Micropipeline controller design and verification with applications in signal processing

George Taylor

A thesis submitted for the degree of Doctor of Philosophy.
The University of Edinburgh.
September 1998

Abstract

Asynchronous circuits, in comparison with synchronous circuits, have the potential to offer power and speed advantages combined with improved design reuse and composition. Continual improvements in fabrication technology increase die sizes and decrease device sizes, increasing the difficulty of clock distribution and timing validation in synchronous designs. As a consequence there has been a resurgence of interest in asynchronous circuits and design methods. This work examines aspects of asynchronous micropipeline controller design, verification and application.

A micropipeline controller circuit is presented and compared with other controller circuits. A method for modelling asynchronous circuits using process algebra at an individual gate level is examined and used to verify the controller circuit. Two applications in the context of the discrete cosine transform (DCT) are then explored.

The first application is an area and power efficient circuit for bit serialisation and matrix transposition. This can be used either to embed a synchronous bit-serial processing core into a bit-parallel environment or to perform matrix transposition as part of a DCT. Key elements are modelled using process algebra. The second application is an initial attempt at an asynchronous application specific processor which is used to implement the DCT, and is intended to be extendible to other signal transforms.

The presented micropipeline controller was found to be superior to other controllers for linear micropipelines, which are key parts in the applications studied. The modelling method used has been found suitable for the verification of manually designed gate-level circuits. Finally the applications have illustrated that the use of asynchronous methods makes new or simpler architectures possible.

Declaration of Originality

I hereby declare that the research recorded in this thesis (excluding the exception stated below) and the thesis itself, is the original and sole work performed by myself while studying in the Department of Electronic and Electrical Engineering at The University of Edinburgh.

The research recorded in Chapter 4, Asynchronous circuit modelling with CCS, was performed jointly by myself and Graham Clark from the Department of Computer Science at The University of Edinburgh.

George Taylor

Acknowledgements

I would like to thank Gerard Blair and David Renshaw for their supervision and conversation. I would also like to thank all the Signals and Systems office members for interesting and fun times both in and out of the office. Special acknowledgement is due to Rudy and Sarat for their discussions, daily humour and advice on preparing a thesis.

I acknowledge EPSRC, award number 95305666, for funding this work.

Contents

List of figures

List of tables

Chapter 1
Introduction

Asynchronous circuits, circuits without clocks, have recently been attracting renewed interest as the problems of clock distribution, timing and high power consumption are becoming more commonplace with increasing size, speed, complexity of design and demand for mobile and low power products.

Asynchronous circuits are not new. Early work, on asynchronous sequential circuits and timing models, was performed by Huffman [1, 2], Muller [3, 4] and later by Hollaar [5]. Since the late 1980's and the Turing award winning paper by Ivan Sutherland [6] there has been an increasing amount of interest in asynchronous design. An overview of more recent work and design methodologies can be found in [7]. Currently there are over sixty groups involved, see [8] for comprehensive details.

Potentially, asynchronous circuits have the capability to be more modular, power efficient, faster and less dependent on technology changes than synchronous circuits.

This thesis examines aspects of asynchronous micropipeline controller design and verification, with applications oriented around a common signal processing algorithm, the Discrete Cosine Transform (DCT). Micropipeline circuits are introduced by Sutherland [6] and is the design methodology used by Amulet [9], an asynchronous implementation of the ARM processor.

1.1 Motivation, aims and chronological order of work

Initially this project started with a review of implementations of the DCT. The motivation for this was to learn about the algorithmic and circuit design techniques used in VLSI implementation of a common signal processing algorithm. One of the circuit techniques found was that of complementary pass-transistor logic [10], an alternative to standard CMOS logic gates. At the same time asynchronous design looked a promising method for reducing power consumption. The decision was made to pursue an objective of using a combination of pass-transistor logic and asynchronous circuits to produce a new DCT circuit as an example application, thereby combining a mixture of low-level and higher level design techniques.

The micropipeline style of asynchronous design was chosen, partly because it is simple (with blocks of combinatorial logic between pipeline stages equivalent to combinatorial logic between synchronous registers) and partly because it is well known.

After examination of existing micropipeline controller circuits, a new micropipeline controller circuit was developed. Although pass-transistor style logic was found to be useful in the implementation it seemed of secondary importance to the circuit itself and the modelling of the circuit. The new circuit was later found to have been previously discovered, but not formally published and was without a quantitative comparison.

The new circuit needed to be verified and it was decided to use the CCS [11] (Calculus of Communicating systems) process algebra because expertise was available locally. Initially intended simply to verify the circuit, this work expanded as subtle issues in the modelling of asynchronous circuits emerged.

In addition to the original application idea of a DCT, a new application, ideally suited to the developed micropipeline controller was devised. This application, the conversion of bit-parallel to bit-serial data, for example for a bit-serial DCT, again required verification using process algebra to ensure key parts behaved as expected. Later, independent literature, using a similar method to perform matrix transposition as part of a DCT, was found and the circuit architecture revised.

Finally, in keeping with the original project aims whilst there was remaining time, initial work on the design of an asynchronous application specific processor, with emphasis on the DCT, was performed.

1.2 Structure

Chapter 2 provides background information on asynchronous circuits and summarises the differences from synchronous circuits. The timing models used in later chapters are informally defined. The chapter then focuses on communicating data with micropipelines, the design methodology used in later chapters. Finally, the important difference between verification and simulation is stated and a method commonly used in the specification of micropipeline circuits is introduced.

Chapter 3 introduces the concept of the micropipeline latch controller. Focusing on circuit level implementations, existing latch controllers using the two-phase and four-phase protocol are reviewed. A simplified two-phase latch controller circuit is then developed, quantitatively compared with the existing circuits and shown to be favourable in certain situations.

Chapter 4 provides a short tutorial on how to model circuits using CCS. Initially believed to be a straightforward application of CCS to verify the simplified latch controller circuit, issues concerning interference between successive transitions on a wire and isochronic forks arose and are discussed. The latch controller circuits from Chapter 3 are revisited with a focus on verification, including the verification of the simplified latch controller.

Two example applications originally intended to be suitable for the latch controller circuit are then discussed. The example applications are both oriented around the Discrete Cosine Transform (DCT), an operation commonly used in digital video compression.

Chapter 5 discusses the first application, a two-dimensional structure based upon micropipelines, suitable for performing conversion between bit-parallel and bit-serial data, or for matrix transposition. The use of a micropipeline based structure results in a large power and an area saving compared to the equivalent synchronous structure. Two variations of the architecture are discussed and key aspects are modelled using the method from Chapter 2.

Chapter 6 reviews recent synchronous implementations of the DCT. Although this work is self-contained, from the review an algorithm is selected for use with initial work on the second application, an asynchronous application specific processor (ASP) for signal transforms, developed in Chapter 7.

Finally Chapter 8 provides a summary of the work, identifying the achievements made and discusses limitations and potential for future work.

Chapter 2
Asynchronous circuits and micropipelines

This chapter provides basic background information on asynchronous circuits with a focus on micropipelines, the design methodology used in later chapters.

2.1 Introduction

In a circuit, composed from sub-circuits, information is communicated via one or more signals from a *sender* sub-circuit to a *receiver* sub-circuit. In a digital circuit these signals are discrete sampled values (typically binary).

A typical digital sub-circuit repeats the following steps:

1. Receive some input,
2. spend some time processing, during which the sub-circuit is unable to receive further input (and often such that further input may result in incorrect behaviour),
3. produce some output,
4. become ready to receive input.

The majority of digital circuits require reliable communication, such that information is not lost, corrupted or duplicated. Synchronisation between senders and receivers is needed to ensure that,

- the sender produces output only when the receiver has read and has finished with the sender's previous output,
- the receiver knows when the sender has produced new output.

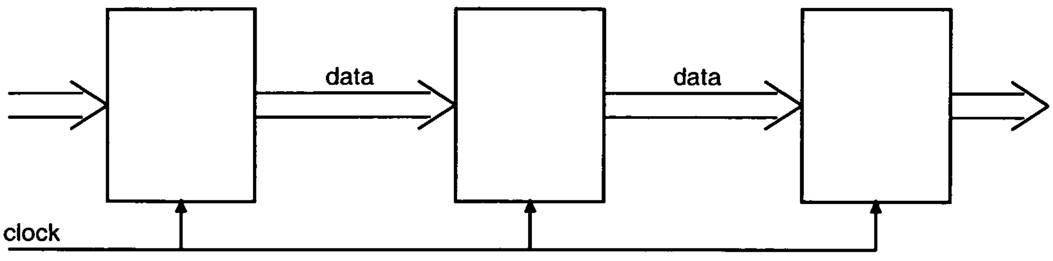


Figure 2.1: *Synchronous pipeline*

Design approaches to achieve reliable communication between sub-circuits can be divided into two categories: *synchronous* and *asynchronous*. *Synchronous* circuits are controlled by a global clock. All sub-circuits operate in lock-step regulated by this clock, thus providing the synchronisation needed between each sender and receiver. Each sub-circuit generates output and reads new input upon a change in the global clock (typically the rising edge). A typical example is the simple linear pipeline is shown in Figure 2.1. Upon every clock cycle each item of data is processed and moved one place to the right. Changes in the global clock must occur at (or at least at a sufficient approximation to) the same time everywhere in the circuit, otherwise it is possible for a receiver to read data either before the corresponding sender has sent data, or during a transient period when the data signals are changing. The majority of digital VLSI design [12] is based upon this framework.

Asynchronous circuits do not operate in lockstep and do not require a global clock. Instead the synchronisation between a sender and receiver takes place using *local handshaking*. Here a protocol is used such that the sender has a means of informing the receiver "here is some data" and the receiver had a means of acknowledging "ready for more data". Such protocols are discussed later in section 2.4.

2.2 Asynchronous advantages

Asynchronous circuits are not new, some early computers (1960s) were asynchronous. Synchronous design emerged as a way of making the design process easier. However, with a rapid increase in circuit complexity and size, various problems with large synchronous designs are starting to emerge. Potential commonly cited benefits [7] of asynchronous systems include:

Simple module interfaces: The interfaces between sub-circuits are simplified. Complex timing constraints underlying synchronous systems (for example, setup time and hold time) are no longer required. This eases specification, allows increased design reuse, eases porting to other process technologies and aids synthesis from high level languages.

Low power: By their nature most asynchronous circuits only consume power when performing useful work. By comparison the global clock present in synchronous designs runs continuously, is the most rapidly changing signal present and is distributed across a large area. This is undesirable for low power operation, although disabling the clock to idle subsystems (clock gating) can help overcome this.

No clock skew: Clock distribution problems, in particular that of clock skew, where changes in the clock signal do not arrive simultaneously at all destinations, are avoided.

Average case performance: In a synchronous system the maximum clock frequency is chosen such that the slowest sub-circuit operates correctly under the worst case conditions of high temperature, low power supply voltage and worst case fabrication parameters. Many types of asynchronous circuits do not need this (usually large) safety margin and can deliver average case rather than worst case performance. For example, decreasing the supply voltage (within limit) decreases the rate of computation, rather than causing erroneous operation.

Fewer global timing issues: For a specified minimum data processing rate, in a synchronous system all sub-circuits must be suitably optimised such that the clock rate can be high enough to support this minimum data rate. In an asynchronous system only the parts used frequently need to be optimised and less design time can be spent on rarely used features.

Electromagnetic compatibility: In a synchronous system most of the activity happens around clock edges, as a result power consumption tends to occur in bursts of high peak current with a low current between bursts. This results in a large amount of noise at the clock frequency and its harmonics. Asynchronous circuits may exhibit a 'smoother' consumption of power, the resulting noise is much more well spread throughout the frequency spectrum at a lower power level without large narrow-band peaks [13].

The design of ad hoc synchronous circuits is straightforward, the designer uses combinatorial logic to implement the desired function, registers to hold the results and by making the clock period long enough the problems of hazards, races and other dynamic behaviours are avoided.

Simulation can then be used to exercise the circuit under 'sufficiently many' input conditions. In comparison, asynchronous circuits are harder to design and generally require the application of formal design and verification techniques (introduced later in section 2.7). Asynchronous circuits may be harder to test [14], as both logical and delay faults must be tested for.

Asynchronous circuits typically incur both speed and area overhead due to local handshaking, although the impact of this depends on the 'granularity'. Finally, it is possible to combine asynchronous and synchronous design, through a scheme known as *globally asynchronous local synchronous* [15, 16]. Here many of the advantages of synchronous design (for example ease of design and low overhead) and the advantages of asynchronous design (for example reusable modules) are combined.

2.3 Timing models

Various timing models are used in the design of asynchronous circuits [17]. Definitions of common models are given here. An overview of timing models and where they are used with an emphasis on asynchronous design methodologies can be found in [7].

Bounded delay model: In a *bounded delay* model both wire delays and gate delays have upper and usually also lower bounds.

Speed independent model: In a *speed independent* model wire delays are assumed to be negligible (small enough to make no difference to circuit behaviour) compared to gate delays. The gate delays are unbounded, such that a delay may be of any size, or vary whilst the circuit is active.

Delay insensitive model: A circuit is *delay insensitive* if an arbitrary unbounded delay can be inserted anywhere in the circuit, at any time, even whilst the circuit is active and the circuit will continue to operate correctly. Because a delay insensitive circuit is free of timing assumptions circuit correctness is independent of layout and transistor sizing.

Isochronic fork: This is a fork in a wire such that the delay from the single source to both destinations is identical under all operating conditions. In a real circuit this additionally means that the circuits at both destinations must use the same voltages in the decision between a logic low and logic high. In practice it is impossible to make an isochronic fork, but usually a close approximation is suitable and achievable within a localised circuit area.

Quasi-delay insensitive A *quasi-delay insensitive* circuit is a delay insensitive circuit but with the concession that some forks may be isochronic. For all practical purposes quasi-delay insensitive circuits and speed independent circuits are identical and circuits expressed in one delay model can be expressed in the other [7].

2.4 Protocols and Micropipelines

There are two main approaches to the communication of data between a sender and receiver, delay insensitive handshaking (using a dual rail encoding) and the bundled data interface.

2.4.1 Dual rail encoding

To communicate binary data bits in a fully delay insensitive manner, two binary signal wires are required per data bit. Typically a 0 binary digit is transmitted as 01 and a 1 as 10. Between digits a 00 spacer is inserted. The receiver knows that a multi-bit datum has arrived when after receiving a spacer one wire out of each pair of signals has changed. A single *acknowledgement* wire is used by the receiver to acknowledge receipt of new input.

2.4.2 Bundled data interface

The *bundled data interface* [6] is depicted in Figure 2.2. Here a single *request* wire is used to tell the receiver the sender has sent new data. The data must reach the receiver before the request, in order that the receiver is at no risk of reading the data wires too soon. This scheme is called the bundled data interface because the data wires and the request wire are considered a bundle, the timing of which must be preserved during routing.

2.4.3 Two-phase and four-phase

Two possible protocols exist for use with the bundled data interface, the *two-phase* protocol [6] and the *four-phase* protocol [18].

The two-phase protocol uses *transition signalling*. Here both rising and falling edges of a

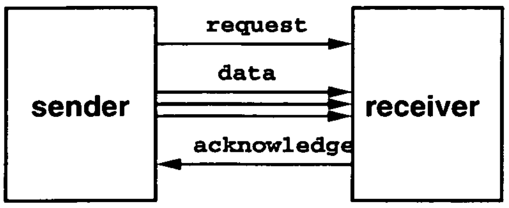


Figure 2.2: *Bundled data interface*

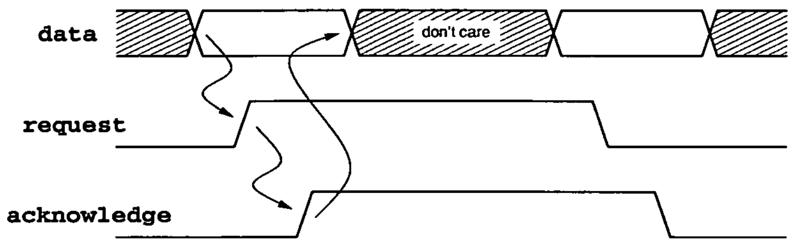


Figure 2.3: *Two phase protocol*

signal are semantically equivalent and are called *events*. Figure 2.3 shows two items of data being transferred using the two-phase protocol. Such a scheme is conceptually simple.

In the four-phase protocol only one edge (rising or falling) is significant and the other is needed to reset the signal ready for another significant edge. Compared to the two phase protocol twice as many request and acknowledge signal transitions are needed to transfer the same amount of data. Figure 2.4 shows one version of the four-phase protocol, in which rising edges are significant for both handshake signals.

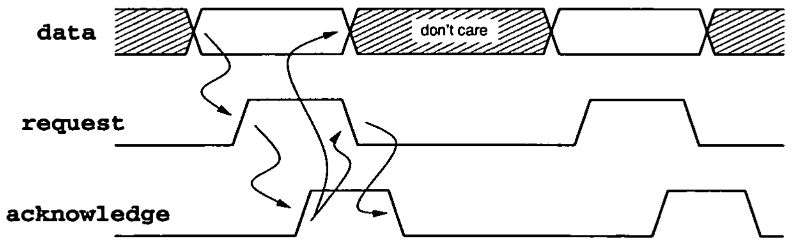


Figure 2.4: *Four phase protocol*

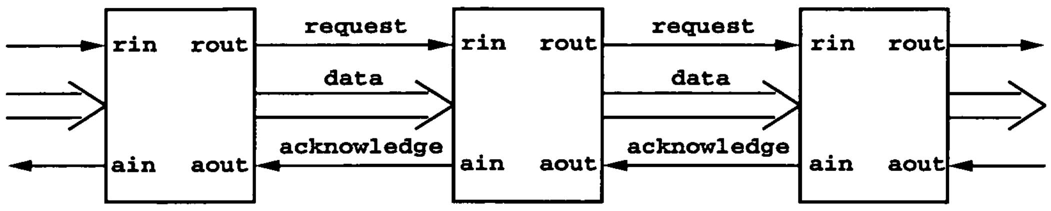


Figure 2.5: *Micropipeline FIFO*

2.4.4 Micropipeline FIFO

Multiple sub-circuits, each of which act both as a sender and a receiver (with synchronisations between inputs and outputs), communicating using the bundled data interface can be combined to form a *micropipeline* [6]. The micropipeline is the asynchronous equivalent of the synchronous pipeline. A simple linear first-in first-out (FIFO) queue is shown in Figure 2.5. There is no global clock signal to regulate the flow of data; instead local handshake signals are used. Each pipeline *stage* can request that the next stage accept new data and can acknowledge receipt of data from the previous stage. Each stage can act as soon as its immediate environment permits. Note that the conventional labelling of handshake signals uses 'in' to indicate signals on the data input interface and 'out' to indicate signals on the data output interface, irrespective of whether the signals are physical inputs or outputs.

An important feature of micropipelines is their *elasticity*. This means that the number of data items in the micropipeline can vary over time. With the addition of other synchronising circuit elements [6] (for example, Call, Select and Arbiter) more complex structures involving forking, combining and feedback paths are possible.

2.5 Event based logic

With conventional logic the building blocks are NOT, AND, OR and XOR. When the logical state of wires is unimportant and instead transitions are used, a different set of primitives is useful. Two common ones are XOR and the *Muller C-element* [6], shown in Figure 2.6. XOR acts as the OR for events, and the C-element as the AND for events. Additionally one other primitive is used later, the *Toggle* element [6]. A Toggle element has one input and two outputs `dot` and `blank`. Both outputs are initialised low. Upon each input event an event is generated on one output, alternating between outputs, starting with the `dot` output.

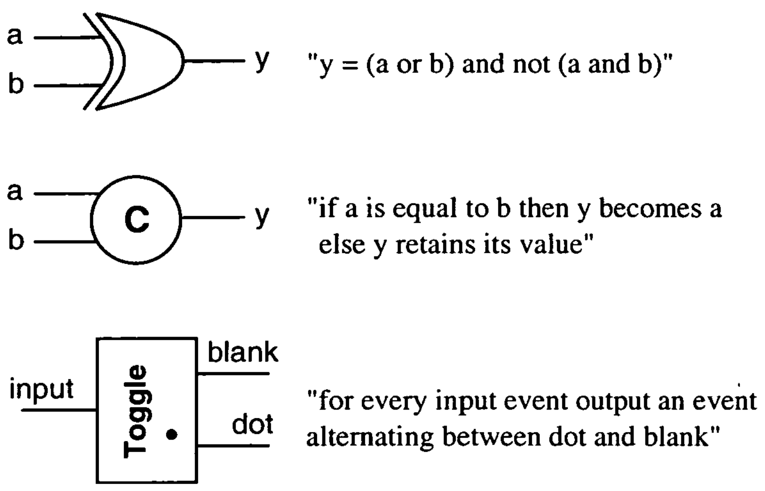

a
b
y
"y = (a or b) and not (a and b)"
a
b
C
y
"if a is equal to b then y becomes a else y retains its value"
input
Toggle
blank
dot
"for every input event output an event alternating between dot and blank"

Figure 2.6: *XOR, Muller C-element and Toggle*

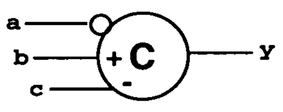


Figure 2.7: *Example of asymmetric C-clement*

More complex versions of the C-element are possible, known as *Asymmetric C-elements* [18]. An example is shown in Figure 2.7. For the output y to become high b must be high, indicated by the '+' symbol, b can not be used to stop the output becoming low. For the output to become low c must be low, indicated by the '-' symbol, c can not be used to stop the output becoming high. Input a is a 'normal' input, except that its logic value is inverted. Any combination of '+', '-' and inversion is permissible. Such asymmetric C-elements are no longer purely event based and are often used with the four-phase protocol where rising and falling edges are not equivalent.

2.6 Completion detection

One of the advantages of asynchronous design is that the time taken to complete an operation can depend on the operands. For example, in a binary adder some additions may take longer than others. By comparison, in a synchronous system, the clock period must be long enough such that there is always time for the worst case addition. This raises the question of how to know when an operation is complete.

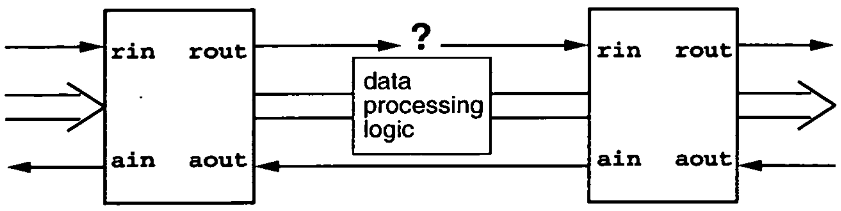


Figure 2.8: *Micropipeline with data processing logic*

The simple FIFO shown previously in Figure 2.5 can be expanded to explicitly show data processing logic inserted between stages which store data, see Figure 2.8. The issue of *completion detection* is to decide when the data processing logic has finished and permit the corresponding request event to reach the next stage. Three main methods are outlined here:

Delay matching: Here a delay equal to the worst case processing time is inserted into the request signal. Whilst conceptually simple and allowing 'conventional' combinatorial logic to be used for the data processing, each operation will effectively require the worst case delay. Careful simulation is needed (like that in a synchronous design) to ensure the delay is worst case and not shorter.

Detection logic: For some processing operations a side effect might be to indicate a result is available. Usually though it is necessary to use extra logic or dual rail encoding as a means of knowing when the output is ready. Hazard free logic is required to ensure a false 'finished' signal is not generated. Extra circuitry is required, but with the advantage of not requiring simulation to obtain the correct delays. Another variation is to use the input data to select one of several matched delays.

Current sensing: With static CMOS combinatorial logic, after an input change and before the circuit has settled with new output, a pulse of current somewhat larger than the leakage current is drawn from the supply. It is possible to insert circuitry to measure this current and indicate when the combinatorial logic has settled [19]. However, certain pathological input changes may result in current changes too small to detect and the detection circuitry may have a significant standby power consumption.

2.7 Verification

It is important to know that a circuit has been designed correctly. For synchronous circuits *simulation* is often sufficient to test a design; with asynchronous circuits the more formal approach of *verification* is generally needed.

Asynchronous circuits are free from discrete timing intervals, hence traditional discrete event simulation may only test a small fraction of the possible behaviour. The test space of an asynchronous circuit involves not only the (already potentially large) space of input stimuli and circuit states, but also many possibilities of wire and gate delays.

For example, it is impossible show using simulation that a circuit is capable of deadlock, unless the simulation is performed with the conditions required for deadlock. This would require the designer to know those conditions in advance, or be lucky. It follows that, to be certain the example circuit is not capable of deadlock, all possible conditions must be simulated. This is generally intractable, even for small circuits and the designer must resort to more formal methods to reason about circuit behaviour.

There are two approaches, which are often combined, to solving this problem:

Synthesis: A method of specifying designs is used such that only designs which do not have certain undesirable properties can be specified. The design is then (automatically) synthesised from this specification.

Verification The designer starts with an ad hoc design and uses a formal verification method to check the design is equivalent to some (simple and known to be correct) specification.

Many formal synthesis and verification methods exist, although there is lack of tool support for many methods; for example [7, 20, 21] provide a good introduction. The circuits discussed in later chapters were designed 'by hand' and verified using Milner's Calculus of Communicating Systems (CCS) [11]. The motivations for this choice and the verification method are discussed in Chapter 4.

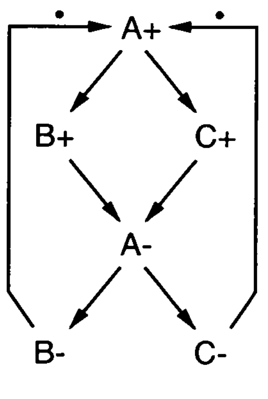


Figure 2.9: *Example Signal Transition Graph*

2.8 Specification

To verify a circuit, or to synthesise it, it is necessary to start with a specification. One method (which is used later) of specifying the behaviour of a circuit is to use a Signal Transition Graph (STG). In particular the STG/MG form [20, section 15.6] is introduced here.

An example STG is shown in Figure 2.9. The label A^+ corresponds to "signal A going high" and A^- to "signal A going low", likewise for B and C. Initially tokens are placed where marked with a dot. Initially A^+ is free to happen as there is one token on both of its input arcs. Informally, when a signal transition occurs one token is removed from each input arc and one token is added to each output arc. More than one token can be placed on an arc. (Note the overloading of the terms 'input' and 'output'. A,B and C could be circuit inputs, outputs or a mixture.) Following this rule, once A^+ occurs, B^+ and C^+ can occur in either order or at the same time, however, A^- cannot occur until both B^+ and C^+ have occurred. Likewise A^+ cannot occur again until B^- and C^- have occurred.

2.9 Summary

In this chapter the concept of and motivation for asynchronous circuits, in particular that of micropipelines have been introduced. Timing models and a commonly used specification method have been described along with the motivation for verification instead of simulation.

Chapter 3
Micropipeline latch controllers

This chapter is concerned with how data is stored in a micropipeline stage. Firstly existing circuits, to control the storage of data with latches, are introduced; then a new[1] circuit is presented and compared with the existing circuits. This chapter focuses on circuit level details, later in Chapter 4 the same circuits are revisited with a focus on formal verification.

3.1 Introduction

In [6] Sutherland describes an *event controlled latch*, which behaves like a level-sensitive latch, except that it is controlled by two inputs, *capture* and *pass*, which respond to events rather than logic levels. Sutherland also explains how a two-phase micropipeline can be constructed, where each stage is composed of an event controlled latch and a C-element. The event controlled latch is conceptually 'clean', as all control signals are event based, however, compared to a simple level-sensitive latch [12, Section 5.5.2], there is a large amount of circuitry per data-bit. It is therefore desirable, especially for wide data paths, to use instead a multi-bit *data latch* consisting of many level-sensitive latches controlled by a single enable signal. This requires a *latch controller* circuit to control the data latch and the local handshake signals [18]. Another method is to use edge-triggered registers, however, this would require increased circuitry (and hence area and power) per data bit.

Figure 3.1 shows a single stage. The data latch enable signal en is routed through the latch to produce en$'$, indicating when the data latch has completed an operation. The latch controller may present a two-phase or a four-phase interface. Notionally the latch controller is delay insensitive; in practice the latch controller may have internal timing assumptions. This is an

[1]Acknowledgement is due to Mark Josephs for pointing out, subsequent to this work being performed, that the simplified latch controller circuit has been previously discussed at a workshop by himself [22] and by I.W. Jones in 1994 at a special session at the First International Symposium on Advanced Research in Asynchronous Circuits and Systems in Utah. However, neither presentations appear in published proceedings or contain a quantitative comparison.

acceptable concession provided that such assumptions are strictly internal to the latch controller and can not be exposed by the surrounding circuitry.

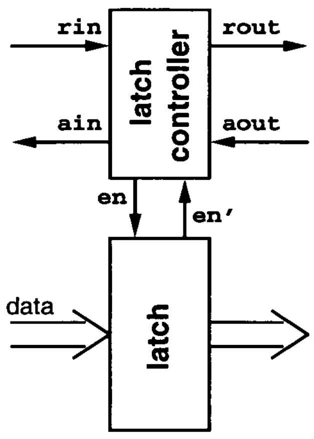


Figure 3.1: *Single pipeline stage with latch controller*

In principle, the two-phase protocol should be faster and use less power than the four-phase protocol, as there is no time wasted on return-to-zero transitions and only one transition (rather than two) per signal per datum is needed. However, in practice circuits designed to use the two-phase protocol are often more complex; transistors are level controlled not event controlled. Latch controllers are a good example of this additional complexity and this provided the motivation to produce a two-phase latch controller which is superior or at least on a par with the four-phase controllers.

3.2 Standard two-phase latch controller

Figure 3.2 [18, Figure 10] shows what will be referred to from now on as the *standard two-phase latch controller.* The C-element is used to block requests on `rin` whilst there is data stored in the data latch. The XOR detects if there has been a request passed by the C-element which has not been acknowledged by the following micropipeline stage, if so the data latch is instructed to capture the data (the data latch is transparent when en is high). The Toggle detects when the data latch has completed an operation. Both the Toggle and the C-element are initialised (reset signal not shown) after power-up. In [18] this controller is shown to be unfavourable compared to a four-phase latch controller.

A brief clarification about the positioning of the data latch should be made at this point. In

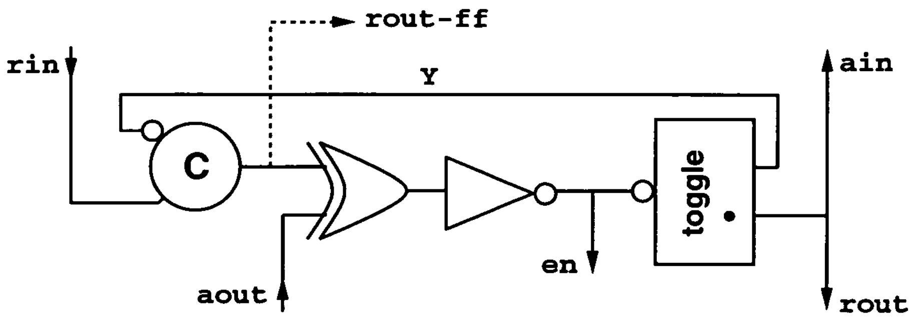


Figure 3.2: *Standard two-phase latch controller*

Figure 3.1 the data latch is shown 'in' the signal path, with output en′ a delayed copy of input en. The literature makes the assumption with the en signal that all the delay is present in gates and the wire delay is negligible, this corresponds to the speed independent timing model. In Figure 3.2 only the en signal is shown, the delay is moved into the inverter-buffer driving en. It is also assumed that when en changes the data latch acts no later than the Toggle. This assumption is not needed in the circuit of Figure 3.1, as en′ is defined to change only when the data latch has completed an operation. Additionally, when the stage captures data it is assumed that the data will have propagated through the latch no later than when an event on rout is produced; this can be considered a data bundling constraint. The assumptions surrounding en are not considered here further, the reader who dislikes these assumptions can use the method of Figure 3.1 and accept a small loss in performance. For the remainder of this work, circuit diagrams use the convention of showing only the en signal. Signal transition diagrams show actions cap (capture) and pass to indicate the data latch changing to opaque and transparent states, irrespective of the logical value of en required to make the data latch transparent.

The standard latch controller, at the hierarchical level shown in Figure 3.2, is delay-insensitive. A variant of the circuit, known as *fast-forward* involves using the rout-ff output instead of rout where rout-ff is produced in advance of the data latch capturing data. Two assumptions are made. The first assumption is that the data will have propagated through the transparent data latch, before rin propagates through the C-element to rout-ff. The second assumption is that the right-hand environment does not respond to the rout-ff by producing an aout before the latch controller is ready to accept it—an aout occuring too soon would interfere with the request travelling through the XOR gate. In practice these assumptions are often reasonable to make, especially if the right-hand environment is another similar latch controller, or, particularly in the case of the second assumption, if data-processing logic (and hence a delay on `rout`) is inserted before the following stage. The use of fast-forward can lead to a significant speed increase.

The specification for a two-phase latch controller (without fast-forward) is given in Figure 3.3. The STG could be simplified to show only events, rather than distinguishing positive and negative transitions. However, it is shown this way for consistency with the four-phase STGs shown later and to include the exact initial conditions.

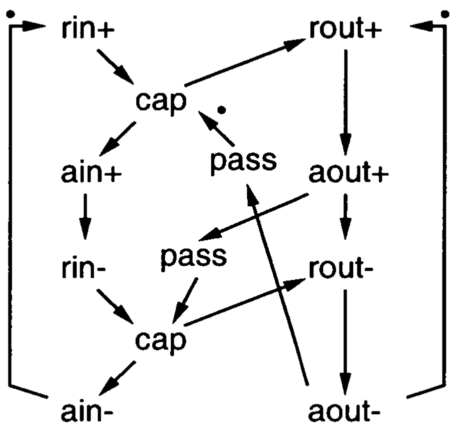


Figure 3.3: *Two-phase specification*

3.3 Four-phase latch controllers

The four-phase latch controllers discussed here all use the 'normal' variation of the four-phase protocol, where rising edges are significant for both request and acknowledge signals. Other controllers [23] may use other variations of the protocol and are intended for use where dynamic logic requiring pre-charge is inserted between micropipeline stages. Excluding these, there are four four-phase controllers for use with level-sensitive latches. These circuits are designed with the speed-independent timing model.

Simple four-phase latch controller The *simple four-phase latch controller* [24, Figure 6] (shown also in Figure 3.4) is composed from only a C-element and buffer. However, due to unwanted synchronisations between the input and output interfaces, a micropipeline based on this controller does not permit adjacent stages to hold data simultaneously. This is inefficient and, as in the literature, this controller is not considered here further.

