems [Bow95].

Asynchronous design, as discussed and presented in previous chapters, is often proposed as a viable solution, removing the clock altogether [DGY93, Hau95, Pav94, SSM94]. In this chapter, however, we propose a new clocking method which both saves power and facilitates synchronization.

A possible solution to the clock distribution problem lies in non-synchronized operation, wherein the various modules on the chip do not maintain a known relative clock phase to each other, and intercommunicate asynchronously. However, the design of non-single clocked synchronous systems should be closely woven with the concern for synchronization. In addition, it seems useless to synchronize far-apart clocks, since data lines spanning large portions of the chip may be subject to substantial propagation delay (close to, or in excess of, the clock cycle) and be out of sync in any case. On the other hand, since the relative skew of the clock as it arrives at the various modules is immaterial for non-synchronized operation, the clock signal can be distributed over networks that are designed to dissipate minimal power and occupy minimal area.

This chapter discusses some previous work done on synchronization issues, and defines multi-synchronous systems. Then, in contrast with clock-manipulating techniques, an *adaptive synchronization* is presented, to adjust data delays, and substantially reduce the probability of synchronization failure. It is further proposed that the adaptive synchronization be performed semi-statically, adapting various data delays from time to time. The suggested adaptive synchronization approach is compared with synchronizers and stoppable clock schemes.

6.1.1 Previous work

The most tightly coupled, highest performance systems today (such as high end microprocessors) operate on a single common clock with minimal skew (dissipating a lot of power to achieve that), avoiding synchronization issues altogether. Variations include multiple synchronized clocks operating at several frequencies and phase locking. For low bandwidth communications among systems with uncorrelated clocks, synchronizers are employed successfully [CM73, CW75, Mar81, Pec76, RMC+88, Sei80, Sei94, Sto82, Vee80]. Alternative methods that have been proposed include stretchable clocks and clock tuning [Cha84, Cha87, Keh93, Pec76, PN95, RMC+88, Sei80, YD96]. Self-clocked data (such as Manchester coding on Ethernet [MB76], and start/stop bits on RS232 serial communications) are exchanged when even lower bandwidth is needed.

The synchronization problem has received a lot of attention [Cha87, CM73, CW75, Gre95, Keh93, Mar81, Pec76, PN95, RMC+88, Sei80, Sei94, Sto82, Vee80, YD96]. Solutions have been developed for a wide range of applications, from intra-chip communications to wide area

networks. As technology progresses, integration levels and computational speeds increase, and systems which used to require multi-board implementations are expected to fit inside single chips. Likewise, the synchronization methods that were once applicable to backplanes and multiple boards should now be considered for the inner circles of chips.

Low level clock/data synchronization is typically handled by synchronizers. However, they are principally suitable for low bandwidth communications, and a number of issues render them less effective in high performance chips. First, synchronizers may occasionally fail due to metastability [CM73, CW75, Mar81, Pec76, Sto82, Vee80]: A synchronizer might enter a metastable state, or take abnormally long time to settle. While the probability of failure has been kept very low, this is exponentially more difficult to achieve when the cycle time becomes aggressively shorter (as described below in Sect. 6.5). Second, in high performance systems, modules may receive many data inputs concurrently from many other modules and at high rates; consequently, the probability of at least one input switching at the same time as the clock is growing beyond negligible levels. Third, synchronizers incur at least one clock cycle delay; this may lead to unacceptable long latencies accumulating over multi-module paths, and be especially limiting on cyclic paths such as between a reservation station and the execution units of a high performance processor.

Stretchable (or stoppable) clocks [Cha84, Cha87, Pec76, RMC+88, Sei80, YD96] have been proposed as an alternative to synchronizers. A ring-oscillator based clock generator is attached to each synchronous module. An arbiter detects clock/data conflicts and stretches the ‘off’ phase of the clock (thus trading failure for a long delay). Stretchable clocks are subject to two drawbacks. First, the multiple clock generators typically develop frequency variations, due to temperature and supply voltage in-die variations. As a result, relative inter-module phase shifts drift continuously, causing frequent recurrences of conflicts. Second, as with synchronizers, high bandwidth communications received over many channels increase the probability of clock/data conflicts. This fact leads to a high rate of clock stretching events, severely impeding performance.

Many other variations have also been proposed. [Keh93] suggests clock (phase and frequency) tuning for performance enhancement. [Sei94] hides some synchronization latency by inter-module FIFO buffers. In the STARI protocol [Gre95], asynchronous FIFOs are employed; synchronization is achieved on the first data transfer, and is automatically maintained thereafter. The FIFO must be kept about half full, and each insertion and removal operation must complete within one cycle. If these requirements are violated (e.g., on FIFO underflow), synchronization is lost. [PN95] employs analog adjustable clock generators, achieving local self-alignment of all clocks.

We have developed a clock synchronization method that applies an external crystal clock, rather than a self generated one. That method and its limitations are presented in Appendix A. It was

found less desirable than data adaptive synchronization, which is the subject of this chapter.

6.1.2 Multi-Synchronous Systems

Consider a 0.5B transistor chip comprising 100 synchronous modules of 5M transistors each. Various clocking schemes may be employed. A single clock is feasible, but as mentioned above the cost in power and area may be prohibitive for some applications. In *multi-clocked* systems each module is clocked independently of the others, and as explained above, using synchronizers severely limits performance. Instead, we propose a *multi-synchronous* (multisync) scheme (as defined above in Sect. 1.1), whereby all modules feed off the same external crystal clock, while arbitrary relative clock phases are permitted.

Most synchronization methods assume that the arrival (switching) time of data at any module is uniformly distributed over the clock cycle, as in Fig. 6-1(a). However, in multisync systems, the arrival time of certain data channels incident upon a certain module may be distributed unevenly, e.g., as in Fig. 6-1(b). When one synchronous module outputs data to another, data output is synchronized with the local clock of the sender. Since the phase difference between the receiver and sender clocks, as well as the data interconnect delay, are stationary, data arrival time at the receiver is correlated with the receiver clock. Stationarity can be assumed because the delays and phase differences among the modules in the system are functions of the implementation, of physical parameters, and of temperature and supply voltages, and they typically change very slowly during operation. However, in systems with a high degree of connectivity the combined distribution of all channels incident upon a specific module looks more like Fig. 6-1(c), and the danger of clock/data conflicts cannot be ignored.

A multi-synchronous system is presented in Fig. 6-2. The common clock is distributed over thin wires (saving area and power, compared to minimal skew clock distribution nets). While clock frequency is the same for all modules, the actual phase shifts are considered unknown (a similar clocking is described in [Gre95], but there the system has to be reset upon synchronization loss). While the clock phase differences among modules are affected by supply voltage and temperature, these variations appear like very slow drifts, and take a huge number of cycles to be noticed. Thus, we may safely assume that the phases do not shift for relatively long time, and the phase differences can be considered stationary.

Consider modules *A* and *B* in Fig. 6-2, which are tightly coupled over an asynchronous channel without a FIFO, for high bandwidth communication. Similar to clock delays, the data delay δ_{AB} is also stationary and is considered unknown. Module *A* generates output transitions on D_A at a fixed phase difference relative to its own clock C_A. The data propagate to module *B*, which

samples *DataRdy* on the rising edge of its own clock C_B. New data are sent over from *A* to *B* at a high rate, e.g., on almost every clock cycle. Since the relative clock phase difference Δ_A-Δ_B of modules *A* and *B* is presumed unknown, the data may arrive at *B* simultaneously with the rising edge of clock C_B, creating a clock/data conflict and possibly resulting in a metastable state at the input of *B*, and in loss of data. If the relative clock phases and data delays remain fixed (stationary), and since both modules operate at the same clock frequency, this unfortunate situation is most likely to recur. The use of synchronizer in this case does not solve the problem, because the synchronizer may enter a metastable state on every conflict, increasing the probability of failure.

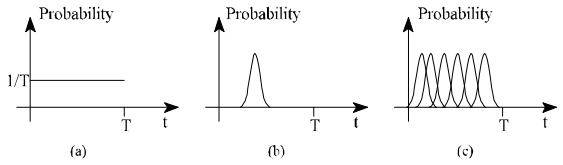


Figure 6-1: Arrival time distribution of inputs (over a clock cycle *T*): (a) uniform distribution (asynchronous input); (b) clustered distribution when the sender and receiver clocks are correlated (single synchronous input); (c) combined distribution of many independently correlated inputs is similar to uniform distribution (multiple synchronous inputs).

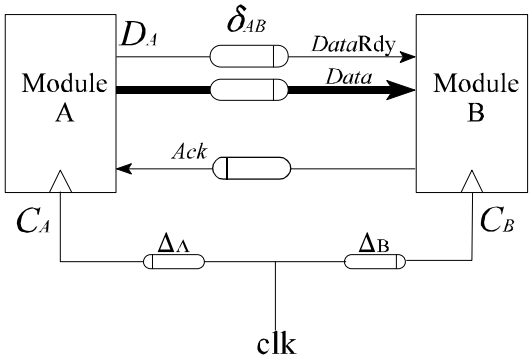


Figure 6-2: A multi-synchronous system.

The novel method we propose suggests adjusting data timing rather than the clock, thus converting data arrival time distributions into forms like Fig. 6-1(b) and substantially reducing the synchronization problem. [Sei94] has employed pipeline synchronizers in order to convert the uniform distribution of asynchronous inputs into a non-uniform distribution, useful for a synchronous receiver. The latency of the pipeline synchronizer is rendered unnecessary when the sender is also synchronous, as discussed above.

In the following we assume the system architecture to be asynchronous (as is the case for *Kin*), thus altering various delays do not affect system correctness. We describe systems operating with a four-phase handshake protocol, but the results may also be applied to two-phase handshake.

6.2 Data Adaptive Synchronization

Data adaptive synchronization adjusts the delays on the data lines instead of adjusting the local

clock phase. Since the communication channels are connected point to point, the delays on them can be changed so that they do not conflict with the local clock, without affecting the other channels (this approach also applies to bus taps). We add a data coordination circuit for each communication channel, as in Fig. 6-3(a). When a conflict is detected, the data delay is adjusted to prevent conflicts in future communications.

Note that three different phases are assumed stationary in the multisync model (Fig. 6-2): The clock phase difference Δ_A-Δ_B, the sender data phase D_A-C_A, and the data delay δ_{AB}. Consequently, the phase of the arriving *DataRdy* at module *B* relative to C_B is also stationary. In other words, the arrival time distribution is represented in this model by Fig. 6-1(b), and is highly non-uniform. In the following, we take advantage of this fact and control data delays so as to assure that the center of this distribution is safely remote from the clock transition for every data line in the system. This is achieved by tuning the data delay δ_{AB}.

The adaptive mechanism architecture for a specific module is shown in Fig. 6-3(a). Data input channels DI_i are subject each to given data delays δ_{Ii} (ref. δ_{AB} in Fig. 6-2). Adaptive synchronization circuits A_i, clocked by the local clock CK_{in}, monitor the *DataRdy*$_i$ lines, and control adjustable data delays δ_i, whose value is in the range $0 \le \delta_i < T$ (T is the clock cycle). The function of the A_i is to separate the clock and data transitions. The multiple input delays can be adjusted independently of each other, so the combined data arrival time distribution at the entry to the module looks like Fig. 6-3(b). Adaptive synchronization applies equally well to single sender, multiple receivers buses (Fig. 6-4).

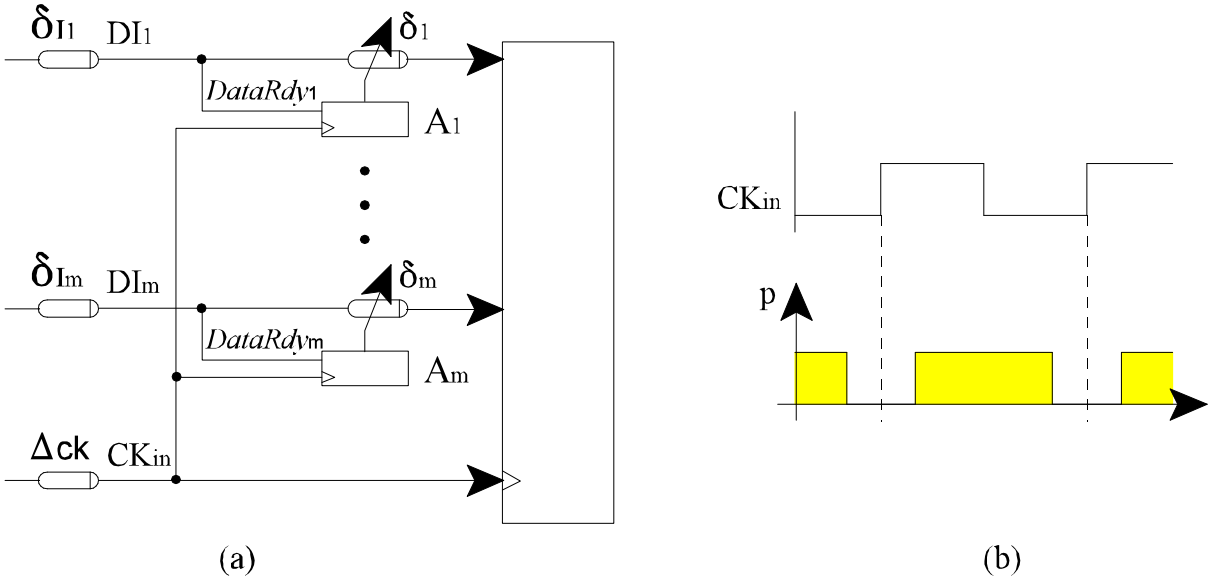


Figure 6-3: (a) Adaptive Synchronization; (b) Combined data arrival time distribution — data delays are adjusted to avoid conflicts.

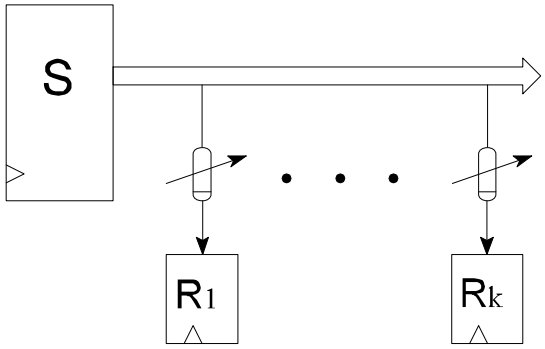


Figure 6-4: Adaptive Synchronization for single sender, multiple receivers buses; each receiver adjusts its own data input.