Buggy semi-decoupled four-phase latch controller This controller [18, Figure 12] removes

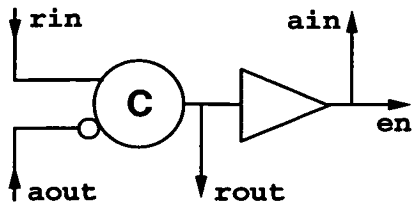


Figure 3.4: *Simple four-phase latch controller*

some, but not all, of the unwanted synchronisations present in the simple controller. The term *semi-decoupled* is used as there is still some unwanted coupling (synchronisation) between the input and output interfaces. This circuit was later shown to be non-persistent [24], however under realistic real-world conditions it is quite usable. For this reason and to distinguish it from the following variant this circuit is referred to here as the *buggy semi-decoupled four-phase latch controller.* The circuit, shown in Figure 3.5, is simplified by showing `naout`, `nain` and `nen`, the logical negations of `aout`, `ain` and `en`. The timing assumption causing the non-persistency is discussed later in Chapter 4.

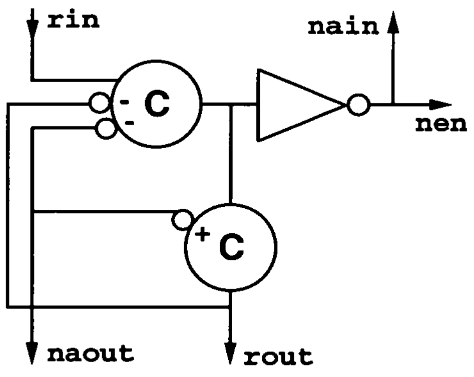


Figure 3.5: *Buggy semi-decoupled four-phase latch controller*

Fixed semi-decoupled four-phase latch controller In [24] another semi-decoupled controller is presented, shown here in Figure 3.6. This circuit, synthesised from an STG specification, is almost identical to the 'buggy' circuit except that it does not exhibit the non-persistent behaviour and is slightly slower. The circuit is referred to here as the *fixed semi-decoupled four-phase latch controller.* The STG specification is shown in Figure 3.7 with the internal signals removed and with the actions `cap` and `pass` added. The arcs $\texttt{aout}^{+} \rightarrow \texttt{ain}^{-}$ and $\texttt{rin}^{-} \rightarrow \texttt{rout}^{-}$ represent the unwanted synchronisations making this controller semi-decoupled.

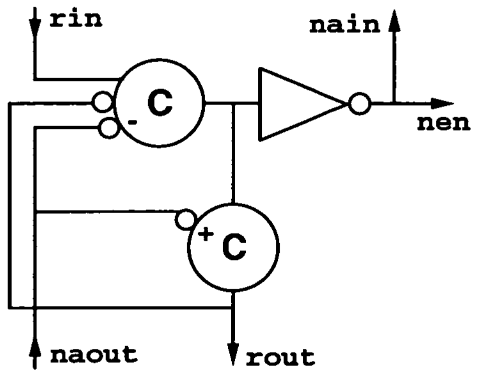


Figure 3.6: *Fixed semi-decoupled four-phase latch controller*

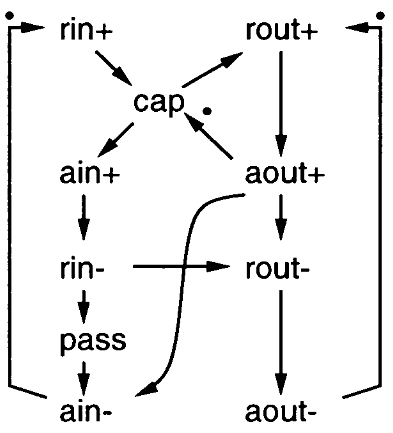


Figure 3.7: *Semi-decoupled four-phase specification*

Fully-decoupled four-phase latch controller Also in [24] a *fully-decoupled four-phase latch controller* is presented (shown here in Figure 3.8); it is fully-decoupled as there exist no unwanted synchronisations between the input and output interfaces. This lack of coupling is desirable to maximise throughput when processing logic is inserted between stages [24]. This circuit is not examined further, except as part of the performance comparison in section 3.5. The STG specification for a fully-decoupled four-phase controller is shown in Figure 3.9. It is equivalent to one half of the two-phase STG with the addition of return-to-zero actions.

Both semi-decoupled four-phase controllers are similar to the fast-forward two-phase controller, in that the data latch buffer delay is not present in the generation of `rout` but is for `ain`. For the STG specifications given here, the data latch is assumed to be in both the path from `rin` to `rout` and from `rin` to `ain`. This can be achieved by placing the data latch in

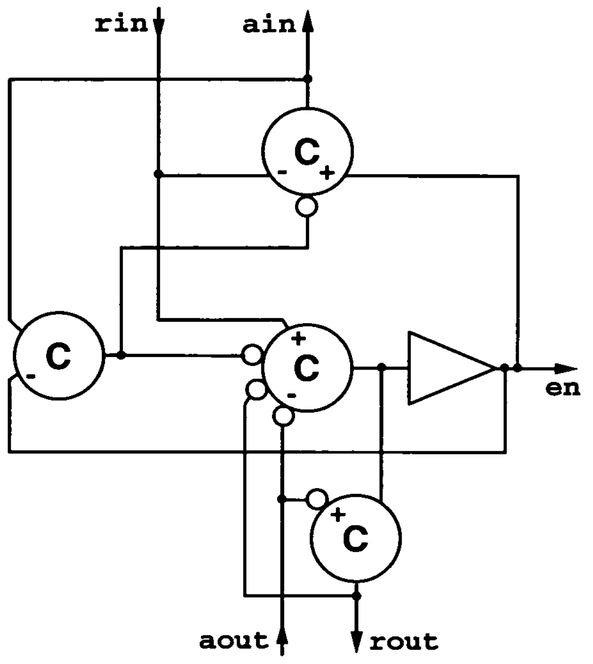


Figure 3.8: *Fully-decoupled four-phase latch controller*

'series' with the output of the 3-input C-element, as shown for the buggy semi-decoupled controller in Figure 3.10, corresponding to the model of Figure 3.1. Finally it should be noted that two-phase latch controllers, are by the definition of the two-phase protocol, fully-decoupled.

3.4 New two-phase latch controller

In this section a *simplified two-phase latch controller* is presented. This circuit has less overhead than the standard two-phase controller, whilst still using simple transparent latches in the data path. The first circuit uses the undesirable method of delay matching, the second version, which is verified later in Chapter 4, does not.

3.4.1 Latch controller with delay matching

For the standard two-phase latch controller shown in Figure 3.2, an event on `rin` causes an event on `ain` and `rout`, once the data latch has captured data. An event on `aout` causes an event on the feedback path `Y`, once the data latch has returned to being transparent. Therefore, `ain` and `rout` can be generated by delaying `rin`, and `Y` can be generated by delaying `aout`.

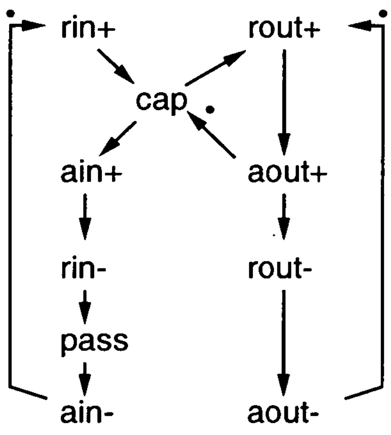


Figure 3.9: *Fully-decoupled four-phase specification*

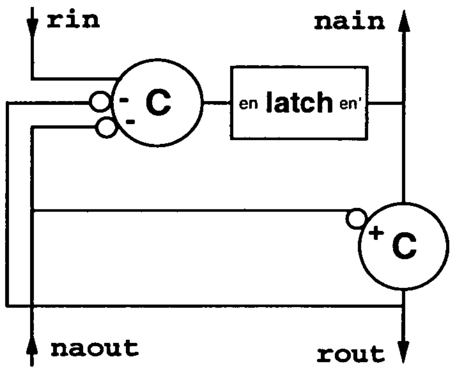


Figure 3.10: *Buggy semi-decoupled four-phase latch controller without fast-forward*

This arrangement is shown in Figure 3.11. Delay D1 ensures all data latch bits are transparent for the minimum time required to capture new data, delay D2 ensures an acknowledgement and request are not sent before the data has been captured.

Separate delays for `ain` and `rout` may be used. For example, if the data latch passes the data an appreciable time before the data is captured, a shorter delay may be used to generate `rout` than for `ain`. A fast-forward version can be obtained by taking `rout` from the input of D2, in which case the assumptions given toward the end of section 3.2 must be made.

Both D1 and D2 can be removed to give a similar micropipeline to that used by Yun *et al* [25], in which both `rout` and `ain` are generated early. Similar timing assumptions to those made by Yun would be required, with the additional constraint that the data latch must be transparent for a setup time between captures.

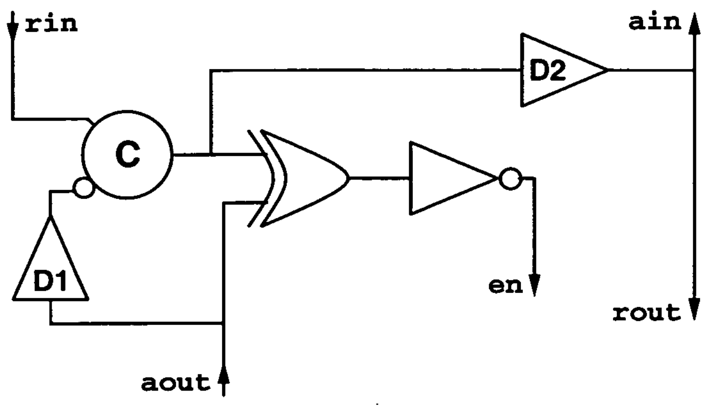


Figure 3.11: *Delay matching two-phase latch controller*

Although the circuit of Figure 3.11 can be implemented, careful simulation to find the correct delays is required and, if the load on en is large, a long chain of inverters may be required to produce each delay.

3.4.2 Simplified two-phase latch controller

In Figure 3.11 the purpose of D1 is to delay priming of the C-element in order to delay further requests on `rin` until en is high. Likewise the purpose of D2 is to delay generation of `rout` and `ain` until en is low. The circuit shown in Figure 3.12 achieves this behaviour by using level-sensitive latches L1 and L2 to block or pass events. The C-element becomes redundant and is removed. A latch acts as an AND between a transition sensitive signal and a level sensitive signal. Latch L1 blocks events on `rin` until the data latch is able to capture new data. Latch L2 delays generation of events on `rout` and `ain` until the data latch has captured the new data.

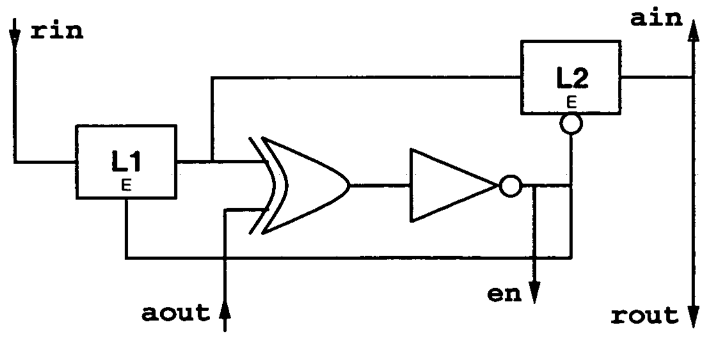


Figure 3.12: *Simplified two-phase latch controller*

The circuit operates as follows: initially all signals except `en` are low, L1 and the data latch are transparent and L2 is opaque. An event on `rin` will pass through L1 and the XOR causing `en` to become low and the data latch and L1 to capture. L2 will then become transparent allowing the event to proceed to `rout` and `ain`. A further event on `rin` would be blocked by L1. An event on `aout` will now cause `en` to become high, the data latch to become transparent, L2 to capture and L1 to pass a blocked or future event on `rin`.

It is assumed that L1 and L2 operate at the same time, i.e. that there is an isochronic fork from `en` to L1 and L2. In practice isochronic forks can be difficult to implement and so it is possible for L1 and L2 to be transparent at the same time. In the first case, if L1 is late capturing, the event passed by L2 to `ain` could result in an event on `rin` whilst L1 is still transparent. However, this path involves the `aout` to `rout` delay of the previous stage, which is an acceptable safety margin. In the second case, if L2 is late capturing, an event on `rin` may pass directly through L1 and L2. This race hazard does not involve an external path but is similar to one found in many synchronous designs where flip-flops are built from two latches, including the implementation of the toggle element (for example that in [25]).

A possible implementation is shown in Fig. 3.13. Inverter B0 provides the buffering to drive the multi-bit data latch. Latches L1 and L2 are constructed from a pass-transistor and two inverters [18, Figure 5] (using a weak inverter with resistive elements as feedback to keep state). These latches invert the data. At initialisation, the output of L1 is set high and the output of L2 is set low. The complementary signals required to drive the pass-transistors are explicitly shown and inverter B4 is shared. Inverter B4 is required, rather than taking the output of the XOR, to ensure both of the latch control signals are subject to delay in charging and discharging the `en` signal.

For a fair comparison with the four-phase controllers it is necessary to construct the fast-forward version. This requires inverter B2 to obtain the correct polarity for `rout-ff`. Since B2 is therefore required anyway, an efficient complementary pass-transistor XOR gate is used, requiring B1 to generate the complement of the other input.

To counter the race hazard (the second case described above) the input to L2 is obtained via B3 and not directly from L1. It is intended that the delay through B2 and B3 be larger than delay between L1 acting and L2 acting. Note that passing request events from `rin` to `ain` and `rout` via B2 and B3 does not change the critical path as events will be delayed by L2 until the data latch has captured.

It should be noted that the use of complementary logic signals in this way increases the number of timing assumptions present—it is assumed that the inverter delays present in generating the complements are negligible. The diagram of Figure 3.13 could be drawn not to show the complements, for example by generating them 'inside' the latch and XOR components. In hindsight `rout-ff` as shown should probably not be shared with the XOR input as external loading could affect internal circuit behaviour. An alternative and perhaps more robust approach would be to use a standard CMOS XOR gate. Finally the use of standard circuit elements may ease implementation using a standard cell library which does not contain the 'non-standard' C-element.

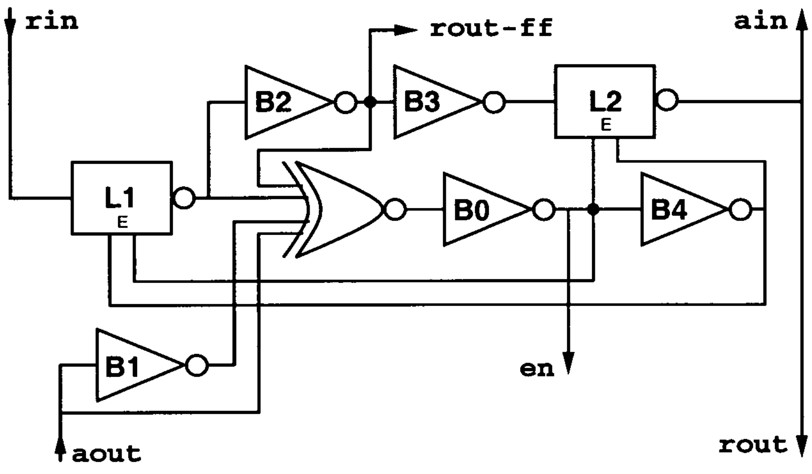


Figure 3.13: *Simplified two-phase latch controller: possible implementation*

Yun *et al.* [25] discuss a strategy for increasing speed using double-edge triggered flip-flops, one for each data bit. The control circuit is then only a C-element and buffer giving a much shorter cycle time than both the standard two-phase controller and four-phase controllers. However, this is at the expense of increased power dissipation and area due to the complex flip-flop circuit required per data bit. Their speedup is also partly due to additional timing assumptions.

3.5 Comparison

To quantitatively evaluate the simplified latch controller in comparison with other latch controllers, transistor net-lists of a six stage FIFO were created for simulation using SPICE with

Circuit	tr count	energy (pJ)	cycle time (ns)	energy-delay
Simplified 2-ph FF (Fig. 3.13)	29 (4)	12.6 (1.00)	4.17 (1.00)	(1.00)
Standard 2-ph FF (Fig. 3.2)	48 (2)	14.1 (1.12)	4.92 (1.18)	(1.32)
Semi-dec. 4-ph ([18, Fig.12])	18 (4)	13.0 (1.03)	4.99 (1.20)	(1.23)
Fully-dec. 4-ph ([24, Fig.13])	42 (8)	17.8 (1.41)	5.54 (1.33)	(1.88)
Simplified 2-ph (Fig. 3.13)	29 (4)	12.8 (1.02)	5.63 (1.35)	(1.37)
Standard 2-ph (Fig. 3.2)	48 (2)	14.2 (1.13)	6.46 (1.55)	(1.75)

Table 3.1: *Results of comparison*

ES2 5V 0.7μm parameters. All transistors were sized to give minimum area except for devices used as resistive elements. Each circuit is connected into a test framework where the left and right-hand environments supply and read data as fast as the pipeline will allow, so that the pipeline determines the speed.

This method of measuring the speed corresponds to the method used in [24] rather than in [18] in which individual timings (for example `rin` to `rout` delay) are summed to give an overall cycle time. It was found the later approach lead to a significant error if the timings were always measured at the fifty percent of full voltage swing point, rather than taking into account the exact voltage at which the logic starts to switch.

Capacitive loading is placed on `en` to simulate a 32-bit single-phase latch [18, Figure 9]. The two-phase circuits with the toggle element employ the improved toggle circuit given in [25]. The four-phase circuits compared are the buggy semi-decoupled controller and the fully-decoupled controller. The simple four-phase controller is not included, as it does not allow adjacent stages to simultaneously hold data and such a comparison would be unfair.

Table 3.1 lists the results. The cycle time was measured as the delay between requests from one stage to another and was confirmed to be the same between all pairs of adjacent stages. The values shown are for a single pipeline stage, normalised values are shown in parentheses. The transistor count excludes devices used as resistive elements in the weak inverters in the latch and C-element implementations (these are shown in parentheses). ‘FF’ indicates a fast-forward version.

The four-phase circuits generate `rout` excluding the data latch buffer delay time, so they should be compared with the fast-forward two-phase circuits. The results indicate that the simplified two-phase circuit is faster, smaller and consumes less power than the standard two-phase circuit and compares favourably with the four-phase circuits. This comparison is valid for simple linear micropipelines, however, such an analysis for general micropipelined systems would need to take into account the efficiency of other primitive elements such as those mentioned in section 2.4.4. In Chapter 5 a circuit for which the simplified controller is ideal is presented.

3.6 Summary

In this chapter latch controllers have been introduced. A new latch controller circuit has been presented and shown to compare very favourably with previous latch controllers and to be superior when used in appropriate circumstances.

Chapter 4
Asynchronous circuit modelling with CCS

This chapter describes a method of modelling asynchronous circuits using process algebra, this method is then used to verify some of the circuits discussed previously in Chapter 3. The aim of this modelling is to verify that circuits behave as specified and do this without unknown timing assumptions.

4.1 Introduction

Milner's Calculus of Communicating Systems (CCS) [11], a process algebra, provides a formal semantic basis for reasoning about concurrent systems. CCS was chosen in preference to other methods because of former experience with it and that tool support and expertise are available locally.

4.2 Relation to previous work

Initially the intention was to verify the simplified two-phase latch controller (section 3.4.2) and it was believed that this would be a straightforward application of CCS. However, various non-trivial aspects soon emerged, such as those of quenching and the implementation of isochronic forks (these are discussed later). The first two modelling styles (sections 4.4.2 and 4.4.3) were developed before investigating the literature. It was encouraging to find that the second of these styles (section 4.4.3) had been used before [26, 27]. The modelling method was then extended (sections 4.4.4 and 4.4.5) and when this work was submitted to the UK Asynchronous Forum (see Appendix D) literature was found [28] which discusses essentially the same extension as given in section 4.4.4. To date it would seem that the modelling style of section 4.4.5 is unique in that the strategy to allow quenching behaviour to occur has not been used before.

An alternative approach involving process algebra is the Rainbow system [29]. Rainbow promises support for modelling asynchronous circuits at a variety of levels, for example with a low level hardware description language, at an algorithm control-flow level or with temporal logic. The system is built on a formal basis using a process algebra called APA. This gives a sound foundation on which support for analysis, simulation, verification and synthesis should be possible. At the time of the modelling work discussed here, Rainbow did not offer verification. Rainbow is intended for the design of systems built from modules communicating with the bundled data interface, rather than for the design of gate level circuits. One key feature is that, for simulation, data values as well as control signals can be manipulated.

Another process algebra approach designed for asynchronous circuits is described in [30], but no tool support is available for this, and automated model checking was desirable. This approach does however lead to more concise models for components, as properties such as all possible orders of actions do not need to be specified and isochronic forks are less 'messy'. Initially the idea of specifying all behaviours was preferred, at the risk of missing some out, instead of specifying some at the risk of including implicit unwanted behaviours.

A different approach to verification is to use Petri nets. For example one property verifiable with this technique is that of persistency; checking that when a signal has made a transition it cannot make a further transition until all signal transitions depending on this transition have occurred. This corresponds to the concept of an event 'catching up' with another event resulting in quenching (see section 4.4.5). The compositional approach of CCS was preferred to the Petri net approaches which tend to flatten the circuit structure. Also for large circuits the Petri net graphs become complex and lose their intuitive graphical appeal. However, it should be noted that recent Petri net approaches [31] look suitable for the tool support of modelling circuits with large state spaces; an intractable state space is easy to achieve with CCS. Since both Petri nets and CCS are both essentially ways of describing a labelled transition system, it would be reasonable to speculate that equivalent 'tricks' could by employed in CCS by algebraic manipulation.

The most notable difference is that these other approaches consider the behaviour of a signal 'catching up' with another as an error. The method discussed in section 4.4.5 allows this to happen, provided that the circuit model still satisfies the specification it is compared with.

4.3 CCS overview

Systems described in CCS consist of *processes* composed using the basic constructors provided by CCS, for example those of non-deterministic choice and parallel composition. A process can perform *actions* to evolve into a new process, possibly involving communication with another process. An informal description of the syntax of processes is given below.

Prefix: If P is a process, then $\mathsf{a}.P$ is a process that can perform an a action, evolving into P. Actions fall into three categories: *input* actions, for example a, *output* actions, for example $\overline{\mathsf{a}}$, and a distinguished action, τ, that represents the silent action, an action produced by a communication internal to a process.

Definition: If Q is a process and $P \overset{\text{def}}{=} Q$, then P is a process that can only behave in exactly the same fashion as Q. This constructor must be used to create recursive definitions.

Summation/Choice: If P and Q are processes, then $P+Q$ is a process that non-deterministically chooses to behave like either P or Q.

Composition: If P and Q are processes, then $P|Q$ is a process that can behave like P and Q, acting independently of each other, or which can evolve further by a communication between P and Q if they possess complementary input and output ports. This means that if P can perform the input action a, and Q can perform the output action $\overline{\mathsf{a}}$, $P|Q$ can perform either a, $\overline{\mathsf{a}}$, or the silent action τ resulting from an internal communication between processes P and Q.

Restriction: If P is a process and L is a set of actions (excluding the silent action τ), then $P\backslash L$ is the process that can behave exactly the same as P, except that it cannot perform any actions contained in L. Restriction is used with composition to force synchronisation on input and output actions. For example by extending the composition example above to give $(P|Q)\backslash\mathsf{a}$, processes P can only perform input action a when Q also performs output action $\overline{\mathsf{a}}$, together these are observed as a τ action.

Relabeling: If P is a process and a and b are actions, then $P[\mathsf{b}/\mathsf{a}]$ is the process that behaves exactly the same as P except that if P can perform a or $\overline{\mathsf{a}}$ then $P[\mathsf{b}/\mathsf{a}]$ can perform b or $\overline{\mathsf{b}}$ respectively. This is useful when several copies of the same process with differing action names are required.

It is possible to compute the state space for a process. This can be thought of as a number of states interconnected by transitions, a labelled transition system. Each state corresponds to a process and transitions between processes are labelled with actions. For example the recursive process $P \stackrel{\text{def}}{=} \mathtt{a}.\mathtt{b}.P$ has two states and two transitions.

With the CCS outlined above it possible to describe systems, however further support is required to reason about these systems, for example to ask "Can action $\mathtt{a}$ ever happen?". In fact for the circuits verified here, it is usually not necessary to ask such questions as the verification is performed by testing the equivalence between a model of the circuit and a specification. However, during development of the models it was useful to ask such questions, and when the circuit and specification are not equivalent a method is required to state distinguishing behaviours.

Hennessy-Milner logic [32] and its temporal extension, the modal mu-calculus [33] are modal logics which can be used to express properties of labelled transition systems. The full syntax is not given here, but informal definitions of parts of the logic used later are introduced when needed.

Tool support exists in the form of the Edinburgh Concurrency Workbench (CWB) [34]. The CWB computes the state space for a CCS process, and provides support for: determining the equivalence of processes, automatically generating logical formulae which distinguish processes, interactively simulating a process, analysing the state space and allowing properties of a process to be checked using Hennessy-Milner logic and the modal mu-calculus.

The most commonly used feature of the CWB was the ability to test for observational equivalence [11]. Informally, two processes are observationally equivalent if they can not be distinguished by an observer with which the processes interact. Specifically silent τ actions are not included, this means that it is possible to show that a circuit model is equivalent to a specification without the τ actions 'getting in the way'.

4.4 Modelling circuits with CCS and the CWB

During the development of the modelling method several issues relating to the choice of CCS to model asynchronous circuits were observed:

Suitability: Circuits are composed from sub-circuits, the compositionality of CCS means this structure is captured in the model. CCS has no notion of explicit timing delays, this can be thought of as inserting arbitrary but finite delays between actions. This property combined with non-deterministic choice is ideal for testing for delay insensitivity.

Accuracy: All models abstract from the real circuit implementation. For simple circuit elements such as XOR it is easy to see the correctness of the model. For more complex elements, such as the C-element, some care is needed. It would be possible to use CCS to model circuits at a transistor switching level (simple on-off behaviour), but in this work only logic gate level modelling was performed.

Clarity: CCS allows the specification of structured, clear and concise models. However as will be seen later some of this clarity and much conciseness is lost when isochronic forks are added to the model.

Next some basic assumptions are given and then the modelling method is developed; the intention is that models of circuits are to be compared, using observational equivalence, to a simple and therefore 'correct by inspection' specification. Initially only simple circuit elements, focusing on XOR, are considered. CCS circuit models for commonly used elements are given later in section 4.5.

4.4.1 Assumptions

The following assumptions are made.

1. The environment (circuitry connected to the circuit being verified) is assumed to operate correctly.

2. Gates are modelled as digital devices using two logic levels, as with traditional discrete event simulators. Communication between gates is assumed to be reliable. If needed the modelling method could be extended to support more than two logic levels, for example weak and strong, but this would increase the state space of the models considerably.

3. It is assumed all gates are initialised into a known state after power-on. For state keeping gates, for example latches, an explicit reset signal is required. This reset mechanism is not modelled.

4. No assumptions about the delays in gates or wires are made; all delays are unbounded but finite.

5. For the latch controller circuits studied the circuit model described in section 3.2, Figure 3.1 where the data latch is in the signal path is used. Additionally all data bundling constraints are assumed. Likewise verification does not include the assumptions involved in the fast-forward variation (section 3.2).

4.4.2 Modelling with logic levels

The first method captures the logic value (high or low) of signals, much like that in discrete event simulation. The CCS model for an XOR gate with two inputs $\mathtt{in}_1, \mathtt{in}_2$ and output $\mathtt{out}$ is given below. The subscripted numbers in the process names refer to the logic levels of the inputs. The action names indicate input and output 'values'. Processes XOR_{10}, XOR_{01} etc are defined in a similar manner.

$$
\begin{aligned}
XOR_{00} &\overset{\text{def}}{=} \mathtt{in_1hi}.XOR_{10} \\
&+ \mathtt{in_1lo}.XOR_{00} \\
&+ \mathtt{in_2hi}.XOR_{01} \\
&+ \mathtt{in_2lo}.XOR_{00} \\
&+ \overline{\mathtt{outlo}}.XOR_{00} \text{ etc}
\end{aligned}
$$

The XOR process has four states, corresponding to the possible input combinations. There are disadvantages to this method. The model can always accept input and offer output even though no change in signal level occurs, this is meaningless in circuit terms and not intuitive. This may also be undesirable for analysis as it is always possible for the model to exhibit activity (for example silent τ transitions) even though the circuit is in a stable state awaiting further input. Finally, the gate is modelled at an unnecessary level of detail resulting in a larger than necessary state space.

4.4.3 Modelling with events

Instead of modelling logic levels it is possible to model events, abstracting from the actual logic levels. XOR becomes:

$$XOR \overset{\text{def}}{=} \mathtt{in}_1.\overline{\mathtt{out}}.XOR + \mathtt{in}_2.\overline{\mathtt{out}}.XOR$$

This XOR process has two states, one state which allows an input action on either input and the other which offers an output action. This model is an improvement, actions only occur when signals change and fewer states are required. There is no loss of generality by modelling events instead of logic levels, when modelling gates where the initial state of the inputs are important (for example an AND gate), it is only necessary to arrange the internal states of the model appropriately.

However, there is a flaw with this modelling style. Consider XOR, once the action corresponding to an event on one input has occurred, it is impossible for the other input action to occur until the output action occurs; the model blocks further inputs until an output occurs. This is not a property true of real circuits which can always receive input, even if such input causes undesirable behaviour. This is a significant flaw—the model cannot spot delay *sensitive* behaviour.

For example, consider the 'transition detector' circuit shown in Figure 4.1. The intention is that, when an event occurs on one of the XOR inputs, an event occurs on the output of the XOR clocking the '1' into the double-edge triggered flip-flop (DET-FF). In the model this circuit could be verified as correct by testing the property that any input event to the XOR must eventually lead to an event on the DET-FF output. However, in the real circuit, it is possible for a second input event to arrive at the XOR before the XOR has produced output. Such a second event may cancel the due change in output, or it may not, this is a race condition. This behaviour is not captured by the model as all events input to the XOR must be propagated to the XOR output.

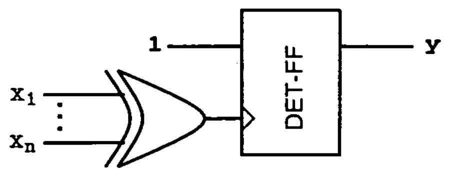


Figure 4.1: *Transition detector*

This modelling style is suitable for the verification of circuits provided it is known in advance that they do not exhibit delay sensitive behaviour, for example a circuit composed properly from modules communicating with a two-phase bundled data interface. This is the modelling style used by [26]. To observe delay sensitive behaviour it is necessary to ensure that inputs to a circuit can not be blocked. The next two modelling styles overcome this.

4.4.4 Overcoming blocking with wires

In this style 'wires' are added between gates. The obvious definition of a wire might be:

$$\mathit{Wire} \stackrel{\text{def}}{=} \texttt{in}.\overline{\texttt{out}}.\,\mathit{Wire}$$

The wire process acts as a one place buffer, however this does not solve the problem, it merely postpones it. Whereas before only one input event to the *XOR* gate was required to block further input, an *XOR* followed by *Wire* requires two input events for blocking to occur. A delay insensitive circuit should not permit circumstances in which one event might 'catch up' and interfere with another. Therefore one solution to the blocking problem is to consider the circuit in error if an event is blocked. The model can then be tested by using the CWB to check if any of the states in the model satisfy the mu-calculus formula for "eventually, an error may occur". This can be achieved with the following definition for *Wire*:

$$\begin{aligned}
\mathit{Wire} &\stackrel{\text{def}}{=} \texttt{in}.\,\mathit{Wire}' \\
\mathit{Wire}' &\stackrel{\text{def}}{=} \overline{\texttt{out}}.\,\mathit{Wire} + \texttt{in}.\mathit{Error} \\
\mathit{Error} &\stackrel{\text{def}}{=} \texttt{error}.\mathit{Error}
\end{aligned}$$

Whilst this modelling style would work, there is a penalty of increased state space as each wire now has three states. In particular the majority of states in the model may only be reached once a single wire has entered its error state. An alternative approach would be to build the error state into the gates, avoiding the need for wires, but this still increases the state space and does not aid clarity. The final modelling method, discussed next, does not add to the state space.

4.4.5 Final modelling method

This approach directly models the possibility that a gate which has been *excited* by an input event may return to an unexcited state via a further *quenching* input action. *XOR* becomes: (see also Appendix A.1).

$$\begin{aligned} XOR &\overset{\text{def}}{=} \texttt{in}_1.XOR_e + \texttt{in}_2.XOR_e \\ XOR_e &\overset{\text{def}}{=} \texttt{in}_1 XOR + \texttt{in}_2.XOR + \overline{\texttt{out}}.XOR \end{aligned}$$

Like the original event based model (section 4.4.3) only two states are required. This method takes the approach that quenching can occur, and is not an error, provided that the circuit behaviour still satisfies the specification. In fact it is no longer possible to detect quenching. The disadvantage to this approach is that the definitions of gates lose some clarity, especially when synchronisation mechanisms are added to permit isochronic forks, as shall be seen later. However this approach was found to work well in practice.

4.5 CCS models of common circuit elements

The CCS models for the basic circuit elements used in two-phase and four-phase latch controllers are given here, except for XOR which was given in section 4.4.5. This is followed by a simple example to illustrate composition.

The C-element, toggle and latch models assume delay-insensitivity even though the implementations may contain timing assumptions. These timing assumptions are either completely internal, or are the result of an internal fork for a feedback path required to keep state. In the later case, it is possible to change the inputs before the feedback path has settled resulting in incorrect operation. In practice these assumptions are reasonable as they are very localised, and the internal feedback path delay is very short. This is an example of 'hierarchical' delay-insensitivity where the circuit is being tested for delay-insensitivity given the assumption that its building blocks are delay insensitive.

4.5.1 C-element

$$\begin{aligned} C &\overset{\text{def}}{=} \texttt{in}_1.C_1 + \texttt{in}_2.C_2 \\ C_1 &\overset{\text{def}}{=} \texttt{in}_1.C + \texttt{in}_2.C_e \\ C_2 &\overset{\text{def}}{=} \texttt{in}_1.C_e + \texttt{in}_2.C \\ C_e &\overset{\text{def}}{=} \texttt{in}_1.C_2 + \texttt{in}_2.C_1 + \overline{\texttt{out}}.C \end{aligned}$$

To model a C-element for which one input is inverted, for example that used in the standard two-phase latch controller, a different starting state is used, for example C_1 instead of C. Asymmetric C-elements are created in a similar manner, see Appendix A.2.