6.3 Data Adaptive Synchronization Circuit

The principles of adaptive synchronization resemble self-clocking communication mechanisms, such as in UARTs. The challenge is to obtain proper operation even at the presence of metastability. Consider the adaptive synchronization circuit in Fig. 6-5, with an adjustable delay (Fig. 6-6). A four-phase data signaling discipline is assumed, wherein *DataRdy* rises to '1' after the new data are available (the circuit may be readily extended to two-phase operation as well). The receiving module (cf. Fig. 6-2) latches the inputs upon the positive edge of its local clock, and only if *DataRdy* is '1'. Thus, the purpose of the adaptive synchronization circuit is to detect the phase of *DataRdy* relative to the local clock, and to adapt the δ_i delay if that phase is dangerously close to 0 or T (2π).

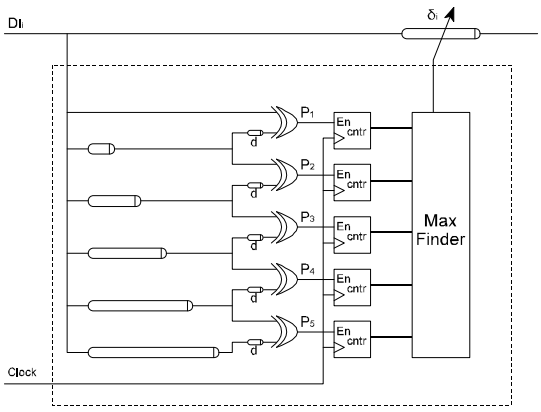


Figure 6-5: Adaptive Synchronization circuit.

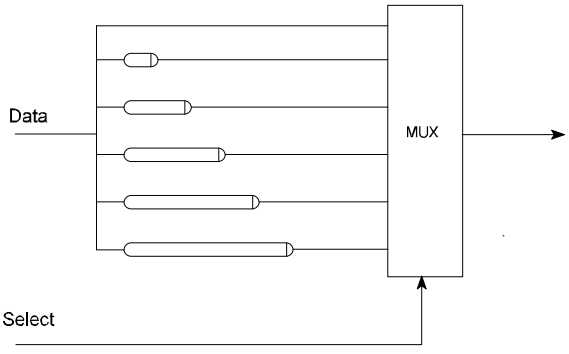


Figure 6-6: Adjustable delay circuit, consisting of multiple delay lines and a selector.

First, the data is fed into a phase detector. Let's assume that the *DataRdy* lines switch (up or down) on every cycle. In [Keh93], several delayed phases of the clock are used to detect data transition time. In Fig. 6-5, several delayed versions of the data are employed instead. The xor gates generate a sequence of pulses, as in Fig. 6-7. The delays marked 'd' assure a small pulse overlap. The outputs of the xor gates are the enable signals of counters which are triggered by the local clock edge. On the rising edge of the clock, one or two of the counters increment their count. This is repeated for a large number of cycles, e.g., 1000 times. At the end of that time, the counters are expected to show a distribution similar to either Fig. 6-8(a) or Fig. 6-8(b). In either case, two or three counters show large counts, and the remaining ones are close to zero. The spread is caused by pulse overlap, by clock and delay jitter, and by pulse/clock conflicts which may result in metastable states, in long settling times, and in indeterminate counting. In spite of such physical difficulties, the phase detector is robust thanks to many repeat counts, and it produces a very clear indication of the relative phase of the *DataRdy* line. The circuit in Fig. 6-5 is similar to delay lock loop (DLL) circuits, except that the proposed circuit is digital rather than analog, and its operation is algorithmically controlled.

Next, the *MaxFinder* circuit determines, according to which counter has won, if the δ_i delay of the data lines should be changed, and by how much. For example, the count depicted in Fig. 6-8(a) indicates no change, while that of Fig. 6-8(b) calls for adding a delay of at least T/5. The adjustable delay consists of multiple parallel delay lines and a selector (Fig. 6-6). Notice that although the phase detector examines only the *DataRdy* line, δ_i is applied to all data lines of the *i*'th channel.

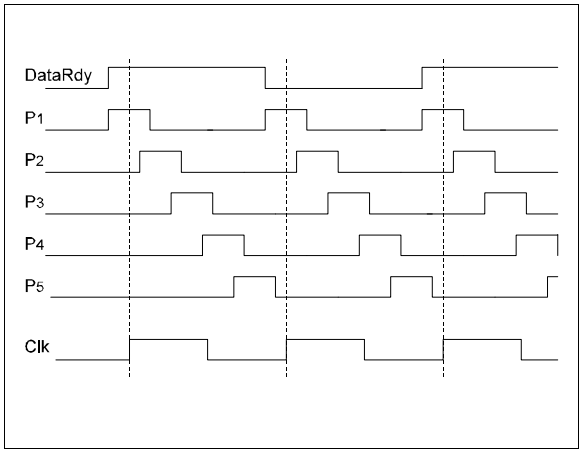


Figure 6-7: Phase detection waveforms.

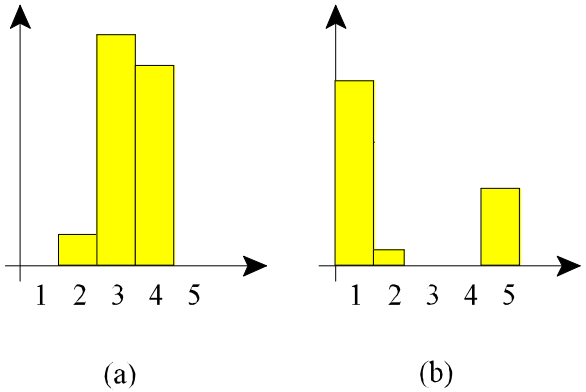


Figure 6-8: Typical phase detection counter outputs: (a) data transition is safely within the cycle; (b) data delay should be increased to avoid clock/data conflicts.

Although the examples present circuits for the case wherein the clock cycle is divided into five periods, at high frequencies it might be simpler to implement using only three such periods (since the clock cycle time may be only a few gate delays long). The circuit complexity of a proper adaptive synchronization circuit, for a 32-bits data path, is approximately 5,000 transistors (comprising the delays, xor gate, counters, comparators and switches in the max finder circuit, and the adjustable delay circuits). Thus, the total overhead per a 5M transistor module with 10 input channels is about 50,000 / 5M = 1% (recall that it replaces a massive clock distribution network). The extra power consumption is similarly marginal.

Adaptive synchronization is suitable for a wide range of applications. Typical data delay range is $0.1T < \delta_{Ii} < 1.5T$ for a 0.5B transistors, 1GHz chip, but the delay may be much larger than T for multi-chip and MCM configurations. In such cases, new asynchronous data signaling methods could be used, such as multiple message windows (wherein multiple messages are sent before an acknowledge is expected). As long as relative delays are stationary, adaptive synchronization remains applicable.

6.4 Training Sessions

Adaptive synchronization may be performed continuously, in parallel with normal circuit operation. However, modifying the data delays may cause timing problems at the time of change, so this is best carried out while the system is not performing any real task. In addition, during normal operation it cannot be guaranteed that all *DataRdy* lines switch frequently enough. And continuous adaptation may be unnecessary if all delays are highly stationary and stable.

Consequently, special training sessions are proposed for adaptive synchronization. During a training session the system stops performing all real computations. Instead, all *DataRdy* lines are toggled every cycle, and all adaptive synchronization circuits operate and adjust the δ_i delays. Any synchronization failures during a training session can obviously be ignored. The training session requires a relatively small number of counting cycles. Since all adaptive synchronization circuits operate in parallel, 100,000 clock cycles (0.1mS at 1GHz) seems a safe bound on the required session duration.

A training session is always employed after reset, for initial adjustment of all delays. Thereafter, training sessions can be invoked either periodically or as required. Periodical training frequency depends on process parameters (especially delay stability) and operational parameters (such as clock frequency and dynamic temperature and voltage variations), but it is estimated that at 2010 technology much less than one training per second will be required. The expected performance overhead in thus much less than 10^5 cycles / 10^9 Hz = 0.01%.

Training sessions are also proposed in [SCI92], wherein a point-to-point communication ring architecture is defined. Training sessions are utilized to send sync packets at ringlet initialization, and once every time interval appropriate for normal operation of the particular implementation. Clock skew in [SCI92] is handled (using Phase Lock Loop circuits) by observing incoming clock and local clock phases.

Significant temperature and voltage variations may be sensed on-chip by special sensors in order to invoke a training session when a problem seems imminent. Alternatively, the adaptive synchronization circuits themselves may be modified to act as the sensors. If any such circuit detects that any switching phase approaches 0 (or 2π) closer than some safety threshold, a hardware interrupt is invoked to start a training session. In addition, a training session can also be triggered when a higher level logic (or software) detects a synchronization or communication failure. A similar tuning idea is used in [Keh93].

6.5 Probability of Synchronization Failure

In this section we analyze the failure probability of the adaptive synchronization (A/S), and compare it to synchronizers.

Synchronization failure might happen at a training session, failing the delay adaptation process and causing the system to fail, or during regular operation, after a successful training. Failures during training sessions do not affect system operation, and might only cause the training itself to fail. These synchronization failures can happen in the phase detection circuit (Fig. 6-5), when one of the counters enters a metastable state while incrementing its count, due to marginal triggering. Since a training session takes many cycles, the counters are allowed sufficient time to resolve any metastability before their outputs are read. Thus, the probability of failure of the training session is practically zero. After a successful training session, all delays are adapted properly so that data is expected to arrive at a module around the middle of the local clock cycle, and avoid synchronization failures. However, due to possible jitters in clock phase and line delays, the data arrival time might randomally change from cycle to cycle, and become dangerously close to a clock edge. The system model for the failure analysis is described in Fig. 6-9. The phase of the clock at module *B* is affected by the delay along the clock distribution network from the clock source to module *B*. Data sent from module *A* will arrive at module *B* with a phase affected by the delay of the clock signal to module *A*, the internal logic delay from clock edge to data output, and data propagation delay to the input of *B* (Fig. 6-9(a)). We assume normally distributed jitters, and define two random variables with normal (Gaussian) distribution, X_c and X_d, representing the phases of the clock and data at module *B*, respectively. $X_c = N(\mu_c,\sigma_c)$ is normally distributed with mean μ_c (equal to the clock cycle time T) and standard deviation σ_c (caused by jitter

effects). Without limit of generality, we take the phase of the clock to be 0 (i.e., μ_c=T), since we are only interested in the relative phase of data to clock, and cyclically the phase is $2\pi k$ (k an integer). X_d = $N(\mu_d,\sigma_d)$, where μ_d is the expected arrival time within a clock cycle. After a training session, μ_d is expected to be at the middle of the clock cycle, i.e., μ_d= T/2, assuming X_c is centered on $0+2\pi k$ (see Fig. 6-9(b)). Note that X_d is actually a sum of three normally distributed variables, so its variance (σ^2_d) is calculated as the sum of three variances. Assume ϵ is the time window within a clock cycle (Fig. 6-9(c)) in which data must be stable (generally considered to be the setup-and-hold period) to avoid metastability.

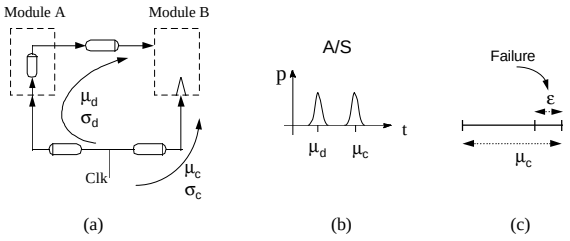


Figure 6-9: Model for analyzing synchronization failure.

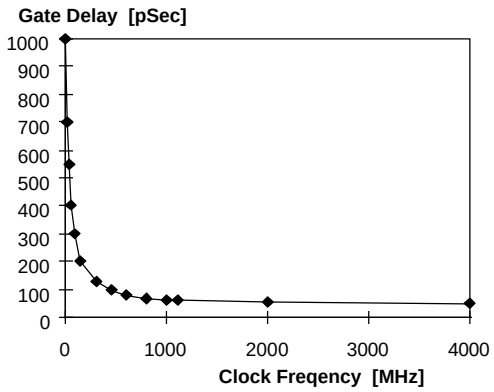


Figure 6-10: Gate delay vs. clock frequency.

When using a synchronizer, there is no knowledge of the data arrival time, so uniform arrival time distribution is assumed. Once a synchronizer has entered the metastable state, the probability that it will still be metastable some time later has been shown to be an exponentially decreasing function [Cha83, RC82]. The probability of synchronization failure of a synchronizer is given by

(6-1) $$P_{failure}(Synchronizer) = P[meta|t=0] \times P[meta|t=(\mu_c-\epsilon)] = \frac{\epsilon}{\mu_c} \times e^{-\frac{(\mu_c-\epsilon)}{\tau}}$$

It equals to the probability that a synchronizer which enters a metastable state (at time $t=0$), still remains in the metastable state at the time its output should be stable for sampling in the next clock cycle. The parameter τ is the exponential time constant of the decay rate of the metastability (discussed below).

The probability of failure of the adaptive synchronization is the probability that the values of the two random variables X_c and X_d are too close (within ϵ) to each other, i.e., the data switches too close to the clock edge. This probability can be calculates as the probability that a random

variable, equal to the difference of the two random variables, has a value in the forbidden range:

(6-2) $$P_{failure}(A/S) = P[-\epsilon \le (X_c - X_d) \le \epsilon] + P[-\epsilon \le (X_d - X_c) \le \epsilon]$$

Note that the normal distribution of the difference random variable spans beyond [0,T], and because of the 2π cycling, X_C should be considered at both 0 and T. Since we assumed normal distributions, each of the probabilities in Eq. 6-2, can be calculated by the Gaussian function, with the proper parameters [Pap91], e.g.,

(6-3) $$P[-\epsilon \le (X_c - X_d) \le \epsilon] = \mathbf{G}\left(\frac{\epsilon - (\mu_c - \mu_d)}{\sqrt{\sigma_c^2 + \sigma_d^2}}\right) - \mathbf{G}\left(\frac{-\epsilon - (\mu_c - \mu_d)}{\sqrt{\sigma_c^2 + \sigma_d^2}}\right)$$

The value of the Gaussian function is determined by the error function, *erf(x)*, whose value can be obtained by the *ERF(x)* function with parameter transformation:

(6-4) $$\mathbf{G}(x) = \frac{1}{2} + \mathrm{erf}(x); \qquad \mathrm{erf}(x) = \frac{1}{2}\,\mathbf{ERF}\left(x/\sqrt{2}\right)$$

Technology is defined by the gate delay, which also limits the highest clock frequency that can be used. However, the clock frequency increases faster than the gate delay decreases, as can be observed from Fig. 6-10 (based on data from [KG97, SIA94, Wei96]). Since gate delay does not scale linearly with frequency, less gates are available in a clock cycle time, as frequency rises. The probability of failure goes up because the clock cycle time $T(=\mu_C)$ shrinks faster than ϵ (the settling window). To compare the failure probabilities, we assume the following model: The metastability window ϵ width is assumed to be equal to a gate delay, the parameter τ to be 1/3 of a gate delay, and the jitter (which equals 6σ) to be half a gate delay (and no more than 15% of the clock cycle). Figure 6-11 presents a logarithmic graph comparing the synchronization failure probabilities of a synchronizer relative to the A/S scheme. The graphs of the probability of failure for the synchronizer and A/S were calculated according to the assumed model for various gate delays, and the lines show the trend, as explained below.

For high communication bandwidth (e.g., almost every cycle), the mean time between failures (MTBF) is given by

(6-5) $$\mathbf{MTBF} = \frac{1}{P_{failure} * f_c}$$

The failure probabilities required to achieve an MTBF of once a year and once a minute at the various technologies are also presented in the graph. As can be observed from the graph, using a synchronizer can be practical for lower frequencies, but as clock frequency increases, the synchronizer has less time to resolve and the probability of failure rises rapidly. Using a sequence of synchronizers decreases the failure probability, but increases the latency and affects performance. Note also that the failure probability presented is of a single synchronizer, and since many synchronizers are required (for every bit in every bus between modules), the failure probability is worse than drawn on the graph. When synchronizers fail to deliver a flawless operation at higher frequencies, A/S still applies. The zero values of A/S failure probability cannot be plotted on the logarithmic graph. The inter-module clock jitter will be the limiting factor on maximum clock frequency in A/S scheme. At even higher frequencies, when A/S fails, it can be used together with a synchronizer, to decrease the probability of the synchronizer entering a metastable state. Beyond a certain technology (e.g., when the jitter is more than 15% of the clock cycle), all synchronization methods fail, and the only solution is to use a complete asynchronous design, with asynchronous communication, as described in other chapters of this thesis.

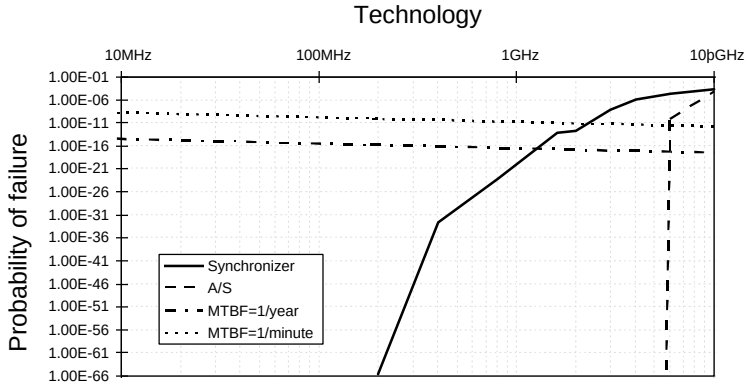


Figure 6-11: Probability of synchronization failure.

6.6 Concluding Remarks

The architecture of *Kin* (described in Ch. 2) and its support of Avid execution (detailed in Ch. 3) implies a large and complex microprocessor. A single clock is either impractical or impossible for such very high performance chips, e.g., as predicted by the SIA technology roadmap for the year 2010 (over 0.5B transistors operating at over 1GHz clock) [SIA94]. We have presented an adaptive synchronization solution for multi-synchronous systems. Multi-synchronous architectures (locally synchronous, globally asynchronous) could be a viable alternative to a fully asynchronous design. We focus on common clock multi-synchronous systems, where a single crystal clock is

distributed over minimal area, minimal power networks, so that all modules operate on the same clock frequency (or its frequency divided versions) but at unknown phase differences.

We take advantage of the stationary nature of clock and data delays, and of the consequential non-uniform arrival time distribution of asynchronous signals. Data timing is dynamically adjusted to avoid clock/data conflicts.

We have presented a novel adaptive method addressing the synchronization problem. While most previously proposed methods manipulate the clocks, adaptive synchronization adjusts data delays. The method exploits the high stability of delays and the stationarity of most relative phases. The probability of synchronization failures is reduced substantially. Timing adaptation can be limited to special training sessions (as commonly practiced in data communication networks). Thus, the synchronization monitoring circuits are kept off the critical paths. The adaptation circuits incur only marginal overhead in area, power and performance. A study of alternative methods (such as synchronizers and stretchable clocks) shows that they may not be as usable as adaptive synchronization.

The solution presented in this chapter was developed as a possible implementation methodology for *Kin*, but it can also stand by itself and be used in any other systems.

Chapter 7 : Adapting Statecharts Methodology for Asynchronous Design

This chapter describes how to apply a novel methodology, based on statecharts, to the design of large scale asynchronous systems. The design is specified at multiple levels, simulated, animated, and compiled into synthesizable VHDL code by using the Statemate Magnum CAD tool . We add a validation sub-system to check correct operation. Statemate Magnum is originally synchronous, but we employ it for asynchronous design by avoiding any design dependence on the clock, and simulating with fast clock and on-line delays. We have used statecharts to specify and simulate several systems, large and small, including the asynchronous instruction length decoder (described in Ch. 4), the doubly-latched asynchronous pipelines from Ch. 5, and *Kin* model (as explained in Ch. 2). The methodology is demonstrated here through a simple FSM design example.

7.1 Introduction

Numerous applicable methodologies have been developed for the design of asynchronous logic [Hau95]. Most of those methodologies and tools were developed for the design of small systems. Only a few tools and methodologies have addressed large scale system level design. They include the CHP [Mar90] and Tangram [vBKR+91] compilers, the combination of commercial and special tools for the PostOffice [CDS93] and AMULET [Pav94]. However, no single and complete methodology and tool set have been demonstrated as yet for the design of large scale asynchronous systems.

The research described in this thesis focuses on the architecture and design methodology of a microprocessor, which is a large scale asynchronous system. We were looking for one complete CAD system, based on a well-understood methodology, suitable for all levels of the system, for all timing disciplines, and for all design tasks including specification, design, simulation, animation, validation, verification, debugging and synthesis. Naturally, since the CAD system is the tool rather than the research, and since such a grand CAD system requires immense resources to develop and maintain, we have turned to the domain of commercial CAD products in our quest. Unfortunately, no large scale commercial CAD systems are available for asynchronous design.

Thus, we employed a commercial synchronous CAD system and adapted it for the design of

asynchronous circuits. A design discipline is developed by which any explicit dependence on the clock is carefully avoided. The circuit is synthesized by the tool into a synchronous structure, but it is subsequently converted into an asynchronous one, as described in Sect. 5.7.

This chapter describes the adaption of the novel commercial high-level CAD system Statemate MAGNUM™ [iLo96] to the design of large scale asynchronous systems. Statemate MAGNUM (*Magnum* for short) is based on statecharts [Har87], and is introduced succinctly in Sect. 7.2. *Magnum* provides an environment for a hierarchical graphical specification of the design, and also facilities for simulation, animation, verification, and compilation into either a software program (in C or Ada) or a hardware description (in VHDL or Verilog, at either the behavioral or RTL levels).

The application of *Magnum* is described through the design of a small quasi-delay insensitive finite state machine (qDI FSM) [DGY92b]. Section 7.3 defines the FSM and Sect. 7.4 explains its design with *Magnum*. In Sect. 7.5 we describe special validation statecharts for asynchronous logic. Simulation is discussed in Sect. 7.6.

7.2 The Statechart-based Statemate MAGNUM CAD System

Statecharts [Har87] constitute a specification formalism [HP96b, HPS+87] for reactive systems, and can be used to design complex discrete-event systems and communication protocols. Asynchronous systems can be considered as reactive systems, since data are transferred by using handshake protocols and each module reacts to changes in its inputs, does some processing, and signals to other modules when done. Statecharts extend conventional state-transition diagrams of finite state machines by adding hierarchy, concurrency and communications. While statecharts describe system behavior, structural, functional and data-flow aspects of the system are specified with the related *activity charts*. The nature of these charts is demonstrated in Sect. 7.4 wherein they are applied to the design of qDI FSM.

Historically, statecharts were applied to the design of real-time, reactive software systems. Recently the same methodology has also been applied to hardware specification. *Magnum* presently compiles the specification into either VHDL or Verilog. It can compile into either behavioral or RTL styles, and it can target the code for various commercial synthesizers and simulators. We have successfully employed the Compass RTL synthesizer on the Synopsys-targeted VHDL.

The applicability of tools like *Magnum* to asynchronous design is far from obvious. *Magnum* is inherently a tool for designing synchronous logic. It generates RTL VHDL that is commonly

synthesized according to the ‹register-and-cloud' model [Per94] where clouds of combinational logic are interconnected through synchronous registers. *Magnum* employs a unit delay model, and simulates the design by advancing a clock. To bypass these difficulties, we have defined a design and simulation discipline, as described in Sects. 7.4 and 7.6. We also investigated post-synthesis robust conversion algorithms that convert the generated synchronous circuit into a legal asynchronous circuit, as discussed in Sect. 5.7.

7.3 The qDI FSM

The qDI FSM [DGY92b] consists of a combinational logic block (CL, Fig. 7-1) and a register (REG). It is based on the dual rail, four phase design methodology. The inputs and outputs are either *defined*, or *undefined*, or in transition between those states. The ACK line is a single rail control line. The system/environment protocol is shown in Fig. 7-2. The corresponding protocols for the CL [DGY92a] and the REG components are shown in Figs. 7-3 and 7-4, respectively (note their duality). This FSM is the asynchronous version of the synchronous Mealy machine (the outputs are defined only when the inputs are).

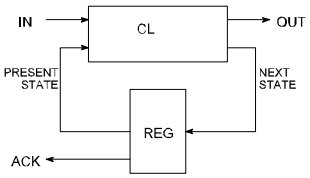


Figure 7-1: qDI FSM.

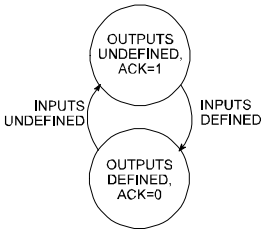


Figure 7-2: FSM handshake.

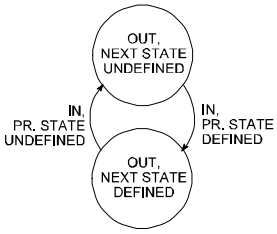


Figure 7-3: CL handshake.

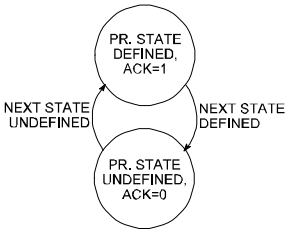


Figure 7-4: REG handshake.

7.4 Specifying the qDI FSM with Statecharts

To demonstrate the use of *Magnum* consider a two inputs, two outputs FSM. Figure 7-5 shows a simplified activity chart of the qDI FSM (the validation parts are explained in Sect. 7.5 below). Activities are drawn as boxes, with CL and REG nested inside the qDI_FSM activity (they are hierarchically described in separate activity charts, as indicated by '@'). The environment ENV, being an external activity, is drawn as dotted boxes, and can appear multiple times for clarity. Besides providing inputs and outputs, ENV also supplies circuit delays for simulation (which can validate delay insensitivity).

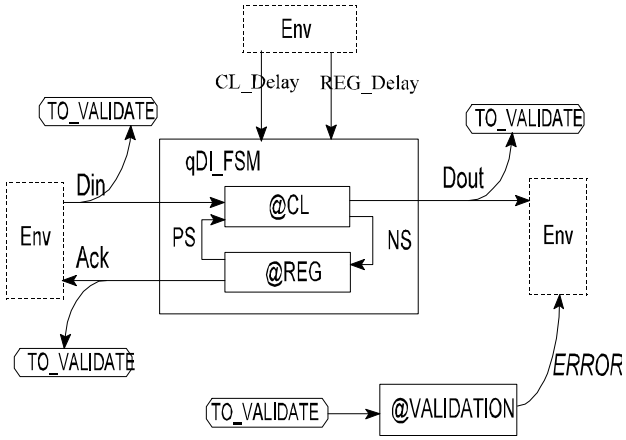


Figure 7-5: Activity chart, FSM with validation.

The detailed hierarchical description of the CL and REG activity charts can be found in [KGS96]. Two alternative CL statecharts are shown in Figs. 7-6 and 7-7 (note that states are drawn with round corners). The former is designed for implementation and simulation with bounded delays, and the latter assumes (quasi-) delay insensitive operation.

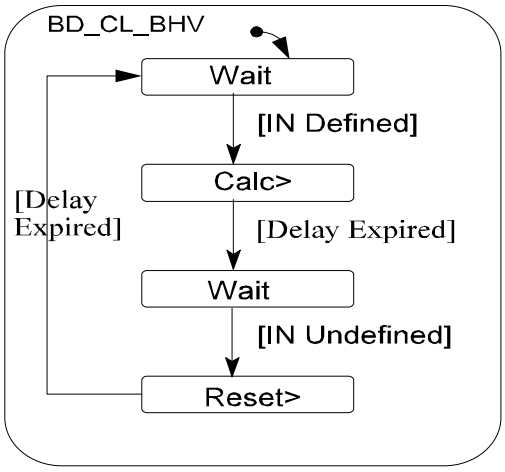


Figure 7-6: CL Bounded Delay statechart.

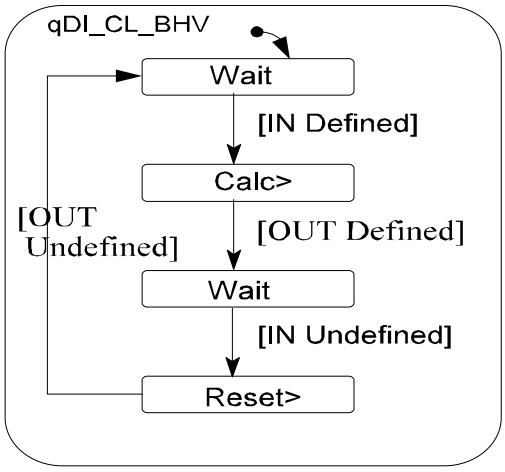


Figure 7-7: CL qDI statechart.