4.5.2 Toggle

The toggle element is specified below.

$$\begin{aligned} \mathit{Toggle} &\stackrel{\text{def}}{=} \mathtt{in}.(\overline{\mathtt{dot}}.\mathit{Toggle}_2 + \mathtt{in}.\mathit{Toggle}) \\ \mathit{Toggle}_2 &\stackrel{\text{def}}{=} \mathtt{in}.(\overline{\mathtt{blank}}.\mathit{Toggle} + \mathtt{in}.\mathit{Toggle}_2) \end{aligned}$$

4.5.3 Latch

A simple transparent latch: the latch can be enabled allowing events to pass from input to output, or disabled such that output events are not possible. Note that when disabled, internal state changes must still take place upon input events in order to know if an output needs to be offered upon re-enabling (the output need not necessarily occur, for example if a quenching enable or input event occurs). The *Latch* process below is initially enabled (transparent).

$$\begin{aligned} \mathit{Latch} &\stackrel{\text{def}}{=} \mathit{Latch\text{-}En} \\ \mathit{Latch\text{-}En} &\stackrel{\text{def}}{=} \mathtt{in}.\mathit{Latch\text{-}En}_e + \mathtt{enable}.\mathit{Latch\text{-}Dis} \\ \mathit{Latch\text{-}En}_e &\stackrel{\text{def}}{=} \mathtt{in}.\mathit{Latch\text{-}En} + \mathtt{enable}.\mathit{Latch\text{-}Dis}_e + \overline{\mathtt{out}}.\mathit{Latch\text{-}En} \\ \mathit{Latch\text{-}Dis} &\stackrel{\text{def}}{=} \mathtt{in}.\mathit{Latch\text{-}Dis}_e + \mathtt{enable}.\mathit{Latch\text{-}En} \\ \mathit{Latch\text{-}Dis}_e &\stackrel{\text{def}}{=} \mathtt{in}.\mathit{Latch\text{-}Dis} + \mathtt{enable}.\mathit{Latch\text{-}En}_e \end{aligned}$$

4.5.4 Data latch

In checking the correctness of a latch controller it is useful to know when the multi-bit data latch captures data. The simple definition below, which generates an observable `capture` action, can be used to achieve this. The `release` action is optional and can be left out, or ignored. In verifying that a latch controller is suitable for use in a micropipeline it is not necessary to know when the data latches are transparent as the transparent state serves no useful purpose (except when using the fast-forward variant). It is however essential to check that the latch controller has captured the data before sending an `ain` event to the previous stage. There is no need to explicitly permit quenching on the input. The *Datalatch* process cannot block its input provided that the output is connected to an input of another process which permits quenching and that no synchronisations with the `capture` or `release` actions are made.

$$\mathit{Datalatch} \stackrel{\text{def}}{=} \mathtt{in}.\mathtt{capture}.\overline{\mathtt{out}}.\mathtt{in}.\mathtt{release}.\overline{\mathtt{out}}.\mathit{Datalatch}$$

4.5.5 Forks

Most circuits contain at least one fork. In modelling terms this represents an event which needs to be propagated to more than one destination. In a delay insensitive circuit the events may be propagated to the outputs in either order: a non-deterministic fork. Furthermore, a quenching input event can occur before or after the previous input event has reached one destination. For example an input event may have reached one destination, a second input event then occurs, resulting in no events reaching the second destination; this may result in a second event reaching the first destination. The full version, as given below, of this fork must be used; the optimised version [27], in which the output events occur in a deterministic order, cannot be used, as the circuits discussed later do not fall into the class of circuits for which this optimisation is valid.

$$\begin{aligned}
Fork &\stackrel{\text{def}}{=} \texttt{in}.Fork_e \\
Fork_e &\stackrel{\text{def}}{=} \overline{\texttt{out}_1}.Fork_{2e} + \overline{\texttt{out}_2}.Fork_{1e} + \texttt{in}.Fork \\
Fork_{1e} &\stackrel{\text{def}}{=} \overline{\texttt{out}_1}.Fork + \texttt{in}.Fork_{2e} \\
Fork_{2e} &\stackrel{\text{def}}{=} \overline{\texttt{out}_2}.Fork + \texttt{in}.Fork_{1e}
\end{aligned}$$

Isochronic forks can not be implemented as a separate circuit element like the non-deterministic fork above. Instead it is necessary to use synchronisation between internal state changes in the gates at each destination of a non-deterministic fork. The circuit level parallel to this is that not only should the wire delay to each destination of the isochronic fork be the same, the gates at the destination should switch at the same voltage levels.

Unfortunately this synchronisation adds to the complexity and doubles the state space of such gates. Consider the isochronic fork present in the simplified latch controller circuit from section 3.4.2. Here both latches must operate at the same time. This is modelled by each latch having a set of 'mirror' states which are entered upon an `enable` action in which a further `enable` action is not possible but the latches behaviour with regard to which other actions are permitted is unchanged. To allow a change in this behaviour, the other latch must supply a synchronising action. In this way it appears as if both latches are enabled or disabled at the same time, without affecting the need to allow quenching. The resulting code for the latch can be found in Appendix A.3. One noteworthy point is that the generation of the additional CCS code is methodical and could probably be automated, though the clarity of the model is lost. This is especially noticeable for the isochronic fork required between two asymmetric three-input C-elements in the four-phase latch controllers.

4.5.6 Composition example

This example illustrates *XOR* (from 4.4.5) driving a *datalatch*. The resulting model has input actions $\mathtt{in}_1, \mathtt{in}_2$ and output action $\overline{\mathtt{out}}$. Relabeling is used to 'connect' the XOR to the data latch and restriction is used to ensure that the two processes communicate to produce a silent τ action, that is to say the `w` action is not externally observable.

$$\mathit{Example} \stackrel{\mathrm{def}}{=} (\mathit{XOR}[\mathtt{w}/\mathtt{out}] | \mathit{Datalatch}[\mathtt{w}/\mathtt{in}])\backslash\{\mathtt{w}\}$$

4.6 Verification of latch controllers

This section discusses the verification of two and four-phase latch controllers using CCS and the relation of the CCS specifications to the STG specifications given in Chapter 3.

The CCS model of each latch controller circuit is verified by placing it in a correctly operating environment to simulate the previous and next stages in a micropipeline. The environment, see Appendix A.4, is modelled by two processes: *Env-L* which is always able to supply new data and *Env-R* which is always able to receive data. Observable actions $\mathtt{rin}, \overline{\mathtt{ain}}, \overline{\mathtt{rout}}, \mathtt{aout}$ are used to observe changes in the handshake signals to the latch controller and the observable action `capture` indicates when the latch controller instructs the multi-bit data latch to capture data. The environment processes for two-phase and four-phase circuits are identical. This is because the environment for a four-phase circuit, when modelled using events, performs the same sequence of actions, but at twice the rate as for a two-phase circuit to process the same number of items of data.

The verification, in particular claims to delay insensitivity with allowance for isochronic forks, is subject to the assumptions stated previously in sections 4.4.1 and 4.5.

4.6.1 Standard two-phase controller

A CCS model, given in Appendix A.6, of the standard two-phase latch controller (introduced in section 3.2) was constructed and shown to be observationally equivalent to the specification below (also given in Appendix A.5). This is consistent with the accepted fact that this circuit is delay insensitive.

$$\begin{aligned}
\mathit{SpecTwoPhase\text{-}L} &\stackrel{\text{def}}{=} \mathtt{rin}.\mathtt{sync}.\mathtt{capture}.\overline{\mathtt{sync}}.\overline{\mathtt{ain}}.\mathit{SpecTwoPhase\text{-}L}\\
\mathit{SpecTwoPhase\text{-}R} &\stackrel{\text{def}}{=} \overline{\mathtt{sync}}.\mathtt{sync}.\overline{\mathtt{rout}}.\mathtt{aout}.\mathit{SpecTwoPhase\text{-}R}\\
\mathit{SpecTwoPhase} &\stackrel{\text{def}}{=} (\mathit{SpecTwoPhase\text{-}L}|\mathit{SpecTwoPhase\text{-}R})\backslash\{\mathtt{sync}\}
\end{aligned}$$

4.6.2 Simplified two-phase controller

The CCS model, given in Appendix A.7, of the simplified two-phase latch controller (see section 3.4.2) was also shown to be observationally equivalent to *SpecTwoPhase* given above. This verifies that, assuming the isochronic fork discussed in section 3.4.2, the simplified controller is equivalent to the two-phase specification, and therefore is equivalent to the standard two-phase latch controller.

A version of the model without the isochronic fork was also produced. For this the CWB produces a distinguishing formula. The formula given below is satisfied by *SpecTwoPhase* but not by the circuit model.

$$\langle\langle\mathtt{rin}\rangle\rangle\,[[\mathtt{capture}]]\,\langle\langle\overline{\mathtt{ain}}\rangle\rangle\langle\langle\mathtt{rin}\rangle\rangle\langle\langle\overline{\mathtt{rout}}\rangle\rangle\,[[\mathtt{aout}]]\,\left[\left[\overline{\mathtt{ain}}\right]\right]\,\mathtt{ff}$$

At this point an informal explanation of part of the Hennessy-Milner logic is needed: If a process P satisfies the formula $\langle\langle\mathrm{a}\rangle\rangle\phi$ (pronounced 'diamond ϕ') then *there exists* a process Q which satisfies ϕ such that P can become Q by some sequence of actions which consists of a only once and an arbitrary but finite (and possibly zero) number of τ actions. If a process P satisfies the formula $[[\mathrm{a}]]\,\phi$ (pronounced 'box ϕ') then *for all* processes Q which can be reached from P via some sequence of actions which consist of a only once and an arbitrary but finite (and possible zero) number of τ actions, all Q must satisfy ϕ. All processes satisfy `tt` and no process satisfies `ff`.

Consider the above formula, with reference to Figure 3.12 the following sequence of events is possible. An `rin` arrives from the previous micropipeline stage and propagates through transparent latch L1 and the XOR resulting in a `capture` action. The signals continue to propagate such that L1 becomes opaque and L2 transparent. The latch controller produces $\overline{\mathtt{ain}}$ and the previous stage responds to produce a second `rin`. The latch controller also produces an $\overline{\mathtt{rout}}$. The next micropipeline stage responds to the $\overline{\mathtt{rout}}$ by producing `aout`, which will eventually cause L1 to become transparent and L2 to become opaque. However, if L1 becomes transparent whilst L2 is still transparent (the fork is not isochronic), then the waiting `rin` can pass through L1 and L2 to produce an $\overline{\texttt{ain}}$. The specification can not produce this $\overline{\texttt{ain}}$ without first producing `capture`. This corresponds to the formula above. The formula in fact states a stronger version of this because of the 'boxed' actions.

4.6.3 Equivalence to two-phase STG specification

Using a simple tool constructed to compute the state space of an STG and convert this to a CCS process description, the two-phase STG specification (see Figure 3.3) was converted to CCS and shown to be observationally equivalent to *SpecTwoPhase*. This acts as an extra check that the specification is correct. Appendix A.8 lists the conversion tool input and output.

4.6.4 Buggy four-phase latch controller

To test the CCS modelling technique, an attempt to confirm the existence of a timing assumption in the buggy semi-decoupled four-phase latch controller (see Figure 3.10) was made. The CCS model of this controller was tested against a four phase CCS specification converted from the STG of Figure 3.7. The CCS model assumes that the fork from `aout` to the two C-elements is isochronic. In [24] all forks are assumed isochronic as the circuit is speed independent but this is unnecessary. When tested for observational equivalence the distinguishing formula below is produced, this formula is satisfied by the specification but not by the circuit model.

$$\langle\!\langle \texttt{rin} \rangle\!\rangle \langle\!\langle \texttt{capture} \rangle\!\rangle \, [[\overline{\texttt{ain}}]] \, [[\texttt{rout}]] \, \langle\!\langle \texttt{aout} \rangle\!\rangle \langle\!\langle \texttt{rin} \rangle\!\rangle \, [[\overline{\texttt{ain}}]] \, \langle\!\langle \texttt{rin} \rangle\!\rangle \, [[\texttt{capture}]] \, \texttt{ff}$$

This expresses that the circuit model can perform the following sequence of actions (with reference to Figure 3.10).

1. An `rin` arrives from the previous micropipeline stage resulting in a `capture` action and the event propagates through the three-input C-element leading to an $\overline{\texttt{ain}}$ event.

2. The latch controller circuit continues to propagate the effect of the `rin`, which causes an $\overline{\texttt{rout}}$ to be sent to the next stage. The next stage acknowledges this by generating an `aout` event.

3. Since the previous stage's `rin` has been acknowledged by the circuit with an $\overline{\texttt{ain}}$, the previous stage can and does produce another `rin` (the high-to-low reset).

4. Next the circuit acknowledges the high-to-low `rin` with an $\overline{\texttt{ain}}$. This implies the `rin` must have propagated through the three-input C-element. Therefore, both the event propagating round the fork from the $\overline{\texttt{rout}}$ event and the event propagating along the fork caused by the `aout` event have reached the 'negative' inputs of the three-input C-element.

5. Because the circuit has acknowledged the high-to-low reset, the previous stage can and does produce another `rin` event (indicating new data).

At this point a problem exists. The circuit should not allow this `rin` event to propagate because the effects of the previous high-to-low reset transition may not yet have propagated fully. However, two of the three inputs to the C-element cannot prevent propagation of a rising transition on `rin`. This leads to possible interference (a high catching up with a low) on the wire leading to the two-input C-element. The specification does not allow the final `rin` to propagate through the three-input C-element. This non-persistent behaviour is stated in [24]; the confirmation of this provided some reassurance in the correctness of the CCS modelling method.

4.6.5 Fixed four-phase latch controller

Additionally a CCS model of the fixed semi-decoupled four-phase latch controller (section 3.3) was tested for observational equivalence with a CCS specification obtained from the STG of Figure 3.9. The same fork as in the previous section is assumed isochronic, but other forks are not. The circuit model and specification were found to be observationally equivalent.

4.7 Evaluation

The method is well suited to the verification of delay insensitive circuits, but as discussed earlier the clarity and conciseness of CCS is lost with the addition of isochronic forks. CCS would not therefore be well suited for verifying speed independent circuits. For larger or unconstrained circuits (for example a latch controller without a constraining environment) the state space 'explosion' is likely to make verification with the CWB prohibitive. It should be noted that this is a limitation of using the CWB to check properties not a limitation of CCS itself. Despite these problems, the tool support is very good and CCS and the associated logic are quick to learn and apply. Overall the CCS modelling method was found to be appropriate for the complexity and size of the circuits verified.

With the other modelling techniques discussed earlier (section 4.2) quenching is not permitted, and such interference between signals is unconditionally considered an error. In this technique quenching is allowed to occur, provided that this behaviour does not affect the circuit correctness when the circuit model is compared to a specification. To date evidence has not been encountered which suggests this reasoning is flawed. An example circuit, in which quenching can occur during correct operation, has not been found, without the example circuit containing redundant circuitry (which is removable at no loss to correction operation). From this it is possible to speculate that such a circuit may not exist. However, allowing quenching to occur is useful; it reduces the state space of a model and seems more 'natural', in that the circuit can do whatever it wishes internally, provided its externally observable behaviour is correct.

4.8 Summary

This chapter has introduced a method of modelling asynchronous circuits using the CCS process algebra. During the development of the final method several approaches were tried. In all of the approaches a circuit can be informally mapped to CCS as detailed below, and the CWB used to check properties of the model.

- Each circuit element is modelled with a corresponding CCS process.
- CCS input and output actions are used to represent the transmission of events from one circuit element to another.
- CCS parallel composition with restriction is used to 'connect' the models of circuit elements together. This can be done in either a hierarchical or linear fashion to reflect the modularity of the circuit. Once models of basic circuit elements have been designed the composition is straightforward.
- Processes acting as specifications are used to model the environment surrounding the circuit.
- Observational equivalence is used to compare a model against a specification.

Modelling with logic levels: Signals were modelled as the communication of high and low logic levels. This was found to be undesirable as the model can always exhibit activity (even whilst the circuit can not) and an unnecessary level of detail is involved.

Modelling with events: Events (transitions) are modelled instead of logic levels. This reduces the state space and the model only exhibits activity when the corresponding circuit does. By arranging the initial state appropriately there is no loss of generality. However, without further development there is a flaw, gate inputs in the model can block (prevent further input changes temporarily) whereas gate inputs in a real circuit can not. Such blocking is undesirable when testing for delay insensitivity.

Modelling with events and wires: A 'wire' process is placed between gates. The wire enters an error state to indicate when an event is about to be blocked by the gate following the fire. This increases the state space in the model and so the next method is preferred.

Modelling with events and quenching: With this style the process representing each gate never blocks its inputs. It permits quenching, where one input event can cancel a previous input event without an output event from the gate. This method differs from others in the literature in that quenching is permitted, provided that the externally observable interface is still equivalent to the specification. As an optimisation, quenching does not need to be permitted on a gate input when it is known that it will not occur, for example in the case of an input from a correctly behaving environment.

The modelling method has been successfully applied to the following circuits. The first two confirm existing known facts as a means of testing the method in practice, and the second two involve the verification of new circuit structures.

- Standard two-phase latch controller
- Buggy (and fixed) four-phase latch controller
- Simplified two-phase latch controller
- Circuits for full- and empty-detection of pipelines and multi-state FIFO's (see Chapter 5).

In theory the CCS technique could be applied to any complexity of circuit, but in practice the use of the CWB limits this to small or well constrained circuits which are close to being fully-delay insensitive. The circuits modelled in Chapter 5 demonstrate a possible method of including timing assumptions in the model, but at the expense of clarity and conciseness. In practice the tool support is good and the method was found to work well for the circuits studied and is expected to perform well for other circuits of similar complexity.

Chapter 5
Two-dimensional micropipelines

In this chapter the linear micropipeline structure, introduced in Chapter 2, is developed into a two-dimensional structure which can perform matrix transposition or convert between word-serial bit-parallel data and word-parallel bit-serial data. The circuit structure is an ideal application of the simplified latch controller developed in Chapter 3. The CCS modelling style from Chapter 4 is used to check properties of the circuit. Two ways of capturing timing assumptions within this modelling style are used, one a direct CCS approach involving examination of the state space, the other, somewhat simpler, involving STG specifications.

Two variants of the circuit are discussed, termed Method I and Method II. Method I was developed for parallel-serial conversion, prior to knowledge of independent work for matrix transposition by Tierno and Kudva [35]. Method II was developed upon re-examination of the circuit after reading the work by Tierno and Kudva. Both methods can be applied to either matrix transposition or parallel-serial conversion. Method I is modelled using the direct CCS approach, Method II is modelled using the STG based approach. Although not performed, it should be possible to model both circuit variants with either modelling method. A review of the two circuit variants and the work by Tierno and Kudva is presented.

Matrix transposition is similar in operation to that of parallel-serial conversion (each bit of data is replaced with a word of data). Matrix transposition is commonly used in the calculation of the 2-D DCT, see Chapter 6. This chapter focuses on the task of bit-parallel word-serial to bit-serial word-parallel conversion, but the methods apply to both the reverse conversion and matrix transposition.

5.1 Motivation and application

Figure 5.1 shows a typical setup for parallel to serial and serial to parallel conversion, where a bit-serial core is required to interface with a bit-parallel environment. Block I performs word-serial bit-parallel (n words of width w bits) to word-parallel bit-serial conversion, block O the reverse. An example application might be a bit-serial DCT, such as one based upon the Arai algorithm discussed in Chapter 6.

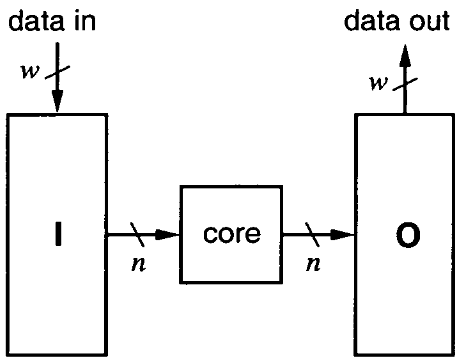


Figure 5.1: *Bit-serial core in a bit-parallel environment*

Typically a two-dimensional array of synchronous registers performs the conversion. This approach has several disadvantages:

- The two-dimensional array of edge-triggered registers requires a large area and consumes significant power.
- The input and output interfaces must both be synchronous (and with the bit-serial core).
- The data input and output rates are not constant, a burst of input and output is followed by a burst of processing.

Many DCT implementations use a DRAM block, with address counters, to perform matrix transposition, thus avoiding the large area of a two-dimensional array of registers. The DRAM method is slower because only one word of data can be accessed at a time, but some improvement can be made by using a dual-port DRAM with concurrent read and write access. For parallel to serial conversion the DRAM method would involve reading and writing one bit at a time and would hence be too slow.

5.2 Architecture

5.2.1 Concept

An alternative scheme is shown in Figure 5.2. The parallel to serial conversion block is based on two micropipelines, a vertical pipeline IV for reading the bit-parallel input data and a horizontal pipeline IH for providing bit-serial output. The data is kept in a two-dimensional array of level-sensitive latches IA. Each latch cell in the array can be supplied with data from the cell to the left (except those in the left hand column) or from the cell above.

The optional column of latches IX may be used to sign extend the bit-serial output concurrently with new data entering IV. The optional linear micropipeline buffer IB buffers the input, permitting a constant data input rate. The data processing core remains synchronous, the clock for which is generated by global control circuitry coordinating the operation of all the blocks shown in Figure 5.2.

The serial to parallel conversion block has a similar structure, with horizontal pipeline OH to read bit-serial data from the core processor and vertical pipeline OV to supply bit-parallel output. The optional linear micropipeline buffer OB permits a constant data output rate. Circuits II and OI allow the handshake signals at the top of the vertical pipelines to be 'isolated' from the buffers; wide pass-transistors are suitable for this.

Pipeline IV should have an even number of stages if IB is used, otherwise the handshake signals between IV and IB will be out of phase if the two pipelines are isolated, IV is reset and then IV and IB are reconnected. A similar argument holds for OV and OB.

Pipelines IH and OV produce an `empty` signal which indicates when there is no data held in the pipeline, likewise pipelines IV and OH produce a `full` signal which indicates when the pipeline is full and, in the case of Method I, there is a request waiting at the input of the pipeline.

Each pipeline has a reset input, for all pipelines except IH and OV this would reset the pipeline into an empty state; the reset inputs to IH and OV reset the pipelines to a full state where each stage is initialised holding data. The reset input follows a lower bounded delay model, upon issuing a reset a lower bounded amount of time must pass before the reset input can be removed and the pipeline used. A global control circuit, not shown in the diagram, is required to coordinate the activity of the various circuit blocks.

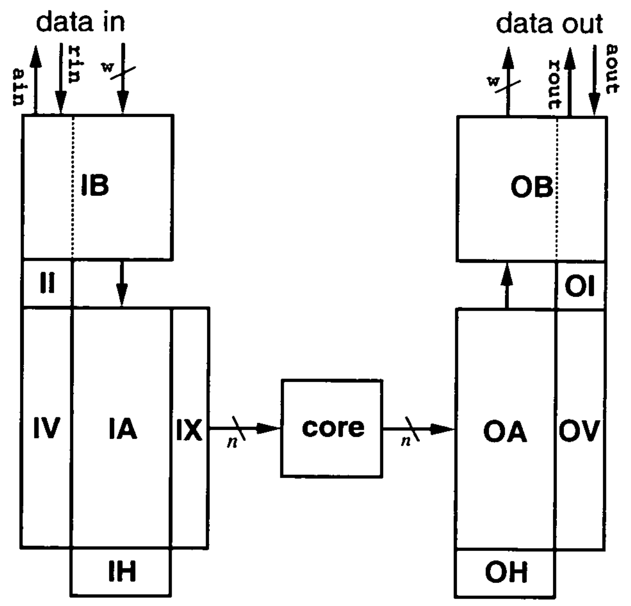


Figure 5.2: *Parallel-to-serial, processing core and serial-to-parallel*

This scheme has several advantages when compared to the synchronous shift-register approach:

- The use of transparent latches leads to an area and power saving.
- The input and output interfaces are asynchronous.
- The input and output data rates may be constant, provided suitable buffer sizes at the top of each converter block are chosen.

In principle, the input and output interfaces are asynchronous; for applications where the surrounding environment is synchronous this is likely to be a disadvantage. Since the data processing time is constant, it may be reasonable to assume a bounded-delay model for the input and output interfaces. The simulations required to ensure this assumption are likely to be made easier by the fact that the blocks are composed from regular cells. This is to say that, for each request to input data (by providing an event to `rin` on IB) there will be a bounded delay before the `ain` from IB occurs. The worst case for this delay can be found by simulation, in a manner similar to that used to find the maximum clock frequency for the synchronous equivalent. Hence the `ain` from IB can be ignored (unconnected) and the `rin` to IB treated as an input clock. Care is needed to ensure that IB is large enough to allow a continuous constant frequency input clock, or a scheme of stopping and starting input would be needed. A similar argument holds for the output interface, where `aout` from OB would be used as an output clock.

Again a similar arrangement can be used for the reading of data from IH by the core and for the writing of data to OH from the core. The highest core clock frequency is limmited by the slowest of the worst case response times of IH to `aout` and OH to `rin`. Alternatively and perhaps preferably because it involves fewer timing assumptions, `rout` from IH and `ain` from OH may be used to control the generation of each clock cycle.

Note that `rout` from IV and `ain` to OV are unconnected and that `aout` to IV and `rin` to OV are kept at logic zero. These interfaces are referred to from now on as a left and right hand *null environment.*

5.2.2 Operation

Typical operation of the system in Figure 5.2 is summarised below. The overall operation appears complex because there is much concurrent activity, the description below lists one possible sequence of operation.

1. IH and OV are initialised to be full and IV and OH are initialised to be empty. OV is isolated from OB. Additionally if this is the first cycle of operation, IB and OB are initialised empty.
2. Data flows into IB and from there into IV. Concurrently, the previous output (if not the first cycle) from the bit-serial core which is held in OV flows into OB and is output. Because IB and OB are elastic micropipelines data can continue to arrive into IB and leave OB.
3. When IV is full and OV is empty, the handshake signals between IB and IV are isolated and the core clock is started. Data is read from IH, processed and written into OH.
4. The clock driving the `aout` input to IH is stopped whilst the core continues to process and read sign extended bits.
5. Once the core has read enough sign extended bits, IH can be reinitialised to be full, IV reinitialised empty and new data permitted to flow from IB into IV. Concurrently data is permitted to leave OV and enter OB. If IX is used then IV can be reinitialised and reconnected to IB before all the sign extended bits have been read.
6. Once OV is empty and IV is full a new cycle of processing starts.

5.2.3 Two-dimensional micropipeline

The circuit for a two-dimensional micropipeline, equivalent to the parallel to serial converter (blocks IV, IH, IA) is shown in Figure 5.3. Note that reset circuitry and the sign-extender IX is not shown. Blocks FD and ED detect when IV is full (*full-detection*) and IH is empty (*empty-detection*). Two variants of the circuits for full- and empty-detection are discussed later.

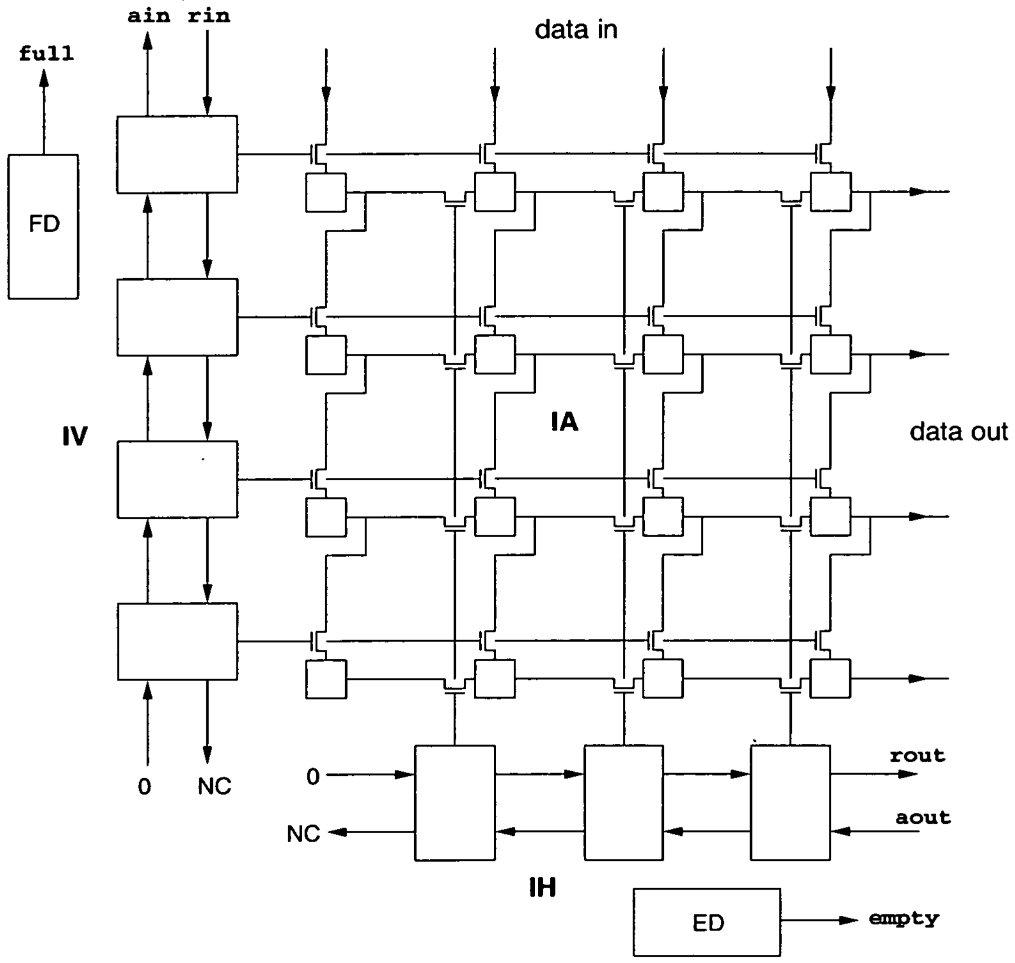


Figure 5.3: *Two-dimensional micropipeline*

An alternative to using empty-detection in block IH is to connect an inverter between ain and rin of the left most stage such that the horizontal pipeline always stays full. The number of items of data read from IH would then need to be counted, this might be an existing function of the processing core. This avoids having to reset IH to be full for each cycle of operation. However, the reset circuitry would still be needed to ensure correct operation after power on.

An alternative approach to full- and empty-detection would be to use a counter to count the number of events requesting input, or requesting output. This is, however, a somewhat less elegant and more expensive approach. A counter would need to be placed between the buffering pipelines and the vertical pipelines. Since the complexity of a counter is much higher than one of the pipeline stages, it would seem reasonable to assume that the highest data rate would be restricted by the counter rather than by the pipeline.

It should be noted that the simplified two-phase latch controller, from section 3.4.2, is the ideal choice of latch controller, because it is used here in a simple linear pipeline (see section 3.5).

5.3 Method I – circuit

Although the circuits for full- and empty-detection are dissimilar for this method, they are grouped together here as they were developed at the same time and the same modelling style is used (see section 5.4) to verify both circuits.

5.3.1 Full detection

Full-detection is concerned with deciding when an initially empty micropipeline has become full and can not hold any more data. A single pipeline stage can not 'know' when the whole pipeline is full, this is because all valid circuit states, for a single stage, can occur during normal operation before the pipeline is full. A simple way to implement full-detection might be to define the pipeline to be full when all stages are holding data (occupied) and hence no data latches are transparent. However, consider an arbitrary length pipeline with a right hand null environment, into which one item of data is inserted into the left hand side. It is possible, although unlikely for a non-trivial number of stages, that each stage will capture data and produce an `rout` to the next stage until the data reaches the right most stage. No stage has yet acknowledged receipt of data to the stage to the left. There is only one item of data in the pipeline, yet each stage in the pipeline is occupied and so the pipeline, by the current definition, is full. A similar argument can be made for any number of data items being inserted into the pipeline. A more sophisticated mechanism is needed and one such mechanism is introduced next.

The simplified two-phase latch controller with circuitry to detect *request-pending* is shown in Figure 5.4. Note that the right most stage in the pipeline which will be connected to a right hand null environment may be simplified.

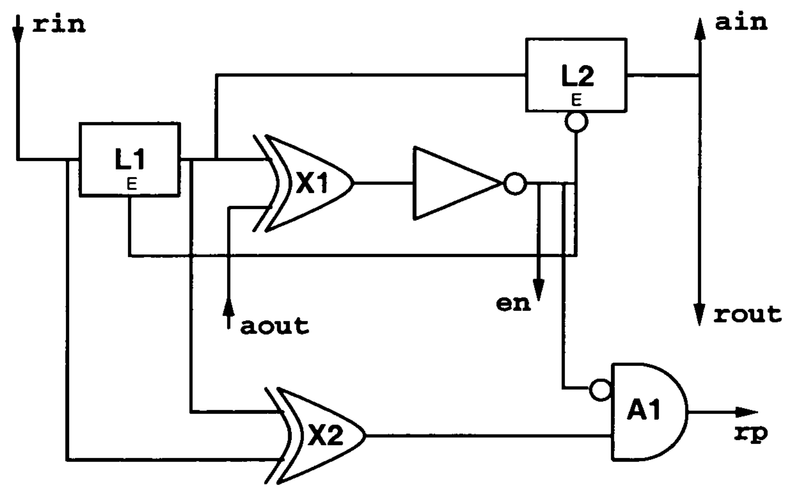


Figure 5.4: *Two-phase latch controller with request-pending*

The output rp (request-pending) is high when the pipeline stage is occupied and there is a request waiting on rin. The XOR gate X2 is used to detect that there is a request event waiting on rin which cannot pass through L1. The XOR output may, however, also glitch high if L1 is transparent as a request event passes through L1. To counteract this glitch A1 is used to ensure that not only is there a request waiting, but the stage is occupied. It is not possible for an event to pass through L1 and cause a glitch in the XOR output whilst the stage is occupied, because L1 would not be transparent. Nor is it possible for such a glitch to occur between the data latch becoming occupied and L1 being disabled. This is because such a request on rin can only occur in response to an ain and this ain cannot occur until L1 and L2 have acted at the same time (by definition of the isochronic fork to the enable inputs of L1 and L2). It is possible to replace X2 with an inverter and AND gate to give a smaller implementation, as the actual logic values at the two sides of L1 at the time of a request-pending are known and alternate between even and odd stages.

The circuitry, to detect request-pending, does not interfere with the normal latch controller operation, because it acts strictly as an ‘observer’. A consequence of this is that the request-pending circuitry is not delay insensitive. For example, suppose the latch controller reaches a state where the request-pending signal should change from logic low to high. This change could occur after an arbitrary delay after the latch controller reaches this state. This potential change could, but not necessarily, be quenched if the latch controller subsequently changed state such that the request-pending signal should return low. Likewise for an initial change from logic high to logic low.

For a single stage pipeline, with a right hand null environment, the stage will indicate that there is a request-pending (`rp` high) upon the second `rin`. Since there is only one stage this request-pending signal is directly used to indicate the pipeline is full. Note that the single stage pipeline is only defined to be full when the second, unable to be processed, `rin` is received.

For n stages (n an integer > 1) the circuit shown in Figure 5.5 is used. The signal $\texttt{full}_{\texttt{i}}$ is the AND of $\texttt{full}_{\texttt{i-1}}$ and the request-pending signal $\texttt{rp}_{\texttt{i}}$. The right most stage is connected to a null environment and $\texttt{full}_{\texttt{1}}$ is simply $\texttt{rp}_{\texttt{1}}$.