Statechart transitions are labeled *trigger/action*, where *trigger* is either an *event* and/or a *[condition]*. All parts are optional in the synchronous *Magnum*. Unlabeled transitions are triggered by the next clock in synchronous systems, so they are ruled out in our case. *Magnum* events, unlike the common concept of events in asynchronous design, are dangerous, as they can go unnoticed and disappear unless some receiver is ready and waiting for them. Since *Magnum*

has no inherent countermeasure for this problem, we require that regular statechart transitions are labeled with conditions instead (we do allow the use of some special events in Sect. 7.5 below). Asynchronous signal transition events (such as *'X↑')* are replaced by conditions (such as *'[X=1]')*.

Initial state is marked with the default entry arrow. The CL starts with inputs and outputs undefined. When all CL inputs (i.e., the inputs to the FSM, and the present state (PS) from the registers) become defined, the statechart moves to the next state wherein the Boolean functions are evaluated. The actual Boolean expressions are hidden in the data dictionary (as indicated by '>'). In Fig. 7-6, exit from that state occurs when the computation delay expires. Alternatively, in qDI implementation (Fig. 7-7), exit from the same state depends on CL outputs becoming defined, rather than on the delay, as appropriate for a delay insensitive circuit. A similar sequence occurs on the ‹return-to-zero' part. The statechart of the registers is presented in [KGS96].

This FSM design is generic by nature, and only a small part of it is affected by the details of the specific FSM being designed.

7.5 Validation

As with any asynchronous methodology, the design of qDI circuits depends on assumptions about correct operation by both the circuit and its environment. Those assumptions can be proven correct through a formal verification framework. However, in many cases this is an elusive goal, so we resort to on-line *validation*, namely continuous checking that all assumed properties are never violated. Validation may be limited to the design, simulation, and debugging phases, after which all validation sub-systems are removed from the design. Alternatively, all or part of the validation sub-system may be retained during synthesis, so that they continue to function through the lifetime of the hardware. This latter safety measure may be valuable especially for the interfaces between the circuit and the external world.

In the qDI FSM, we wish to validate the following properties: (a) The inputs and outputs must carry only legal dual-rail values {*0,1,undefined*} ≡ {01,10,00}; (b) Changes of inputs and outputs must be monotonic (from all-undefined to all-defined, without some lines becoming defined and then returning to undefined, etc.); (c) The handshake protocols of Figs. 7-2 - 7-4 are adhered to.

The activity chart of the qDI FSM with validators is shown in Fig. 7-5. The validation statechart (Fig. 7-8) demonstrates hierarchy and concurrency — all enclosed statecharts, separated by the dotted lines, are active concurrently. They continuously monitor the relevant signals of the FSM. The four instances of the generic DR_STATUS (Fig. 7-9) perform two tasks: First, while in

super-state DR_NORMAL_OP, if both wires are '1,' the chart exits immediately to the DR_MALFUNCTION terminal state and issues the ERROR event. Incidentally, this is one of the few cases wherein we do use *Magnum* events, since another statechart is continuously watching for that event (see below). ERROR event can be designed to ring some bells. Note that the transition into DR_MALFUNCTION exits from the external boundary of DR_NORMAL_OP, meaning that this transition takes place regardless of which internal state has been active at the time of error. The second task is to detect the direction of the transitions on the lines ('up' or 'down'), for use by the monotonic validator.

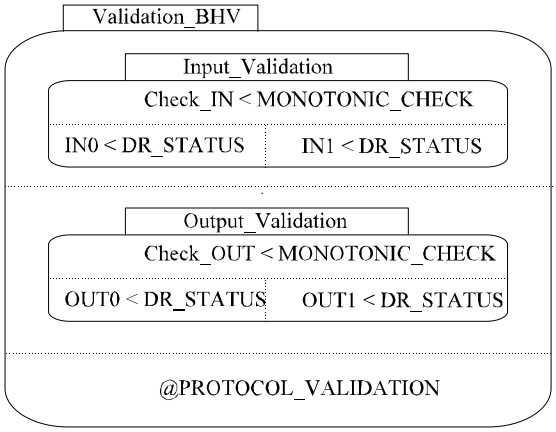


Figure 7-8: Validation statechart.

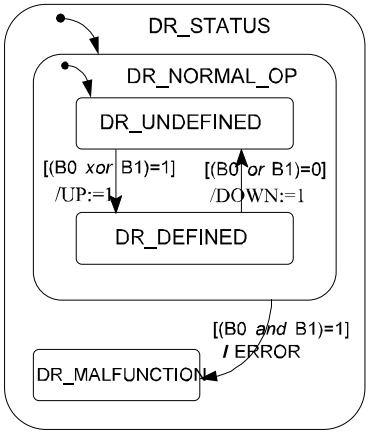


Figure 7-9: Dual-rail validation statechart.

Monotonicity is validated by the regular expression $[\mathit{Undef}\,[\mathit{up}]^{+}\,\mathit{Def}\,[\mathit{down}]^{+}\,]^{*}$, as described in Fig. 7-10. The MONOTONIC_CHECK statechart monitors the DR_STATUS statecharts, watching for non-monotonic transitions. For an *N*-bit dual rail line, two *N*-bit vectors are defined, VUP and VDOWN, consisting of the corresponding UP and DOWN flags that are set by the dual rail validators. When at least one of them marks its UP flag, the MONOTONIC_CHECK validator moves to the UP_GOING state. It stays there as long as not all DR_STATUS validators have set their UP flags. If any of the lines returns to undefined value, its DOWN flag is set and the monotonic validator immediately escapes to the NON_MONOTONIC terminal state, and generates an ERROR event. Upon normal transition to VALUE_DEFINED (after all *N* bits are set) the VUP vector is reset. A similar process takes place on the way down. Note that while input validation checks the environment, output validation checks our own system.

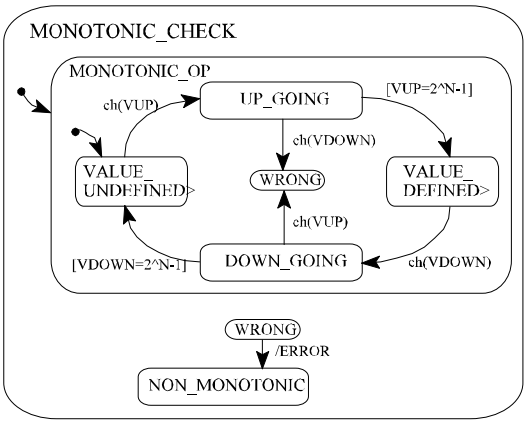


Figure 7-10: Monotonic validation statechart.

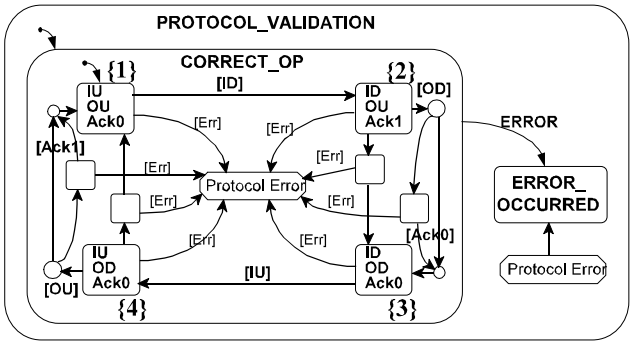


Figure 7-11: Protocol validation statechart.

The PROTOCOL_VALIDATION statechart (Fig. 7-11) checks adherence to the four phase protocol, of both the environment and the system. Any deviation is considered an error, which results in immediate escape to the terminal state ERROR_OCCURRED. Note how the chart contains all the intermediate transitional states, as well as the stable ones {1-4}. The protocol validator also watches for ERROR events generated elsewhere. Upon such an event, the validator jumps to the error state, and can alert the user. For clarity reasons, the statechart in Fig. 7-11 is somewhat simplified, and does not contain all the state transition details. A full description (with all the conditions for each transition) can be found in [KGS96].

While validation relates to dynamic checking of properties, and the checks are limited to the cases that are actually simulated, verification usually involves exhaustive checks or proofs that guarantee that certain properties are always true. Verification aspects of statecharts have been discussed by [Day92].

7.6 Simulation

The simulation facilities of *Magnum* include features such as graphic feedback in color, animation, links to external code, and multiple manners of affecting the simulation and observing the results, either interactively or in batch mode. However, the interesting question is how to perform asynchronous simulations with a synchronous simulator.

The most practical solution we have found is to simulate the design with an extremely fast clock (relative to the circuit delays). Even when all conditions are set for a transition to take place in a statechart, the (synchronous) simulator would not make the transition unless the clock has ticked. Having that clock toggle very fast results in transitions happening almost immediately after

they are enabled. Again, this approach relies heavily on a strict discipline assuring no hidden synchronous transitions in the design, and avoiding all functional dependencies on the clock.

Another issue is the generation and simulation of delays. For bounded delay and timed disciplines it is useful to simulate the design having delays varying within the (lower and upper) bounds set for them. For qDI disciplines, it is useful to simulate with arbitrary delays. Most important, the circuit should be simulated at many different 'timing corners,' namely different combinations wherein some parts of the circuit are very fast and other parts are very slow, in order to detect illegal behavior. The solution to all these requirements lies in *on-line delays*.

On-line delays are specified neither in the design nor in the simulator. Rather, they are supplied to the simulator, on demand in real-time, as the simulator executes. Every time a delay is needed, it is re-read from an external, concurrently executing procedure, the *delay generator*. The generator can be programmed to either use fixed delays, draw them from a table, draw them from a predefined statistical distribution, or execute special functions to create data-dependent delays. Each of those delay models is important for some aspects of asynchronous design.

7.7 Concluding Remarks

We have presented the applicability of a complete CAD system, which was originally developed for the design of synchronous circuits, to asynchronous logic design. *Magnum* is based on statecharts, a suitable methodology for the design of asynchronous architectures. Specific design rules have been developed for the specification (no dependence on clock transitions, use of conditions rather than events), and for simulating asynchronous circuits on the synchronous simulator (use extremely fast clock, apply on-line delays). Other aspects of *Magnum*, such as static and dynamic verification, animation, and compilation are discussed in [KGS96] and [iLo96]. Validation statecharts have been added to monitor the simulations, and they can also be synthesized into validating hardware. They provide an important complement of formal verification for large and complex systems.

Simulations were used for debugging while developing the models and also for performance evaluation (using online delays generation). As explained in Chs. 2 and 3, a model for *Kin* was built based on statecharts and external C-code. Statechart models of various asynchronous pipelines (cf. DLAP in Ch. 5) were also built to analyze their operation before implemented at transistors level. The asynchronous instruction length decoder from Ch. 4 was fully described by statecharts, for complete architecture specification and module interface.

Chapter 8 : Summary and Further Research

Analysis of future semiconductor technology (such as 1 billion transistors per chip and over 1GHz clocks planned for the year 2010) shows that it places severe constraints on the design of high performance microprocessors. In particular, the chip is too large and the clock is too fast for single clock synchronous operation. Rather, new forms of distributed architectures and asynchronous interconnects are called for. In this thesis it is argued that the technological constraints necessarily lead to asynchronous solutions.

This research describes the architecture of the asynchronous microprocessor *Kin*, which supports a novel aggressive speculative *Avid* execution method (necessary for high speed and suitable for asynchronous processors). The development of *Kin* included addressing and solving problems at the architecture level, as well as developing architectural concepts and design methodologies for the required building-blocks. Using some of these basic blocks is not restricted to the design of a microprocessor, and they can be applied to other systems as well.

The thesis also discusses a number of associated issues: Fully asynchronous design of one module (an asynchronous instruction length decoder), algorithmic conversion of synchronous pipelines into (doubly-latched) asynchronous ones (DLAP), mixed timed globally asynchronous locally synchronous systems (with adaptive synchronization), and the design methodology (based on statecharts) suitable for high level asynchronous design.

Some future research directions can be identified. They relate either to the near-term evolutionary path from present day CAD and architecture towards asynchronous systems, or to the longer term issues of what's beyond *Kin*.

Regarding the near term, substantial research must be devoted to transitional methodologies described in the thesis:

- Automatic conversion of synchronous circuits into asynchronous ones should be further investigated.
- Adaptive synchronization for multi-sync systems may be studied, together with clocking methodologies.
- Synthesis tools must be adopted to produce and verify GALS systems.
- Seamless interfaces of various low level synchronous synthesis tools to high level specification tool like Statecharts should be developed.

Further, the architectural aspects need additional study:

- *Kin* architecture can be refined. For instance, alternative organizations of the reservation stations and instruction schedulers can be simulated and compared.
- More Avid execution simulations should be performed and analyzed. We have only simulated a constant Avid scheme, but Avid performs best when used as an adaptive mechanism, where the Avid depth is adjusted according to the quality of the prediction of each branch.
- We have presented the fully asynchronous design of an instruction length decoder for *Kin*. Other modules can be implemented according to various asynchronous methodologies.
- DLAP design can be refined, and two-phase DLAP schemes may be researched.

On the longer term:

- The technological and architectural constraints that may be posed by technologies beyond the 2010 SIA prediction should be investigated.
- More aggressive deviations from contemporary architectures should be considered. In particular, both the instruction sets and the computing paradigms most suitable for asynchronous processing must be identified.

Last but not least, *Kin* could be implemented in silicon!

Appendix A : Clock Coordination for Synchronization of Multi-Synchronous Systems

This appendix describes the *adaptive coordinated synchronization* for effective communications among non-synchronized synchronous blocks, based on adapting the clock. When a collision between the incoming data and clock is detected, the phase of the clock is changed and adjusted to avoid the conflict. Unlike stretchable clocks which use local clock generators, the schemes presented here are designed for multi-synchronous systems operated by an external crystal clock, thus guaranteeing a nominal frequency. We propose three alternative clock coordination methods (clock-disable, clock-toggle, and clock-shift), and analyze their effect on system performance and their limitations.

A.1 Introduction

In clock coordination schemes, we take advantage of the non-uniform arrival time distribution (explained in Ch. 6) and adapt the clock of the receiver to switch when incoming data is stable. Incoming data and clock are processed by a clock/data coordinator prior to their entry into the synchronous module, as in Fig. A-1. The purpose of the coordination circuit is to avoid clock/data collision, by modifying the clock. Unlike symmetric arbitration, the coordination circuit gives higher priority to the data input and is designed to delay the clock in cases of conflict. A proper delay, matching the delay of the coordination circuit, is added on the data lines, to preserve the relative timing of *Data* and *DataRdy*.

We consider three alternative methods of coordination (Fig. A-2). The *clock-disable* method disables the local clock for a complete cycle. The *clock-toggle* method toggles the clock, delaying the rising edge for a one half cycle. The *clock-shift* method adds a small delay to the clock, i.e., it shifts the phase of the local clock for all future cycles.

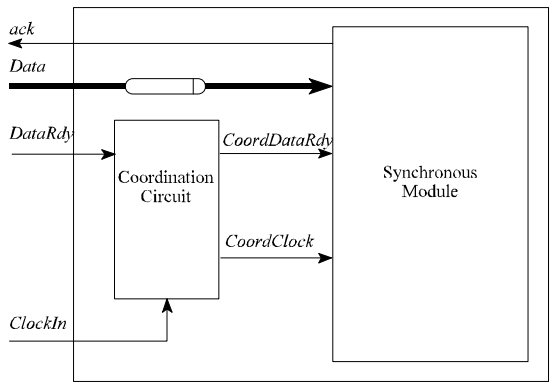


Figure A-1: A clock/data coordination circuit for each synchronous module.

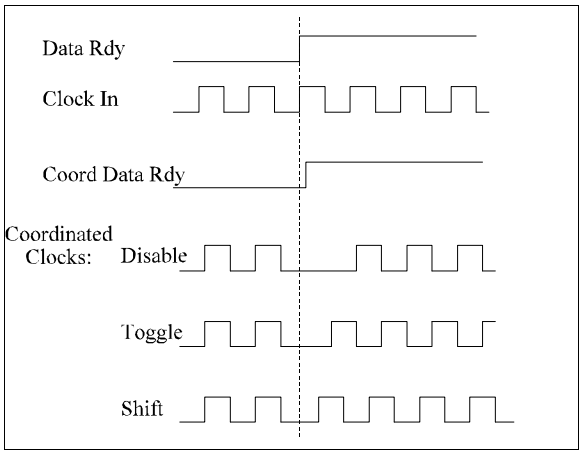


Figure A-2: Alternative coordination methods.

A.2 Clock Coordination Circuit

For all three methods, the coordination circuit comprises three components (Fig. A-3): The Conflict Detection (CD) block detects when data and clock edges have occurred simultaneously. The Clock Phase Delay (CPD) block delays the clock by a selectable amount of time. Its actual function varies from one method to another (full cycle, half cycle, or a variable delay). The Clock Regeneration (CR) block recovers the duty cycle of the input clock. Upon an edge conflict, CD generates $\overline{hold}$ and *SelectNext*, to keep *CoordClock* in its low phase, and to change the delay, respectively.

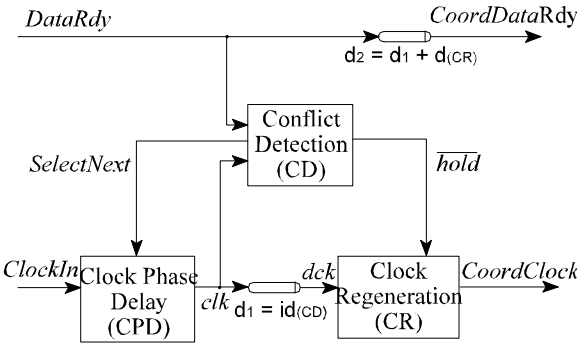


Figure A-3: Coordination circuit block diagram.

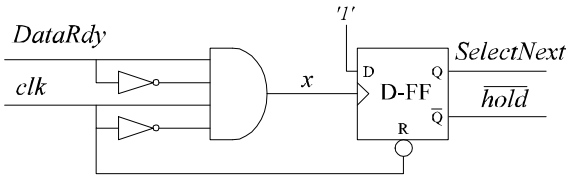


Figure A-4: Clock/data edge Conflict Detection.

Inertial delay line d_1 matches the delay of CD (as explained below). Thus, upon edge conflict, CR is disabled sufficiently early to block the faulty clock transition. Delay d_2 matches d_1+delay(CR), to preserve the relative timing of the clock and data.

When *SelectNext* is on, CPD *clk* output is first reset (also signaling CD to reset and release $\overline{hold}$), the clock 'off' phase is stretched as much as necessary, and *clk* is set again to signal the beginning of the next valid 'on' phase.

In this discussion we assume that *DataRdy* rises to '1' after the new data are available and ready to be sampled. A simplified CD implementation is proposed in Fig. A-4. When both inputs to CD rise simultaneously, a pulse is generated at *x*, setting the D-FF (used as a SR latch with an edge-triggered 'set' input), and generating $\overline{hold}$ and *SelectNext*. The latch is cleared when the *clk* input turns low. Possible metastability of the latch is discussed in Sect. A.4.1 below.

In addition to matching CD delay, the positive inertial delay d_1 (implemented by the circuit in Fig. A-5) also filters short positive *clk* pulses. Such pulses are created on clock/data conflicts, when *clk* is reset shortly after it has risen (Figs. A-10, A-12). Delay d_1 is asymmetric [Sei80]: the positive delay is longer (by Δ) than the negative delay. To preserve the duty cycle of the input clock, a matching negative inertial delay is appended to CR (as described in Fig. A-6).

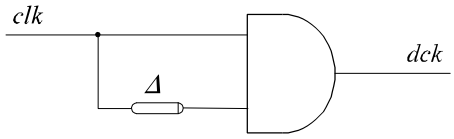


Figure A-5: Positive inertial delay d_1.

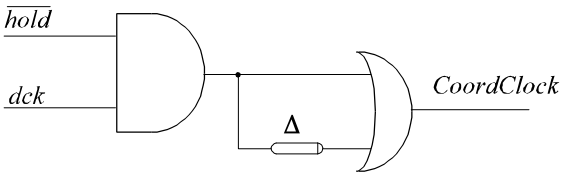


Figure A-6: Clock Regeneration with a negative inertial delay.

The content of the CPD block depends on the coordination method implemented (see Figs. A-7, A-9, A-11). In the clock-disable case CPD is a wire. For clock-toggle it selects either *ClockIn* or its inverse. A multiplicity of delayed clock signals for the clock-shift method are generated by a tapped delay line, and the proper phase is selected by a sequential selector, which switches inputs on every rising edge of its control input. Typical waveforms of the three coordination circuits are shown in Figs. A-8, A-10, A-12, wherein conflicts are marked by dotted lines.

Note that, for very high clock frequencies, the total latency of the coordination circuit (d_2) may approach, or even exceed, a whole clock cycle. While resembling the latency incurred by a synchronizer, coordination is still preferred to the latter. Since sender and receiver clocks are correlated in multi-synchronous systems, clock/data conflicts are highly likely to recur (and cause recurring metastability situations) if the relative clock phase remains fixed. Coordination circuits are designed to adapt local clocks and thus counter this problem.

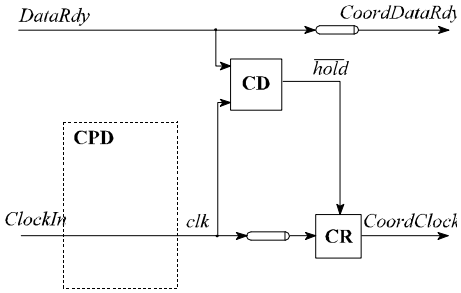


Figure A-7: Clock Disable.

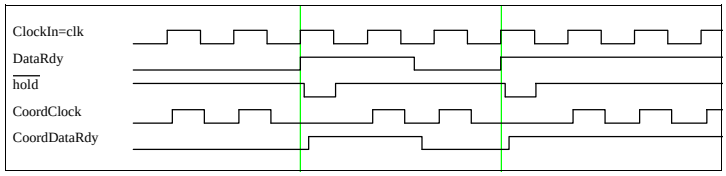


Figure A-8: Waveforms of Clock-Disable method.

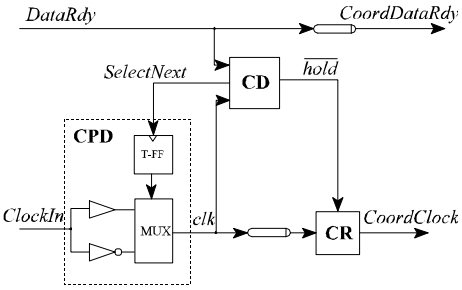


Figure A-9: Clock Toggle.

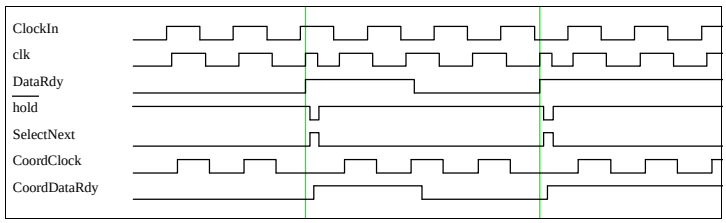


Figure A-10: Waveforms of Clock-Toggle method.

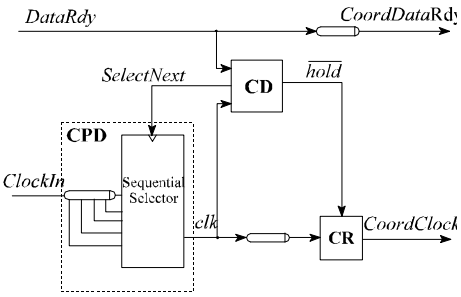


Figure A-11: Clock Shift.

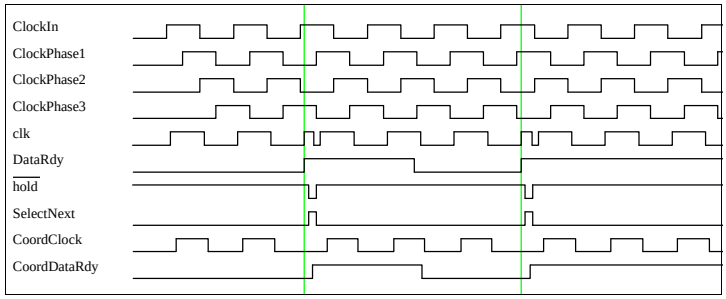


Figure A-12: Waveforms of Clock-Shift method.

A.3 Performance Analysis of Clock Coordination

The Clock-Disable method is the simplest one to implement, but its obvious limitation is that it may cause a substantial performance degradation, and the module is doomed to suffer from the same problem repeatedly. When a clock/data edge conflict is detected, the local clock is disabled

for one cycle. We assume that the data arrival time is uniformly distributed over the clock cycle. Although we have claimed above that this distribution is non-uniform, it is valid to assume uniformity when the expected relative arrival time is yet unknown, or when it has changed, e.g., due to temperature or supply voltage drifts. Thus, the probability of conflict is

(A-1) $$p = P(conflict) = \frac{t_{su} + t_h}{C}$$

where t_{su} and t_h are the setup and hold times, respectively, and C is the clock cycle. If data arrive every R cycles, then a computation which takes R cycles if there were no conflicts would need an extra cycle with probability p due to conflicts. Thus, the slow-down S_R is

(A-2) $$S_R(Clock-Disable) = \frac{Num.\ cycles\ due\ to\ conflict}{Num.\ cycles\ without\ conflict} = \frac{R+p}{R} = 1+\frac{p}{R}$$

If the incoming data arrive from N different sources, and all the bits arriving over the same bus are synchronized, then the probability of conflict is

(A-3) $$P(c,N) = 1-P(\neg c,N) = 1-P(\neg c,1)^N = 1-[1-P(c,1)]^N = 1-(1-p)^N$$

The slow down in such a case is

(A-4) $$S_N(Clock-Disable) = 1+\frac{P(c,N)}{R} = 1+\frac{1-(1-p)^N}{R}$$

According to the Clock-Toggle method, the clock is delayed one half cycle upon a conflict. Hence:

(A-5) $$S_R(Clock-Toggle) = \frac{R+\frac{1}{2}p}{R} = 1+\frac{p}{2R}$$

(A-6) $$S_N(Clock-Toggle) = 1+\frac{P(c,N)}{2R} = 1+\frac{1-(1-p)^N}{2R}$$

For the Clock-Shift method, we estimate the shift resolution to be 3 gate delays. If δ=(shift resolution delay)/C, then

(A-7) $$S_R(Clock-Shift) = \frac{R+\delta p}{R} = 1+\frac{\delta p}{R}$$

(A-8) $$S_N(Clock-Shift) = 1+\frac{\delta P(c,N)}{R} = 1+\frac{\delta[1-(1-p)^N]}{R}$$

Figure A-13 shows the slow-down (S_N) of the three coordination methods relative to R (assuming p=0.05, N=10, and if $C \approx 15$ gate delays, then δ=0.2). While S_N*(Clock-Disable)*$\rightarrow$*1.04* for large R, it grows to 1.2-1.4 for the higher bandwidth case (R=2 or R=1, respectively). The slow-down in case of clock-toggle is only 1.2 for R=1, and diminishes to 1.08 in the clock-shift case. The clock-shift method causes the least performance degradation: Only 8% slow-down for high rate data transfer (i.e., every cycle), and as low as 0.8% for low rate. This performance analysis underestimates the performance of the clock adjusting approach, since after the clock phase is adjusted, it is likely to guarantee the successful synchronization of subsequent data transmissions. After adaptation, the probability is not uniform any longer and hence p is substantially smaller.

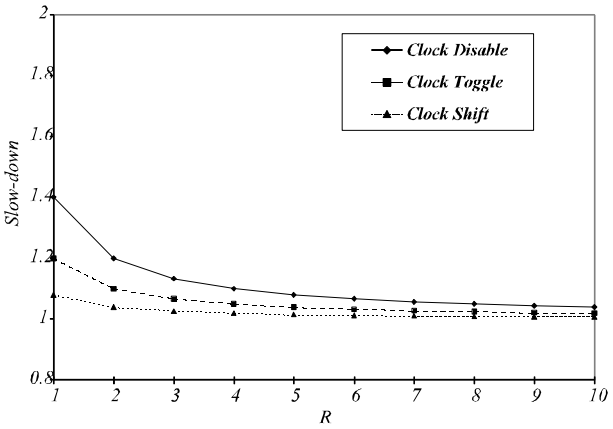


Figure A-13: Clock coordination slow-down.