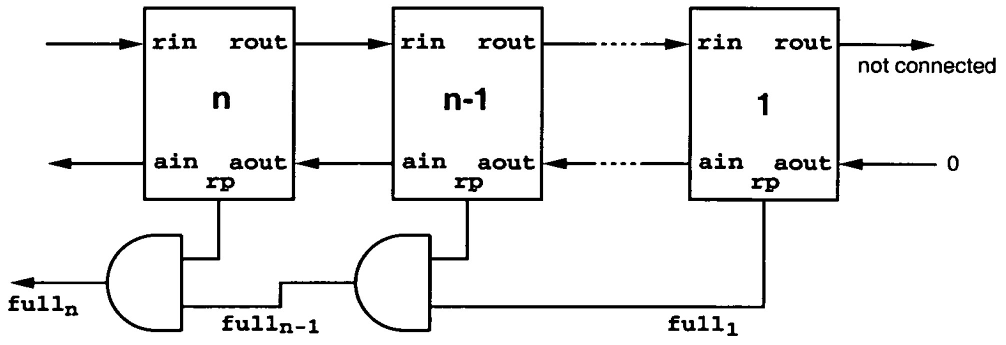


Figure 5.5: *Micropipeline with request-pending full-detection*

For the correct detection of a full pipeline, the indication of full must eventually occur if the pipeline is full and there is a request waiting at the pipeline input. This indication of full must never occur erroneously, even momentarily, before then.

Whilst data is being passed from left to right through the pipeline it is possible a stage i may set $\texttt{rp}_{\texttt{i}}$ high and then low again. This can occur if stage $i-1$ has not yet acknowledged a request from stage i and stage i receives a request from stage $i+1$. This will not matter, provided stage i cancels $\texttt{rp}_{\texttt{i}}$ before there exists the possibility for $\texttt{full}_{\texttt{i-1}}$ to occur, if not then $\texttt{full}_{\texttt{i}}$ could be generated erroneously.

Therefore without any timing assumptions it is clear that the circuit will not work correctly. Consider this scenario with an n stage pipeline: All the stages to the right of stage n are occupied, stage n is occupied with the same data as stage $n-1$, as stage $n-1$ has not yet acknowledged to stage n and there are $n-1$ pieces of data in the pipeline. Stage n receives the nth request from the left hand environment and signals $\texttt{rp}_{\texttt{n}}$. Stage $n-1$ then acknowledges to stage n, but $\mathtt{rp_n}$ does not yet become low. Stage n sends a request to stage $n-1$, the stages to the right of stage n can now correctly signal $\mathtt{full_{n-1}}$ which, combined with the still high $\mathtt{rp_n}$, will produce $\mathtt{full_n}$. This is incorrect, $\mathtt{full_n}$ should not be produced until the $(n+1)$th request is received from the left hand environment, $\mathtt{rp_n}$ is still free to become low and when it does $\mathtt{full_n}$ will also return low.

Note that this sequence of events also demonstrates that the possibility of using just the data latch enable signal without X2 in stage n, with an aim to signalling $\mathtt{full_n}$ after n requests from the left hand environment, will not work correctly either. It is therefore necessary to have $n+1$ requests before $\mathtt{full_n}$ is produced.

The behaviour of each stage i assumes that the $\mathtt{full_{i-1}}$ signal is generated correctly without glitches, therefore stage i must ensure that $\mathtt{full_i}$ is generated correctly. As shown above, without timing assumptions, $\mathtt{full_i}$ may be generated erroneously. In the example scenario above, this error occurs because a stage i is allowed to keep $\mathtt{rp_i}$ high whilst generating a request to stage $i-1$. If the assumption is made that, by the time stage i can generate an `rout`, $\mathtt{rp_i}$ has returned low (unless there really is a request-pending and it is not the case that $\mathtt{rp_i}$ is high because it has not returned low yet) then the problem is avoided.

Restating the above assumption, for a single pipeline stage, `rp` must become low before the stage produces an `rout` in response to the waiting `rin` when `aout` is received. The `aout` will cause the data latch to become transparent and will indicate to A1 that `rp` should become low. Once the data latch is transparent L1 will also become transparent, indicating to X2 that `rp` should become low and the data latch will then capture again, possibly quenching the previous change to the inverted input of A1. If the delay through X2 and A1 is less than the delay through X1 and the data latch and the delay for L2 to act, then `rp` will become low before `rout` is produced. Driving the data latch involves a large load, and so this assumption would seem reasonable. Furthermore, if X2 is replaced by an inverter and AND, with a shorter delay than an XOR, this is in favour of the assumption. Finally, what correct operation really assumes, is that `rp` is low before the stage to the right is able to act on `rout` and produce a full signal. There will be a delay, in addition to the margin of safety already attained, in the stage to the right acting on `rout`. This delay is comparable to the delay in `rp` becoming low as the same circuit arrangement is involved.

However, making this assumption does not imply that there is not another sequence of events by which the circuit can produce an erroneous full signal. To be sure of this it is necessary to consider all possible states the circuit can be in and to check that the final full signal is only generated once. This should be immediately before the pipeline deadlocks when it cannot accept more input and after the correct number of requests have arrived from the left hand environment. To examine this state space the modelling method from Chapter 4 is used; this is discussed in section 5.4.3.

5.3.2 Empty detection

Empty-detection is concerned with deciding when an initially full micropipeline (but without a request waiting at the input) has emptied such that no more data can be read from the pipeline. Compared to full-detection, detecting that the pipeline is empty is more straightforward. Whereas for full-detection all stages can be occupied even if the pipeline contains only one distinct item of data, a pipeline for which all stages are unoccupied (and hence transparent) cannot contain any data and is therefore empty. The simplified two-phase latch controller with an `em` output (empty signal, or more accurately, unoccupied signal) is shown in Figure 5.6.

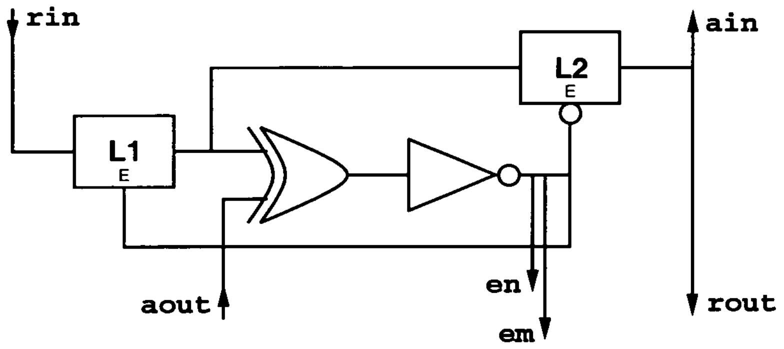


Figure 5.6: *Two-phase latch controller with 'empty' output*

An n stage pipeline is shown in Figure 5.7. The AND of the `em` signals from stage i and the empty signal $\texttt{empty}_{i-1}$ forms the empty signal $\texttt{empty}_i$, $\texttt{empty}_1$ is simply $\texttt{em}_1$. As with full-detection, the circuitry to detect the empty condition does not interfere with normal operation and is therefore not delay insensitive. The $\texttt{empty}_n$ signal should become high if and only if all the stages are unoccupied.

Without any timing assumptions the circuit will not work correctly. The scenario with an n stage pipeline is as follows: All pipeline stages are unoccupied except the right most two stages

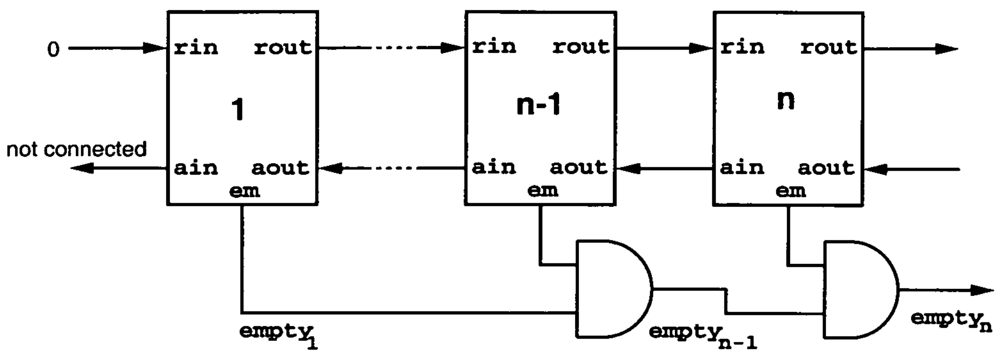


Figure 5.7: *Micropipeline with* em *empty-detection*

n and $n-1$. The right hand environment delivers an aout to stage n. Stage n signals it is unoccupied by making $\mathtt{em_n}$ high. Stage n responds to the waiting request from stage $n-1$ and captures new data, however it does not yet return $\mathtt{em_n}$ to low (for example a long wire delay). Stage n then acknowledges to stage $n-1$ which can then signal $\mathtt{empty_{n-1}}$, this combines with the high $\mathtt{em_n}$ to give a high on $\mathtt{empty_n}$, even though there is one item of data still in stage n. Eventually $\mathtt{em_n}$ will go low and $\mathtt{empty_n}$ will return low.

To avoid this problem, the assumption is needed that $\mathtt{em_n}$ has returned low, before stage n acknowledges receipt of the waiting request to stage $n-1$. This assumption on its own is somewhat 'tight'. Within a single stage em would be required to become low before an ain is produced. This involves a race between the latch enable to L2 permitting the ain to occur and the same latch enable signal reaching the AND gate connected to em. This is the assumption that will be made for the purposes of verification later as the assumption is localised to a single pipeline stage. In practice, however, stage $n-1$ will take some time to respond to the ain, as an event on aout must pass through X1 and the data latch before being able to influence em and hence $\mathtt{empty_{n-1}}$.

Again, making this assumption does not imply that there is not another sequence of events by which the circuit can produce an erroneous empty signal. Once again, verification using formal methods is used to test this; this is discussed in section 5.4.4.

5.4 Method I – modelling

In this section CCS is used to verify the mechanisms described earlier for full- and empty-detection. The term 'the timing assumption' refers interchangeably to the timing assumption described in section 5.3.1 for full-detection or in section 5.3.2 for empty-detection.

5.4.1 Overview

The method used to confirm the correct operation of full- and empty-detection involves the creation of a model for a single pipeline stage. Several of these pipeline stages are then combined and composed with a model of a suitable 'chain' of AND gates. The resulting composition is tested against a simple specification which is 'correct by inspection'.

Three approaches were identified for how to model a pipeline stage consisting of a simplified two-phase latch controller and full- or empty-detection circuitry.

1. Model the circuit at a gate level by composing a CCS process from processes describing each circuit element.
2. Extend the specification for a two-phase latch controller (*SpecTwoPhase* from section 4.6.1) to include `rp` or `em` actions.
3. Create a new 'high-level' model of a pipeline stage including request-pending or unoccupied detection.

The first approach suffers from two drawbacks. The first, a potential problem, is state space. Several new circuit nodes have been introduced with the circuitry to generate `rp`. In particular these nodes are not synchronised with the original latch controller circuitry giving rise to a larger state space. The generation of `em` is simpler. However, in both cases the correctness of the original latch controller circuit would be re-verified, this is unnecessary as the additional circuitry does not synchronise with and hence can not interfere with the latch controller part (assuming an absence of implementation issues such as fan-out). The second drawback is that for both full- and empty-detection it would be hard to incorporate the timing assumption without 'hacking-on' some CCS to ensure the assumption is maintained throughout all possible states in the model.

The second approach would seem the most attractive. In this the specification of a two-phase latch controller operating in a correct environment would be extended to include the generation of rp and em actions. The specification *SpecTwoPhase* is composed from two processes, one which models the left-hand (input) interface of a pipeline stage and another which models the right-hand (output) interface. These left and right-hand processes synchronise in a straight-forward manner corresponding to the capturing of data. This synchronisation does not vary depending on if there is a waiting request or not. Generating rp and em actions involves a more complex synchronisation between left and right-hand sides if the timing assumption (for example relating rout and rp) is to be met. To state when rp or em *may* change and when they *must* change by, requires an examination of the state space for a pipeline stage. This leads to the third approach.

The third approach is to examine the state space for a standard two-phase pipeline stage (a stage without full- or empty-detection) operating in a correct environment and annotate this to indicate when rp or em may change and must change by. Furthermore, the timing assumption only involves ensuring that rp or em are at a logic low (rather than high as well) by a particular time. The pipeline stage is modelled as two processes, one which is derived directly from the state space of *SpecTwoPhase* and one, discussed next, which aids with the modelling of the timing assumption.

5.4.2 Modelling the timing assumption

The process *EL*, given below and in Appendix A.9, can be thought of as a set-reset flip-flop which converts events into a logic level. Upon a lo action the output, represented by action $\overline{\texttt{out}}$, can be thought of as becoming or remaining at logic low, and likewise *hi* for logic high. Note that the output is still modelled using the event based modelling method discussed in section 4.4.3. The action $\overline{\texttt{losync}}$ is to provide a synchronisation which can only happen when the output state is a logic low.

$$\begin{aligned}
EL &\stackrel{\text{def}}{=} EL_0\\
EL_0 &\stackrel{\text{def}}{=} \texttt{lo}.EL_0 + \texttt{hi}.EL_{0e} + \overline{\texttt{losync}}.EL_0\\
EL_{0e} &\stackrel{\text{def}}{=} \texttt{lo}.EL_0 + \texttt{hi}.EL_{0e} + \overline{\texttt{out}}.EL_1\\
EL_1 &\stackrel{\text{def}}{=} \texttt{lo}.EL_{1e} + \texttt{hi}.EL_1\\
EL_{1e} &\stackrel{\text{def}}{=} \texttt{lo}.EL_{1e} + \texttt{hi}.EL_1 + \overline{\texttt{out}}.EL_0
\end{aligned}$$

In use EL is composed with another process, P, representing a latch controller and the composition is restricted with the set {`lo`, `hi`, `losync`}. EL always accepts `hi` and `lo` actions allowing P to freely request that the output may become high or low regardless of the current state of the output. Action $\overline{\texttt{losync}}$ may only occur when the output is low and is going to remain low until a `hi` occurs. Process EL hence supplies an output which when requested to change can change 'at any time' thereafter, but with the feature of P being able to block until the output is low and will remain low until otherwise requested. This blocking of P removes the states from the model for which the timing assumption is not met.

5.4.3 Full-detection

To model full-detection a model of a single pipeline stage including request-pending is created, and then several instances of this are composed with a model of the AND gate chain to form a pipeline with full-detection.

5.4.3.1 Single stage model

The model of a single pipeline stage, see Figure 5.8, is intended to be equivalent to the circuit shown in Figure 5.4. In this model the output of the inverting buffer following X1 passes through the data latch (see discussion in section 3.2 about positioning of the data latch), into a non-isochronic fork to the request-pending detection circuitry and the isochronic fork between L1 and L2.

Figure 5.8 shows the state space of process LCU, a two-phase latch controller which is initially unoccupied of data. The CWB code for LCU can be found in Appendix A.10. To create LCU the process *SpecTwoPhase* was minimised with the CWB "min" command and re-entered by hand to have simple state names. To guard against errors being introduced LCU was then checked for observational equivalence to *SpecTwoPhase*.

By inspection, there are two places in the state graph where request-pending (`rp`) can change to logic high, after an `rin` event when in state LCU_4 or LCU_6. Note that LCU_1 is a state in which an `rin` has occurred, either the `rin` has just arrived from the left hand environment (edge from LCU to LCU_1), or there was a `rin` waiting (edge from LCU_8 to LCU_1). Both LCU_4 and LCU_6 are states in which a previous `rin` has already occurred (causing the stage to become occupied) but an $\overline{\texttt{aout}}$ (causing the stage to become unoccupied) has not yet occurred. There are two places, rather than one, because of the different possible orders in which $\overline{\texttt{rout}}$, $\overline{\texttt{ain}}$ and `rin` may occur. To indicate that `rp` may become high a $\overline{\texttt{hi}}$ action is inserted after the `rin` in both places and *LCU* is composed with *EL* as described in section 5.4.2. The addition of $\overline{\texttt{hi}}$, $\overline{\texttt{lo}}$ and $\overline{\texttt{losync}}$ described below, are shown in the state diagram of the new process *LCUrp* in Figure 5.9. The CWB code for *LCUrp* is given in Appendix A.10.

Again by inspection, there is one place where `rp` can change to logic low. When in state LCU_8 there is a request-pending condition and upon the `aout` which must follow, the latch controller will return to being unoccupied and hence `rp` can be cleared. A $\overline{\texttt{lo}}$ action is inserted after the `aout`.

To meet the timing assumption `rp` must change to logic low before $\overline{\texttt{rout}}$ can occur when in state LCU_2 and LCU_4. This is done by the insertion of a `losync` action which can not occur unless `rp` is low and will remain low until a `hi` action. The existing definition of LCU_2 is:

$$LCU_2 \overset{\text{def}}{=} \overline{\texttt{rout}}.LCU_3 + \overline{\texttt{ain}}.LCU_4.$$

The obvious way to add `losync` is to change this definition to become:

$$LCU_2 \overset{\text{def}}{=} \texttt{losync}.\overline{\texttt{rout}}.LCU_3 + \overline{\texttt{ain}}.LCU_4.$$

However, there is a problem with this. Provided `rp` is low, or can become low and will remain low, `losync` can freely occur (see discussion below). This `losync` is observable as a τ action from the composition $LCU|EL$. This τ action can 'just occur' and after doing so the choice between $\overline{\texttt{rout}}$ and $\overline{\texttt{ain}}$ is restricted to just $\overline{\texttt{rout}}$. This means that the internal communication to ensure `rp` is low would be able to restrict the state space representing the latch controller operation. What is needed is a way to express the property that $\overline{\texttt{rout}}$ can occur only after `losync` has occurred, but that if `losync` does occur the $\overline{\texttt{ain}}$ choice is still available, as if the `losync` had not occurred. This can be done by using the definition below, a similar arrangement is used for extending the definition of LCU_4.

$$LCU_2 \overset{\text{def}}{=} \texttt{losync}.(\overline{\texttt{rout}}.LCU_3 + \overline{\texttt{ain}}.LCU_4) + \overline{\texttt{ain}}.LCU_4.$$

The addition of both $\overline{\texttt{lo}}$ and $\overline{\texttt{hi}}$ actions does not influence the latch controller operation, this is because *EL* always permits the complementary `lo` and `hi` actions to occur.

The `losync` synchronisation removes states from the model which correspond to circumstances when the timing assumption is not met. Assuming that the single stage model, $LCUrp$, is correct enough such that the `losync` synchronisation can not occur following a $\overline{\texttt{hi}}$ (which would result in deadlock and hence be detected) `losync` may freely occur once the `rp` output is at logic low. This freedom is ensured because the `rp` output is 'connected to' an AND gate process (part of the full-detection circuit outside the pipeline stage) which always permits input events (in order to permit quenching) and hence the AND gate can not block `rp` and in turn block the latch controller operation. For this reason in the $LCUrp$ process `losync` can be treated as a free to occur τ action when in state $LCUrp_2$ or $LCUrp_4$, as was assumed previously.

In summary, the use of *EL* and `losync` provides a way of ensuring that the input to the AND gate driven by `rp` has become low before particular states in the latch controller can occur (hence modelling the timing assumption) and without the additional circuitry restricting operation of the latch controller.

The $LCUrp$ process does not always accept input events `rin` and `aout`, hence breaking the requirement from Chapter 4 that circuit inputs should not block. However, this does not matter here, as it is already known that the latch controller implements the two-phase protocol correctly. The controller hence provides a correct left and right-hand environment for adjacent stages and therefore can not produce output events which would cause quenching on the inputs of adjacent stages.

The final model of a pipeline stage $LCURP$ is composed from $LCUrp$ and *EL*:

$$LCURP \stackrel{\text{def}}{=} (LCUrp|EL[\texttt{rp}/\texttt{out}])\backslash\{\texttt{lo}, \texttt{hi}, \texttt{losync}\}$$

To confirm that $LCURP$ still operates correctly as a latch controller, $LCURP[\tau/\texttt{rp}]$ was tested and found observationally equivalent to *SpecTwoPhase*. This confirms that the addition of `lo`, `hi` and `losync` does not interfere with correct operation of the latch controller, provided that the `rp` output is not blocked, that is to say that the $\overline{\texttt{rp}}$ action is free to occur. Note that this test does not confirm that the generation of `rp` is correct.

To test that `rp` generation is correct a single stage pipeline connected to a right hand null environment was modelled and shown to be equivalent to the specification:

$$SingleStageSpec \overset{\text{def}}{=} \mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\overline{\mathtt{rp}}.0$$

Whilst this shows that $\overline{\mathtt{rp}}$ correctly occurs when the single stage pipeline is full, it does not confirm that the correct timing assumption is being modelled. Modelling with multiple pipeline stages can not test this either. It is possible for an error introduced during the construction of the $LCURP$ process to model a more restrictive timing assumption than the one intended and the model would still confirm correct circuit behaviour. In a previous attempt, prior to use of the EL process, the state space of LCU was extended by hand directly to give the $LCURP$ process. Due to the many orderings in which signal changes may occur this manual method is error prone and the first attempt implemented a more restricted version of the timing assumption than was necessary. The use of the EL process was then chosen as a way to simplify this.

5.4.3.2 Multiple stages

To test that full-detection operates correctly, several pipeline stages are modelled along with a chain of AND gates and a right hand null environment as shown in Figure 5.5. This was performed for two, three, four and five pipeline stages. Appendix A.11 contains the CWB code for a three stage pipeline. The three stage pipeline was shown to be observationally equivalent to the specification:

$$ThreeStageSpec \overset{\text{def}}{=} \mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\overline{\mathtt{rp}}.0$$

Whilst this verifies full-detection for the number of stages tested, strictly speaking it does not show that it will work for an arbitrary number of stages, however likely or 'obvious'. Such a check can not be performed with the CWB as the CWB expands the state space of the model and so it must be of limited size.

5.4.3.3 Request-pending with only XOR

A further optimisation which could be tested is to remove the AND gate A1 from the request-pending detection circuit leaving just X2. This was found (using the CWB) not to work correctly for even a two stage pipeline. The pipeline is capable of erroneously indicating it is full because each stage can signal request-pending as a request is passing across latch L1 and these signals combine in the AND gates to produce a glitch on the `full` output. If further timing assumptions were introduced it would be possible to remove A1 but this has not been investigated. It was decided that the circuit with `A1` provided a good balance between circuit complexity and timing assumptions. On a similar note, it might be possible to create a circuit which does not have the timing assumption, for which the latch controller operation is influenced by the request-pending detection. However, this would probably require an isochronic fork between the pipeline stage and the AND gate chain in order to 'know' when the AND gate input is low and would probably incur additional circuitry.

5.4.4 Empty-detection

Modelling and verification of the empty-detection circuitry, from Figure 5.7, is performed in an identical manner as with full-detection. The main difference is that the pipeline must be initialised to be full, but without a waiting request on the left most pipeline stage. Two single stage pipeline models are hence required, one for the left most stage in the pipeline and one for the other stages, differing only in initial state. As with full-detection the EL process is used to supply the `em` output and similar care is required with the insertion of `losync` actions.

Figure 5.10 shows the state space of process LCO, an initially occupied latch controller with a request waiting from the stage to the left. By inspection the positions where $\overline{\texttt{lo}}$, $\overline{\texttt{hi}}$ and `losync` should be inserted are found. After an `aout` is received the latch controller becomes unoccupied and so `em` should go high, therefore after each `aout` a $\overline{\texttt{hi}}$ action is inserted. The timing assumption states that `em` should be low before $\overline{\texttt{ain}}$ occurs, provided the stage really is occupied rather than the $\overline{\texttt{ain}}$ is just slow in occurring. Therefore `losync` is inserted before the $\overline{\texttt{ain}}$ from state LCO_2 and LCO_4. The state space for the resulting process $LCOem$ is shown in Figure 5.11.

The final model of a single stage, $LCOEM$, is composed from $LCOem$ and EL and was tested for observational equivalence with *SpecTwoPhaseFull* (*SpecTwoPhase* with a different initial state). The CWB code for this and the other processes involved in empty-detection is given in Appendix A.12. In a similar manner to full-detection, empty-detection was tested for one to five pipeline stages. The three stage pipeline was shown to be observationally equivalent to the specification:

$$\mathit{ThreeStageSpec} \overset{\text{def}}{=} \texttt{aout}.\overline{\texttt{rout}}.\texttt{aout}.\overline{\texttt{rout}}.\texttt{aout}.\overline{\texttt{empty}}.0$$

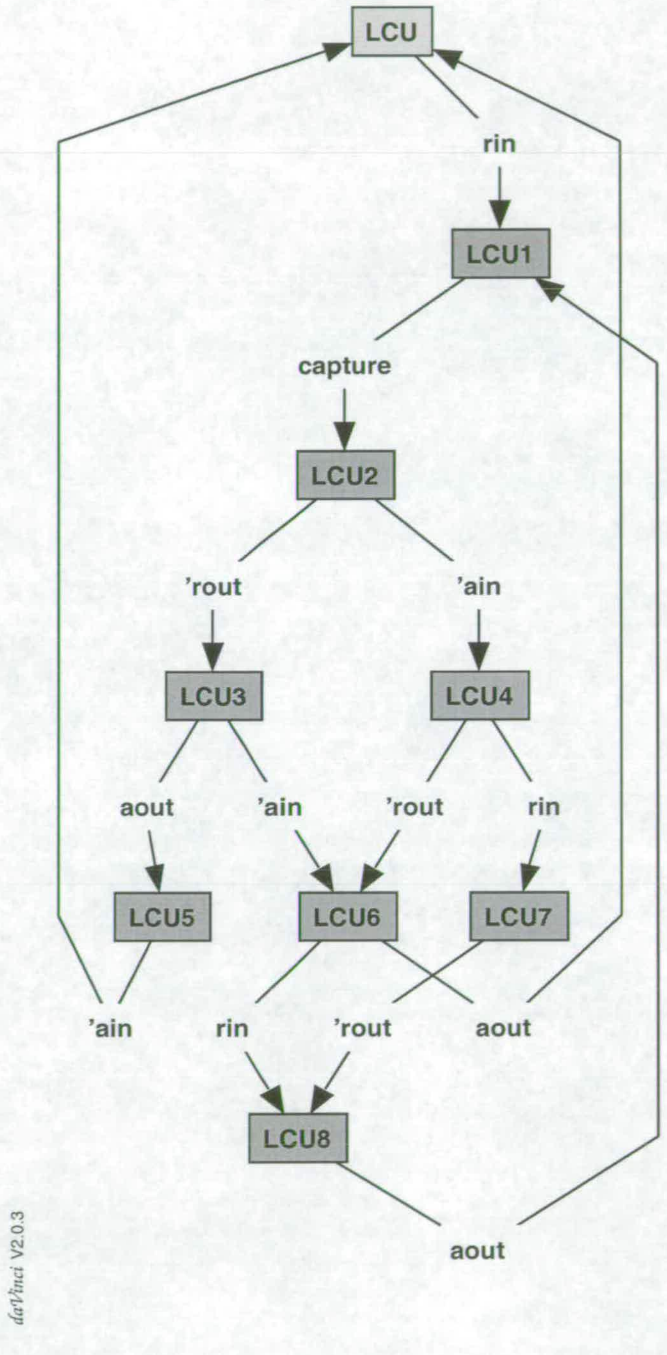


Figure 5.8: *State space of initially unoccupied latch controller*

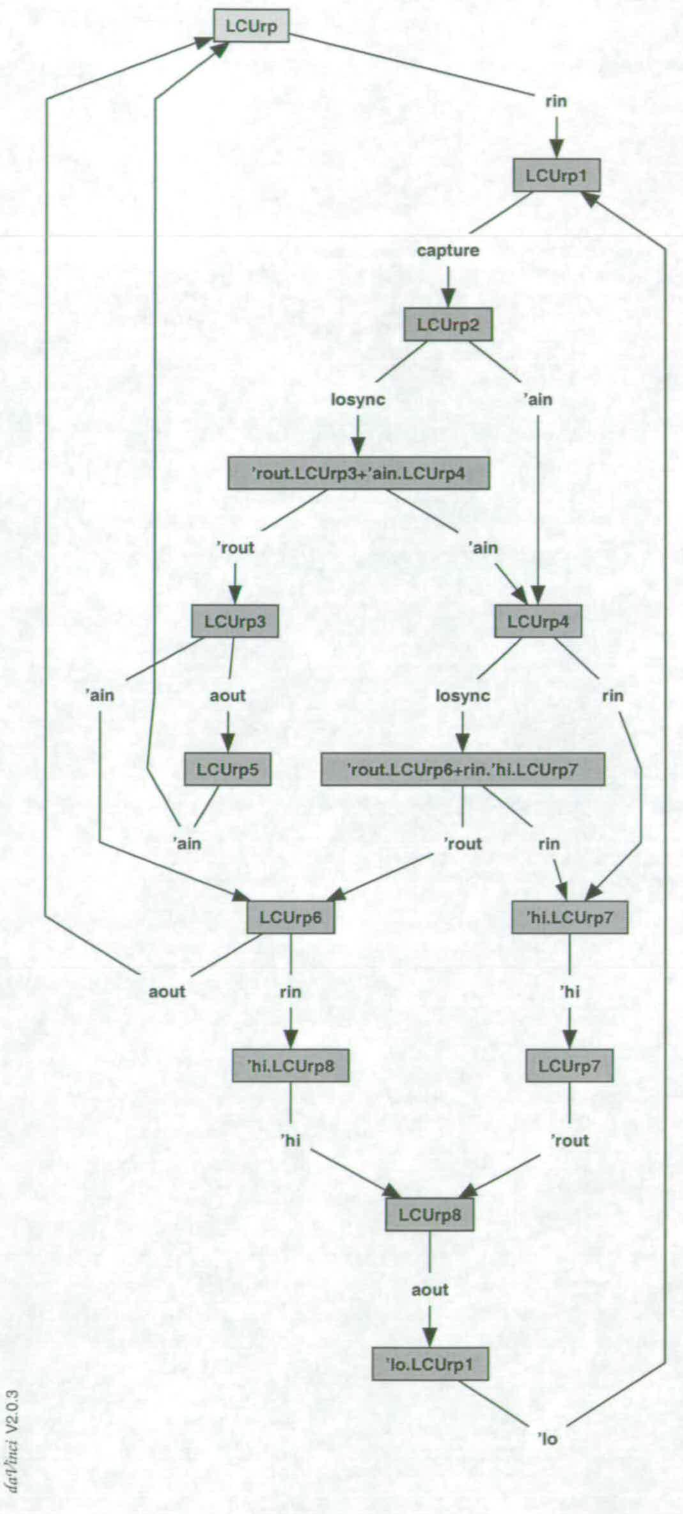


Figure 5.9: *Modified LCU process*

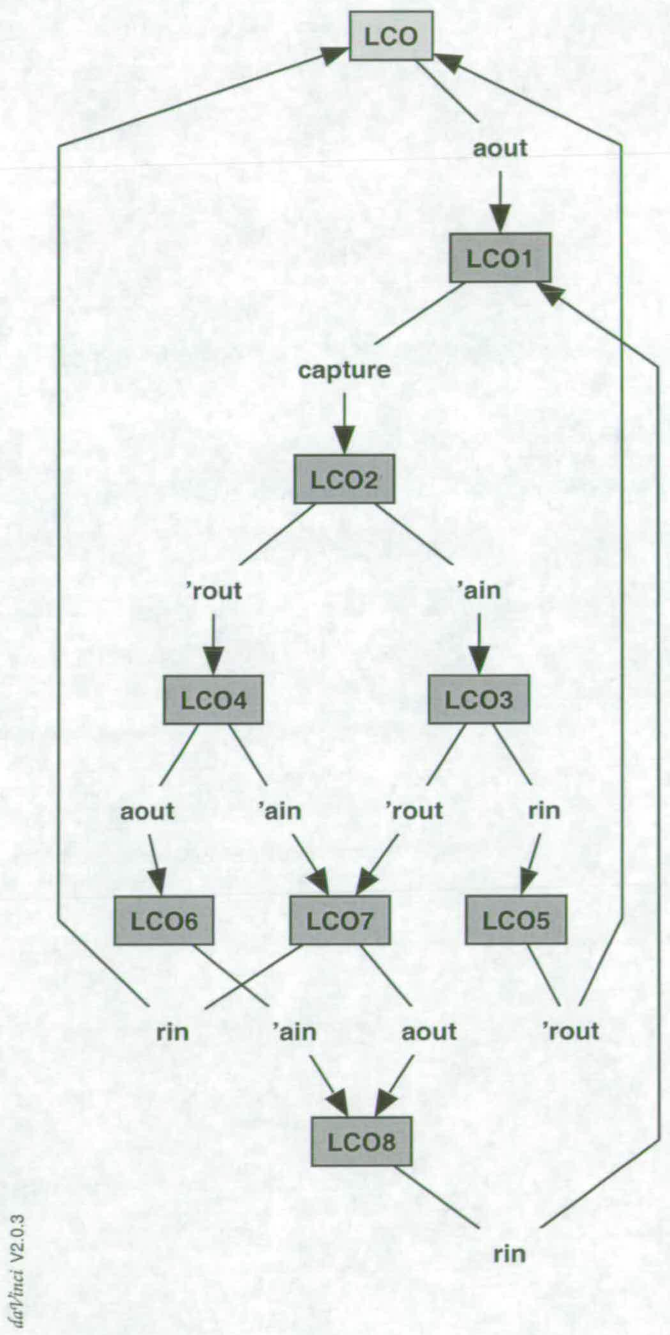


Figure 5.10: *Initially occupied latch controller*

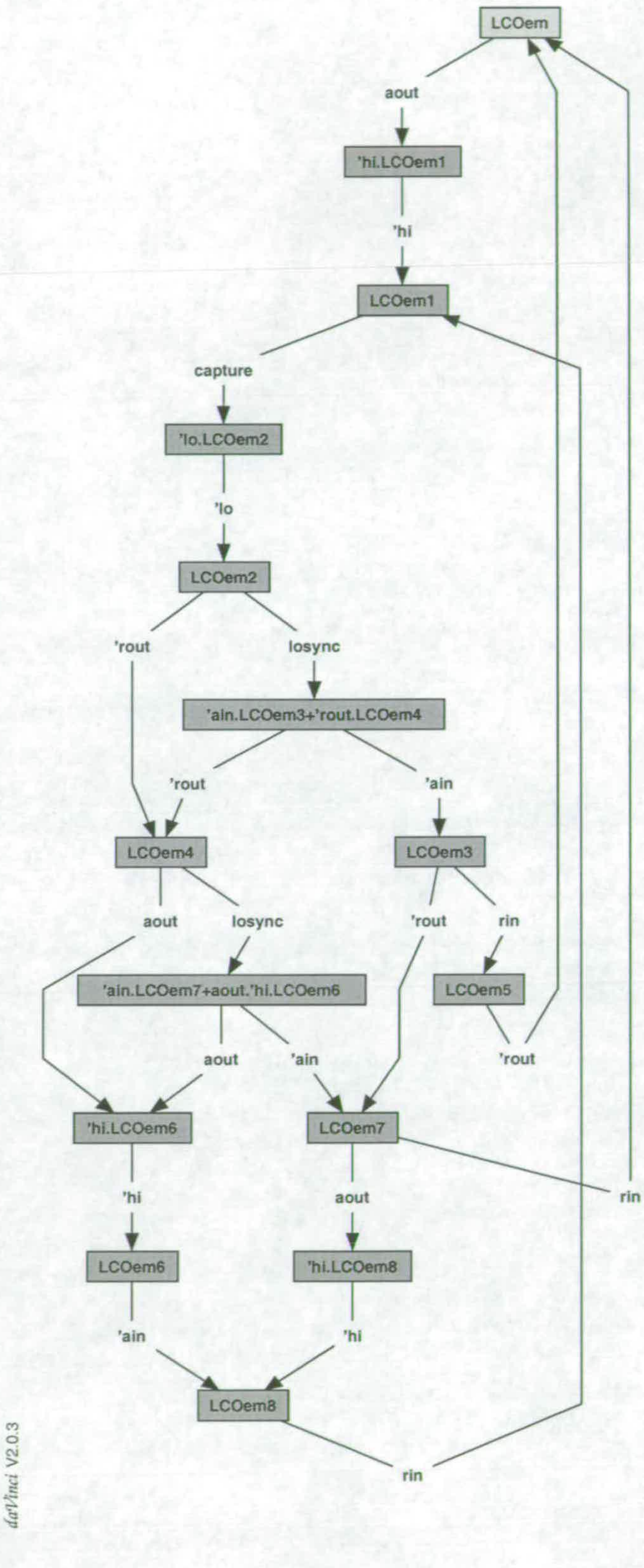


Figure 5.11: *Modifed LCO process*

5.5 Method II – circuit

This section describes a second method of full- and empty-detection. The circuits for full- and empty-detection are similar in structure and involve a simple function of the handshake signals between pipeline stages. Unlike Method I, for this method any two-phase latch controller circuit may be used without modification. Additionally a four-phase version should be possible.

5.5.1 Full detection

Figure 5.12 shows an n (n even) stage micropipeline with full-detection. Initially the pipeline is empty and all rout, f and full signals are 0. Once the pipeline is full alternate rout signals will alternate between 0 and 1 and full will change to 1. A chain of AND gates (with an inverter on every second stage) is used to detect when the pipeline is full.

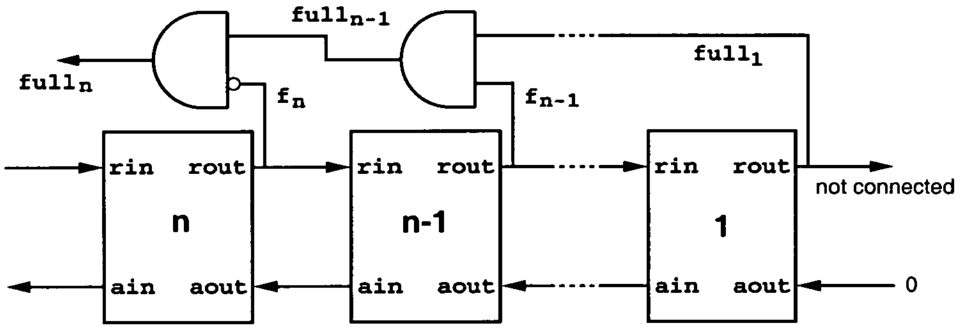


Figure 5.12: *Micropipeline with full-detection*

As before it is essential that $\mathtt{full_n}$ does not glitch high. Consider this scenario for an even number of stages n: (i) Assume the stages to the right of stage $n-1$ are full and stages n and $n-1$ are empty so that $\mathtt{full_{n-2}}$ is high and $\mathtt{rin_n}$, $\mathtt{rout_n}$, $\mathtt{rin_{n-1}}$, $\mathtt{rout_{n-1}}$ are low. (ii) $\mathtt{rin_n}$ goes high. (iii) $\mathtt{rout_n}$ becomes high. (iv) $\mathtt{rin_{n-1}}$ goes high but $\mathtt{f_n}$ does not change and remains low. (v) $\mathtt{rout_{n-1}}$ goes high, permitting $\mathtt{f_{n-1}}$ and hence $\mathtt{full_{n-1}}$ to become high. At this point it is possible, albeit unlikely, for $\mathtt{full_n}$ to become high. $\mathtt{f_n}$ may then go high and $\mathtt{full_n}$ returns low.

To avoid this problem the assumption that $\mathtt{f_i}$ changes before stage $i-1$ responds to $\mathtt{rout_i}$ is made. This assumption is valid because before stage $i-1$ can respond to $\mathtt{rout_i}$, it must capture data, which involves a large delay (due to the high capacitance of the row latch enable line) compared to the propagation of $\mathtt{f_i}$ plus the internal response of the AND gate (plus the inverter delay on every second stage). This assumption is a little strict; the assumption need be enforced on only either rising or falling edges of f_i depending on whether i is even or odd. However, more importantly the assumption restricts all activity in stage $i-1$ to occur (for example the generation of ain_{i-1} and rin_{i-2}) after f_i has changed.

Assuming (for only one edge) that f_i must change before f_{i-1}, unless f_i changes back again first, should be sufficient. However, the simpler (and still valid as argued above) assumption that f_i changes before $rout_i$ influences the next pipeline stage is simpler to model, as this assumption can be incorporated into a model of just one pipeline stage, without additional synchronisation between pipeline stages. Again verification is required to check that this timing assumption is sufficient; this is discussed in section 5.6.

5.5.2 Empty detection

Empty-detection is performed in a similar manner to full-detection, as shown for an even number of stages n in Figure 5.13. For an even number of stages the aout signals from each stage are combined with a chain of AND gates to form empty, for an odd number of stages a chain of OR gates is used and the final output inverted. This is because the waiting request (which will be a 0 or 1 depending on whether the number of stages is even or odd) present at the left hand input will determine the value of all the aout signals once the pipeline is empty. This corresponds to the left most column of latches in Figure 5.3 and is the reason for the left most OR/AND gate being connected to ain_1 ($aout_0$). This is equivalent to the special left hand stage used in the Method I model.

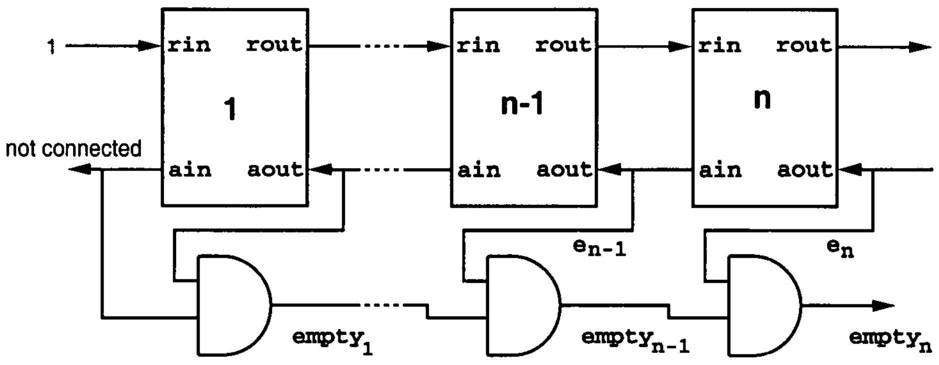


Figure 5.13: *Micropipeline with empty-detection*

Using a similar analysis to that for full-detection, it is assumed that e_i changes before ain_i can change and thus influence stage $i-1$ and in turn $\mathtt{e}_{\mathtt{i-1}}$. This assumption is valid, as the response to $\mathtt{ain}_{\mathtt{i}}$ by stage $i-1$ involves the long delay changing a column latch enable line, compared to the short delay on $\mathtt{e}_{\mathtt{i}}$. As with full-detection again the assumption is over strict and the more relaxed version involving $\mathtt{e}_{\mathtt{i}}$ and $\mathtt{e}_{\mathtt{i-1}}$ (defined in a similar manner to that for full-detection) should be sufficient.

5.6 Method II – modelling

The simple assumptions from sections 5.5.1 and 5.5.2 can be easily formalised by the addition of arcs to the STG for a two-phase latch controller (originally shown in Chapter 2, Figure 3.3).

The STG for a two-phase latch controller with the addition of the `f` signal is shown in Figure 5.14. The addition of arcs `capture` $\dashrightarrow$ `f` and `f` $\dashrightarrow$ `rout` correspond to the assumption made in section 5.5.1. The STG corresponds to the circuit for a single stage shown in Figure 5.15, note carefully the new position of the `rout` label. Events rather than separate low and high transitions are modelled. The inverter present at every second stage is modelled by setting the appropriate initial condition of the AND gate input.

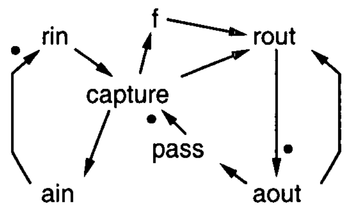


Figure 5.14: *STG of two-phase latch controller with* `f` *signal*

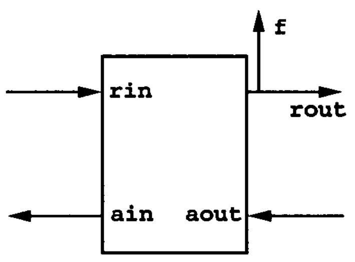


Figure 5.15: *Single pipeline stage with separate* `f` *and* `rout` *signals*

In a similar manner a two-phase latch controller with the `e` signal is modelled using the STG shown in Figure 5.16. The addition of arcs `aout` $\dashrightarrow$ `e` and `e` $\dashrightarrow$ `ain` correspond to the assumption made in section 5.5.2, the corresponding single stage circuit is shown in Figure 5.17.

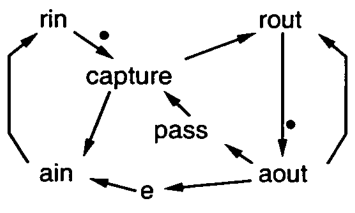


Figure 5.16: *STG of two-phase latch controller with* e *signal*

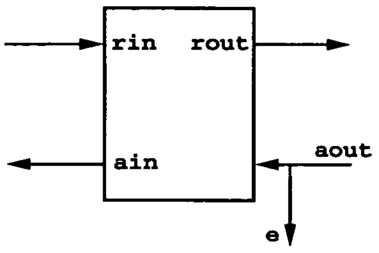


Figure 5.17: *Single pipeline stage with separate* e *and* aout *signals*

To form the overall circuit model first the STG for a single stage was converted to a CCS description (see section 4.6.3). A multiple stage model was then composed and this in turn composed with a model of the chain of AND or OR gates. The resulting process was then tested for observational equivalence to a simple specification process using the CWB.

The specifications for three stage pipelines with full-detection and empty-detection are shown below. *FullSpec* is more complex than *EmptySpec*, as $\overline{\mathtt{full}}$ can be generated before the final $\overline{\mathtt{ain}}$ and the left hand environment may also generate an additional rin.

$$\begin{aligned}
\mathit{Full} &\stackrel{\text{def}}{=} \overline{\mathtt{full}}.\overline{\mathtt{ain}}.\mathtt{rin}.0 + \overline{\mathtt{ain}}.(\overline{\mathtt{full}}.\mathtt{rin}.0 + \mathtt{rin}.\overline{\mathtt{full}}.0) \\
\mathit{FullSpec} &\stackrel{\text{def}}{=} \mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\overline{\mathtt{ain}}.\mathtt{rin}.\mathit{Full} \\
\mathit{EmptySpec} &\stackrel{\text{def}}{=} \mathtt{aout}.\overline{\mathtt{rout}}.\mathtt{aout}.\overline{\mathtt{rout}}.\mathtt{aout}.\overline{\mathtt{rout}}.\mathtt{aout}.\overline{\mathtt{empty}}.0
\end{aligned}$$

This has verified that, using only one simple assumption in the control circuitry, full- and empty-detection work for a few pipeline stages. Note that the delays in each OR/AND gate in the full- and empty-detection are unbounded. However, as with Method I, only a limited number of stages can be tested using the CWB; up to five stages have been tested.

5.7 Review

5.7.1 Comparison with synchronous equivalent

When compared to a fully synchronous version an area saving should be possible due to the use of latches instead of registers. The overhead of the micropipeline latch controllers is small; one latch controller is needed for each row and each column. For comparison of power consumption the following points can be observed, all of which suggest a significant power saving over the synchronous version.

- The number of data copy operations, where a latch is loaded with new data, in the synchronous version is double that of the micropipeline version. This is because registers, composed from two latches, are used.
- The load per bit on the clock signal in the synchronous version may be twice that of the load per bit on the latch enable signal in the micropipeline version (again because registers instead of latches are used).
- The number of latch enable transitions to fill or empty an n stage micropipeline is $\sum_{i=0}^{n-1} 2i + 1$, which is half that of the number of clock transitions, $2n^2$, required to fill or empty the synchronous equivalent (assuming single-edge sensitive registers).

The work by Tierno and Kudva [35] also offers a comparison in favour of the micropipeline version, although they compare asynchronous bit-parallel matrix transposition with synchronous bit-serial matrix transposition.

5.7.2 Comparison between Method I and Method II

For full-detection, Method II involves less circuitry and does not require modification of the latch controller. For empty-detection Method I and Method II involve the same amount of circuitry, but Method II might be preferable for layout reasons because it does not require access to the latch enable signals.

5.7.3 Comparison with previous work

Tierno and Kudva [35] present both two- and four-phase matrix transposition architectures. The two-phase method uses two latches per data bit, with alternating latches used for the two phases in the two-phase protocol, permitting continuous data flow. Completion detection is used to avoid delay matching the switch between row and column modes of operation. However, this is all at the expense of control circuitry, including counters to decide when the rows and columns are full or empty.

Their four-phase method is similar to Method II. Each pipeline stage outputs 'empty' and 'full' signals and these are combined with a single multi-input AND gate. A timing assumption similar to one made in Method II is made in the generation of these signals and in their delivery to the AND gate. One difference is that Method II uses a chain of gates rather than a single gate and so the timing assumption remains localised. This arrangement should also be faster, due to both the simple CMOS gate implementation and because the full or empty signal propagates in the direction in which the pipeline is filling or emptying.

A further difference is that their four-phase method assumes a square matrix and uses a single 1-D micropipeline, multiplexed between rows and columns. Switching between rows and columns is a bounded-delay operation which is assumed to happen between clock edges of the synchronous input and output clock. This bounded-delay is likely to be of similar magnitude to the reset delay needed in both Method I and II. Finally, to permit continuous operation, rather than add elastic buffers, two four-phase transpose circuits are used in a double-buffering arrangement. These approaches could also be applied to Method I and II.

5.7.4 Modelling method

The way Method II was modelled is somewhat simpler than the approach used for Method I. This is because specifying the circuit behaviour with additional timing assumptions was simpler using an STG than extending a flat state-space model. However, the use of the EL process in section 5.4.2 is an approach which could be applied to circuits for which the STG specification is too large to be manageable, for example a circuit composed from many sub-circuits, provided that it is sufficiently clear where the synchronising $\overline{\texttt{lo}}$, $\overline{\texttt{hi}}$ and $\overline{\texttt{losync}}$ actions should be placed.

Although the verification of the circuits shows that the circuits work without unknown timing assumptions for a small number of pipeline stages, tests with the CWB can not show the result for an arbitrary, but finite, number of stages. The recursive nature of the definition of the full-and empty-detection circuits suggest that an inductive proof should be possible. Such an inductive proof would take the form of assuming correct behaviour for n stages and showing that an $n + 1$ stage model must be correct if an n stage one is. A base case can then be tested using the CWB. The CCS composition of an n stage model, a single pipeline stage, the corresponding AND/OR gate and a suitable environment is not small and an initial uncompleted but time-consuming attempt suggests that such a proof would be laborious. One could say it seems 'obvious' the result must hold for n stages, but this is not proof. Additionally the more relaxed timing assumptions for both Method I and II could be modelled. However, given the existing timing assumptions have been argued to be safe, the extra modelling would seem unnecessary. Finally the timing assumptions made and the modelling performed does not rule out the possibility of using a fast-forward version (see Chapter 2) of the latch controller to improve performance.

5.8 Conclusion

This chapter has presented an architecture for performing parallel-serial conversion or matrix transposition. By using a two-dimensional micropipeline like structure with level sensitive latches instead of edge sensitive registers both area and power can be saved. Key features of the architecture operation, namely full- and empty-detection, have been verified using extensions to the modelling method from Chapter 4. Future work could involve circuit level simulations and layout to confirm area and power saving, a comparison of speed with the synchronous version and an inductive proof to extend the verification to an arbitrary number of pipeline stages.

Chapter 6
Review of discrete cosine transform VLSI accelerators

6.1 Introduction

As an introduction to the discrete cosine transform (DCT), prior to the discussion of an asynchronous implementation in Chapter 7, this chapter provides a review of VLSI implementations of the DCT in which the distinct roles of algorithmic and multiplier design are identified and key circuit and logic innovations are highlighted. To select from the large number of DCT implementations in existence, only implementations which offer reasonable performance and for which fabricated implementations and quantitative results have been reported are studied. In general these implementations are aimed at high performance applications, but a few lower performance implementations (notably the bit-serial based ones) are included for completeness. All of the implementations reviewed in this chapter are synchronous.

6.2 Overview

The DCT forms a key role in several image compression standards including JPEG [36] for still picture compression, ITU-T H.261 [37] and H263 for teleconferencing, and ISO MPEG-1 and MPEG-2 [38] for audio-visual compression and communication. Because of the existence of these standards, new consumer markets are emerging including digital direct satellite television broadcasting, high definition television (HDTV), digital video disc (DVD) and multimedia personal computers. All of these standards use an 8x8 two-dimensional (2-D) DCT which has therefore become the focus for VLSI implementation.

The 2-D DCT can be described as a transform from a 2-D matrix of pixels to a 2-D matrix of 'spatial frequency information'. The DCT is used for image compression because the transformed matrix contains many small or zero entries. Further the response of the human eye is frequency dependent and this can be exploited by weighting the transform results according to the ability of the human eye to perceive them. For transmission the resulting matrix is quantised (commonly resulting in many zero values) and encoded using run-length and Huffman encoding to remove sequences of zeros. By changing the amount of quantisation, image quality and bit rate can be traded which thus enables a controllable loss of the perceived image quality.

In this chapter several recent VLSI implementations are reported since they illustrate many different algorithmic and architectural decisions. For many applications such as HDTV, the receiver is the cost critical component and so implementations for the inverse DCT (IDCT) are described in some cases without the corresponding forward DCT. There are three factors which differentiate these designs: the underlying DCT algorithm, the digital architecture for multiplication and specific circuit and logic techniques.

6.3 The 2-D Discrete Cosine Transform

For an input matrix $x(m,n)$ and an output matrix $z(k,l)$ with $\{0 \leq m,n,k,l < N\}$ the forward $N \times N$ 2-D DCT is defined as

$$z(k,l) = \frac{2}{N}\alpha(k)\alpha(l)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)\cos\frac{(2m+1)\pi k}{2N}\cos\frac{(2n+1)\pi l}{2N} \tag{6.1}$$

and the inverse $N \times N$ 2-D DCT as

$$x(m,n) = \frac{2}{N}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} \alpha(k)\alpha(l)z(k,l)\cos\frac{(2m+1)\pi k}{2N}\cos\frac{(2n+1)\pi l}{2N} \tag{6.2}$$

where $\alpha(0) = \sqrt{\frac{1}{2}}$ and $\alpha(k) = 1$ for $k \neq 0$.

A naïve implementation of (6.1) or (6.2) requires N^4 multiplications. However, by noting that each cosine part only varies with one of the summations, the transform can be calculated by a *row-column decomposition* with only $2N^3$ multiplications; $2N$ multiplications per input pixel. The DCT and IDCT become:

$$Z = AXA^T \tag{6.3}$$

$$X = A^T ZA \tag{6.4}$$

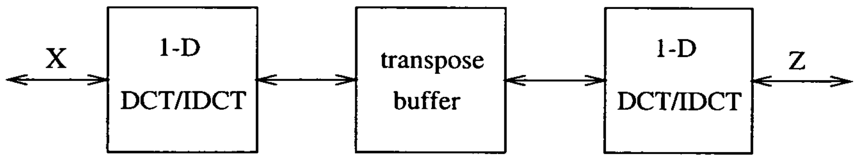


Figure 6.1: *Row-column decomposition*

where X is the source pixel (time domain) data, Z the DCT output coefficients (frequency domain) and A is an orthogonal matrix defined as:

$$a(u,v) = \sqrt{\frac{2}{N}}\alpha(u)\cos\frac{(2v+1)\pi u}{2N} \tag{6.5}$$

This row-column decomposition is equivalent to a 1-D DCT/IDCT followed by transposition and a second 1-D DCT/IDCT, as shown in Figure 6.1.

Direct two-dimensional methods are also possible with alternative optimisations, these are discussed later.

6.4 Some DCT algorithms

In this section two 8-point 1-D algorithms for the row-column decomposition and two direct 8x8 2-D approaches are considered.

6.4.1 Algorithm by Chen et al

Matrix A for the 8-point DCT can be written as:

$$A = \begin{bmatrix} d & d & d & d & d & d & d & d \\ a & c & e & g & -g & -e & -c & -a \\ b & f & -f & -b & -b & -f & f & b \\ c & -g & -a & -e & e & a & g & -c \\ d & -d & -d & d & d & -d & -d & d \\ e & -a & g & c & -c & -g & a & -e \\ f & -b & b & -f & -f & b & -b & f \\ g & -e & c & -a & a & -c & e & -g \end{bmatrix} \tag{6.6}$$

The multiplier coefficients a-g are listed in Appendix B.

The symmetry in this matrix can be exploited and the 1-D DCT rearranged to give:

$$\begin{bmatrix} y_0 \\ y_2 \\ y_4 \\ y_6 \end{bmatrix} = \begin{bmatrix} d & d & d & d \\ b & f & -f & -b \\ d & -d & -d & d \\ f & -b & b & -f \end{bmatrix} \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix}$$
$$\begin{bmatrix} y_1 \\ y_3 \\ y_5 \\ y_7 \end{bmatrix} = \begin{bmatrix} a & c & e & g \\ c & -g & -a & -e \\ e & -a & g & c \\ g & -e & c & -a \end{bmatrix} \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \tag{6.7}$$

The $(N \times N)$ multiplication matrix has been replaced by two $(N/2) \times (N/2)$ matrices, which can be computed in parallel, as can the sums and differences forming the vectors on the right-hand side of (6.7).

The implementations by Madisetti and Willson [39], Uramoto *et al* [40], Matsui *et al* [41] and Jang *et al* [42] are based upon this decomposition which requires 32 multiplications. However, Madisetti and Willson observe that the first part of (6.7) only involves multiplication by three rather than four different constants and so reduce the number of multiplications to 28.

The frequently referenced algorithm by Chen *et al* [43] is also derived from (6.7) but only requires 16 multiplications with 2 multiplications on the critical path. The data-flow graph for Chen's algorithm is shown in Figure 6.2. To compute the IDCT, the role of the inputs and outputs are reversed.

If Chen's algorithm is directly implemented in 4 clock cycles, one for each stage in the data-flow graph, each cycle requires at most 8 additions and at most 8 multiplications. The implementation used by Madisetti and Willson also takes 4 cycles and requires 8 additions per cycle, but only 7 multiplications per cycle are needed (although it is 7 in every cycle); additionally, 1 adder and 1 subtractor are needed to prepare the right-hand side sums and differences.

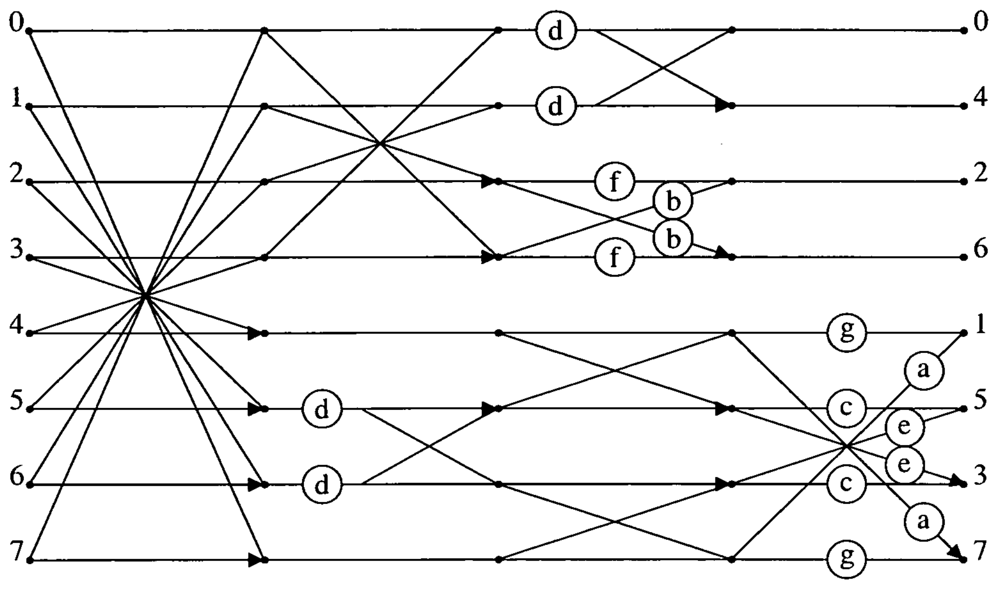


Figure 6.2: *The algorithm by Chen* et al

6.4.2 Algorithm by Arai et al

Although the minimum number of multiplications for the 8-point 1-D DCT algorithm is 11 [44], the algorithm by Arai *et al* [45] requires only 5. This is possible because the outputs are scaled: to obtain the true DCT value each output requires a further multiplication. However, in many systems this can be incorporated into multiplication coefficients used in the subsequent stage, for instance the perceptual weights in the video encoding algorithm. This type of algorithm is known as a scaled-DCT. If the two-dimensional transform is produced by row-column decomposition, then the scaling for both layers of one-dimensional transforms can be combined and performed after the second layer. This algorithm forms the basis of the asynchronous DCT discussed in Chapter 7.

As can be seen from Figure 6.3, the Arai algorithm also has the useful properties that the 5 multiplications can be performed in parallel and that no path through the data-flow graph includes more than one multiplication. Details of the inverse transform may be found in [45].

The Arai algorithm provides a good basis for a low cost scaled-DCT of other orders since it views the scaled N-point DCT as the real part of a $2N$-point discrete Fourier transform, which can be effectively implemented using fast DFT algorithms based upon cyclic-convolutions and Winograd's algorithm. Such solutions will display the same basic structure as the 8-point data-

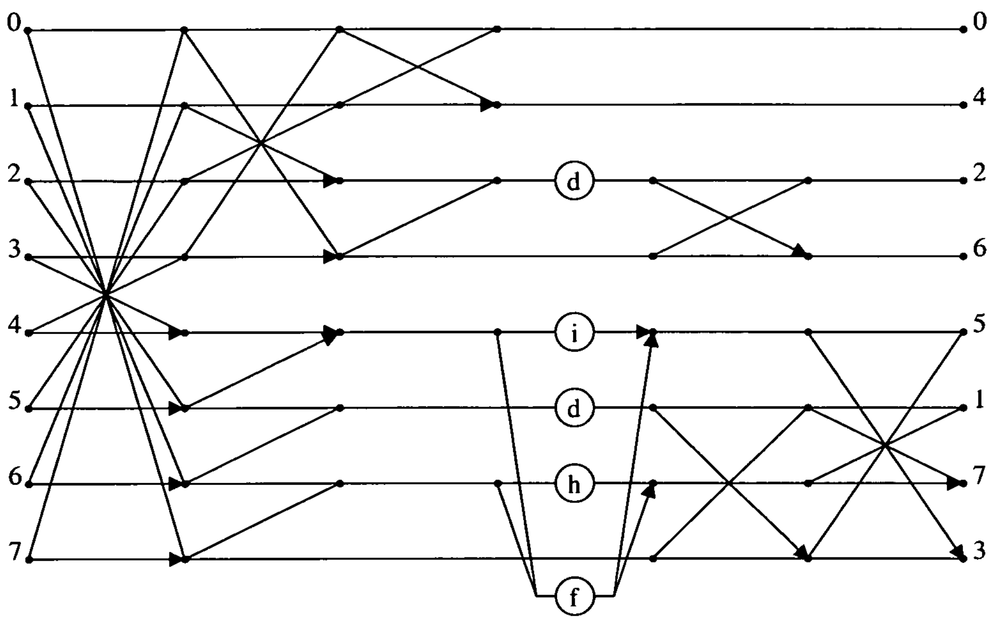


Figure 6.3: *Forward algorithm by Arai* et al

flow graph.

6.4.3 Algorithm by Chang and Wang

Chang and Wang [46] describe an algorithm which performs a direct 2-D transform using row-column decomposition within a systolic array. The $N \times N$ 2-D DCT in (6.3) is split by applying row-column decomposition into two steps of calculation, with intermediate result Y. Denoting $c_{ij} = \cos\frac{(2j+1)i\pi}{2N}$ from (6.5), and neglecting the scale factor, the matrix multiplications can be written as:

$$Y_{kn} = \sum_{m=0}^{N-1} X_{mn} c_{km} \tag{6.8}$$

$$Z_{kl} = \sum_{n=0}^{N-1} Y_{kn} c_{ln} \tag{6.9}$$

$$\tag{6.10}$$

By using the symmetries of the cosine function, and assuming that N is even, they define

$$U_{mn} = X_{mn} + (-1)^k X_{(N-1-m)n} \tag{6.11}$$

and

$$V_{kn} = Y_{kn} + (-1)^l Y_{k(N-1-n)}. \tag{6.12}$$

to give:

$$Y_{kn} = \sum_{m=0}^{N/2-1} U_{mn} c_{km}. \tag{6.13}$$

$$Z_{kl} = \sum_{n=0}^{N/2-1} V_{kn} c_{ln} \tag{6.14}$$

The algorithm is implemented as a systolic array in four stages corresponding to (6.11), (6.13), (6.12), (6.14). This requires a total of 64 multipliers for an 8x8-pt DCT.

6.4.4 Algorithm by Liu and Chiu

A different approach, taken by Liu and Chiu [47–49], is to calculate a running (or recursive) DCT in which the values of the DCT are updated with each new sample. Given a sequence of input data a 1-D DCT of the last N input values is output. Each DCT utilises the previous DCT result, the next DCT is obtained by adding the difference between it and the previous DCT. The discrete sine transform (DST) is needed in this calculation of the DCT and hence both DST and DCT outputs are available.

The 1-D DCT for the sequential input starting at $x(t)$ and ending with $x(t+N-1)$ is:

$$X_c(k,t) = \frac{2\alpha(k)}{N} \sum_{n=t}^{t+N-1} x(n) \cos \frac{\pi[2(n-t)+1]k}{2N} \tag{6.15}$$

for $k = 0, ..., N-1$ and $\alpha(k)$ is as before. Liu and Chiu then derive a recursive expression for the DCT X_c and DST X_s to give:

$$\begin{aligned} X_c(k,t+1) = {} & \{X_c(k,t) + \frac{2}{N}[-x(t) + (-1)^k x(t+N)] \cos \frac{\pi k}{2N}\} \cos \frac{\pi k}{N} \\ & + \{X_s(k,t) + \frac{2}{N}[-x(t) + (-1)^k x(t+N)] \sin \frac{\pi k}{2N}\} \sin \frac{\pi k}{N} \end{aligned} \tag{6.16}$$

where $k \neq 0$; $k = 0$ is a special case. $X_s(k, t+1)$ is calculated in a similar manner, the first

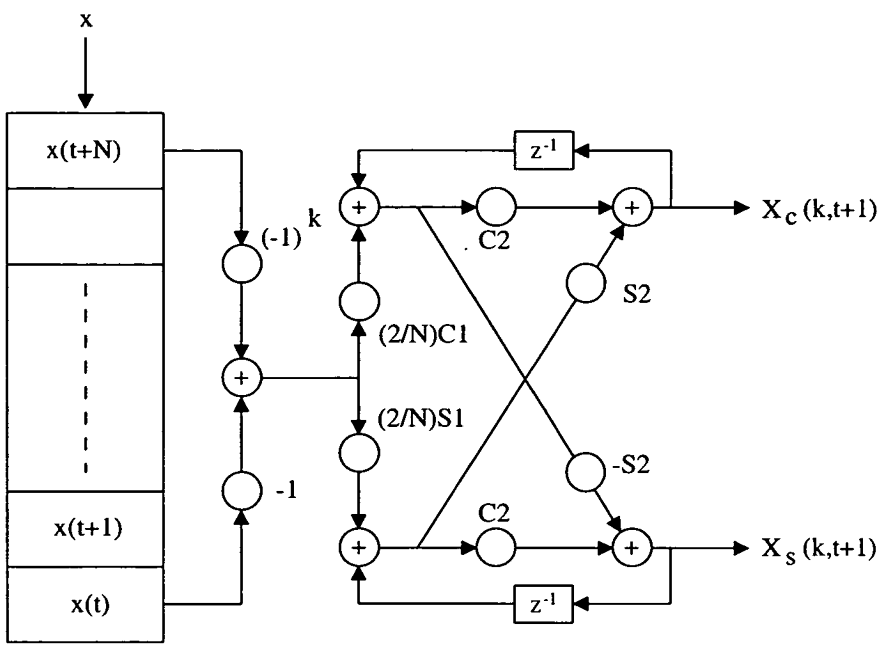


Figure 6.4: *DCT lattice by Liu and Chiu*

cosine and sine term in each half of (6.16) are swapped. $X_c(k, t+1)$ and $X_s(k, t+1)$ are obtained from $X_c(k, t)$ and $X_s(k, t)$ by subtracting the effect of $x(t)$ and adding the effect of $x(t+N)$. This is called the time recursive DCT.

The lattice structure to compute $x_c(k)$ for $k = 1, ..., N-1$ is shown in Figure 6.4 for a block size of 1; $Cn = \cos(\pi kn/2N)$ and $Sn = \sin(\pi kn/2N)$. A parallel array of this lattice is used to generate all N DCT outputs in parallel (with a special reduced form for $k = 0$).

The 2-D version is achieved using a circular shift register between two 1-D DCTs and N-length delays in each of the second layer filters. These together replace the usual transposition RAM and networks. An implementation is provided by Srinivasan and Liu [50, 51]. Aburdene *et al* [52] describe a similar time recursive approach for a 1-D DCT based on Clenshaw's recurrence formula [53].