A.4 Limits of Clock Coordinated Synchronization

Clock coordination suffers from three difficulties, namely metastability, oscillation, and lack of convergence. In this section we explain the problems and show how some of them can be avoided in practice.

A.4.1 Metastability

When a clock/data conflict is detected, a pulse is generated in CD (x, Fig. A-4) to set the latch, disable the clock, and shift the phase. The width ω of the pulse depends on the time difference Δt between the rising edges of *clk* and *DataRdy* (Fig. A-14). Maximal ω is achieved when *clk* and *DataRdy* rise simultaneously; the longer $|\Delta t|$ is, the shorter the pulse. For certain high values of Δt (In the τ regions of Fig. A-14(b)), ω is dangerously short and the pulse might cause the latch

to either enter a metastable state, or to take abnormally long time to settle [Mar81, Sto82]. Note that CD is designed such that if $\Delta t \in \tau$, data and clock are safely separated, non-conflict operation is desired, and CD should not declare a conflict. Long propagation delays do not affect the circuit. If the latch enters a metastable state but does not exit it before *clk* goes low, the latch will be forced out of metastability by *clk* (Fig. A-4). However, if the latch exits metastability while *clk* is still high, and (erroneously) declares a conflict, the clock is aborted in mid-cycle and may cause the module to fail. Normally, the CD has enough time to resolve the metastable state since the *DataRdy* will not change until an acknowledge is returned. This most likely will take more than one cycle, depending on the handshake protocol and the processing time.

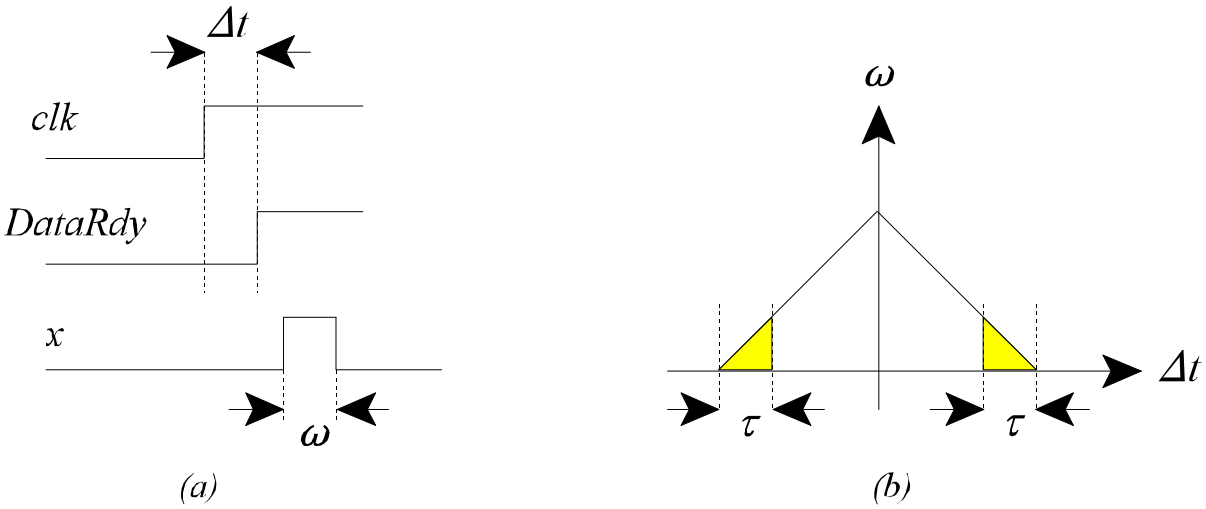


Figure A-14: Conflict detection pulse width ω as a function of *clk/DataRdy* concurrency Δt.

The adaptive synchronization brings the system, after a convergence period, to a stationary state. As explained above, the final state is stationary because the delays and phase differences among the modules in the system are functions of the implementation, of physical parameters, and of temperature and supply voltages, and they typically change very slowly during the operation. After adjusting the clock phase in all modules to avoid conflicts, there is no need to make any more changes and the system is expected to operate without synchronization problems for a long time. Thus, adaptive synchronization is best used during special *training sessions*.

A training session is dedicated to adaptation of clock delays, and does not perform real computations. All the modules intercommunicate heavily in order to entice the coordination circuits to shift phases as needed. Synchronization failures during training sessions are ignored (no real data is lost). After a training session, clock/data conflicts are less likely. The system should be trained upon reset and periodically (but infrequently) during operation. Training is expected

to affect system utilization by less than 1% (as explained in Ch. 6). Each module should send messages on all its output channels during training, so that all potential conflicts are exercised.

A.4.2 Oscillation

When many modules communicate by clock-shift coordinated synchronization, an oscillation may develop. Each module shifts its own clock phase every time a clock/data conflict is detected, and subsequently sends its data to other modules using the newly shifted clock. This change of clock phase might cause another module to develop a conflict, and its own local clock is consequently shifted. The propagating clock shifts may settle down after a few adjustments, or they may cycle and affect the first module again. Thus, the clock phases may continuously be updated and shifted forever. In the following we show that the oscillations of the adaptation process problem may be predicted and avoided at design time.

The system can be described as a graph, wherein modules are modeled by vertices, and communication channels are modeled by directed edges. The danger of oscillating clock shifts can only occur due to cycles in the graph, and only in a very special case. In Fig. A-15, a cycle of three communicating modules is shown.

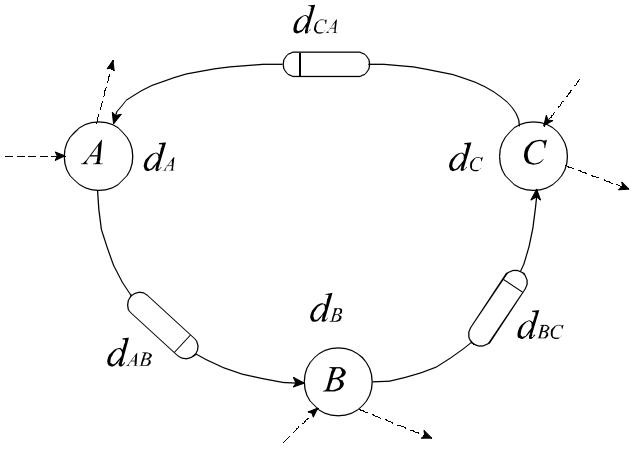


Figure A-15: Communicating modules cycle.

Let d_A be the delay in module *A* from its local (coordinated) clock edge till the data are ready at its output. Let d_{AB} be the delay along the communication channel from module *A* to module *B*. When Module A sends data to module B, they would arrive at module B after $d_A + d_{AB}$ time (relative to some transition of the coordinated clock at module *A*). If a clock/data conflict occurs now in module B, it will update its local clock phase by the shift resolution delay δ, and will send

data to module C using this new clock. The data will arrive at module C (d_B+d_{BC}) after the new clock edge of module B. Those data may cause module C to shift its local clock by δ. The data sent from module C to module A will take (d_C+d_{CA}) to arrive. The total time required for data to be transferred from module A, to B, to C, and back to A (including clock shifts at B and C), is $(d_A+d_{AB}+\delta+d_B+d_{BC}+\delta+d_C+d_{CA})$. To cause a conflict and clock shift in module A (i.e., an oscillation), this sum of delays along the cycle has to be equal to an integral multiple of the clock cycle (i.e., *mC*). In the general case, when there are *k* modules along the cycle, the danger of oscillation exists when:

(A-9) $$(\sum_{k-cycle} (d_i + d_{ij})) + (k-1)\delta \cong mC$$

where module *j* follows module *i* along the directed *k* cycle.

To avoid the oscillation problem, the graph model of the system architecture should be searched (at the design time) for cycles with this property. A small delay is added to each such cycle to avoid the situation. This approach is limited, however, by the accuracy of delay modeling. Alternatively, the coordination circuits may be redesigned to add varying delays to the clocks, so as to break cycles.

The same oscillation problem is also expected to occur when stoppable clocks are applied. Since each module changes its own clock phase, it might cause others to adjust accordingly and vice versa. Data adaptive synchronization (Sect. 6.2) breaks the dependency cycle and avoids oscillation problems.

A.4.3 No Convergence

When a module is interconnected to many other modules, and each of the other modules has a different relative clock phase shift, it is possible that they will not be able to communicate without failure, since the local clock will be shifted constantly. This might happen when there is no common phase they all agree on, and the combined distribution of data arrival times on the many inputs is uniform (Fig. 6-1). Stretchable or pausable clocks solutions cannot be used in this case since the clock will be kept in its ‘off’ phase for arbitrarily long periods of time. This can be avoided by adaptively coordinate delays on the data lines (as described in Section 6.2) rather than change the local clock.

A.5 Concluding Remarks

Alternative methods and circuits for clock coordination are described and performance is analyzed. An external common clock is used, and based on the stationary nature of clock and data delays, a proper phase of the clock is dynamically selected to avoid synchronization failures. Metastability, oscillation, and convergence problems are discussed. As done for the data coordination, the clock adaptation can be limited to special training sessions.

References

[AML97] S. S. Appleton, S. V. Morton, and M. J. Liebelt, "Two-Phase Asynchronous Pipeline Control," *3rd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'97)*, pp. 12-21, Apr. 1997.

[Async] The Asynchronous Logic Home Page, at: *http://www.cs.man.ac.uk/amulet/async/index.html*

[BG92] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed., Prentice Hall, 1992.

[BGK+97] P. Beerel, R. Ginosar, R. Kol, C. Myers, S. Rotem, K. Stevens, and K. Yun, "Design of a High-Speed Asynchronous Instruction Length Decoder," *paper in preparation*, 1997.

[Bow95] W. J. Bowhill, et. al., "Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," *Digital Technical Journal*, **7**(1), pp.100-115, 1995.

[Bru93] E. Brunvand, "The NSR Processor," *Proc. of the 26th Annual Hawaii Int. Conf. on System Sciences*, Vol. 1, pp. 428-435, 1993.

[CDS93] W. S. Coates, A. L. Davis, and K. S. Stevens, "The Post Office Experience: Designing a Large Asynchronous Chip," *Integration - the VLSI Journal*, **15**(3), pp. 341-366, Oct. 1993.

[Cha83] T. J. Chaney, "Measured Flip-Flop Responses to Marginal Triggering," *IEEE Trans. on Computers*, **32**(12), pp. 1207-1209, Dec. 1983.

[Cha84] D. M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*, PhD thesis, Dept. of Computer Science, Stanford Univ., 1984.

[Cha87] D. M. Chapiro, "Reliable High-Speed Arbitration and Synchronization," *IEEE Trans. on Computers*, **36**(10), pp. 1251-1255, Oct. 1987.

[Chr96] D. Christie, "Developing the AMD-K5 Architecture," *IEEE Micro*, pp. 16-26, Apr. 1996.

[Chu87] T.-A. Chu, *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*, PhD thesis, MIT Laboratory for Computer Science, Jun. 1987.

[CKK+96] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, “Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers,” Technical Report, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, Apr. 1996 (See also: *http://www.ac.upc.es/~vlsi/petrify/petrify.html*).

[CM73] T. J. Chaney and C. E. Molnar, “Anomalous Behavior of Synchronizer and Arbiter Circuits,” *IEEE Trans. on Computers*, **22**(4), pp. 421-422, Apr. 1973.

[Cra92] H. G. Cragon, *Branch Strategy Taxonomy and Performance Models*, IEEE Computer Society Press, 1992

[CW75] G. R. Couranz and D. F. Wann, “Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region,” *IEEE Trans. on Computers*, **24**(6), pp. 604-616, Jun. 1975.

[Day92] N. Day , “A Comparison between Statecharts and State Transition Assertion,” *Proc. Int. Workshop on Higher Order Logic Theorem Proving and its Application - HOL'92*, Sep. 1992.

[Dea92] M. E. Dean, *STRiP: A Self-Timed RISC Processor*, PhD thesis, Stanford Univ., 1992.

[DGY92a] I. David, R. Ginosar, M. Yoeli, “An Efficient Implementation of Boolean Functions as Self-Timed Circuits,” *IEEE Trans. on Computers*, **41**(1), pp. 2-11, Jan. 1992.

[DGY92b] I. David, R. Ginosar, M. Yoeli, “Implementing Sequential Machines as Self-Timed Circuits,” *IEEE Trans. on Computers*, **41**(1), pp. 12-17, Jan. 1992.

[DGY93] I. David, R. Ginosar, and M. Yoeli, “Self-Timed Architecture of a Reduced Instruction Set Computer,” in *Asynchronous Design Methodologies*, S. Furber and M. Edwards editors, IFIP Transactions Vol. A-28, Elsevier Science Publishers, pp. 29-43, 1993.

[DGY95] I. David, R. Ginosar, M. Yoeli, “Self-Timed is Self-Checking,” *Journal on Electronic Testing: Theory and Applications*, **6**, pp. 219-228, Apr. 1995.

[DW95] P. Day and J. V. Woods, “Investigation into Micropipeline Latch Design Styles,” *IEEE Trans. on VLSI Systems*, **3**(2), pp. 264-272, Jun. 1995.

[End95] P. B. Endecott, *SCALP: A Superscalar Asynchronous Low-Power Processor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1995.

[FD96] S. B. Furber and P. Day, “Four-Phase Micropipeline Latch Control Circuits,” *IEEE Trans. on VLSI Systems*, **4**(2), pp. 247-253, Jun. 1996.

[FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "A micropipelined ARM," *VLSI'93*, 1993.

[FL96] S. B. Furber and J. Liu, "Dynamic Logic in Four-Phase Micropipelines," *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pp. 11-16, Mar. 1996.

[Flo74] I. Flores, "Lookahead Control in the IBM System 370 Model 165," *IEEE Computer*, **7**(11), pp. 24-38, Nov. 1974.

[Fri95] E. G. Friedman, editor, *Clock Distribution Networks in VLSI Circuits and Systems*, IEEE Press, 1995.

[GM90] R. Ginosar and N. Michell, "On the Potential of Asynchronous Pipelined Processors," *ACM SIGARCH Computer Architecture News*, **18**(4), pp. 27-34, Dec. 1990.