6.4.5 Algorithm by Hsia et al

In [54] Hsia *et al* present an algorithm to calculate the 2-D IDCT directly by skipping non-zero coefficients. The algorithm may be understood by considering the symmetries of the matrix formed by the product of the two cosines in (6.2). For each value of (k, l), the matrix can be

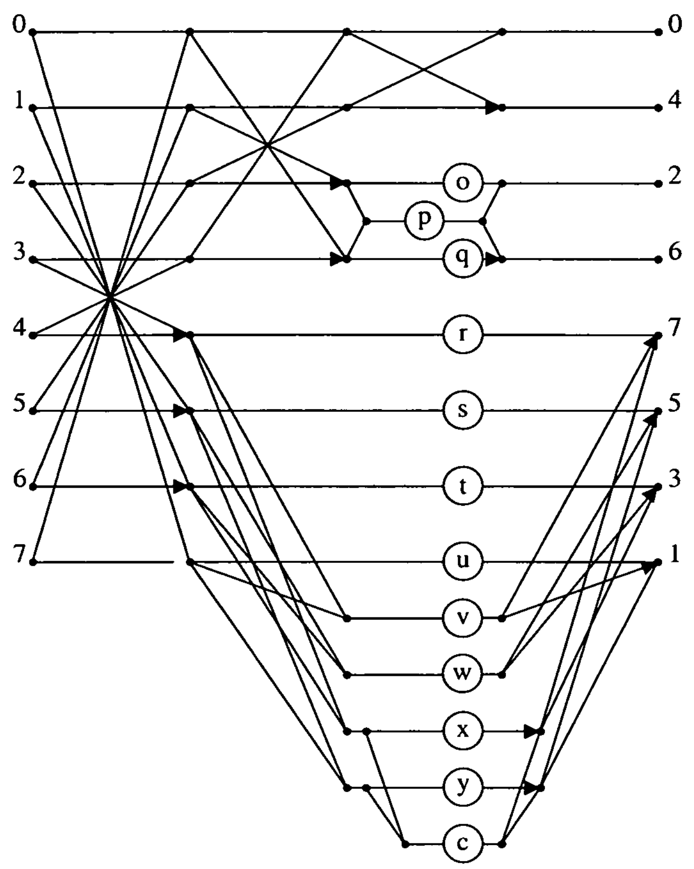


Figure 6.5: *Forward algorithm by Loeffler* et al

divided into 4 quadrants where the values of each are either the same as, the negative of, or a reordering of values of the first quadrant. Furthermore, the values of the first 4x4 quadrant can be calculated from the sum and difference of just four multiplications. Thus for each non-zero input (DCT coefficient) to the IDCT, there are four multiplications, various additions and negations.

6.4.6 Possible developments

For the un-scaled DCT, the family of 11 multiplier algorithms given by Loeffler *et al* [55] seems attractive as does an alternative in the same paper which requires 12 multipliers (only one more than the minimum of 11) and 32 adders. The forward algorithm with 11 multipliers is shown in Figure 6.5. As with the Arai algorithm, this has the advantage that no path includes more than one multiplier and that all the multiplications can be performed in parallel. McGovern *et al* [56] propose a similar algorithm.

A direct 2-D algorithm which has not yet been implemented in hardware, is a 2-D development of the Arai *et al* [45] scaled-DCT algorithm proposed by Feig and Winograd [57]. In this, the multiplications from the two stages of the row-column decomposition are combined. The result is a complete 8x8 2-D DCT with only 54 multiplications (with some shift operations for 1/2) and only 7 distinct coefficients (combined with a further shift operation for an eighth). The latter property leads to either low multiplier coefficient storage requirements or a practical number of hardwired multipliers. Furthermore, the networks for additions are formed by the repeated application of only two sub-networks. The main disadvantage is that the algorithm requires two matrix transpositions.

6.5 Multiplier architectures

There are three main multiplier architectures:

Combinatorial

Combinatorial multipliers take an n bit word and multiply by an m bit word to give an $n+m$ bit word. In many DCT implementations, purely combinatorial logic is used without pipelining. Such multipliers are typically implemented as a tree structure of adders and consume a large chip area, but are very fast. A variation is the hard-coded combinatorial multiplier which multiplies by a constant coefficient and therefore can be optimised for both area and speed.

Distributed arithmetic

Distributed arithmetic replaces combinatorial multipliers with a lookup table and accumulator. The computation of the vector product multiply-accumulation is broken down as follows:

Given a multiplicand vector x_k for $k = 0, ..., K$ and the constant multiplier coefficient vector a_k for $k = 0, ..., K$ the vector product is

$$y = \sum_{k=0}^{K} a_k x_k. \tag{6.17}$$

If x_k is in N bit two's complement fractional form, i.e.

$$x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \tag{6.18}$$

where b_{kn} represents the value, 0 or 1, of bit n of x_k then (6.17) can be computed as

$$y = \sum_{k=0}^{K} a_k(-b_{k0}) + \sum_{n=1}^{N-1} \left[\sum_{k=0}^{K} a_k b_{kn} \right] 2^{-n}. \tag{6.19}$$

The partial products $[\sum_{k=0}^{K} a_k b_{kn}]$ are stored in a ROM and accumulated for each bit of the multiplicand. Distributed arithmetic leads to a smaller area than a combinatorial multiplier but only allows multiplication by a fixed set of coefficients. However, the coefficients are easier to program (by changing the ROM contents) than the corresponding set of hardwired combinatorial multipliers.

Serial

The bit-serial architectures balance the long processing time of each operation (only one-bit per clock cycle) by operating on several words in parallel. Thus the input is word-parallel (a row/column of 8 words) and bit-serial. A typical bit-serial system takes $n + 1$ cycles to add two n bit values and $n + m$ cycles to multiply an n bit word by an m bit coefficient. As with the other multiplier designs a hard coded version can be made to multiply by a constant coefficient.

Bit-serial systems can be made to operate at very high clock frequencies due to the small propagation delays involved in the small blocks of combinatorial logic, however, this is not evident from the reported DCT designs to date. Bit-serial DCT implementations are simply a direct mapping of a data-flow graph.

Conventional two's complement bit-serial multiplication progresses least significant bit first. It has a high latency since the more significant half of the product is only generated after the less significant half, which is generally discarded. On-line arithmetic avoids this delay by progressing most significant digit first and by using redundant number encoding. Bruguera and Lang [58] describe a DCT implementation using this technique, although no performance figures are given.

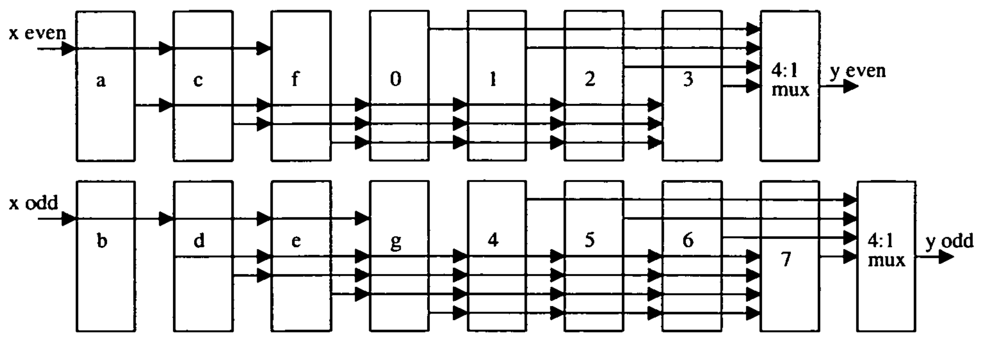


Figure 6.6: *Multiply accumulate by Madisetti and Willson*

6.6 Implementations

The following recent implementations were chosen to illustrate the use of specific logic and circuit techniques to enhance performance.

6.6.1 1-D based designs

Madisetti and Willson [39] implement the Chen based algorithm discussed in section 6.4.1. The 1-D DCT unit consists of 7 combinatorial multipliers, one for each of the matrix elements $a, ..., g$ as shown in Figure 6.6, and 8 combinatorial accumulators to sum the outputs. To minimise the critical path within the DCT unit, hardwired multipliers are used. These multiply by a constant coefficient which is fixed at design time leading to a faster and smaller module at the expense of programmability. Thus one multiplier is needed for each each unique multiplier coefficient. Each accumulator contains a multiplexer to select a particular multiplier output and logic to perform either addition or subtraction according to the sign of the coefficient.

A 'data reorder unit' (not shown in Figure 6.6) prepares the sums and differences for the DCT, and reorders the data into even and odd inputs for the IDCT. The whole 1-D DCT/IDCT takes 4 clock cycles. Rather than use a second 1-D DCT unit, a single 1-D DCT unit is multiplexed between the input data and the output of the transpose buffer implemented by a 64 word DRAM (shown in Figure 6.7). However, since the single 8-pt transform requires only 4 clock cycles and each data point is processed in two transforms, the clock rate is actually equal to the data rate.

With conventional two's complement representation an n bit fractional number f can be ex-

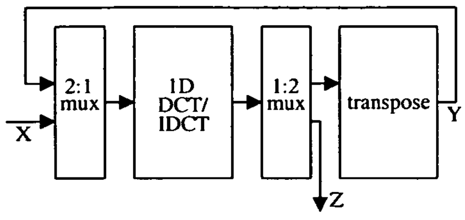


Figure 6.7: *Multiplexed 1-D DCT by Madisetti and Willson*

pressed as

$$f = -b_0 + \sum_{k=1}^{n-1} b_k 2^{-k} \qquad b_k \in \{0,1\}. \tag{6.20}$$

The multipliers store the coefficients using radix-2 signed digit representation in which an n bit fractional number can be expressed as

$$f = \sum_{k=0}^{n-1} s_k 2^{-k} \qquad s_k \in \{-1,0,1\}. \tag{6.21}$$

This form of representation allows most numbers to be represented with fewer non-zero digits than in two's-complement representation. Note that the multiplicand and the product use two's complement representation. In the hardwired implementation one adder (also capable of subtraction) is required per non-zero digit, thus giving a smaller implementation. Between 4 and 6 adders are required for the constants $a, ..., g$ to give 12-bit accuracy. The products are accumulated in 22-bit wide carry-select adders.

In total 7 multipliers and 8 accumulators are needed. Using 0.8μm CMOS the $N = 8$ implementation by Madisetti and Willson has an area of 10mm^2. A complete 2-D transform is computed every N^2 cycles and an input sample rate of 100MHz is possible.

Kovac and Ranganathan [59] describe a JPEG encoder based on the algorithm by Arai *et al* (section 6.4.2) using row-column decomposition with two separate 1-D units. Although not designed for real time MPEG a 100MHz input pixel rate is possible.

Each 1-D unit, shown in Figure 6.8, consists of 6 stages corresponding to the 6 columns of operations shown in Figure 6.3. Each stage contains a pipeline register set (RS) and a combinatorial adder capable of negating the inputs, except for stage 4 which includes a single Wallace-Tree

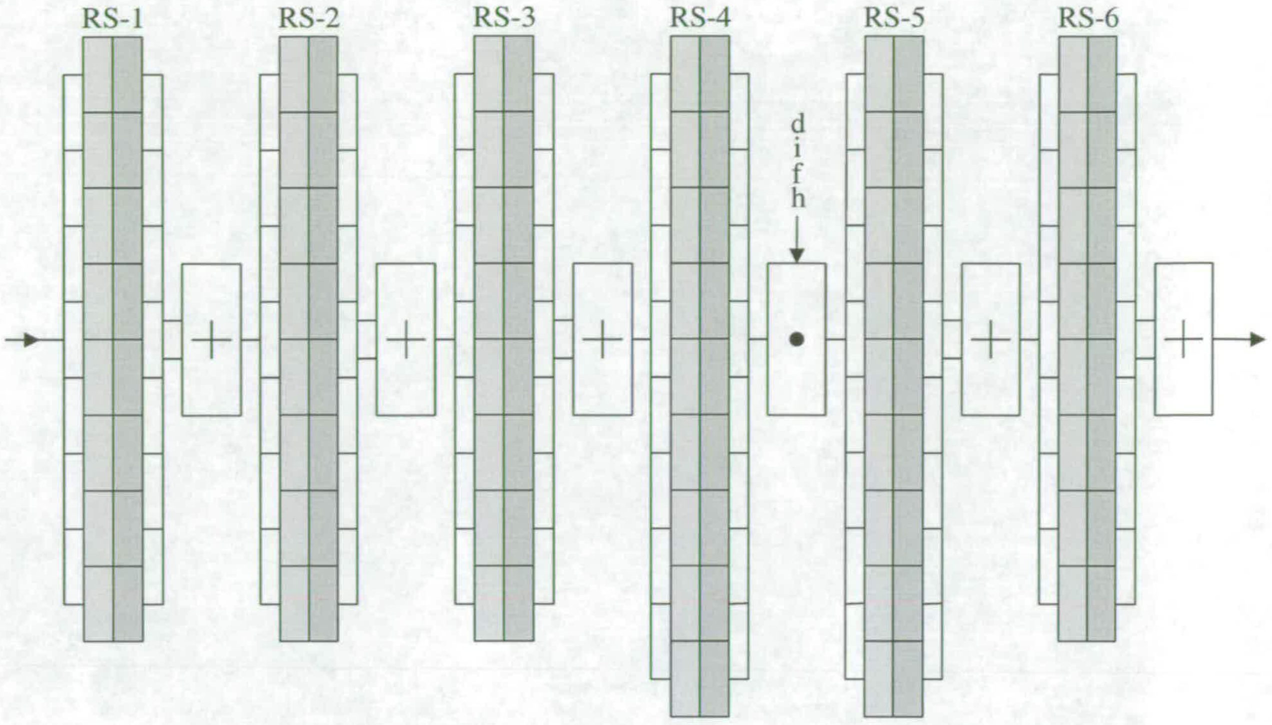


Figure 6.8: *1-D DCT by Kovac and Ranganathan*

combinatorial multiplier. Each register set consists of two columns of 14-bit registers which act as data buffers. Data from the previous stage is written into the left column and, once complete is moved into the right column where it is loaded into the next stage.

Uramoto *et al* [40] implement the Chen based algorithm using distributed arithmetic with two separate 1-D units. To halve the computation time the input values are processed in pairs so that two partial products are accumulated at a time; this requires a dual port ROM. Throughput is further enhanced by pipelining.

By changing the ROM contents the same circuit can be used to calculate the DCT or the IDCT. Alternatively a ROM holding twice as many words can be employed and the appropriate half selected. Uramoto *et al* have developed a 'dual plane ROM' which holds two banks of values where only one bank can be accessed at a time. The dual plane ROM gives a 30% area saving and a speed increase: combined with carefully designed accumulators and routing a 100MHz input pixel rate is possible.

Karathanasis [60] also proposes an algorithm for use with distributed arithmetic, simulated to be capable of a 100MHz pixel rate, but with a smaller ROM requirement, only 10% of that needed in the implementation by Uramoto *et al* [40].

Matsui *et el* [41] give another implementation using the Chen based algorithm capable of a 200MHz input pixel rate. Each of the two 1-D DCT units contain 8 distributed arithmetic multiply-accumulators. The key feature is the high-speed logic design based upon differential NMOS pass transistor logic [61]. Here a logic signal is transmitted as two complementary signals with sense-amplifiers to detect the differential voltage (as small as 100mV) between the signals thus reducing the effect of parasitic capacitance.

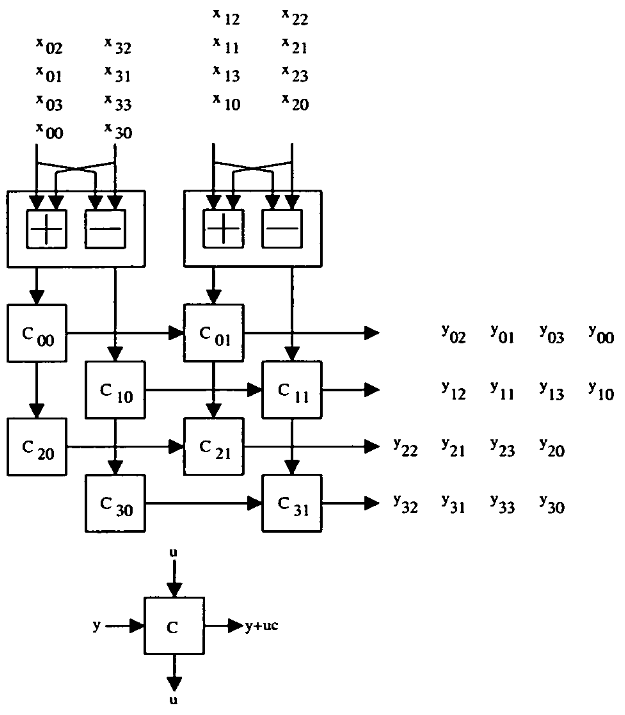
Masaki *et al* [62] describe another distributed arithmetic implementation of the IDCT using the Chen algorithm with specific focus on producing a single chip MPEG decoder. A single 1-D IDCT unit with pipelined Wallace-Tree adders is multiplexed with the input coefficients and transpose buffer. A more sophisticated (more bits at a time) distributed arithmetic scheme is used resulting in a 200MHz pixel rate with only a 100MHz clock.

For comparison two bit-serial implementations are mentioned. Cucchi and Fratti [63] implement the Chen algorithm with bit-serial arithmetic. Two bits are processed at a time so that the internal clock rate is equal to the input pixel rate (40MHz). McGovern *et al* [56] also use bit-serial arithmetic for an algorithm similar to Loeffler *et al* [55] requiring the minimum number of adders and multipliers. Both of these implementations use 2 1-D DCT units and a transpose buffer.

6.6.2 Direct 2-D

Hsia *et al* [54] implement the algorithm in section 6.4.5 using pipelined combinatorial multipliers to decrease the critical path. A K-bit by K-bit multiplier is composed from a tree structure of four $K/2$-bit by $K/2$-bit multipliers, registers are inserted into the tree such that the critical path involves only a K-bit adder instead of a K-bit multiplier. A bank of 64 accumulators holds the current total for each of the IDCT output pixels. The implementation achieves an average pixel rate varying from 150MHz to a maximum of 400MHz as the compression ratio varies from 4 to 16. This corresponds to an average of 33% to 9% non-zero input coefficients.

Another direct 2-D architecture exploiting zero valued DCT coefficients to reduce power consumption is proposed by Xanthopoulos *et al* [64]. Further power reduction is achieved by connecting latches to the inputs of each adder to stop unwanted transitions passing through



x_{02} x_{01} x_{03} x_{00}
x_{32} x_{31} x_{33} x_{30}
x_{12} x_{11} x_{13} x_{10}
x_{22} x_{21} x_{23} x_{20}
c_{00} c_{01} c_{10} c_{11} c_{20} c_{21} c_{30} c_{31}
y_{02} y_{01} y_{03} y_{00}
y_{12} y_{11} y_{13} y_{10}
y_{22} y_{21} y_{23} y_{20}
y_{32} y_{31} y_{33} y_{30}
u
y
c
y+uc
u

Figure 6.9: *First two stages of array for $N = 4$ by Chang and Wang*

the adder. Finally the supply voltage and clock frequency are lowered when there are fewer non-zero coefficients to be processed.

A systolic array is a structure of identical cells with many local and few global interconnections. Such arrays are ideally suited to VLSI implementation due to good regularity, modularity and concurrency. The algorithm by Chang and Wang [46] is implemented using systolic structures. The structure to implement the first two stages, see section 6.4.3, is given here in Figure 6.9 for $N = 4$. In a similar way structures for the second and third stages can be created (as detailed in [46]). The IDCT can be implemented by simply inverting the order of the structures.

The adders (16 bit) and multipliers (12 × 12 bit) are combinatorial; a total of N^2 multipliers and $N^2 + 3N$ adders are required; a complete 2-D transform is computed every N cycles. The implementation for $N = 8$ reads the data in word-parallel bit-parallel format (8 words per clock cycle) and can support a pixel rate of over 320MHz—well in excess of current HDTV requirements.

Srinivasan and Liu [50, 51] implement the 2-D lattice algorithm by Liu and Chiu [47–49] (section 6.4.4) using distributed arithmetic multipliers. The partial product lookup ROM is addressed by a 12 bit word. This would give 4096 rows, to reduce area the ROM is split into two 6 bit addressed 64 row ROMS. One ROM output is then shifted, sign extended and added to the other. Although only 32 multipliers are needed, the algorithm requires a large number of bit-parallel registers.

6.7 Comparison and conclusions

Figure 6.10 shows the area each implementation requires, scaled to a constant 0.8μm process technology. The multiplier styles are combinatorial multiplier (CM), distributed arithmetic (DA) and bit-serial (BS). Also shown is the number of multipliers for the 8x8 transform. Figure 6.11 shows the pixel rate achievable of each implementation, along with the feature size in microns and whether the implementation is electrically switchable (E) between IDCT and DCT, IDCT only (I), DCT only (D), or both but not electrically switchable (D/I). The pixel rates are un-scaled due to the difficulties in scaling operation frequency with feature size (which may vary from constant to quadratic depending on which scaling model is used). For the implementation by Hsia *et al* [54] a pixel rate of 400MHz is assumed, this corresponds to the average number of non-zero coefficients in a typical MPEG sequence reported by both [54] and [64].

The direct 2-D implementations require a larger area, but deliver a considerably higher pixel rate, except the implementation by Srinivasan and Liu [50, 51]. The relatively poor performance of Srinivasan and Liu's implementation may be due to the fact that the underlying algorithm was originally developed for overlapping blocks (a running DCT) for which a smaller implementation is attained.

The bit-serial designs (Cucchi and Fratti [63], McGovern *et al* [56]) perform badly, neither of them being fast enough to meet HDTV requirements, and the implementation by Cucchi and Fratti requires over twice the area of the other 1-D with transpose implementations.

The 1-D designs use either combinatorial multipliers or distributed arithmetic and all employ one of the algorithms from section 6.4.1. The design by Madisetti and Willson [39] achieves a small area by multiplexing a single 1-D DCT, however the design by Matsui *et al* [41] achieves twice the pixel rate (200MHz) for a 30% increase in area, due to use of a novel sense-amplifier and pass-transistor logic. The Masaki *et al* [62] implementation although also capable of 200MHz is larger and performs the IDCT only.

Based upon this study the following guidelines for designers are concluded: Higher performance 1-D implementations require multipliers which are either hardwired-combinatorial logic or sophisticated ROM architectures and/or circuit techniques for distributed arithmetic. However, high-speed circuit techniques for the multiplier/accumulator will be significant to performance. If even higher throughput is required, then this may be traded-off against area with a full 2-D approach (which often, however, avoids the area cost of the transpose memory). In decoder only applications, for example digital TV receivers, IDCT algorithms which exploit zero coefficients would seem attractive.

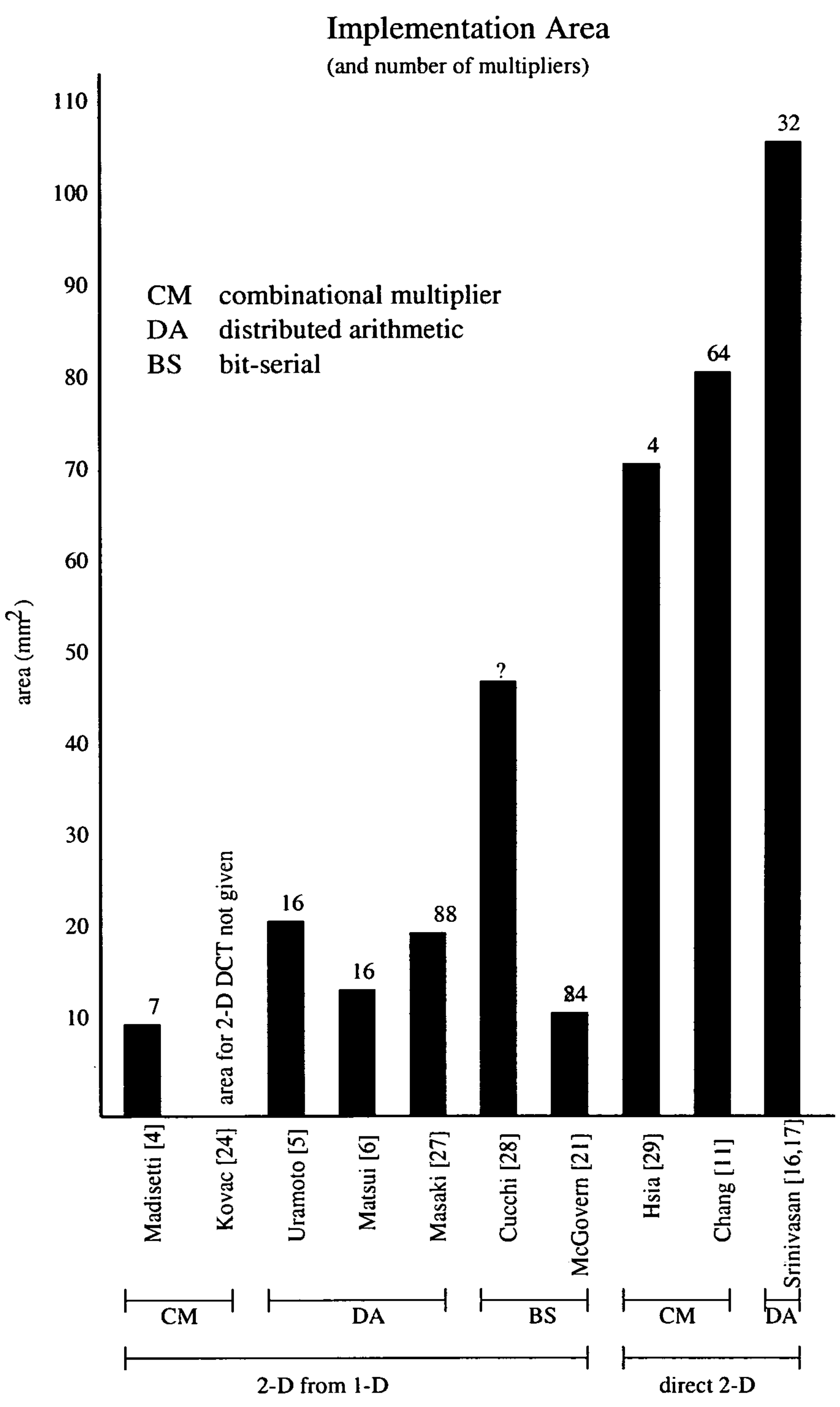


Figure 6.10: *Implementation area scaled to* 0.8μ *process*

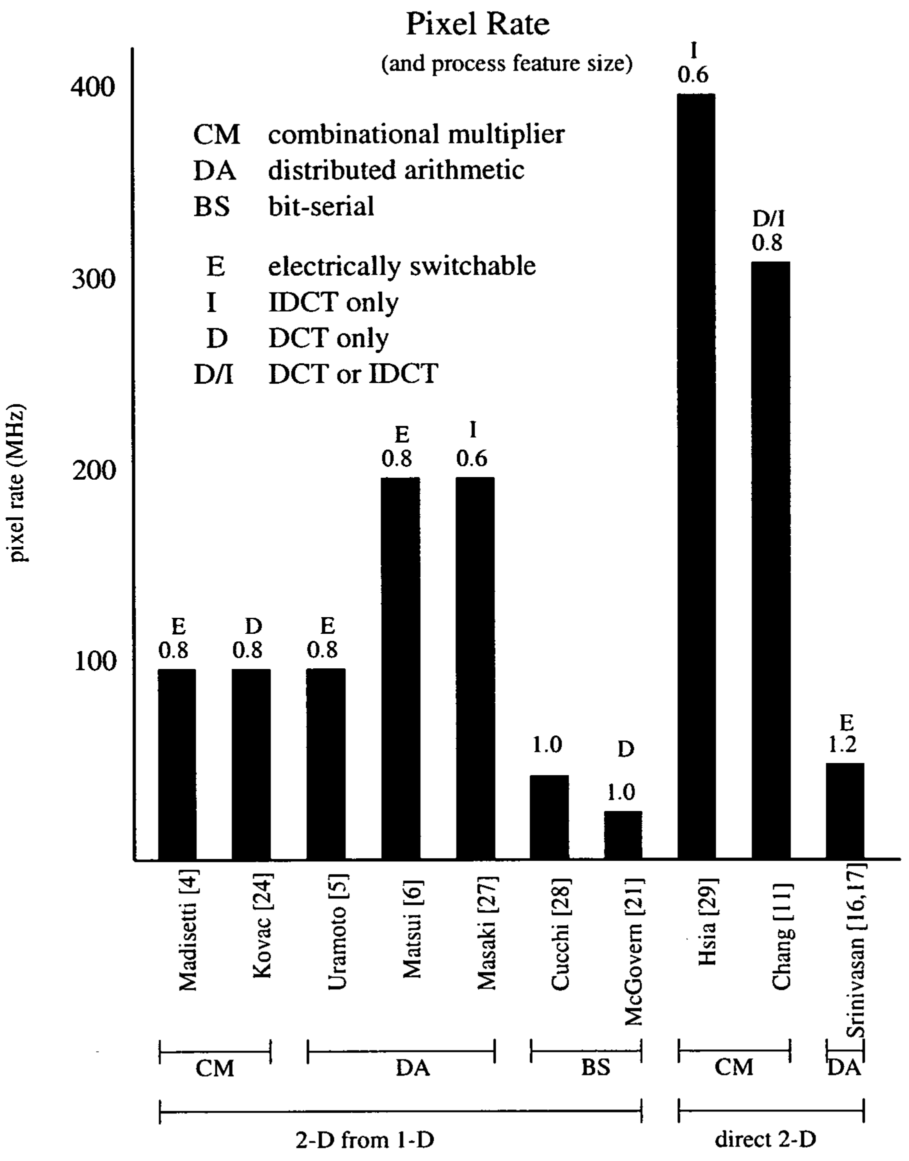


Figure 6.11: *Pixel rate (no scaling)*

Chapter 7
An asynchronous discrete cosine transform

7.1 Introduction

This chapter discusses initial work on an asynchronous application specific processor (ASP) architecture intended for the computation of waveform transforms, with particular emphasis on the DCT. The DCT algorithm by Arai *et al* [45], discussed previously in section 6.4.2, is chosen as it only requires 5 multiplications (although the output values need to be scaled) and has a fairly regular structure. For example, the 'butterfly' operation permits a sequence of additions for which one operand remains constant between pairs of additions. Only the 1-D DCT has been implemented, to form a 2-D DCT two 1-D units and matrix transposition would be required. The flexibility offered by the programmable architecture enables the inverse DCT to be implemented and potentially other algorithms as well. The architecture uses a modest amount of circuitry, at the expense of performance.

7.2 Previous asynchronous DCT work

It would appear there are very few previous asynchronous DCT designs. The design by Stott *et al* [65] and the design by Lipsher [66] are similar in architecture and are both based upon a micropipelined distributed arithmetic implementation of the the algorithm by Chen *et al* [43] (see section 6.4.1). After the work discussed later in this chapter was performed, a further design by Smith *et al* [67] was found. Their design is based upon building blocks constructed from threshold logic gates with hysteresis [68] and they have simulated a prototype (using a mapping from threshold logic to conventional logic) with FPGAs. Their design uses the DCT algorithm by *Lee* [69], which involves 13 multiplications and 29 additions. Compared to synchronous implementations all the asynchronous implementations are slow (10-30MHz pixel rate for 2-D DCT), however the authors of each design discuss various ways in which the performance can be vastly improved with the experience gained from the initial prototype.

A related paper, briefly mentioned in Chapter 6, is the synchronous IDCT processor by Xanthopoulos *et al* [64]. This design skips over zero coefficients (note this can only be done for the inverse transform). When there is less work to be done (more zero coefficients) the supply voltage and clock frequency are lowered to save power. An asynchronous version of this design would reduce the need for careful design and simulation to ensure suitable setup and hold times under the wide range of clock frequency and supply voltage.

7.3 Application specific processors

Application specific processors (ASPs) [70–72] attempt to combine the flexibility of a programmable general purpose processor with the high performance of a dedicated algorithm specific architecture. A large ASP might contain dedicated function blocks in addition to programmable control logic and local program storage. For example the MPEG encoder chip in [73] contains dedicated blocks to perform a DCT, IDCT, motion estimation and quantisation with a programmable processor module with down-loadable microcode to enable tailoring to the individual application.

Asynchronous architectures may ease the modular design of application specific processors, the use of local handshake protocols and the subsequent ease of composition should enable functional blocks to be easily added to a base architecture. For example, in [13] a commercial asynchronous DSP processor is presented and compared to a synchronous equivalent. The use of asynchronous techniques was shown to give low-power, low noise emission properties and a highly configurable architecture to which additional functional units could be easily added.

7.4 Architecture

7.4.1 Overview

The data-path of the proposed architecture is shown in Figure 7.1. From top to bottom the architecture can be divided into three main sections. The upper section takes data from either the external input or the lower section and passes it to one of four elastic FIFO queues or to the external output. Towards the right is a multiplier which multiplies the output of a queue with a selectable constant to give two outputs which are then summed to obtain the product. The lower section consists of an adder which reads data from either a pair of queues or the multiplier. Multi-bit XOR gates (inv) to invert the adder inputs combined with a carry input to the adder permit subtraction and an elastic FIFO buffers the adder output. Both the external data input and data output can be buffered with FIFO queues.

The four queues and the adder output FIFO are micropipeline circuits, the adder and multiplier are intended to use completion detection but delay matching is also possible. The multiplexers and XOR inverters have a short (and probably data independent) delay and can be delay matched. It should be noted that the micropipeline handshake signals are not always connected to functional units as shown with the data path and many handshake signals are controlled directly by the control logic units.

Input pixel data arrives at the top and data items are directed into the appropriate queues A-D. The control logic permits two methods of reading data from a queue:

Read The current data output from the queue is read and an acknowledge signal is sent to the queue output and hence the item of data is removed from the queue.

Copy The current data output from the queue is read but an acknowledge signal is not sent and so the item of data remains in the queue.

In both cases the request output from the queue is used by the control logic to detect when new data is available.

Three kinds of operation are possible on the data stored in the queues.

Multiply A single item of data is read or copied from a queue and multiplied by a constant held within the multiplier unit. The two outputs of the multiplier are then summed to form the product.

Add Two items of data are read or copied (or a combination of) from queues A and C, or from queues B and D and are added (or subtracted).

Copy A single item of data is read or copied from a single queue and is added with zero (a hardwired input to the adder input multiplexers).

In all three cases the result is available at the output of the adder. The result is then buffered by the adder output FIFO and is either an output result, or is an intermediate result which is written back into one of the queues. Although not used in the DCT algorithm it is possible to transfer data directly from input to output without processing.

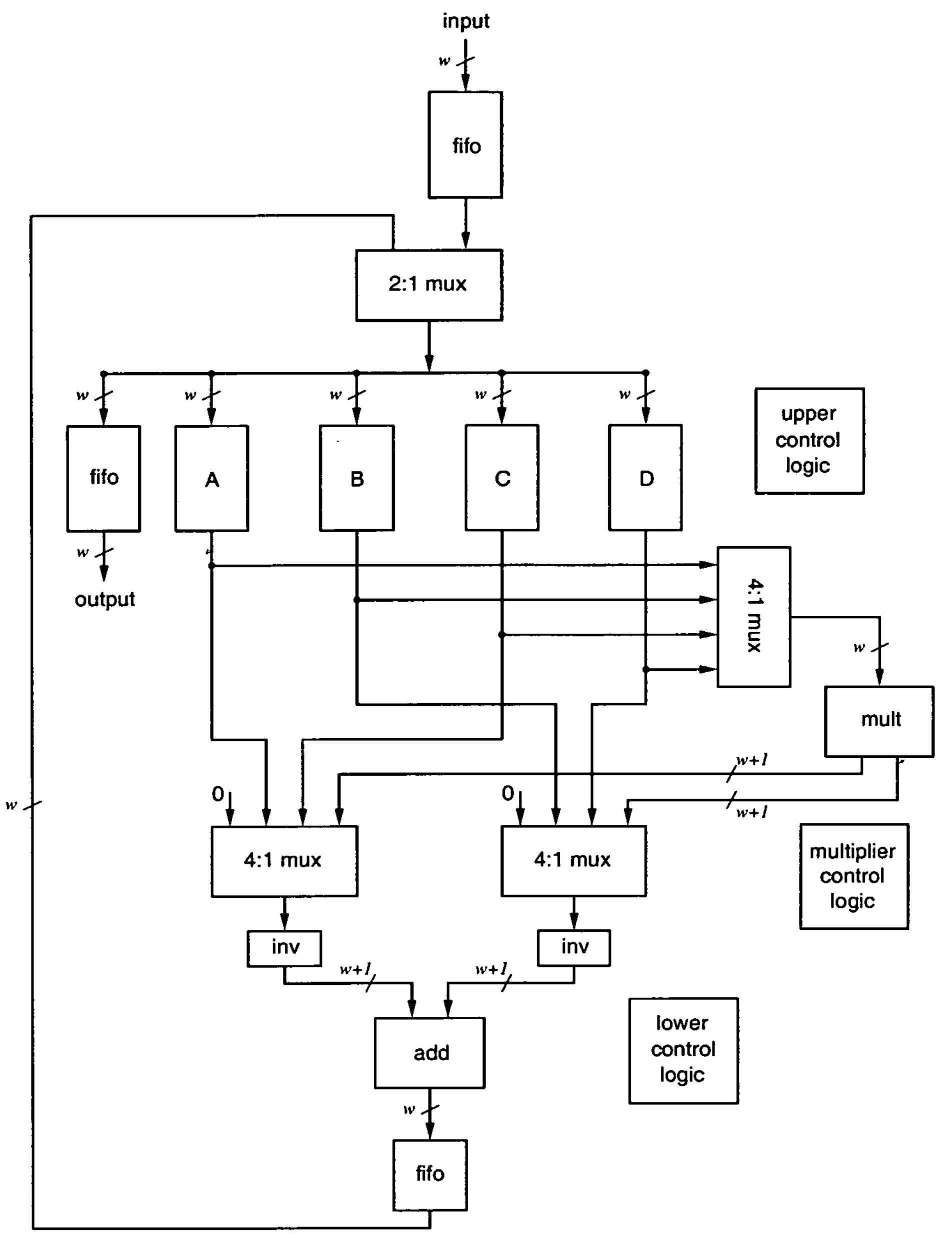


Figure 7.1: *Asynchronous ASP architecture for DCT*

7.4.2 Control logic

The control logic is split into three units, upper, lower and multiplier control logic. Each control unit contains a ROM holding microcode like instructions. The functions performed by each control unit are summarised below. The three control units operate independently and asynchronously with separate internal control ROMs and ROM address counters. In all three control units the ROM lookup time and ROM address counter could be delay-matched or completion detection could be used; for the purposes of simulation these parts were implemented as delay matched behavioural models.

Upper control unit

The upper control unit connects to the request and acknowledge signals from: the queue inputs, the external input, the external output and the output of the adder output FIFO. Additionally the upper control unit controls the 2:1 multiplexer to select between the adder output FIFO and the external input. Each operation cycle of the upper control unit transfers an item of data from either the adder output FIFO or the external input to either one queue or to the external output.

The upper control ROM is a list of number pairs, indicating from which source data is read and to which destination data is written. Every time the destination acknowledges receipt of the data the next ROM entry is read, the appropriate source request is waited for (or the source may already be ready) and another item of data transferred.

Lower control unit

The lower control unit connects to the request and acknowledge signals from the queue outputs and the acknowledge output from the adder output FIFO. Both the adder and multiplier have a ‘go’ input to start an operation and a ‘done’ output to signal completion. The adder ‘done’ signal supplies requests to the input of the adder output FIFO. The remaining handshake signals are connected to the lower control unit. Additionally the lower control unit controls the multiplier input multiplexer, the adder input multiplexers, the XOR inverters and the adder carry input. Each operation cycle of the lower control unit performs one of the three operations, add, multiply or copy.

This unit contains the most complex ROM and decoder. Several multi-bit fields are used to indicate from which queues data is to be read from and to control the multiplexers and logic surrounding the adder and multiplier.

Multiplier control unit

The multiplier and multiplier control unit consist of a bit-parallel combinatorial multiplier, a ROM to hold the multiplier coefficients and a control ROM indicating which multiplier coefficient to use. A more efficient scheme might be to use just one ROM containing the (possibly duplicated) multiplier coefficients. Each operation cycle of the multiplier is to perform a single multiplication, note that a cycle only commences when requested by the lower control unit.

7.4.3 Adder and multiplier

Both the adder and multiplier are intended to be asynchronous with a 'go' signal, completion detection and a 'done' output signal. In the simulations performed both the adder and multiplier use delay matching rather than completion detection.

The multiplier produces two outputs instead of the normal product, this is intended to allow the adder to be reused for multiplications, perhaps reducing the complexity of the multiplier depending on the circuit implementation. A possible algorithm for the multiplier, without completion detection, is given as a Verilog behavioural model in Appendix C.2. This corresponds to the multiplier described in [12, section 8.2.7.1]; the final summation required is performed by reusing the existing adder. Each cell in the two-dimensional array of cells forming the multiplier consists of a full-adder and multiplexer, which can both be efficiently implemented using pass-transistor logic [10].

The multiplier outputs are one bit wider than the rest of the data path, as a consequence the other inputs to the adder input multiplexers are extended by concatenating a 0 LSB bit (before optional inversion for subtraction). After addition a $w + 2$ bit result is produced, the top (carry out) bit is removed as the choice of the number of bits before the binary point ensures the carry is never needed and the bottom bit, which is only non-zero in the case of a multiplication, is removed.

The FIFO buffer at the adder output is required to avoid deadlock when a result from the lower section is to be written back into a queue which is already full. A two stage FIFO is required for the DCT algorithm as at the start of the algorithm two additions need to be performed and both results written into a queue which is still full. An alternative strategy would be to increase the length of some or all of the four queues.

7.4.4 DCT Algorithm

The 1-D DCT algorithm by Arai *et al* [45] was modified to remove and reposition some subtractions. The intention was to arrange that only one adder input require negation. However, the initial version of the microcode contained an error, which when corrected required both adder inputs to be capable of negation (though not both at the same time). This alteration would most likely be needed for other algorithms anyway. The modified Arai algorithm is shown in Figure 7.2. A C version of the algorithm, directly reflecting the data flow in the architecture, is given in Appendix C.1.

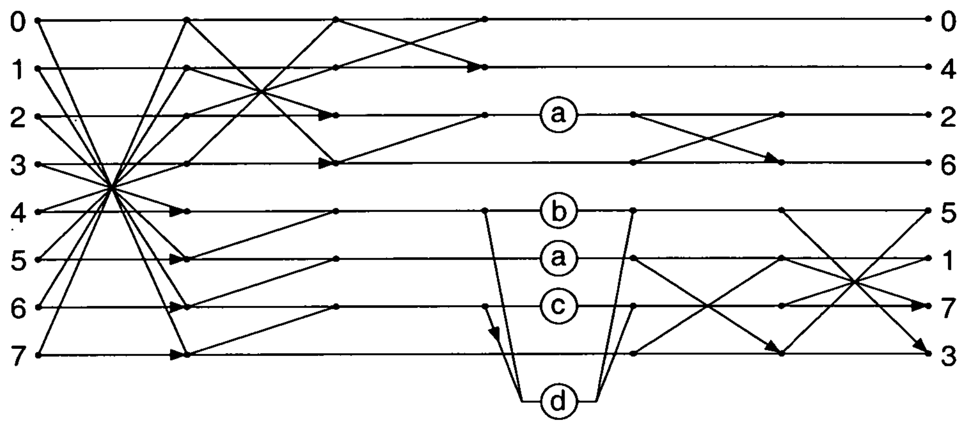


Figure 7.2: *Modified Arai DCT algorithm*

7.4.5 Fixed point arithmetic

Throughout the system all data is encoded using two's complement fixed point representation. The main data path is w bits wide. Input pixel data (typically 12 bits) and output DCT coefficients (typically 9 bits) should be sign-extended and shifted accordingly, this can be done after the input FIFO and before the output FIFO buffers. The signal flow graph (Figure 7.2) can be divided into three sections: (i) stages (columns in the diagram) before the multiplications, (ii) the multiplication stage and (iii) the stages after multiplication. To avoid shifting operations to correct bit alignment and avoid overflow it is necessary to determine the range of values possible at each of the three stages.

Input pixel data is in the range $[-1, +1]$. After a maximum of three additions the range is $[-8, +8]$. The largest multiplier coefficient is approximately 1.3, which when rounded up to a power of two gives a data range after multiplication of $[-16, +16]$. A further three additions can then occur giving an output range of $[-128, +128]$. Thus the input pixel data is represented with four digits before the binary point (4bpp) and the output DCT coefficients with 8bpp. To avoid shifting operations to re-scale values the multiplier coefficients are stored as 4bpp (4bpp multiplied by 4bpp produces 8bpp product). Values which are not multiplied in the flow graph are multiplied by one (under the 4bpp representation) to attain the correct bit position compared to the other signals. Three such multiplications are required, placed in the 1st, 2nd and 4th rows in the signal flow graph.

7.4.6 Extensions and application to other algorithms

Although the architecture is controlled by microcode ROMs, only the DCT has been implemented. It should be possible, but has not been checked, to implement other waveform transforms, perhaps with minor alterations. As the architecture stands only algorithms in which the control flow is not influenced by data values can be implemented because the three control units operate independently of the data values. Additionally the architecture currently does not provide a division operation and only supports a fixed precision. It is possible to implement the inverse DCT (also based upon the Arai algorithm) simply by modifying the control ROM contents.

To implement other algorithms various design parameters may need to be adjusted, including: the length of the adder output FIFO, the length and number of queues, and the data path width. Additionally if a large number of multiplications are to be performed a FIFO placed between the multiplier and adder would increase concurrency, and a barrel shifter might be faster than multiplying by 'one' to scale values not being multiplied. A second input to the multiplier to permit one data value to be multiplied by another might also be desirable.

Another extension would be to replace the control ROMs with RAM allowing the algorithm to be changed at run-time. A sufficient number of queues of suitable length for the 'largest' algorithm to be used would be required, however, because the queues are elastic, algorithms which do not need the full queue size would still run correctly.

The microcode for the DCT was written by hand from examination of the signal flow graph. It should be possible to automate this process using a tree search type method, although the search space is likely to be quite large.

Finally performance could be increased by extending the architecture to use multiple functional units and permit concurrent writing and reading of the queues, however this would require a more sophisticated control mechanism.

7.5 Simulation

The architecture with the DCT algorithm was simulated in Verilog. A structural model composed from behavioural models of the basic units was used to obtain a good compromise between a complete circuit description and a fast design and simulation time. The three control units, the adder and multiplier exist only as a behavioural model. In particular, details of how to implement the control units have not been examined; a behavioural model is suitable for experimenting with the architecture. Although simulated using the two-phase handshake protocol for simplicity, a four-phase protocol could also be used. The model is fully parameterised, allowing properties such as the number of queues, or the data path width to be changed easily. The simulation shows that the architecture works correctly for the delay parameters used in the simulation. Although from examination it would seem unlikely that other delay parameter values may result in failure, a more formal method of verification would be needed to guarantee this.

A C program to model the algorithm using floating-point arithmetic was implemented as a reference. As an additional check the program computes the DCT using equation (6.1). The C program produces input vectors for the Verilog simulation and output vectors for comparison with the simulation output. The simulation was tested for several input data sequences, including the 'DC input' case where all the input values are the same.

7.6 Review

Performance

The implementation area should be small because only one adder and multiplier are used. The FIFO queues are built from transparent latches rather than registers, this should further reduce area and power consumption. However, because there is only a single adder, many operation cycles are needed (46 for the upper section with the DCT) and performance is slow. Multiple units could be used in parallel, for example sixteen DCT units and a matrix transposition unit to compute a complete 2-D DCT. However this would require sixteen bit-parallel combinatorial multipliers.

Experiments with real image data would need to be performed to assess how much performance gain could be obtained by using completion detection instead of delay matching for the adder and multiplier. The microcode algorithm used for the DCT was arranged to ensure that operations such as $A + B$ and $A - B$ are adjacent, this may increase the benefit of completion detection and should reduce power consumption by avoiding unnecessary changes on one adder input. If little gain is achieved by the use of completion detection it is debatable if this style of DCT algorithm, involving a regular and constant load computation, could benefit from an asynchronous implementation. However, aside from data dependent processing delays an asynchronous implementation does make elastic queues straightforward and permits asynchronous input and output interfaces which need not operate at a constant data rate.

The architecture was simulated using the two-phase protocol for simplicity and ease of design. Although for best performance the linear FIFO queues should be two-phase (see Chapter 3), the four-phase protocol may be preferable when implementing the control circuitry.

Timing assumptions

Much use of delay matching is used, this has the disadvantage of delivering worst-case performance and requiring careful device level simulation prior to producing a physical implementation. In particular the functions performed by the control circuitry include delay matching for the multiplexers. One alternative strategy would be to use dual-rail encoding in the data path, however, this would significantly add to the area and power consumption as the architecture is dominated by the data path.

Correctness

A side effect of the property that the control algorithm operates independently of the data is that it if formal verification were to be used, only the control functions need be verified without the problem of data dependencies leading to a large state space. Provided the control units are deadlock free, the only remaining feedback path (and hence a potential deadlock) is the transfer of data from the lower section back to the upper section. Deadlock as a consequence of this path, for example if the adder output FIFO is too short, would be observed during simulation.

7.7 Conclusion

Initial work on a new ROM programmed architecture, with the potential for run-time reconfigurability, has been presented. The architecture has been programmed to implement the 1-D DCT and is potentially extendible to other waveform transforms. The architecture, with the DCT algorithm, has been successfully simulated using Verilog. Future work could include extensions to support other transforms, or to increase performance.

Chapter 8
Conclusions and discussion

In this thesis aspects of micropipeline controller design, verification and application have been examined. This final chapter summarises the work in earlier chapters. Achievements are identified and with the benefit of hindsight, limitations, potential for future work and more general issues are discussed.

8.1 Summary and review

Chapter 2 introduced asynchronous circuits and their differences from synchronous circuits, with focus on the micropipeline design style used in later chapters.

In Chapter 3 micropipeline latch controllers were examined. From this study a new, independently discovered, two-phase latch controller circuit was developed. SPICE simulations show that in comparison to existing two-phase latch controllers, the new circuit is faster, with a lower power consumption and fewer transistors. The new circuit has the same benefits when compared with the fully-decoupled four-phase controller of equivalent functionality. When compared with the semi-decoupled four-phase controller, the new circuit is faster, but at the expense of more transistors.

However, the comparison results are only valid for a simple linear micropipeline, free from other asynchronous circuit elements required for forking and merging. These other elements are generally known to be faster and smaller when implemented using the four-phase protocol instead of the two-phase protocol, a fact confirmed by the predominance of four-phase circuitry in the asynchronous literature. It would seem therefore that the application of the new controller is limited; however, a suitable application is the two-dimensional micropipeline structure presented in Chapter 5. Although the two-phase protocol is aesthetically desirable, in general using the four-phase protocol requires less circuitry, a consequence of the fact that transistors are level-sensitive rather than edge-triggered.

In Chapter 4 a method of modelling asynchronous circuits using CCS process algebra was developed. Processes are used to represent circuit elements and the parallel composition and restricted synchronisation features of CCS used to connect elements. The same compositional approach can be used to combine modules within a larger circuit. The concurrency workbench (CWB) is used to check properties of the models, or to test for equivalence between a model of a circuit and a specification. Once a model of a module has been shown equivalent to a specification, the specification can be used in place of the more complex module model within a hierarchical composition.

Initially believed to be a straightforward application of CCS, more involved issues soon became apparent, namely those of quenching and timing assumptions, in particular isochronic forks. At first, logic levels were modelled but then the more efficient scheme of modelling events was used. Two ways of permitting the detection of interference between signals were discussed, one involving wires with error states and the other where gate inputs permit quenching. The latter involves a smaller state space and was preferred. During this work previous literature using CCS was found and was found to be consistent with this work. This method has the unique property that the traditionally undesirable behaviour of quenching is permitted inside a circuit, provided that the externally observable behaviour is correct. This seemed a more natural way of modelling a circuit and helps to reduce the state space of the model.

The modelling method was then applied to the verification of several circuits. To test the method in practice, the standard two-phase latch controller and semi-decoupled latch controllers from Chapter 2 were re-verified. In the case of the buggy semi-decouple four phase controller, a well known timing assumption was correctly exposed by the model. In Chapter 5 the method was successfully applied to the verification of empty- and full-detection circuitry for a micropipeline FIFO. Two approaches were tried, one a direct CCS approach and the other using a combination of STG specification and CCS verification. Both approaches were successful but the latter was found to be somewhat simpler.

Isochronic forks were found to be both awkward to model and to give a large increase in state space, especially when used in conjunction with large circuit elements, for example the asymmetric three input C-element. This suggests that, although suitable for the verification of quasi-delay-insensitive circuits with a small number of isochronic forks, the method is perhaps not appropriate for the verification of the larger class of speed independent circuits. In practice the modelling method was found to be well suited to the verification of small gate-level hand-designed circuits.

Next, two applications which are micropipeline intensive and hence potentially suitable for use with the simplified latch controller were studied. The first application has two uses in the implementation of the discrete cosine transform (DCT): (i) bit-parallel to bit-serial conversion for a bit-serial DCT, (ii) matrix transposition for a bit-parallel DCT. The second application is a fully asynchronous microcoded architecture, which was used to implement the DCT.

In Chapter 5 a new two-dimensional micropipeline architecture was developed for converting between bit-parallel and bit-serial data. Independent work, which uses a similar architecture to perform matrix-transposition, was reviewed and the new architecture was further simplified. The architecture has the potential to offer area and significant power savings compared to the synchronous equivalent.

Key aspects, namely those of full and empty detection, in both the complex and simple variants of the architecture, were modelled using the CCS method from Chapter 4, with enhancements to permit the incorporation of timing assumptions. Combined with the verification of the latch controllers, the modelling work has illustrated how verification rather than simulation, which does not test for timing assumptions, is important even for apparently simple circuits.

Chapter 6 reviewed recent synchronous DCT implementations, highlighting the roles of algorithmic and circuit design, with a quantitative comparison of implementation area and speed. From this review it was concluded that row-column decomposition methods require specific circuit techniques to achieve high performance and that higher performance may be obtained by the use of direct 2-D approaches at the expense of area.

From the review of algorithms in Chapter 6 an algorithm with a low number of multiplications was chosen as an algorithm for use with the architecture described in Chapter 7. Compared to the number of synchronous implementations there are very few asynchronous DCT implementations, all of which have poor performance.

Chapter 7 described initial work on an asynchronous application specific processor (ASP) architecture, on which a DCT was implemented. The architecture exploits the elasticity of micropipeline FIFOs to store intermediate results during computation and is controlled by microcode ROMs, permitting the implementation of the inverse DCT or potentially other transforms. Although, involving linear micropipeline FIFOs, the architecture might seem well suited to the simplified latch controller, it is likely that the control circuitry (only developed to a behavioural model) would be most efficiently implemented using four-phase circuitry due to the reasons discussed earlier.

8.2 Future work

Several areas, directly related to the work discussed, in which future work is possible have been identified.

- Although four-phase logic seems to dominate the asynchronous literature, there would appear to be little literature which performs an explicit comparison between two and four-phase circuits. Further work to compare the speed, power, area, complexity of timing constraints and ease of design of two- and four-phase circuits could be performed. The majority of existing literature is primarily concerned with maximum speed, different conclusions might be reached if speed were a secondary concern to minimising power consumption.
- The CCS modelling method would benefit from state space reduction techniques to permit faster modelling of larger circuits.
- The methods of full- and empty-detection in the two-dimensional micropipeline architecture were only verified for a limited number of pipeline stages. A formal proof could be constructed to show the result holds for an arbitrary number of stages.
- Layout could be performed for both the two-dimensional micropipeline and a synchronous equivalent. SPICE simulations with capacitance extraction could then be used to accurately compare power and speed.
- The review of DCT literature highlights some promising algorithms without implementations. For consumer video applications an inverse only DCT implementation, which exploits zero coefficients, of which there are few implementations, may offer advantages.
- Much work is needed in the field of asynchronous DCT implementations if a performance comparable to synchronous implementations is to be achieved.
- Reconfigurable or programmable application specific asynchronous circuits may have advantages over their synchronous counterparts. The use of completion detection and handshake protocols should permit good average case performance whilst avoiding timing validation issues. New reconfigurable and programmable architectures, previously

non-viable with a synchronous implementation, may be possible. This whole area deserves systematic investigation.

8.3 Discussion

This thesis has focused on specific aspects of micropipeline design and verification. On a broader scale the world-wide asynchronous community is performing research on both micropipelines and other asynchronous methodologies.

The common criticisms are that compared to synchronous circuits, asynchronous circuits are:

- hard to design,
- hard to verify,
- hard to test after manufacture,
- slow.

Purely synchronous circuits appear to be easy to design, the use of discrete time intervals removes most of the need to worry about races and other hazards. However, clock distribution and skew is becoming an increasing problem as fabrication improvements push up die sizes and push down feature sizes. In practice, many synchronous circuits are not purely synchronous anyway, in addition inputs to a circuit are ultimately asynchronous.

In a synchronous design a module interface may have many timing constraints, for example setup and hold times. This can make the independent development of modules difficult because the timing constraints of one module influence the design of another. This problem is amplified when modules which have been laid out (hard intellectual property blocks) are exchanged. This style of pre-designed module is likely to be used for high performance data path modules and the high speed leads to tight timing constraints. With most asynchronous interfaces there is a reasonable amount of delay insensitivity, leading to fewer timing constraints and less dependence on the particular implementation or technology. This increased ease of composition from pre-designed modules should aid system design.

A frequent aspect of synchronous circuit design involves simulation using back annotation with delays derived from the layout, for example to test the safety of timing assumptions. Several iterations of layout and simulation may be involved to meet timing requirements imposed by the circuit specification. Such simulations are usually repeated for a range of process parameters and operating conditions. Ideally a circuit would be fully delay insensitive, such that circuit correctness is independent of layout, technology and process parameters, operating temperature and voltage (within sensible limits) and the timing of signals supplied by the environment connected to the circuit. In practice useful asynchronous circuits involve timing assumptions and a compromise has to be made.

For a moderately complex synchronous circuit it is straightforward to simulate a synchronous circuit using a discrete event simulator to test its behaviour, at least at an algorithmic level. For large circuits, for example a microprocessor, formal methods are often useful to assist with demonstrating correctness, but this is to test logical correctness assuming discrete time intervals (i.e. a clock), not correctness of the circuit under the non-deterministic choice resulting from arbitrary (though perhaps bounded) wire and gate delays. Modelling the latter tends to involve a larger state space, as all the 'intermediate' states of signal propagation are needed, not just the states due to state keeping registers.

Formal methods for verification or synthesis are a critical part of asynchronous circuit design. Discrete event simulation involves particular delays being assigned to wires and gates which is not compatible with the ideal of delay insensitivity in an asynchronous circuit. In practice, simulation can be very helpful in the design of an asynchronous circuit, but it does not provide property checking, for example guarantee of freedom from deadlock. Formal methods are usually only appropriate to verify the control paths of a circuit, as the verification of data paths would involve an intractable state space. This is analogous to simulating the circuit for all possible data patterns.

Asynchronous circuits may be hard to test after manufacture. It is not only necessary to test for behavioural errors, it is necessary to test for timing related errors [74]. A search of the asynchronous bibliography [75] suggests that further research in this area is required.

It is widely viewed that the overhead of handshaking in an asynchronous circuit results in lower performance than that of the synchronous equivalent. However, asynchronous circuits permit data dependent processing times which may yield better average case performance even if worst case performance is not as good. In addition asynchronous circuits permit new architectures for which synchronous versions may be impossible or inefficient. Other strategies include globally-asynchronous locally-synchronous systems [76], permitting well established high performance synchronous design methods to be mixed with the compositional advantages of asynchronously communicating modules. However, synchronous designs still dominate in the high performance processor industry. Many (and arguably the majority of by number of units sold) VLSI products do not require the ultimate in high speed, for example devices present in personal digital assistants (PDAs), mobile telephones, automobiles and consumer electronic items. In these products the efficient use of battery power is more important and the reduced electromagnetic noise from asynchronous circuits may be an advantage. For non speed critical applications ease of design may be more important, and the avoidance of clock distribution and lengthy simulation to check timing would reduce design time and hence time to market.

Although the majority of asynchronous interest has been (and still is) academic, commercial interest is on the increase and products are starting to be developed. Berkel, working at Philips Research Laboratories, has developed a method of specification and synthesis, known as Tangram [77], which was subsequently used to design the error correction decoder for DCC players [78]. Theseus Logic have developed an asynchronous DCT [67] and Cogency Technology have produced an asynchronous DSP core [13]. Sun Microsystems have performed research on the counterflow pipeline processor [79]. The Manchester based Amulet group are working on the third version of an asynchronous micropipelined implementation of the ARM processor [9]. Recent additions to this list are the New Media Processor [80] from Sharp and it is known that Intel is working on an asynchronous instruction decoder. Despite these examples, much more commercial interest and CAD support is required if asynchronous circuits are to regain a foothold in a world dominated by synchronous design.

It is perhaps not hard to see why asynchronous design is not popular, synchronous design methods, simulation and tools are well established, the methods used in asynchronous circuit design and verification are still subject to much academic research. Much further work is needed toward making asynchronous circuits easier to implement and use. The majority of the literature discusses very specific aspects of asynchronous design rather than broader design issues. Compared to synchronous design there is a vast shortage of design experience. As discussed briefly above, asynchronous circuits may have a fair amount to offer designers, however, if asynchronous design to become common place, tried and tested design methodologies need to be established.

Appendix A
CWB code for models and specifications

A.1 Basic circuit elements

```
* Xor
* XOR/XNOR gate
*       agent:  Xor
*       input:  in1, in2
*       output: out
*
agent      Xor    =  in1.Xor-e + in2.Xor-e;
agent      Xor-e  =  in1.Xor   + in2.Xor    + 'out.Xor;


* Toggle
* Toggle element
*       agent:  Toggle
*       input:  in
*       output: dot, blank
*
agent    Toggle    =  in.('dot.Toggle-b + in.Toggle);
agent    Toggle-b  =  in.('blank.Toggle + in.Toggle-b);


* And
* Two-input AND gate.
*       agent:                      And
*       ports:                      in1,in2,out
*
* Agent is labelled as And_<in1><in2><out>
* And_000 is shortened to And, this saves need to
* use min to remove the extra state and keeps tidy names.
*
agent   And     = in1.And_100 + in2.And_010;
agent   And_100 = in1.And     + in2.And_110;
agent   And_010 = in1.And_110 + in2.And;
agent   And_110 = in1.And_010 + in2.And_100 + 'out.And_111;
agent   And_001 = in1.And_101 + in2.And_011 + 'out.And;
agent   And_101 = in1.And_001 + in2.And_111 + 'out.And_100;
agent   And_011 = in1.And_111 + in2.And_001 + 'out.And_010;
agent   And_111 = in1.And_011 + in2.And_101;


* Fork
* Fork in wire, allows either output order, and quenching
*       agent:  Fork
*       input:  in
*       output: out1, out2
*
agent     Fork     =  in.Fork-e;
agent     Fork-e   =  in.Fork    + 'out1.Fork-e2 + 'out2.Fork-e1;
agent     Fork-e1  =  'out1.Fork + in.Fork-e2;
agent     Fork-e2  =  'out2.Fork + in.Fork-e1;


* DataLatch
* Simulated multibit data latch (without data), used to produce
* an observable capture action. Note quenching is not needed as
* this is just a buffer.
*       agent:  DataLatch
*       input:  enable
*       output: done
*
agent  DataLatch  =  enable.capture.'done.enable.'done.DataLatch;
```

A.2 Muller C-elements

```
* Muller C-elements
*
* Naming examples:
*
*       Cpbn
*
* A a C-element with one input which can only make the output
* go _Positive_, one input which is a 'normal' input (can make
* the output go _Both_ positive and negative), and one input
* which can only make the output go _Negative_.
*
* Likewise the inputs are named p1, p2, b1, n1 etc.
*
* If an input is _Inverted_ (with a bubble) then the C-element
* name would be like this (but the input names are unchanged):
*
*       Cpibn
*
* 'Internal' agent names do not have inverted inputs marked (as they
* are reused), and they have the current state of inputs and maybe
* also the output in the name. For example, for Cpibn, an agent used
* in its construction might be Cpbn-010-0, indicating p1=0, b1=1,
* n1=0 and out=0.
*


* Standard 2-input C-element
*       agent:   Cbb
*       input:   b1, b2
*       output:  out
*
agent    Cbb       =  b1.Cbb-b1 + b2.Cbb-b2;
agent    Cbb-b1    =  b1.Cbb    + b2.Cbb-e;
agent    Cbb-b2    =  b1.Cbb-e  + b2.Cbb;
agent    Cbb-e     =  b1.Cbb-b2 + b2.Cbb-b1 + 'out.Cbb;


* Standard 2-input C-element with bubble on one input
*       agent:  Cbib
*       input:  b1 (inverted), b2
*       output: out
agent   Cbib       = Cbb-b1;

* Minimise to save one state
min(Cbib, Cbib);


* Asymmetric 2-input C-element, one both input and one + input
*       agent:  Cbp
*       input:  b1,p1;
*       output: out
*

agent Cbp  =  Cbp-00-0 ;
agent Cbp-00-0          = b1.Cbp-10-0    + p1.Cbp-01-0;
agent Cbp-10-0          = b1.Cbp-00-0    + p1.Cbp-11-1-e;
agent Cbp-01-0          = b1.Cbp-11-1-e  + p1.Cbp-00-0;
agent Cbp-11-1-e        = b1.Cbp-01-0    + p1.Cbp-10-0    + 'out.Cbp-11-1;
agent Cbp-11-1          = b1.Cbp-01-0-e  + p1.Cbp-10-1;
agent Cbp-10-1          = b1.Cbp-00-0-e  + p1.Cbp-11-1;
agent Cbp-01-1          = b1.Cbp-11-1    + p1.Cbp-00-0-e;
agent Cbp-01-0-e        = b1.Cbp-11-1    + p1.Cbp-00-0-e  + 'out.Cbp-01-0;
agent Cbp-00-0-e        = b1.Cbp-10-1    + p1.Cbp-01-0-e  + 'out.Cbp-00-0;


* Asymmetric 2-input C-element, one both and one inverted + input
*       agent:  Cbpi
*       input:  b1, p1 (inverted);
*       output: out
*
agent Cbpi = Cbp-01-0;

* Minimise to save one state
min(Cbpi, Cbpi);
```

The CWB code for other C-elements is constructed in a similar manner, and to conserve space is not given here.

A.3 Latches with isochronic fork

```
* Latch
* Pair of level sensitive latches, as used in the two-phase
* controller without toggle. When composing LatchL1 and
* LatchL2 they must be made to synchronise on latchpairsync.
*

* the real definition
*
agent Latch-EN-wait          =  latchpairsync.Latch-DIS-sync
                                    + in.Latch-e-wait;
agent Latch-EN               =  in.Latch-e
                                    + enable.Latch-EN-wait;
agent Latch-DIS-sync-wait    =  latchpairsync.Latch-EN
                                    + in.Latch-DIS-outsync-wait;
agent Latch-DIS-sync         =  enable.Latch-DIS-sync-wait
                                    + in.Latch-DIS-outsync;
agent Latch-DIS-outsync-wait =  latchpairsync.Latch-e
                                    + in.Latch-DIS-sync-wait;
agent Latch-DIS-outsync      =  enable.Latch-DIS-outsync-wait
                                    + in.Latch-DIS-sync;
agent Latch-e-wait           =  latchpairsync.Latch-DIS-outsync
                                    + in.Latch-EN-wait
                                    + 'out.Latch-EN-wait;
agent Latch-e                =  'out.Latch-EN
                                    + in.Latch-EN
                                    + enable.Latch-e-wait ;

* Latch 'l1' (the one which rin is connected to)
*       agent:  LatchL1
*       input:  in, enable
*       output: out
*       also: latchpairsync
*
agent LatchL1 = Latch-EN;

* Minimise to save one state
min(LatchL1, LatchL1);

* Latch 'l2' (the one which drives rout and ain)
*       agent:  LatchL2
*       input:  in, enable
*       output: out
*       also:   latchpairsync
*
agent LatchL2 = Latch-DIS-sync['latchpairsync/latchpairsync];

* Minimise to save one state
min(LatchL2, LatchL2);
```

A.4 Environments

```
* EnvTwoPhaseL, EnvTwoPhaseR
* A valid two-phase environment for a micropipeline to operate in.
* The inputs (that is outputs from the micropipeline) allow quenching.
*
* Example:
*   agent Implementation = ( EnvTwoPhaseL | Micropipeline | EnvTwoPhaseR )
*                          \ ENVTWOPHASESET;
*
*       agents:                        EnvTwoPhaseL, EnvTwoPhaseR
*       observable actions:            rin,ain,rout,aout
*       interface to micropipeline:    rinp,ainp,routp,aoutp
*
agent   EnvTwoPhaseL    =  rin.'rinp.EnvTwoPhaseL-w;
agent   EnvTwoPhaseL-w  =  ainp.('ain.EnvTwoPhaseL + ainp.EnvTwoPhaseL-w);
agent   EnvTwoPhaseR    =  routp.('rout.aout.'aoutp.EnvTwoPhaseR
                                    + routp.EnvTwoPhaseR);
set     ENVTWOPHASESET  =  { rinp, routp, ainp, aoutp };


* EnvFourPhaseL, EnvFourPhaseR
* A valid four-phase environment for a micropipeline to operate in.
* The inputs (that is outputs from the micropipeline) allow quenching.
*
* Example:
*   agent Implementation = ( EnvFourPhaseL | Micropipeline | EnvFourPhaseR )
*                          \ ENVFOURPHASESET;
*
*       agents:                        EnvFourPhaseL, EnvFourPhaseR
*       observable actions:            rin,ain,rout,aout
*       interface to micropipeline:    rinp,ainp,routp,aoutp
*
* Note this is correctly the same as EnvTwoPhase.

agent   EnvFourPhaseL    =  rin.'rinp.EnvFourPhaseL-w;
agent   EnvFourPhaseL-w  =  ainp.('ain.EnvFourPhaseL + ainp.EnvFourPhaseL-w);
agent   EnvFourPhaseR    =  routp.('rout.aout.'aoutp.EnvFourPhaseR
                                + routp.EnvFourPhaseR);
set     ENVFOURPHASESET  =  { rinp, routp, ainp, aoutp };
```

A.5 Two phase specification

```
* SpecTwoPhase
* A correct two phase latch controller operating in a valid
* environment.
*       agent:                  SpecTwoPhase
*       observable actions:     rin,ain,rout,aout,capture
*   ('capture' is to observe when the data latch captures new data)
*
agent   SpecTwoPhase-L  =  rin.sync.capture.'sync.'ain.SpecTwoPhase-L;
agent   SpecTwoPhase-R  =  'sync.sync.'rout.aout.SpecTwoPhase-R;
agent   SpecTwoPhase    =  ( SpecTwoPhase-L | SpecTwoPhase-R ) \ {sync};
```

A.6 Standard two-phase latch controller

```
* Build up the latch controller circuit.
* Interface to "Circuit" is rin,ain,rout,aout.
*

* C-element
agent     C1                    = Cbib[c1/out,t1blank/b1,rin/b2];

* xor
agent     X1                    = Xor[x1/out,c1/in1,aout/in2];

* datalatch (on output of xor)
agent     D1                    = DataLatch[d1/done,x1/enable];

* toggle (on output of data latch)
agent     T1                    = Toggle[t1dot/dot,t1blank/blank,d1/in];

* fork1 from output of toggle to give ain and rout
agent     F1                    = Fork[ain/out1,rout/out2,t1dot/in];

* whole circuit
agent     Circuit               = ( C1 | X1 | D1 | T1 | F1 )
                                \ {c1,x1,d1,t1dot,t1blank};

* rename ports on circuit to be compatible with environment
agent     Circuit'              = Circuit[rinp/rin,routp/rout,
                                          ainp/ain,aoutp/aout];

* attach Circuit' to environment
agent     Implementation         = ( EnvTwoPhaseL | Circuit' | EnvTwoPhaseR )
                                 \ ENVTWOPHASESET;

* hopefully, this will be true
eq(SpecTwoPhase,Implementation);
```

A.7 Simplified two-phase latch controller

```
* Build up the latch controller circuit.
* Interface to "Circuit" is rin,ain,rout,aout.
*

* latch l1
agent     L1                    = LatchL1[l1/out,f21/enable,rin/in];

* fork from output of l1 to XOR and l2
agent     F1                    = Fork[f11/out1,f12/out2,l1/in];

* xor
agent     X1                    = Xor[x1/out,f11/in1,aout/in2];

* datalatch (on output of xor)
agent     D1                    = DataLatch[d1/done,x1/enable];

* fork2 from output of datalatch to l1 and l2
agent     F2                    = Fork[f21/out1,f22/out2,d1/in];

* latch l2
agent     L2                    = LatchL2[l2/out,f22/enable,f12/in];

* fork 3 from output of l2 to ain and rout
```

```
agent      F3                  = Fork[ain/out1,rout/out2,12/in];

* whole circuit
agent      Circuit             = ( L1 | F1 | X1 | D1 | F2 | L2 | F3 )
                               \ {l1,f11,f12,x1,d1,f21,
                                  f22,12,latchpairsync};

* rename ports on circuit to be compatible with environment
agent      Circuit'            = Circuit[rinp/rin,routp/rout,
                                         ainp/ain,aoutp/aout];


* attach Circuit' to environment
agent      Implementation      = ( EnvTwoPhaseL | Circuit' | EnvTwoPhaseR )
                               \ ENVTWOPHASESET;

* hopefully, this will be true
eq(SpecTwoPhase,Implementation);
```

The four-phase latch controller circuits are constructed in a similar manner.