[GM97a] F. Gabbay and A. Mendelson, "An Experimental and Analytical Study of Speculative Execution Based on Value Prediction," Technical Report EE PUB#1098), Department of Electrical Engineering, Technion, Israel, Jul. 1997.

[GM97b] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?," To be published in *the 30th Annual Int. Symp. on Microarchitecture (MICRO-30)*, Dec. 1997.

[Gre95] M. R. Greenstreet, "Implementing a STARI chip,", *ICCD'95*, pp. 38-43, 1995.

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, **8**(3), pp. 231-274, 1987.

[Hau95] S. Hauck, "Asynchronous Design Methodologies: An Overview," *Proc. IEEE*, **83**(1), pp. 69-93, Jan. 1995.

[HP96a] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996.

[HP96b] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*, i-Logix Inc., 1996.

[HPS+87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," *Proc. 2nd IEEE Symp. on Logic in Computer Science*, pp. 54-64, 1987.

[Hor93] M. Horten, "The Hot New Star of Microchips (Pentium Microprocessor)," *New Scientist*, **138**(1871), pp. 31-34, May 1993.

[iLo96] i-Logix, Inc., *Statemate MAGNUM documentation*, 1996
(See also: *http://www.ilogix.com*).

[Int94] Intel Corporation, "Pentium Family User's Manual," Intel, 1994.

[Joh91] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[JRS96] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Prediction," *Proc. of the 29th Int. Symp. on Microarchitecture (Micro-29)*, pp. 142-152, Dec. 1996.

[JS89] J. M. Jaffe and M. Sidi, "Distributed Deadlock Resolution in Store-and-Forward Networks," *Algorithmica*, 4, pp. 417-436, 1989.

[Keh93] T. Kehl, "Hardware Self-Tuning and Circuit Performance Monitoring," *ICCD'93*, pp. 188-192, 1993.

[KG97] R. Kol and R. Ginosar, "Future Processors will be Asynchronous (sub-title: *Kin*: A High Performance Asynchronous Processor Architecture)," Technical Report CC PUB#202 (EE PUB#1099), Department of Electrical Engineering, Technion, Israel, Jul. 1997.

[KGS96] R. Kol, R. Ginosar, and G. Samuel, "Statechart Methodology for the Design, Validation, and Synthesis of Large Scale Asynchronous Systems," *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pp. 164-174, Mar. 1996.

[KGS97] R. Kol, R. Ginosar, and G. Samuel, "Statechart Methodology for the Design, Validation, and Synthesis of Large Scale Asynchronous Systems," *IEICE Trans. on Information and Systems*, **E80-D**(3), pp. 308-314, Mar. 1997.

[LS84] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, **17**(1), pp. 6-22, Jan. 1984.

[LS96] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. of the 29th Annual ACM/IEEE Int. Symp. on Microarchitecture*, Dec. 1996.

[LWS96] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS- VII)*, Oct. 1996.

[Mag80] N. F. Magid, *High Speed Computer Systems as a Result of Concurrent Execution of Sequential Instructions*, PhD thesis, Dept. of Electrical Engineering, Illinois Institute of Technology, Chicago, Illinois, 1980.

[Mar81] L. R. Marino, "General Theory of Metastable Operation," *IEEE Trans. on Computers*, **30**(2), pp. 107-115, Feb. 1981.

[Mar85] A. J. Martin, "A Delay-Insensitive Fair Arbiter," Technical Report 5193:TR:85, Computer Science Dept., Caltech, 1985.

[Mar90] A. J. Martin, "Synthesis of Asynchronous VLSI Circuits," in *Formal Methods in VLSI Design*, ed. J. Straunstrup, pp. 237-283, North-Holland, Amsterdam, 1990.

[MB76] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM.*, **19**, pp. 395-404, Jul. 1976.

[MBL+89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The Design of an Asynchronous Microprocessor," Technical Report, Caltech-CS-TR-89-02, 1989.

[MBM89] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Automatic Synthesis of Asynchronous Circuits from High-Level Specification," *IEEE Trans. on Computer-Aided Design*, **8**(11), pp. 1185-1205, Nov. 1989.

[MBM91] T. H.-Y. Meng, R. W. Brodersen, ann D. G. Messerschmitt, "Asynchronous Design for Programmable Digital Signal Processors," *IEEE Trans. on Signal Processing*, **39**(4), pp. 939-952, Apr. 1991.

[MTM81] N. Magid, G. Tjaden, and H. Messinger, "Exploitation of Concurrency by Virtual Elimination of Branch Instructions," *Int. Conf. on Parallel Processing (ICPP)*, pp. 164-165, Aug. 1981.

[Mye95] C. J. Myers, *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*, PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[NNS+94] L. S. Nielsen, C. Niessen, J. Sparso, and C. H. van Berkel, "Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage," *IEEE Trans. on VLSI Systems*, **2**(4), pp. 391-397, Dec. 1994.

[NUK+94] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor," *IEEE Design & Test of Computers*, **11**(2), pp. 50-63, Summer 1994.

[Pap91] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 3rd ed., McGraw-Hill, 1991.

[Pav94] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1994.

[Pec76] M. Pechoucek, "Anomalous Response Times of Input Synchronizers," *IEEE Trans. on Computers*, **25**(2), pp. 133-139, Feb. 1976.

[Per94] D. L. Perry, *VHDL*, 2nd ed., Ch. 9,10: Synthesis, McGraw-Hill, 1994.

[PN95] G. A. Pratt and J. Nguyen, "Distributed Synchronous Clocking," *IEEE Trans. on Parallel and Distributed Systems*, **6**(3), pp. 314-328, Mar. 1995.

[Ray86] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, 1986.

[RB96] W. F. Richardson and E. Brunvand, "Fred: An Architecture for a Self-Timed Decoupled Computer," *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pp. 60-68, Mar. 1996.

[RC82] F. U. Rosenberger and T. J. Chaney, "Flip-Flop Resolving Time Test Circuit," *IEEE J. of Solid-State Circuits*, **SC-17**(4), pp. 731-738, Aug. 1982.

[RMC+88] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang, "Q-modules: Internally clocked delay-insensitive modules," *IEEE Trans. on Computers*, **37**(9), pp. 1005-1018, Sep. 1988.

[SCI92] IEEE std 1596-1992, *IEEE standard for Scalable Coherent Interface (SCI)*, 1992.

[Sei80] C. L. Seitz, System timing, in C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Ch. 7, Addison-Wesley, 1980.

[Sei94] J. N. Seizovic, "Pipeline Synchronization," *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pp. 87-96, 1994.

[Sha97] H. Shafi, *Avid Execution and Instruction Pruning in the Asynchronous Processor Kin*, MSc thesis, Dept. of Electrical Eng., Technion, Israel, *in preparation*, 1997.

[SIA94] SIA, *The National Technology Roadmap for Semiconductors*, 1994
(See also: *http://www.sematech.orh/public/roadmap/index.htm*).

[Smi81] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Symp. on Computer Architecture*, pp. 135-148, May 1981.

[SS93] J. Sparso and J. Staunstrup, "Delay-insensitive multi-ring structures," *INTEGRATION, the VLSI journal*, **15**(3), pp. 313-340, 1993.

[SSM94] R. F. Sproull, I. E. Sutherland, and C. E. Molnar, "The Counterflow Pipeline Processor Architecture," *IEEE Design & Test of Computers*, **11**(3), pp. 48-59, Fall 1994.

[Ste94] K. S. Stevens, *Practical Verification and Synthesis of Low Latency Asynchronous Systems,* PhD thesis, Univ. of Calgary, Calgary, Alberta, Sep. 1994.

[Sto82] P. A. Stoll, "How to Avoid Synchronization Problems," *VLSI Design*, pp. 56-59, Nov./Dec. 1982.

[Str94] D. Strassberg, "Cooling Hot Microprocessors," *EDN (European Edition)*, **39**(2), pp. 40-44,46,48, Jan. 1994.

[Sut89] I. E. Sutherland, "Micropipelines," *Comm. ACM,* **32**(6), pp. 720-738, Jun. 1989.

[Tom67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, **11**(1), pp. 25-33, Jan. 1967.

[US95] A. K. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. of the 28th Int. Symp. on Microarchitecture (Micro-28)*, pp. 313-325, Nov. 1995.

[vBB+94] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij, "Asynchronous Circuits for Low Power: a DCC Error Corrector," *IEEE Design & Test*, **11**(2), pp. 22-32, Jun. 1994.

[vBKR+91] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI programming language Tangram and its translation into handshake circuits," *EDAC*, pp. 384-389, 1991.

[Vee80] H. J. M. Veendrick, "The Behavior of Flip-Flops Used as Synchronizers and Prediction of Their Failure Rate," *IEEE J. of Solid-State Circuits,* **15**(2), pp. 169-176, Apr. 1980.

[Ver88] T. Verhoeff, "Delay-Insensitive Codes - an Overview," *Distributed Computing*, **3**(1), pp. 1-8, 1988.

[Wei96] U. Weiser, "Future Directions in Microprocessor Design," Invited lecture, presented at *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, Mar. 1996.

[WH90] S. A. Ward and R. H. Halstead, Jr., *Computation Structures*, McGraw-Hill, 1990.

[Wol92] T. L. Wolf, *The A3000: An Asynchronous Version of the R3000*, MSc thesis, Dept. of Computer Science, Univ. of Utah, 1992.

[YBA96] K. Y. Yun, P. A. Beerel, and J. Arceo, "High-Performance Asynchronous Pipeline Circuits," *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pp. 17-28, Mar. 1996.

[YD96] K. Y. Yun and R. P. Donohue, “Pausible Clocking: A First Step Toward Heterogeneous Systems,” *ICCD’96*, pp. 118-123, 1996.

[YP92] T.-Y. Yeh and Y. N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction,” *The 19th Int. Symp. on Computer Architecture (ISCA), ACM SIGARCH Computer Architecture News,* **20**(2), pp. 124-134, May 1992.

[Yu96] A. Yu, “The Future of Microprocessors,” *IEEE Micro,* **16**(6), pp. 46-53, Dec. 1996.

[Yun96] K. Yun, “Automatic synthesis of extended burst-mode circuits using generalized C-elements,” *Proc. of the 1996 European Design Automation Conference*, pp. 290-295, Sep. 1996.

תכנון אסינכרוני מצריך שיטות הגדרה וכלים השונים מהקיים עבור תכנון סינכרוני. רוב השיטות והכלים שפותחו במיוחד לתכנון אסינכרוני מתאימים בעיקר למערכות קטנות יחסית. במסגרת המחקר נזקקנו למערכת תכנון ופיתוח שלמה, המתאימה לרמות שונות של הגדרת הארכיטקטורה של Kin ושל היחידות שבו. לשם כך, השתמשנו במתודולוגיה המבוססת על תרשימי-מיצוב (statecharts) להגדרה הפורמלית והתיאור של הארכיטקטורה והתכנון, ולשם ביצוע סימולציות. הוגדרו כללים מיוחדים (כגון חוסר תלות מוחלט בשעון) לשם התאמת מתודולוגיית תרשימי-המיצוב לתכנון אסינכרוני וביצוע סימולציות. בעזרת תרשימי המיצוב הוגדרו ונבדקו הארכיטקטורה של Kin, מימוש הביצוע הלהוט במעבד האסינכרוני, המפענח האסינכרוני של אורך פקודה והסוגים השונים של צינור אסינכרוני. בנוסף, הוגדרו מעגלי בדיקה לאימות תוקף ונכונות פרוטוקולי תקשורת והתנהגות מערכת.

DLAP מהיר יותר מכל צינור אחר (סינכרוני או אסינכרוני) כאשר ההשהיות תלויות בנתונים, וכן רגיש פחות להאטה זמנית בקצב ההוצאה מהצינור, מכיון שהוא מכיל יותר בועות. השימוש ב-DLAP פותח עבור המחשב Kin, אך הוא ישים גם בהרבה מערכות אחרות הבנויות כצינור. מעגלים אסינכרוניים צפויים להיות בעלי צריכת הספק מופחתת וביצועים גבוהים יותר ע״י הורדת השעון ושימוש בהשהיות תלויות נתונים. במסגרת המחקר פותח אלגוריתם להמרה אוטומטית של מעגלים סינכרוניים לאסינכרוניים. בכך ניתן לנצל חלק מהיתרונות של מעגלים אסינכרוניים, תוך שמירה על ההשקעות בתכנונים וכלים סינכרוניים. ההמרה נעשית על מעגל שנוצר לאחר סינתזה, והמבנה של ה-DLAP נמצא מתאים לשמש כארכיטקטורת היעד של המרה שכזו.