A.8 STG to CCS conversion

STG input

```
rin
'rout
'ain
aout
capture
END
rin -> capture : 0
capture -> 'ain : 0
'ain -> rin : 1
capture -> 'rout : 0
'rout -> aout : 0
aout -> 'rout : 1
aout -> capture : 1
```

CCS output

```
agent TwoPhase = TwoPhase0;
agent TwoPhase0 = rin.TwoPhase1;
agent TwoPhase1 = capture.TwoPhase2;
agent TwoPhase2 = 'ain.TwoPhase3 + 'rout.TwoPhase4;
agent TwoPhase3 = rin.TwoPhase5 + 'rout.TwoPhase6;
agent TwoPhase4 = 'ain.TwoPhase6 + aout.TwoPhase7;
agent TwoPhase5 = 'rout.TwoPhase8;
agent TwoPhase6 = rin.TwoPhase8 + aout.TwoPhase9;
agent TwoPhase7 = 'ain.TwoPhase10;
agent TwoPhase8 = aout.TwoPhase1;
agent TwoPhase9 = rin.TwoPhase1;
agent TwoPhase10 = rin.TwoPhase1;
```

A.9 Additional specifications and environments

```
* Event representation to level representation converter.
*       agent:  EL
*       input:  lo,hi
*       output: out
*       also:   losync
* out is initially "low". Events on hi and lo are used to
* change the state of out. hi, lo, 'losync are always possible
* from all states. Quenching of changes to out by further input
* can occur.
*
* 'losync is used to synchronise with out being low, i.e. after
* synchronisation with losync, 'out can not then occur until hi
* occurs. Note that 'losync does not cause out to go low,
* 'losync cannot happen if the last input was a hi.
*
agent   EL0  = lo.EL0  + hi.EL0e + 'losync.EL0;
agent   EL0e = lo.EL0  + hi.EL0e + 'out.EL1;
agent   EL1  = lo.EL1e + hi.EL1;
agent   EL1e = lo.EL1e + hi.EL1 + 'out.EL0;
min(EL,EL0);
* Restriction set for using with EL
```

```
set     ELSET  =  { lo, hi, losync };


* SpecTwoPhaseFull
* A correct two phase latch controller operating in a valid
* environment. The latch controller starts off full and with
* a request waiting on rin.
*
*       agent:                  SpecTwoPhaseFull
*       observable actions:     rin,ain,rout,aout,capture
*   ('capture' is to observe when the data latch captures new data)
*
agent   SpecTwoPhaseFull-L  =  sync.capture.'sync.'ain.rin.SpecTwoPhaseFull-L;
agent   SpecTwoPhaseFull-R  =  aout.'sync.sync.'rout.SpecTwoPhaseFull-R;
agent   SpecTwoPhaseFull    =  ( SpecTwoPhaseFull-L | SpecTwoPhaseFull-R )
                               \ {sync};
min(SpecTwoPhaseFull,SpecTwoPhaseFull);


* SpecTwoPhaseFullLeft
* A correct two phase latch controller operating in a valid
* environment. The latch controller starts off full but with no
* request waiting on rin.
*       agent:                  SpecTwoPhaseFull
*       observable actions:     rin,ain,rout,aout,capture
*   ('capture' is to observe when the data latch captures new data)
*
agent   SpecTwoPhaseFullLeft    =  ( SpecTwoPhase-L | SpecTwoPhaseFull-R )
                                   \ {sync};
min(SpecTwoPhaseFullLeft,SpecTwoPhaseFullLeft);

* EnvTwoPhaseFullL, EnvTwoPhaseFullR
* A valid two-phase environment for a micropipeline to operate in,
* where the micropipeline starts off full.
* The inputs (that is outputs from the micropipeline) allow quenching.
*
* Example:
*   agent Implementation = ( EnvTwoPhaseL | Micropipeline | EnvTwoPhaseR )
*                          \ ENVSET;
*
*       agents:                         EnvTwoPhaseL, EnvTwoPhaseR
*       observable actions:             rin,ain,rout,aout
*       interface to micropipeline:     rinp,ainp,routp,aoutp
*

agent   EnvTwoPhaseFullL    = ainp.('ain.rin.'rinp.EnvTwoPhaseFullL
                                    + ainp.EnvTwoPhaseFullL);
agent   EnvTwoPhaseFullR    = aout.'aoutp.EnvTwoPhaseFullR-w;
agent   EnvTwoPhaseFullR-w  = routp.('rout.EnvTwoPhaseFullR
                                    + routp.EnvTwoPhaseFullR-w);
size(EnvTwoPhaseFullL);
size(EnvTwoPhaseFullR);


* EnvNullL, EnvNullR
* Null unconnected environments:
*  EnvNullL always accepts ainp but never produces rinp,
*          and produces error and deadlocks if more than one
*          ainp is received.
*  EnvNullR always accepts routp but never produces aoutp,
*          and produces error and deadlocks if more than
*          one routp is received.
*
* Example:
*   agent Implementation = ( EnvNullL | Micropipeline ) | EnvTwoPhaseR )
*                          \ ENVSET;
*
*       agents:                         EnvNullL, EnvNullR
*       observable actions:             error
*       interface to micropipeline:     ainp,routp
agent   EnvNullL        = ainp.ainp.error.0;
agent   EnvNullR        = routp.routp.error.0;
size(EnvNullL);
size(EnvNullR);
```

A.10 Two-phase latch controller with reqeust pending

```
* LCU
* A correct two phase latch controller operating in a valid
* environment.
*       agents:                 LC*
*       observable actions:     rin,ain,rout,aout,capture
*       equivalent to:          eq(LCU,SpecTwoPhase) is true
agent LCU = rin.LCU1;
agent LCU8 = aout.LCU1;
agent LCU5 = 'ain.LCU;
agent LCU1 = capture.LCU2;
agent LCU2 = 'ain.LCU4 + 'rout.LCU3;
agent LCU4 = rin.LCU7 + 'rout.LCU6;
agent LCU3 = aout.LCU5 + 'ain.LCU6;
agent LCU7 = 'rout.LCU8;
agent LCU6 = aout.LCU + rin.LCU8;


* LCURP
* A correct two phase latch controller operating in a valid
* environment with request pending (RP) added.
*
* Request pending corresponds to the circuit with XOR and AND.
*
* This is constructed with the aid of EL from elements.cwb.
* The stage graph for LCU above was plotted, and extended by
* hand to give agent LCUrp below. LCURP is then formed from
* LCU and LCUrp.
*
*       agents:                 LCU*
*       observable actions:     rin,ain,rout,aout,capture,rp
*       note:                   modified from LCU by hand to add RP
*
agent LCUrp  = rin.LCUrp1;
agent LCUrp8 = aout.'lo.LCUrp1;
agent LCUrp5 = 'ain.LCUrp;
agent LCUrp1 = capture.LCUrp2;
agent LCUrp2 = 'ain.LCUrp4 + losync.('rout.LCUrp3 + 'ain.LCUrp4);
agent LCUrp4 = rin.'hi.LCUrp7 + losync.('rout.LCUrp6 + rin.'hi.LCUrp7);
agent LCUrp3 = aout.LCUrp5 + 'ain.LCUrp6;
agent LCUrp7 = 'rout.LCUrp8;
agent LCUrp6 = aout.LCUrp + rin.'hi.LCUrp8;

agent LCURP  = ( LCUrp | EL[rp/out] ) \ ELSET;
min(LCURP,LCURP);
```

A.11 Full-detection (Method I)

```
** First check what should be true is true in case of silly errors
eq(SpecTwoPhase,LCU);            * should be true
eq(SpecTwoPhase,LCURP[tau/rp]); * should be true


** Test single stage
agent Stage1        = LCURP[rinp/rin,ainp/ain,routp/rout,aoutp/aout,
                            capture/capture,full/rp];
agent Circuit       = ( EnvTwoPhaseL | Stage1 | EnvNullR ) \ ENVSET;
size(Circuit);
min(Circuit,Circuit);
agent SingleStageSpec = rin.'ain.rin.'full.0;
eq(SingleStageSpec,Circuit[tau/capture]);


** Three stage pipeline, right most stage has null right hand environment
agent Stage1        = LCURP[rinp/rin,ainp/ain,r1/rout,a1/aout,
                            capture1/capture,rp1/rp];
agent Stage2        = LCURP[r1/rin,a1/ain,r2/rout,a2/aout,
                            capture2/capture,rp2/rp];
agent Stage3        = LCURP[r2/rin,a2/ain,routp/rout,aoutp/aout,
                            capture3/capture,rp3/rp];
agent Fifo          = ( Stage1 | Stage2 | Stage3 ) \ (r1,a1,r2,a2);
agent And-3-2       = And[rp3/in1,rp2/in2,full-3-2/out];
agent And-3-1       = ( And-3-2 | And[rp1/in1,full-3-2/in2,full/out] )
                      \ (full-3-2);
agent Pipeline      = ( Fifo | And-3-1 ) \ (rp3,rp2,rp1);
agent Circuit       = ( EnvTwoPhaseL | Pipeline | EnvNullR ) \ ENVSET;
size(Circuit);
min(ThreeStageCircuit,Circuit);
agent ThreeStageSpec = rin.'ain.rin.'ain.rin.'ain.rin.'full.0;
eq(ThreeStageSpec,Circuit[tau/capture1,tau/capture2,tau/capture3]);
```

A.12 Empty detection (Method I)

```
** First check what should be true is true in case of silly errors
eq(SpecTwoPhaseFull,LCO);                   * should be true
eq(SpecTwoPhaseFull,LCOEM[tau/em]);         * should be true
eq(SpecTwoPhaseFullLeft,LCOLEM[tau/em]);    * should be true
```

```
** Test single stage
agent Stage1      = LCOLEM[rinp/rin,ainp/ain,routp/rout,aoutp/aout,
                          capture/capture,empty/em];
agent Circuit  = ( EnvNullL | Stage1 | EnvTwoPhaseFullR ) \ ENVSET;
size(Circuit);
min(Circuit,Circuit);
agent SingleStageSpec = aout.'empty.0;
eq(SingleStageSpec,Circuit[tau/capture]);


** Three stage pipeline
agent Stage1      = LCOLEM[rinp/rin,ainp/ain,r1/rout,a1/aout,
                          capture1/capture,em1/em];
agent Stage2      = LCOEM[r1/rin,a1/ain,r2/rout,a2/aout,
                          capture2/capture,em2/em];
agent Stage3      = LCOEM[r2/rin,a2/ain,routp/rout,aoutp/aout,
                          capture3/capture,em3/em];
agent Fifo        = ( Stage1 | Stage2 | Stage3 ) \ {r1,a1,r2,a2};
agent And-1-2     = And[em1/in1,em2/in2,empty-1-2/out];
agent And-1-3     = ( And-1-2 | And[empty-1-2/in1,em3/in2,empty/out] )
                    \ {empty-1-2};
agent Pipeline    = ( Fifo | And-1-3 ) \ {em1,em2,em3};
agent Circuit     = ( EnvNullL | Pipeline | EnvTwoPhaseFullR ) \ ENVSET;
size(Circuit);
min(ThreeStageCircuit,Circuit);
agent ThreeStageSpec = aout.'rout.aout.'rout.aout.'empty.0;
eq(ThreeStageSpec,Circuit[tau/capture1,tau/capture2,tau/capture3]);
```

Appendix B
Discrete Cosine Transform Coefficients

Table B.1 gives the multiplier coefficients referred to in Chapter 6 and 7. Define $c_n = \cos n\pi/16$.

a	c_1	o	$\sqrt{2}(c_2 - c_6)$
b	c_2	p	$\sqrt{2}c_6$
c	c_3	q	$\sqrt{2}(c_2 + c_6)$
d	c_4	r	$-c_1 + c_3 + c_5 - c_7$
e	c_5	s	$c_1 + c_3 - c_5 + c_7$
f	c_6	t	$c_1 + c_3 + c_5 - c_7$
g	c_7	u	$c_1 + c_3 - c_5 - c_7$
h	$c_2 + c_6$	v	$c_3 - c_7$
i	$c_2 - c_6$	w	$c_1 + c_3$
j	$1/2c_2$	x	$c_3 + c_5$
k	$1/2c_4$	y	$c_3 - c_5$
m	$1/2c_6$		

Table B.1: *DCT multiplier coefficients*

Appendix C
Asynchronous discrete cosine transform

C.1 C model of algorithm

```
// multiplier constants
double multd=0.7071067811865475244O,
       multi=0.54119610014619698440,
       multh=1.30656296487637652786,
       multf=0.38268343236508977173;

// the queues
Fifo<double> a(5);
Fifo<double> b(5);
Fifo<double> c(5);
Fifo<double> d(5);

// initial inputs
a.i(x[3]); // input->A
a.i(x[2]); // input->A
a.i(x[0]); // input->A
a.i(x[1]); // input->A
b.i(x[4]); // input->B
b.i(x[5]); // input->B
b.i(x[7]); // input->B
b.i(x[6]); // input->B

b.i( a.v() + b.v() ); // A+B->B
d.i( a.v() - b.v() ); a.r(); b.r(); // A-B->D, ack A,B

b.i( a.v() + b.v() ); // A+B->B
c.i( a.v() - b.v() ); a.r(); b.r(); // A-B->C, ack A,B

a.i( a.v() + b.v() ); // A+B->A
c.i( a.v() - b.v() ); a.r(); b.r(); // A-B->C, ack A,B

a.i( a.v() + b.v() ); // A+B->A
```

```
d.i( a.v() - b.v() ); a.r(); b.r(); // A-B->D, ack A,B

a.i( a.v() + b.v() ); // A+B->A
d.i( a.v() - b.v() ); a.r(); b.r(); // A-B->D, ack A,B

b.i( a.v() + b.v() ); // A+B->B;
c.i( a.v() - b.v() ); a.r(); b.r(); // A-B->C, ack A,B

a.i( a.v() + b.v();                          // A+B->A
a.i( a.v() - b.v();  a.r(); b.r();           // A-B->A, ack A,B

y[0] = a.v() *1;  a.r();                  // A*1->output, ack A
y[4] = a.v() *1;  a.r();                  // A*1->output, ack A

a.i( c.v() + d.v() ); d.r(); // C+D->A, ack D
a.i( c.v() + d.v() ); c.r(); // C+D->A, ack C

b.i( c.v() + d.v() ); d.r(); // C+D->B, ack D
b.i( c.v() + 0     ); c.r(); // C+0->B, ack C

c.i( c.v() + d.v() ); c.r(); // C+D->C, ack C

c.i( c.v()*multd );    c.r(); // C*multd->C, ack C

d.i( d.v() *1 );  d.r(); // D*1->D, ack D

y[2]= c.v() + d.v();    // C+D->output
y[6]= d.v() - c.v();  c.r(); d.r(); // D-C->output, ack C,D

c.i( a.v()*multi );     // A*multi->C

c.i( b.v()*multh ); // B*multh->C

d.i( a.v() - b.v() ); a.r(); b.r(); // A-B->D, ack A,B

d.i( d.v()*multf );    d.r(); // D*multf->D, ack D

a.i( a.v()*multd);     a.r(); // A*multd->A, ack A

a.i( c.v() + d.v() ); c.r(); // C+D->A, ack C

d.i( c.v() + d.v() ); c.r(); d.r(); // C+D->D, ack C,D

c.i( a.v() + b.v() ); // A+B->C
b.i( b.v() - a.v() ); a.r(); b.r(); // B-A->B, ack A,B

y[5]= a.v() + b.v();    // A+B->output
```

```
  y[3]= a.v() - b.v();  a.r(); b.r(); // A-B->output, ack A,B

  y[1]= c.v() + d.v(); // C+D->output
  y[7]= c.v() - d.v();  c.r(); d.r(); // C-D->output, ack C,D
}
```

C.2 Behavioural model of multiplier

```
/* Multiplier, two's complement inputs, add the
 * outputs together for two's complement value.
 * Note only the upper half of the result is produced and the
 * outputs are one bit wider than the inputs (and all bits need
 * to be added to get the correct product).
 */
module mult_mult(Sout,Cout,A,B);

  parameter     bits=8;
  parameter     w=bits-1;

  output [bits:0]       Sout,Cout;
  input  [w:0]          A,B;

  reg    [bits:0]       Sout,Cout;
  reg    [w:0]          c,s;

  integer       i, j;

  always @(A or B) begin
    c=0; s=0;
    for (j = 0; j < w; j = j + 1) begin
      for (i = 0; i < w; i = i + 1)
        {c[i],s[i]} = c[i] + s[i+1] + ((A[j] == 1) ? B[i] : 0 );
      {c[w],s[w]} = c[w] +  s[w] + ((A[j] == 1) ? B[w] : 0 );
    end
    for (i = 0; i < w; i = i + 1)
      {c[i],s[i]} = c[i] + s[i+1] + ((A[w] == 1) ? !B[i] : 0 );
    {c[w],s[w]} = c[w] + s[w] + ((A[w] == 1) ? !B[w]  : 0 );
    Cout = {c[w], c[(w-1):0], (A[w] == 1) ? 1'b1 : 1'b0} ;
    Sout =  {s[w], s};
  end
endmodule
```

Appendix D
List of publications

G. S. Taylor and G. M. Blair, "Design for the discrete cosine transform in VLSI," in *IEE Proc - Comp and Dig Tech*, vol. 145, pp. 127–133, March 1998.

G. S. Taylor and G. M. Blair, "Two-dimensional micropipelines: for parallel to serial data conversion," in *IEE Electronics Letters*, vol. 34, pp. 158–159, January 1998.

G. S. Taylor and G. M. Blair, "Reduced complexity two-phase micropipeline latch controller," to appear in *IEEE Journal of Solid State Circuits*, October 1998.

G. S. Taylor and G. M. Blair, "Reduced complexity two-phase micropipeline latch controller," in *ESSCIRC'97*, pp. 304–307, September 1997.

G. Taylor, G. Clark, and G. M. Blair, "Application of CCS to verify a simplified two-phase micropipeline latch controller," in *2nd UK Asynchronous Forum, Newcastle*, July 1997.

G. Taylor, G. M. Blair, and D. Renshaw "Two-dimensional micropipelines for parallel-serial conversion and matrix transposition," in *4th UK Asynchronous Forum, University of London*, July 1998.

G. Clark and G. S. Taylor, "The verification of asynchronous circuits using CCS," Tech. Rep. ECS–LFCS–97–369, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Oct. 1997.

References

[1] D. Huffman, "The synthesis of sequential switching circuits," in *IRE Transactions on Electronic Computers*, vol. 257, pp. 161–190, 1954.

[2] D. Huffman, "The synthesis of sequential switching circuits," in *IRE Transactions on Electronic Computers*, vol. 257, pp. 275–303, 1954.

[3] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Harvard University Press, Apr. 1959.

[4] D. E. Muller, "Asynchronous logics and application to information processing," in *Symposium on the Application of Switching Theory to Space Technology*, pp. 289–297, Stanford University Press, 1962.

[5] L. Hollaar, "Direct implementation of asynchronous control units," in *IEEE Transactions on Computers*, vol. C-31, pp. 1133–1141, 1982.

[6] I. E. Sutherland, "Micropipelines," in *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[7] S. Hauck, "Asynchronous design methodologies: an overview," in *Proceedings of the IEEE*, vol. 83, pp. 67–93, January 1995.

[8] "`http://www.cs.man.ac.uk/amulet/async/async_community.html`.".

[9] S. B. Furber, J. D. Garside, and D. A. Gilbert, "AMULET3: A high-performance self-timed ARM microprocessor," in *Proc. International Conf. Computer Design (ICCD)*, Oct. 1998.

[10] K. Yano, Y. Sasaki, K. Rikino, and K. Seki, "Top-down pass-transistor logic design," in *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 792–803, June 1996.

[11] R. Milner, *Communication and Concurrency*. International Series in Computer Science, Prentice Hall, 2nd ed., 1989.

[12] N. H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design – A Systems Perspective*. Addison-Wesley, 2nd ed., 1993.

[13] N. Paver, P. Day, C. Farnsworth, D. Jackson, W. Lien, and J. Liu, "A low-power, low noise, configurable self-timed DSP," in *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 32–42, March 1998.

[14] H. Hulgaard, S. Burns, and G. Borriello, "Testing asynchronous circuits: a survey," in *Integration, the VLSI journal*, vol. 19, pp. 111–131, January 1995.

[15] D. Bormann and P. Cheung, "Asynchronous wrapper for heterogeneous systems," in *Proceedings International Conference on Computer Design*, pp. 307–314, October 1997.

[16] G. Blair, "Self-generating clocks using an augmented distribution network," in *Proc. IEE Circuits, Devices and Systems*, vol. 144, pp. 219–222, August 1997.

[17] C. L. Seitz, "System timing," in *Introduction to VLSI Systems* (C. A. Mead and L. A. Conway, eds.), ch. 7, Addison-Wesley, 1980.

[18] P. Day and J. V. Woods, "Investigation into micropipeline latch styles," in *IEEE Transactions on VLSI Systems*, vol. 3, pp. 264–272, June 1995.

[19] M. E. Dean, D. L. Dill, and M. Horowitz, "Self-timed logic using current-sensing completion detection (CSCD)," in *Journal of VLSI Signal Processing*, vol. 7, pp. 7–16, February 1994.

[20] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous circuits.* Monographs in Computer Science, Springer-Verlag, 1994.

[21] M. A. Kishinevskii, *Concurrent hardware: the theory and practice of self-timed design.* Wiley, 1994.

[22] M. B. Josephs, "Event register specification and decomposition," in *AMULET1 modelling workshop, Windermere*, July 1994.

[23] S. B. Furber and J. Liu, "Dynamic logic in four-phase micropipelines," in *Proceedings.Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 11–16, March 1996.

[24] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," in *IEEE Transactions on VLSI Systems*, vol. 4, pp. 247–253, June 1996.

[25] K. Y. Yun, P. A. Beerel, and J. Arceo, "High-performance two-phase micropipeline building blocks: double edge-triggered latches and burst-mode select an toggle circuits," in *IEE Proceedings Circuits, Devices and Systems*, vol. 143, pp. 282–288, October 1996.

[26] Y. Liu, *AMULET1: Specification and Verification in CCS.* PhD thesis, Department of Computer Science, Calgary, Alberta, 1995.

[27] C. Tofts, Y. Liu, and G. Birtwistle, "State space reduction for asynchronous micropipelines," in *BCS-FACS Northern Formal Methods Workshop*, 1996.

[28] G. Gopalakrishnan, "Asynchronous circuit verification using trace theory and CCS," June 1995.

[29] H. Barringer, D. Fellows, G. Gough, and A. Williams, "Abstract modelling of asynchronous micropipeline systems using Rainbow," in *CHDL'97*, April 1997.

[30] M. B. Josephs, "Receptive process theory," in *Acta Informatica 29, 17-31, Springer-Verlag*, 1992.

[31] A. Semenov and A. Yakovlev, "Partial order approach to design, verification and synthesis of asynchronous circuits," in *1st UK Asynchronous Forum, Edinburgh*, pp. 47–50, December 1996.

[32] M. Hennessy and R. Milner, "On observing nondeterminism and concurrency," in *Proceedings 7th ICALP,* Noordwijkerhout (J. W. de Bakker and J. van Leeuwen, eds.), vol. 85 of *Lecture Notes in Computer Science*, pp. 299–309, Springer-Verlag, July 1980.

[33] D. Kozen, "Results on the propositional mu-calculus," *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[34] F. Moller and P. Stevens, *The Edinburgh Concurrency Workbench (Version 7).* Dept. of Computer Science, University of Edinburgh.

[35] J. Tierno and P. Kudva, "Asynchronous transpose-matrix architectures," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pp. 423–428, October 1997.

[36] G. K. Wallace, "The JPEG still picture compression standard," in *Communications of the ACM*, vol. 34, pp. 31–44, April 1991.

[37] "CCITT Recommendation H.261," 1990.

[38] D. L. Gall, "MPEG: a video compression standard for multimedia applications," in *Communications of the ACM*, vol. 34, pp. 46–58, April 1991.

[39] A. Madisetti and A. N. Willson, "A 100MHz 2-D 8x8 DCT/IDCT processor for HDTV applications," in *IEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 158–165, April 1995.

[40] S.Uramoto, Y.Inoue, A.Takabatake, J.Takeda, Y.Yamashita, H.Terane and M.Yoshimoto, "A 100-MHz 2-D discrete cosine transform core processor," in *IEEE Journal of Solid State Circuits*, vol. 27, pp. 492–499, April 1992.

[41] M. Matsui, Y. Uetani, L. Kim, T. Nagamatsu, Y. Watanabe, A. Chiba, K. Matsuda, and T. Sakuri, "A 200MHz $13mm^2$ 2-D DCT macrocell using sense-amplifying pipeline flip-flop scheme," in *IEEE Journal of Solid State Circuits*, vol. 29, pp. 1482–1490, December 1994.

[42] Y. Jang, J. Kao, J. Yang, and P. Huang, "A 0.8μ 100-MHz 2-D DCT core processor," in *IEEE Transactions on Consumer Electronics*, vol. 40, pp. 703–709, August 1994.

[43] W. Chen, C. H. Smith, and S. Fralick, "A fast computation algorithm for the discrete cosine transform," in *IEEE Transactions on Communications*, vol. 25, pp. 1004–1009, September 1977.

[44] P. Duhamel and H. H'Mida, "New 2^n DCT algorithms suitable for VLSI implementation," in *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1805–1808, April 1987.

[45] Y. Arai, T. Agui, and M. Nakajima, "A fast DCT-SQ scheme for images," in *The Transactions of the IEICE*, vol. E71, pp. 1095–1097, November 1988.

[46] Y. T. Chang and C. L. Wang, "New systolic array implementation of the 2-D discrete cosine transform and its inverse," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 150–157, April 1995.

[47] K. Ray-Liu and C. T. Chiu, "Unified parallel lattice structures for time-recursive discrete cosine/sine/hartley transforms," in *IEEE Transactions on Signal Processing*, vol. 41, pp. 1357–1377, March 1993.

[48] C. T. Chiu and K. Ray-Liu, "Real-time parallel and fully pipelined two-dimensional DCT lattice structures with application to HDTV systems," in *IEEE Transactions on Circuits and Systems for Video Techonology*, vol. 2, pp. 25–37, March 1992.

[49] C. T. Chiu and K. Ray-Liu, "Real-time recursive two-dimensional DCT for HDTV systems," in *ICASSP-92: IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 205–208, March 1992.

[50] V. Srinivasan and K. Ray-Liu, "VLSI design of high-speed time-recursive 2-D DCT/IDCT processor for video applications," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 87–96, February 1996.

[51] V. Srinivasan and K. Ray-Liu, "Full custom VLSI implementation of high-speed 2-D DCT/IDCT chip," in *IEEE Proceedings ICIP-94*, vol. 3, pp. 606–610, November 1994.

[52] M. F. Aburdene, J. Zheng, and R. J. Kozick, "Computation of discrete cosine transform using Clenshaw's recurrence formula," in *IEEE Signal Processing Letters*, vol. 2, pp. 155–156, August 1995.

[53] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. UK: Cambridge University Press, 2nd ed., 1992.

[54] S. C. Hsia, B. D. Liu, J. F. Yang, and B. L. Bai, "VLSI implementation of parallel coefficient-by-coefficient two-dimensional IDCT processor," in *IEEE Transactions On Circuits and Systems for Video Technology*, vol. 5, pp. 396–406, October 1995.

[55] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Proceedings of ICASSP*, pp. 988–991, 1989.

[56] F. A. McGovern, R. F. Woods, and M. Yan, "Novel VLSI implementation of (8x8) point 2-D DCT," in *Electronics Letters*, vol. 30, pp. 624–626, April 1994.

[57] E. Feig and S. Winograd, "Fast algorithms for the discrete cosine transform," in *IEEE Transactions on Signal Processing*, vol. 40, pp. 2174–2193, September 1992.

[58] J. Bruguera and T. Lang, "2-D DCT using on-line arithmetic," in *IEEE Internation Conference on Acoustics, Speech and Signal Processing*, pp. 3275–3278, 1995.

[59] M. Kovac and N. Ranganathan, "JAGUAR: a fully pipelined VLSI architecture for JPEG image compression standard," in *Proceedings of the IEEE*, vol. 83, pp. 247–257, February 1995.

[60] H. C. Karathanasis, "A low ROM distributed arithmetic implementation of the forward/inverse DCT/DST using rotations," in *IEEE Transactions on Consumer Electronics*, vol. 41, pp. 263–272, May 1995.

[61] J. Pasternak and A. Shubat, "CMOS differential pass-transistor logic design," in *IEEE Journal of Solid-State Circuits*, vol. 22, pp. 216–222, April 1987.

[62] T. Masaki, Y. Morimoto, T. Onoye, and I. Shirakawa, "VLSI implementation of inverse discrete cosine transformer and motion compensator for MPEG2 HDTV video decoding," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 387–395, October 1995.

[63] S. Cucchi and M. Fratti, "A novel architecture for VLSI implementation of the 2-D DCT/IDCT," in *IEEE*, vol. V, pp. 693–696, 1992.

[64] T. Xanthopoulos, A.P.Chandrakasan, C.G.Sodini, and W.J.Dally, "A data-driven IDCT architecture for low power video applications," in *ESSCIRC'96*, September 1996.

[65] B. Stott, D. Johnson, and V. Akella, "Asynchronous 2-D discrete cosine transform core processor," in *International Conference on Computer Design*, pp. 380–385, 1995.

[66] J. Lipsher, "The asynchronous discrete cosine transform core," Master's thesis, Dept. of Electrical and Computer Eng., UC Davis, 1994.

[67] R. Smith, K. Fant, D. Parker, R. Stephani, and C. Wang, "An asynchronous 2-D discrete cosine transform chip," in *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 224–233, April 1998.

[68] K. Fant and S. Brandt, "NULL Convention Logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *International Conference on Application-Specific Systems, Architectures and Processors*, pp. 261–273, 1996.

[69] G. Lee, "A new algorithm to computer the discrete cosine transform," in *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 32, pp. 1243–1245, December 1984.

[70] K. Martin, "Application-specific processor brings high performance to DSP," in *Computer Design*, vol. 25, pp. 30–32, August 1986.

[71] M. Edwards, "Software acceleration using coprocessors: is it worth the effort?," in *Proceedings of the Fifth International Workshop on Hardware/Software Codesign*, pp. 135–139, 1997.

[72] E. Kappos and D. Kinniment, "Application-specific processor architectures for embedded control: case studies," in *Microprocessors-and-microsystems*, vol. 20, pp. 225–232, June 1996.

[73] A. van der Werf, F. Bruls, R. Kleihorst, E. Waterlander, M. Verstraelen, and T. Friedrich, "I.McIC: A single-chip MPEG-2 video encoder for storage," in *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 1817–1823, November 1997.

[74] O. Petlin and S. Furber, "Asynchronous four-phase and two-phase circuits: testing and design for testability," in *1st UK Asynchronous Forum, Edinburgh*, pp. 13–18, December 1996.

[75] "`http://www.cs.man.ac.uk/amulet/bibliography/bibsearch.html`."

[76] D. S. Bormann and P. Y. K. Cheung, "Asynchronous wrapper for heterogeneous systems," in *Proceedings.International Conference on Computer Design. VLSI in Computers and Processors*, pp. 307–314, 1997.

[77] K. v. Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[78] K. v. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij, "A fully-asynchronous low-power error corrector for the DCC player," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 1429–1439, Dec. 1994.

[79] R. Sproull, I. Sutherland, and C. Molnar, "The counterflow pipeline processor architecture," in *IEEE Design and Test of Computers*, vol. 11, pp. 48–59, Fall 1994.

[80] "`http://www.sharp.co.jp/sc/gaiyou/news-e/9710.htm`".