בעוד הרמה העליונה של הארכיטקטורה של Kin היא אסינכרונית, היחידות השונות של המעבד יכולות להיות ממומשות בהתאם למשטרי תזמון שונים. המפענח האסינכרוני של אורך פקודה וכן ה-DLAP הינם דוגמאות למתודולוגית תכנון אסינכרונית, כאשר המעגלים תוכננו במקור כאסינכרוניים, או הומרו מתכנון סינכרוני. בעבודה מוצג גם מימוש אפשרי נוסף של Kin בתור מערכת רב-סינכרונית. מתודולוגיה זו יכולה לשמש גם כשלב ביניים במעבר מתכנון שכולו סינכרוני לתכנון שכולו אסינכרוני. מערכות רב-סינכרוניות הן מערכות המבוססות על שעון משותף המופץ ליחידות השונות דרך חוטים דקים, ובכך נחסך ההספק הרב המושקע בעצי הפצת שעון ומעגלי תיאום פאזה. כל היחידות במערכת פועלות על פי אותו תדר שעון, אך בפאזה לא ידועה. לפיכך, המערכת איננה סינכרונית ברמת הארכיטקטורה העליונה של החיבור בין היחידות. מכיון שאין תיאום פאזות, נתונים הנשלחים בין יחידות עלולים לגרום לכשל סינכרון. הטיפול המקובל כיום בתופעות כאלו במערכות סינכרוניות הוא שימוש במסנכרנים. אולם, המסנכרן (הממומש כרגיסטר נוסף בכל כניסה) אינו פותר את הבעיה, אלא רק מקטין את הסיכוי שתתרחש, וגורם לעיכוב בקליטת הנתונים ועיבודם. מסנכרן עלול להיכנס למצב מטהסטבילי, ולא לצאת ממנו במשך מחזור שעון שלם בזמן עבודה בתדר גבוה. פתרון אחר המוצע בספרות הוא שימוש בשעונים מקומיים הניתנים לעצירה, כאשר עליית השעון נמנעת ומעוכבת במקרה של התנגשות בין זמן הגעת הנתונים לזמן שינוי אות השעון. כאשר הנתונים מגיעים ממספר מקורות, ההסתברות להתנגשות בין השעון והנתונים עולה, ומספר רב של עצירות השעון פוגע בביצועים. עצירת כל שעון מקומי גורמת לשינוי הפאזה בה ישלחו הנתונים ליחידה המקבלת. כאשר מספר יחידות המתקשרות ביניהן מחוברות כמעגל סגור, עלול להיווצר מצב בו כל שינוי פאזת שעון מקומי גורר שינוי השעון המקומי של היחידה העוקבת, וחוזר חלילה. המחקר מציג שיטת סינכרון מסתגלת עבור מערכות רב-סינכרוניות, אשר מקטינה משמעותית את הסיכוי לכשל סינכרון. כאשר יחידה סינכרונית אחת שולחת נתונים ליחידה סינכרונית אחרת, הנתונים הנשלחים מסונכרנים עם השעון המקומי של השולח. מכיון שהפרש הפאזה בין השעונים של השולח והמקבל, וכן ההשהיה על קווי הנתונים, הם נייחים, יש מיתאם בין זמן הגעת הנתונים לבין השעון של המקבל. הנחת הנייחות נובעת מהעובדה שההשהיות והפרשי הפאזה בין היחידות במערכת משתנים בתלות במימוש, בפרמטרים פיזיקליים של תהליך הייצור, ובטמפרטורה ומתחי ההספקה. שינויים אלו מתרחשים לאט מאד בזמן עבודת המערכת, ואינם מורגשים במשך מספר רב מאד של מחזורי שעון. בניגוד לשיטות המטפלות בשעון, הסינכרון המסתגל משנה ומתאים את ההשהיות על קווי הנתונים, ואינו נוגע בשעון. המערכת הרב-סינכרונית יכולה לכן להיות מופעלת ע״י שעון גביש חיצוני מדויק. במהלך פרקי זמן (קצרים, שאינם גורמים לפגיעה משמעותית בביצועים) של אימון, מזוהה הפרש הפאזה בין השעון של היחידה לזמני הגעת הנתונים בערוצים השונים, והשהיית כל ערוץ מכוונת למניעת התנגשות. ההסתברות לכשל סינכרון מנותחת בעבודה ומושווית יחסית להסתברות לכישלון של מסנכרן. אנו מראים כי ישנן טכנולוגיות (המאפשרות עבודה בתדרים גבוהים) בהן השימוש במסנכרן אינו יעיל, בעוד שיטת הסינכרון המסתגל עדיין פועלת. כאשר גם שיטת הסינכרון המסתגל נכשלת, יש לעבור לתכנון אסינכרוני מלא, בהתאם לשיטות האחרות המתוארות בעבודה.

במעבד בשלבי טיפול שונים. רוב המסלולים המובאים ומטופלים נזרקים לבסוף בעזרת מנגנון גיזום דינאמי, שאינו מעכב את פעולת המעבד. קביעת עומק המסלולים האלטרנטיביים נעשית בצורה דינאמית ומסתגלת, בהתאם להסתברות ביצוע הקפיצה וטיב דיוק החיזוי שלה. סמני מסלול מצורפים דינאמית לפקודות המטופלות במעבד לשם זיהוין וביצוע גיזום יעיל. ניתוח אנליטי מראה כי בעזרת ניצול יעיל של תוספת משאבים ליניארית, ביצוע להוט יכול להקטין את התשלום על טעות בחיזוי מלמעלה מ- 30% לפחות מ- 1%, ובכך להביא לשיפור הביצועים ב- 50% יחסית למה שניתן להשיג בהליכה במסלול יחיד עם משאבים דומים. סימולציות של תוכניות בדיקה מקובלות (SpecInt95) מראות שהביצוע הלהוט משיג ברוב המקרים ביצועים טובים יותר מאשר שיטת המסלול היחיד, ושהדרך האופטימלית ליישומו היא בבחירה דינאמית של עומק המסלולים האלטרנטיביים על גבי מחשב אסינכרוני כגון Kin.

הרמה העליונה של הארכיטקטורה של Kin היא אסינכרונית ובנויה ממספר יחידות בעלות תזמון עצמי (אשר מייצרות אותות לסימון סיום פעולתן), המתקשרות דרך ערוצים אסינכרוניים תוך שימוש בפרוטוקולי תקשורת. היחידות השונות של Kin יכולות להיות ממומשות פנימית או כאסינכרוניות או כסינכרוניות. בעיות של מירוץ וקפאון, וניהול הביצוע הלהוט וגיזום פקודות מטופלים ברמה הגבוהה של הארכיטקטורה האסינכרונית של Kin. היחידות של Kin יכולות להיות ממומשות פנימית כאסינכרוניות, לשם קבלת ביצועים טובים יותר (עבודה לפי השהיות ממוצעות ולא לפי המקרה הגרוע ביותר) וחיסכון בהספק (עקב ביטול הצורך בשעון), או כסינכרוניות (תוך ניצול חלקי של תכנונים קיימים). אך בכל מימוש הרמה העליונה של הארכיטקטורה של Kin היא אסינכרונית.

פענוח פקודות בעלות אורך משתנה מהווה צוואר בקבוק במחשבים בעלי ביצועים גבוהים, מכיון שזו פעולה סדרתית מטבעה (תחילת פקודה לא ניתנת לזיהוי לפני שזוהו תחילת הפקודה הקודמת לה ואורכה). כדוגמא לתכנון מלא של מערכת אסינכרונית מרמת הארכיטקטורה עד לרמת מימוש המעגל, פותח במסגרת המחקר מפענח אסינכרוני לאורך פקודה. המפענח האסינכרוני תוכנן כאופטימלי לטיפול במקרים השכיחים ביותר של אורך פקודות וסוגן, ומכיל מנגנונים יעילים לטיפול (איטי יותר) במקרים הנדירים.

שיטה מקובלת להגדלת מקביליות ושיפור ביצועים של מערכות סינכרוניות ואסינכרוניות היא שימוש בצינור (pipeline) כארכיטקטורה בסיסית. ככל שהצינור מהיר יותר, כך הביצועים טובים יותר. הארכיטקטורה של Kin יכולה להיחשב כצינור מורכב לא ליניארי, וכל אחת מיחידות המעבד יכולה להיות ממומשת פנימית כצינור. חלק מהצינורות האסינכרוניים המופיעים בספרות פועלים בניצולת של 50% בלבד של שלבי הצינור, כאשר הדרגות מחשבות לסירוגין, ודרגות שאינן מחשבות מכילות בועות. החלק האחר של הצינורות האסינכרוניים הידועים סובלים (במקרים מסוימים) מזמן התפשטות אחורנית ארוך של אותות האישור. אלמנט זיכרון אינו יכול לשנות את הערך השמור בו עד שהדרגה הבאה סימנה שהיא מוכנה לערך חדש. דבר זה עלול לגרום להגבלת הביצועים במעגלים בעלי צינור עמוק, למשל בטבעות או צינורות ליניאריים עם השהיות דרגה המשתנות בתלות בנתונים המטופלים. במהלך העבודה פותח מבנה של צינור אסינכרוני בעל בריחים כפולים (DLAP=Doubly-Latched Asynchronous Pipeline). זהו צינור אסינכרוני עם רגיסטרי אדון-עבד אשר מאפשר ביצועים מהירים יותר מתכנונים קודמים של צינורות אסינכרוניים במקרים של השהיות משתנות. פותחו שני סוגים של DLAP, האחד מבוסס על רגיסטרים מעוררי קצה, והאחר על בריחים מעוררי רמה. סימולצית SPICE של המימושים השונים מראה ש-DLAP המבוסס על עירור קצה מעט איטי יותר מזה המבוסס על עירור רמה, אך בשני המעגלים התקורה בזמן זניחה יחסית לזמני חישוב טיפוסיים. יחסית לצינור סינכרוני, DLAP דורש מספר רגיסטרים כפול, אך שימוש בבריחים מקטין את התקורה בשטח למינימלית. ההפרש בין זמן המחזור של DLAP לבין זמן המחזור של הצינור האסינכרוני היעיל ביותר מתבטא בזמן טעינת רגיסטר נוסף, אשר זניח יחסית לזמן עיבוד הנתונים בשלבי הצינור. לעומת זאת,

תקציר מורחב

הטכנולוגיה העתידית של מוליכים למחצה (כגון, מיליארד טרנזיסטורים על פיסה ושעונים בתדר גבוה מ-1GHz, כמתוכנן לשנת 2010) יוצרת אילוצים חמורים חדשים על התכנון של מיקרומעבדים בעלי ביצועים גבוהים. בפרט, השבב גדול מדי והשעון מהיר מדי לפעולה סינכרונית עם שעון יחיד. לפיכך, יש צורך לפתח ארכיטקטורות מבוזרות ותקשורות אסינכרוניות מתאימות חדשות.

בעיות הפצת השעון (כגון אי הגעה בו זמנית, ופיזור הספק גבוה) וזמני התפשטות אותות הנתונים מונעים תכנון של מעבד סינכרוני המופעל בשעון משותף יחיד. כיום מעל 40% מההספק הנצרך במעבדים מפוזר ע״י מעגל יצירת השעון והפצתו. זמני התפשטות אותות הנתונים בתוך המעבד הופכים, עם התקדמות הטכנולוגיה, לארוכים יחסית לזמן מחזור השעון המפעיל את המעבד. כתוצאה מכך, נתונים הנשלחים מיחידה אחת יגיעו במחזורי שעון שונים ליחידות יעד שונות. לא ניתן יותר להפריד בין השפעות המימוש הפיזי על פעולת המעבד ברמת הארכיטקטורה. מערכות אסינכרוניות אינן סובלות מבעיות הפצת השעון, והן יכולות לספק ביצועים גבוהים יותר ע״י ניצול השהיות משתנות בתלות בחישוב ובנתונים. מערכות אסינכרוניות אינן מוגבלות לעבודה על פי המקרה הגרוע ביותר של השהיות, כפי שנעשה במערכות סינכרוניות. מערכות בתזמון עצמי מייצרות אותות לסימון סיום פעולתן.

המחקר מתאר את הארכיטקטורה של המעבד האסינכרוני Kin[1], התומך בביצוע ספקולטיבי להוט ובביצוע פקודות מחוץ לסדר בו הן מופיעות בתוכנית. הארכיטקטורה של Kin כוללת מאפיינים ארכיטקטוניים ייחודיים המיועדים להשיג ביצועים גבוהים תוך ניצול משאבי החומרה הנדיבים שיתאפשרו ע״י הטכנולוגיה העתידית. מסקנת המחקר היא כי האילוצים הטכנולוגים מובילים בהכרח לפתרונות אסינכרוניים. התכנון והפיתוח של Kin כללו איתור ופתרון בעיות ברמת הארכיטקטורה, וגיבוש מתודולוגיות תכנון ואבני בנין לבניית מחשב אסינכרוני. פתרונות אלו נחוצים לתכנון של Kin והם מתאימים גם למערכות אחרות הפועלות בצורה דומה. חלק מהפתרונות שפותחו בעבודה זו ישימים אף בטכנולוגיה הנוכחית.

מחשבים מתקדמים מפעילים כיום שיטות מתוחכמות לחיזוי תוצאות פקודות קפיצה כדי להקטין את השפעתן על ביצועי המעבד. מחשבים בעלי מקביליות פנימית גבוהה (מספר יחידות עיבוד, ומספר שלבי צינור) אינם יכולים לנצל ביעילות את המשאבים הקיימים בתוכם בגלל רמת מקביליות מוגבלת בין פקודות בתוכניות נפוצות. התשלום בעיכוב פעולת המעבד בכל פעם שיש טעות בחיזוי של פקודת קפיצה מקטין את ביצועי המחשב. במסגרת המחקר פותחה שיטת הביצוע הלהוט (Avid Execution) לטיפול בפקודות קפיצה. שיטה זו יעילה יותר משיטת ההליכה במסלול יחיד המקובלת כיום, והיא מסוגלת לנצל ביתר יעילות משאבים פנויים במעבד. מסתבר שלשם קבלת שיפור אופטימלי בביצועי מעבד הממומש ביצוע להוט, יש לבנותו כמחשב בעל ארכיטקטורה אסינכרונית המאפשרת הסתגלות לעומסי חישוב משתנים.

שיטת הביצוע הלהוט מוצגת ומנותחת כאסטרטגיית חיזוי מעבר לפקודות קפיצה עבור המחשב Kin. התשלום על טעות בחיזוי מוקטן ע״י הבאה מוקדמת וביצוע ספקולטיבי של מספר מסלולים אלטרנטיביים בתוכנית בצורה יעילה. על פי שיטת הביצוע הלהוט, המסלול החזוי מובא ומבוצע, כפי שנעשה בשיטת המסלול היחיד. בנוסף, גם חלקים מסוימים של מסלולים שנחזו שלא יבוצעו מובאים ומטופלים. במקרה שמתגלית טעות בחיזוי, לפחות חלק מהמסלול הנכון כבר נמצא

[1] Kin הוא שמו של אל הזמן של המאיה.

תוכן העניינים (המשך)

תוכן העניינים (המשך)

תוכן העניינים

המחקר נעשה בהנחיית ד״ר רן גינוסר מהפקולטה להנדסת חשמל ופרופ׳ מיכאל יואלי מהפקולטה למדעי המחשב.

תודתי העמוקה נתונה לד״ר רן גינוסר ופרופ׳ מיכאל יואלי על הנחייתם המסורה ועל עזרתם רבת הערך והערותיהם המאירות. עידודם לחקור תחומים חדשים ומרתקים לימד אותי רבות ומוערך מאד.

לאמא, באהבת אין קץ.

ארכיטקטורה אסינכרונית בתזמון עצמי של מיקרומעבד מתקדם לשימוש כללי

חבור על מחקר

לשם מלוי חלקי של הדרישות לקבלת התואר
דוקטור למדעים

רקפת קול

הוגש לסנט הטכניון - מכון טכנולוגי לישראל
אלול תשנ״ז חיפה ספטמבר 1997

ארכיטקטורה אסינכרונית בתזמון עצמי של מיקרומעבד מתקדם לשימוש כללי

רקפת קול